Front. Comput. Sci., 2017, 11(1): 88–104
DOI 10.1007/s11704-016-6041-1

RESEARCH ARTICLE

An efficient and highly available framework of data recency enhancement for eventually consistent data stores

Yu TANG, Hailong SUN, Xu WANG (✉), Xudong LIU

School of Computer Science and Engineering, Beihang University, Beijing 100191, China

Abstract Data items are usually replicated in modern distributed data stores to obtain high performance and availability. However, the availability-consistency and latency-consistency trade-offs exist in data replication, thus system designers intend to choose weak consistency models, such as eventual consistency, which may result in stale reads. Since stale data items may lead to serious application semantic problems, we consider how to increase the probability of data recency which provides a uniform view on recent versions of data items for all clients. In this work, we propose HARP, a framework that can enhance data recency of eventually consistent distributed data stores in an efficient and highly available way. Through detecting possible stale reads under failures or not, HARP can perform reread operations to eliminate stale results only when needed based on our analysis on write/read processes. We also present solutions on how to deal with some practical anomalies in HARP, including delayed, reordered and dropped messages and clock drift, and show how to extend HARP to multiple datacenters. Finally we implement HARP based on Cassandra, and the experiments show that HARP can effectively eliminate stale reads, with a low overhead (less than 6.9%) compared with original eventually consistent Cassandra.

Keywords eventual consistency, high availability, data recency, stale read

Received January 20, 2016; accepted May 5, 2016

E-mail: wangxu@act.buaa.edu.cn

1 Introduction

Modern distributed data stores often use data replication to chase high performance and high availability to serve large scale of users and provide continuous service. High availability is of importance for many applications. For example, the e-commerce service Amazon.com ensures that customers should always be able to view and add items to their shopping cars even under failures [1], because any unavailable period of time may make customers lose confidence on the company and turn to other competitive companies. Moreover, low latency is also critical for a lot of online applications. As statistic data[1,2] demonstrated, an 100 milliseconds higher latency at Amazon resulted in an 1% drop in sales; a 500 milliseconds increase of latency at Google search leaded to a 20% decrease in traffic, and 2 seconds slowdown at Bing can reduce revenue/user by 4.3%. Therefore, providing high availability and low latency is vital for modern distributed data stores.

However, the trade-offs of availability-consistency and latency-consistency are inevitable for data replication [2, 3] and often have a significant influence on system design. In order to obtain high availability under failures and provide low operation latency as well, distributed data stores often give up strong consistency models which can eliminate stale reads and instead adopt eventual consistency [1, 3–5]. An eventually consistent data store guarantees that eventually all reads will observe the last updated states when new writes are absent [5]. However, it gives no time bound on when all reads

[1] https://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-29.ppt
[2] http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html

can observe the latest updated state, and before that, it makes no guarantees on the behaviors of read operations, which can lead to possible stale read results.

In eventually consistent data stores, they typically write a data item by sending the write request to a set of replicas and read from a possibly different set of replicas. These eventually consistent data stores often employ optimistic replication mechanisms [6] to achieve higher availability and performance. For write requests, optimistic replication algorithms do not wait for all of the replicas to be synchronized during writing, and only need to contact a small subset of replicas before responding to clients. Besides, optimistic replication mechanisms do not block other updates on the same items to improve concurrency which also result in divergent replica states. For read requests, they also only need to read from a small subset of replicas. Therefore, if the replica subset of each write and those of each read are not guaranteed to overlap, read requests may return stale results.

There are two main reasons why stale reads exist in an eventually consistent data store. Firstly, updates of one same data item are usually performed in a continuous and concurrent way as the system runs. Though the PBS model [7, 8] has indicated that eventually consistent configured quorum systems, which are widely deployed as an implementation of eventual consistency, are "often consistent after tens of milliseconds" when failures are absent, continuous updates on items (especially hot items) still can make replicas inconsistent, resulting in possible stale reads. Secondly, when failures occur during updating, such as node failures and network partitions, the PBS model is not applicable. To make matters worse, some failures can make stale items exist for a long time, even without new updates coming.

To reduce the possibility of stale reads, many eventually consistent data stores have used build-in repair technologies, like read repair, hinted handoff[3)] or Merkle trees based anti-entropy mechanism [1, 9]. However, these approaches have their own limitations: read repair works after a read response has been returned, while hinted handoff and Merkle trees based anti-entropy often work periodically and thus cannot eliminate stale data in time.

In this work, we propose HARP, an efficient and highly available framework that can enhance data recency for ongoing reads of eventually consistent data stores. HARP increases the probability of data recency by detecting whether the returned data from an eventually consistent data store is stale and attempting to obtain the most recent version via rereading, with negligible performance overhead. In order to achieve the goal, HARP takes advantage of a memory-based data store (or *WT store* for short) to record each item's write timestamp (physical time or logical time), which is used to detect stale data. As the accessing speed of modern memory is much faster than that of hard disks, the introduced WT store has little performance effect on a disk-based eventually consistent data store (or *DD store* for short). For write and read operations, we analyze all possible cases by considering failures, exceptions and different comparison results of two timestamps observed from the DD store and WT store. Based on the analysis results, we find that HARP should issue a reread operation to eliminate possible stale reads just in some cases, and in other cases HARP does not need to reread, which avoids unnecessary reread operations and thus reduces the performance overheads of HARP. HARP also needs to ensure that the availability of eventually consistent data stores should not be affected. To achieve this goal, HARP adopts an optimistic solution: HARP does not need to wait for the responses from the WT store if it fails. That means, if the DD store is available but the WT store is not, HARP still can return the responses to clients. Thus the availability of HARP is actually the same with the DD store which is often highly available [1, 4]. Moreover, we present some solutions on how to deal with some practical anomalies in HARP, including delayed, reordered and dropped messages and clock drift, and show how to extend HARP to multiple datacenters.

Based on a widely-used eventually consistent data store Cassandra and an in-memory data store Redis, we implement the prototype system. In our experiments, HARP is able to handle all the stale reads caused by failures in the DD store and a majority of stale reads caused by continuous updates, with the overheads of less than 6.9% compared with the Cassandra's typically eventually consistent quorum configuration ($N = 3$, $W = R = 1$ in this paper).

In this paper, we make the following contributions:

- We propose a framework HARP, which can increase the probability of data recency on top of an eventually consistent data store in an efficient and highly available way.
- We present the design and implementation of HARP. In HARP, through analyzing all possible cases of HARP caused by failures, exceptions and comparison results of observed timestamps, we find that reread operations can be performed to eliminate possible stale reads only when needed so as to reduce the overhead. We also discuss how to keep the high availability of HARP, and

[3)] http://docs.datastax.com/en/cassandra/1.2/cassandra/dml/dml_about_hh_c.html

demonstrate how to deal with some practical anomalies including delayed, reordered and dropped messages and clock drift in HARP, as well as how to extend HARP to multiple datacenters.

- Based on the widely deployed eventually consistent data store Cassandra and the in-memory data store Redis, we implement HARP and evaluate it with the benchmark YCSB [10]. The experiments show that, HARP can eliminate all the stale reads caused by failures in Cassandra and a majority of stale reads when there are no failures in Cassandra, with the cost of less than 6.9% overhead.

The rest of the paper is organized as follows. Section 2 introduces the background of recency guarantees and eventual consistency. In Section 3, we describe the design and the implementation of HARP. In Section 4, we provide the experimental results. Section 5 discusses some limitations of HARP. Section 6 introduces the related work. Finally, we come to the conclusion in Section 7.

2 Background

In this section, we provide backgrounds on data recency guarantee which is one kind of application safety property, and eventual consistency, one of the most widely deployed weak consistency model.

2.1 Recency guarantees

Data recency guarantees are a kind of safety property, and different consistency models may provide different recency guarantees. Data recency ensures that reads to data items must return values that should obey some rules about wall-clock time or logical clock. For example, linearizability [11], one of the most famous consistency model in distributed systems, ensures that reads must return the last completed updated value of a data item, and there are also several weaker variants, like safe semantics, regular register semantics [12].

In practice, data recency is of great importance in real-world applications. We give some real scenarios in social network applications like Twitter and Facebook. If stale reads cannot be eliminated, the following scenarios may happen.

1) Tommy makes a friend with Lucy on a bus and follows her on Twitter. However, when he gets home, he cannot find her in his following list. Tommy is very disappointed because he may never meet Lucy again.
2) At the Oscars, the host Ellen tweets a selfie[4], but Lily cannot find the picture as a follower of Ellen, while her friends could. Since then, Lily has lost confidence in the social networking website.
3) Davy's new girl friend Cindy is reading his tweets. Unfortunately, she finds group pictures of Davy and his ex-girlfriend, but Davy has removed those pictures last week. Moreover, Cindy is a jealous girl, and Davy has to go through a terrible experience.

From the examples above, we can see that data recency guarantees are important for online services. Unfortunately, recency guarantees cannot be theoretically achievable for highly available distributed data stores [13]. When an indefinitely long partition occurs, a distributed data store may have to choose to return clients with probably stale values or stop online services. However, it does not mean that we cannot enhance the probability of data recency. After all, that kind of extreme failures is rare and most stale reads can be effectively avoided.

2.2 Eventual consistency

Many modern distributed data stores prefer to employ weak consistency, providing few safety guarantees to pursue availability and performance, and eventual consistency is a most notable and widely used one[5,6] [1]. Eventual consistency mainly provides a property called convergence [14], which is more a liveness property [15] than a safety property: it only ensures that all replicas will eventually agree on the same value on condition that no updates come, but it cannot guarantee that reads could witness the same value, let alone the latest updated value, at any given time.

Despite its weak safety property, there are a variety of real-world distributed storage systems offering a configuration to eventual consistency, such as quorum-based Dynamo-style replication model [1] (e.g., Apache Cassandra [4], Project Voldemort[7], Basho Riak KV[8]), as a result of preference to availability and performance.

In practice, eventually consistent data stores do not provide data recency guarantees, but their built-in mechanisms [1],

[4] http://popwatch.ew.com/2014/03/02/oscars-2014-ellen-selfie-retweeted-record-obama/
[5] https://github.com/apache/cassandra/blob/cassandra-1.2/interface/cassandra.thrift
[6] http://ria101.wordpress.com/2010/02/24/hbase-vs-cassandra-why-we-moved
[7] http://www.project-voldemort.com/voldemort/
[8] http://basho.com/products/riak-kv

including read repair, hinted handoff and anti-entropy, may alleviate staleness.

- Read repair is aiming at repairing replicas that have stale versions of the reading item, and takes place after read responses have been returned to clients. During a process of read repair, a coordinator collects read responses from all available replicas and chooses the latest version. Afterwards, the coordinator updates those replicas that do not hold the latest version to make all replicas consistent.
- Hinted handoff can be used to make the store recover from temporary failures. When a node A becomes unavailable and some update requests come, other available nodes will store corresponding hints including which node was the intended recipient of the updates, and the updates that should be sent to the unavailable node. After detecting that the unavailable node has become available, nodes storing the hints will deliver the corresponding updates to A. Thus, hinted handoff attempts to make the recovered node consistent with others.
- Merkle trees based anti-entropy is applied to handle recovery from permanent failures. Eventually consistent data stores use Merkle trees to detect the inconsistencies among available replicas. If there are inconsistencies, data stores will perform synchronization actions to eliminate staleness.

However, these mechanisms have their own limitations in enhancing data recency. In summarize, read repair works after read responses have been returned, while hinted handoff and Merkle trees based anti-entropy approaches often work periodically and cannot eliminate stale reads in time.

3 HARP

In this work, we take eventually consistent key-value stores into account. In the key-value model, data is represented as a collection of key-value pairs, and each key is unique in the collection. According to whether a read operation can observe the results of the latest successful write, we classify read operations to *recency read*s and *stale read*s: if a read observes the result that reflects the latest successful write, it is called a recency read; otherwise, we name the read operation as a stale read. The classification derives from the linearizability [13] model, which ensures that a read must return the last completed write to a data item. Therefore, a read that obeys the restriction required by linearizability is a recency read. However, we do not intend to provide a strong guarantee like linearizability.

HARP settles between an eventually consistent data store and the application layer (as shown in Fig. 1). To store each update's timestamp efficiently, HARP leverages a memory-based WT store, which is also placed under the HARP. HARP mainly consists of many agents. Each agent manages the write and read operations issued by clients, and attempts to detect and deal with stale reads.

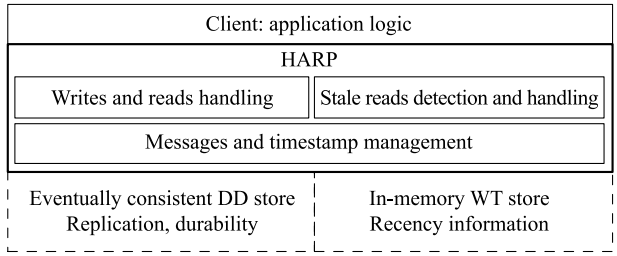

Fig. 1 The logic model of the system

HARP mainly includes three functional modules. The first module handles the write and read processes in HARP (Section 3.1). The second module is designed to detect and handle stale reads (Section 3.2). The third module manages messages and timestamps, and also deals with practical anomalies like delayed, reordered and dropped messages and clock drift (Section 3.3). We also present how to extend HARP to suit for a multiple datacenter environment. In the end of this section, we present an implementation of HARP.

3.1 Handling writes and reads

For write operations, except for storing the item value into the underlying DD store, HARP also needs to store the corresponding write timestamps, obtained from physical clocks or logical clocks, in both DD store and WT store. Compared with that in an eventually consistent data store, a write process managed by HARP requires an extra parameter and an additional write request: the write timestamp and the write request sent to WT store. Denote a write operation handled by the DD store as $w_d(k)$, where k is the corresponding key of the write. Similarly, a write operation handled by the WT store is denoted as $w_t(k)$. Thus, a write operation $w(k)$ handled by a HARP agent will issue $w_d(k)$ and $w_t(k)$, as shown in Fig. 2. To reduce the overall operation latency, $w_d(k)$ and $w_t(k)$ are performed simultaneously. However, the paired operations $w_d(k)$ and $w_t(k)$ may lead to different execution results. Consider the following situations:

W_1: Both $w_d(k)$ and $w_t(k)$ are failed.

W_2: $w_d(k)$ is successful while $w_t(k)$ is failed.

W_3: $w_t(k)$ is successful while $w_d(k)$ is failed.

W_4: Both $w_d(k)$ and $w_t(k)$ are successful.

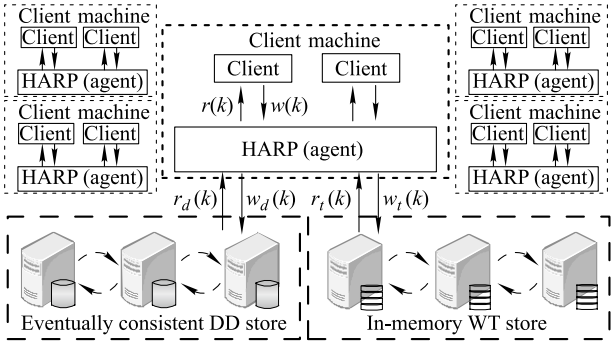

Fig. 2 The architecture to implement HARP (Each agent handles read and write operations issued by clients, and distributes corresponding requests to the underlying stores)

Denote the set $\{W_1, W_2, W_3, W_4\}$ as $\mathscr{W}$. Successful $w_d(k)$ (or $w_t(k)$) means data (or time) is successfully written into the DD store (or WT store) and the corresponding response is received by the agent within a timeout bound; otherwise, $w_d(k)$ (or $w_t(k)$) is considered failed. For W_1 and W_4, $w(k)$ will return *failure* and *success*, respectively, as $w_d(k)$ and $w_t(k)$ are all failed and successful, respectively. For W_2, $w_d(k)$ is successful, but $w_t(k)$ is not. To maintain high availability as the native DD store, the agent will return the client a *success* message. While for W_3, the agent will return a *failure* message. Therefore, for write operations, to achieve high availability as the native DD store, the agent will return a *success* message to the client if $w_d(k)$ is successful; otherwise the agent will return a *failure* message. Among the four situations, W_4 is the most common case because failures are absent most of the time, but other cases also exist in practical data stores. The inconsistency brought in by W_2 and W_3 will be taken into consideration during handling read operations.

Denote a read operation handled by an agent as $r(k)$, where k is the operating key. Similar to a write operation, a read operation $r(k)$ handled by an agent will also issue two paired reads: one that is sent to the DD store is denoted as $r_d(k)$, and the other that is sent to the WT store is denoted as $r_t(k)$. $r_d(k)$ returns the item value and the timestamp that value was updated, which is denoted as $time_d$, while $r_t(k)$ returns the item's latest updated timestamp, which is denoted as $time_t$. There are also several situations for an agent to execute $r(k)$:

R_1: Both $r_d(k)$ and $r_t(k)$ are failed.

R_2: $r_d(k)$ is successful while $r_t(k)$ is failed.

R_3: $r_t(k)$ is successful while $r_d(k)$ is failed.

R_4: Both $r_d(k)$ and $r_t(k)$ are successful, and:

R_{4a}: $time_d < time_t$,

R_{4b}: $time_t < time_d$,

R_{4c}: $time_t = time_d$.

Denote the set $\{R_1, R_2, R_3, R_{4a}, R_{4b}, R_{4c}\}$ as $\mathscr{R}$. Successful $r_d(k)$ means the read handled by the DD store returns the item value (not *null*) and the corresponding updated timestamp within a timeout bound; otherwise, $r_d(k)$ is failed. Successful $r_t(k)$ means the read handled by the WT store returns the timestamp (not *null*) within a timeout bound; otherwise, $r_t(k)$ is failed. Specially, for R_4, both $r_d(k)$ and $r_t(k)$ observe data in the underlying stores, but either may observe stale data. However, we can compare the timestamps obtained from DD store and WT store to predict whether data is stale. According to the comparison result, we classify them into three categories: R_{4a}, R_{4b} and R_{4c}. We use "<" to symbolize that the timestamp on the left side of the operator is smaller than that on the right side (such as R_{4a} and R_{4b}), while "=" stands for that the timestamps on both sides of the operator are equal (such as R_{4c}).

3.2 Handling possible staleness

In this subsection, we firstly present how the execution results of finished writes can influence that of following reads. Then we analyze which cases in $\mathscr{R}$ need second-round reads to eliminate possible stale read, while other cases do not need to.

To begin with, we denote the "result in" relationship between $w(k)$ and the corresponding execution situation W_x as "$\rightarrow$": $w(k) \rightarrow W_x$, where $W_x \in \mathscr{W}$. Denote the "result in" relationship between $w(k)$ and an execution situation of a read as "$\mapsto$": if $r(k)$ follows $w(k)$ and there is no subsequent writes to the same key happens between $w(k)$ and $r(k)$, and the result of $r(k)$ is a situation R_x, where $R_x \in \mathscr{R}$, then $w(k) \mapsto R_x$. Moreover, we denote the "result in" relationship between W_x and R_y as $W_x \Rightarrow R_y$: if $w(k) \rightarrow W_x$ and $w(k) \mapsto R_y$, then $W_x \Rightarrow R_y$. Extensively, $w(k) \mapsto \{R_x\}$ represents $\forall R \in \{R_x\}$ such that $w(k) \mapsto R$. Besides, if $w(k) \rightarrow W_x$ and $w(k) \mapsto \{R_y\}$, then $W_x \Rightarrow \{R_y\}$. Table 1 concludes the usage of the above three "result in" symbols. We divide write operations into two kinds: insert operations and update operations. An insert operation means the DD store has not stored the writing item before, while an update operation means the DD store has already stored the item.

Table 1 The supplement of "result in" symbols

Symbol	Object on the left side	Object on the right side
$\rightarrow$	A write operation	The execution result of the write
$\mapsto$	A write operation	A (or a set of) possible execution result(s) of the following read(s) to the same data item
$\Rightarrow$	An execution result of a write operation	A (or a set of) possible execution result(s) of the following read(s) to the same data item

Understanding how the execution results of finished writes can influence that of following reads can help HARP to know how to efficiently handle every case in $\mathscr{R}$. Therefore, we should analyze the detail "result in" relationships between a write's execution result and a following read's execution result. For simplicity, we assume the WT store always returns the most recent data it has stored, but we will loose the assumption in the end of the analysis.

$W_1 \Rightarrow \mathscr{R}$. Firstly, we present that although $w_d(k)$ and $w_t(k)$ are failed, following reads can still observe the item and its corresponding timestamp. If $w(k)$ is an update operation, though $w_d(k)$ and $w_t(k)$ are failed, previous writes may have already inserted the item and corresponding timestamp into the DD store and WT store. Therefore, following reads can observe an older item and its corresponding timestamp. Besides, though $w_d(k)$ and $w_t(k)$ are failed, it does not mean $r_d(k)$ cannot observe the value $w_d(k)$ attempts to write, because there is a possibility that can cause $w_d(k)$ to fail: data has been written into the DD store, but the agent cannot receive the *success* acknowledgement because of network or node failures. Similar situation can happen between $w_t(k)$ and $r_t(k)$. We call the situation like that a *false negative*. If a false negative only happens to $w_d(k)$, then the following read can get new data from the DD store but older time from the WT store, resulting in R_{4a}; similarly, if a false negative only happens to $w_t(k)$, then the following read can get older data from the DD store but new time from the WT store, resulting in R_{4b}; moreover, if false negatives happen to both $w_d(k)$ and $w_t(k)$, then the following read can get fresh data and fresh time, resulting in R_{4c}. Further, due to various failures, $r_d(k)$ and $r_t(k)$ can be failed during handling read operations. If both $r_d(k)$ and $r_t(k)$ are failed, $W_1 \Rightarrow R_1$; if either $r_d(k)$ or $r_t(k)$ is failed, $W_1 \Rightarrow \{R_2, R_3\}$. Above all, $W_1 \Rightarrow \mathscr{R}$.

$W_2 \Rightarrow \mathscr{R}$. Similar to W_1, $W_2 \Rightarrow \{R_1, R_2, R_3\}$ holds as $r_d(k)$ or $r_t(k)$ may be failed. If a false negative does not happen to $w_t(k)$ and the DD store returns the most recent data, then $W_2 \Rightarrow R_{4b}$. If a false negative happens to $w_t(k)$, there are two cases: if the DD store returns the most recent data, then $W_2 \Rightarrow R_{4c}$; otherwise, as an eventually consistent data store, the DD store returns stale data, then $W_2 \Rightarrow R_{4a}$. Above all, $W_2 \Rightarrow \mathscr{R}$.

Similar to W_2, we can get $W_3, W_4 \Rightarrow \{R_1, R_2, R_3, R_{4a}, R_{4c}\}$. But $W_3, W_4 \nRightarrow R_{4b}$ because the WT store is assumed to return the most recent data it has stored. However, if we loose the assumption that the WT store always returns the most recent data, we can get $W_1, W_2, W_3, W_4 \Rightarrow \mathscr{R}$.

For six kinds of execution results a read may be confronted with, we demonstrate how HARP detects and handles possible stale data during handling read operations. The principle of HARP is increasing the possibility of recency reads in an efficient and highly available way. We assume that failures are rare but may occur, and the WT store always returns the most recent timestamp it has stored.

The PUT procedure in Algorithm 1 concludes the write process. The procedure works in the HARP module and will write an item with the input key k and value v into the DD store DS. Besides, the procedure also needs to get the local timestamp and insert it to the WT store TS. The output of the PUT procedure is whether $w_d(k)$ is successfully executed.

The GET procedure in Algorithm 1 concludes the read process. The procedure works in the HARP module and will fetch an item (including its value v and timestamp $time_d$) corresponding to the specified input key k from the DD store DS. Besides, the procedure also needs to get the corresponding timestamp $time_t$ from the WT store TS to detect possible stale reads. The output of the GET procedure is the item value corresponding to the input key k. Moreover, the GET procedure may invoke the RESOLVE procedure to handle a possible stale read if it is detected. The RESOLVE procedure will then try to get an item value corresponding to a larger timestamp from the DD store DS via reread operations, which are determined by the users' configuration. The RESOLVE procedure also needs the timestamps $time_t$ and $time_d$ respectively obtained from the WT store and the DD store as the input parameters to handle the possible stale read. There is no output for the RESOLVE procedure, as it will update the input item value v if the RESOLVE observes a more recent version and the GET procedure will observe the update.

There are several cases that can lead to R_1: both $r_d(k)$ and $r_t(k)$ catch an exception (e.g., timeout, connection refused), or both receive *null* value, or one catches an exception message, the other receives *null* value. The reasons causing those to happen can be node failures or network partitions, sometimes it is because data or time has been deleted or, actually,

Algorithm 1 Detecting and handling possible stale reads

```
1:  procedure (key k, val v, DD store DS, WT store TS)
2:    // get local incremental timestamp
3:    time ← getCurrentTime()
4:    // w_d(k) and w_t(k) are run concurrently
5:    TS.put(k, time) || result ← DS.put(k, v, time)
6:    // whether a put is successful is up to w_d(k)
7:    return result
8:  end procedure

9:  procedure (key k, DD store DS, WT store TS)
10:   // try to catch exceptions for r_t(k), if they exist
11:   {v, time_d} ← DS.get(k) || {time_t, ex} ← TS.get(k)
12:   if v == null then   // situation R1 or R3
13:     if ex ≠ null or time_t == null then // situation R1
14:       return null
15:     end if
16:     resolve(DS, TS, k, time_t, 0, v)     // situation R3
17:     return v
18:   end if
19:   if time_t == null then     // situation R2
20:     if ex == null then
21:       // time in the WT store is null, so update it
22:       TS.put(k, time_d)
23:     end if
24:     return v
25:   end if
26:   if time_d < time_t then     // situation R4a
27:     resolve(DS, TS, k, time_t, time_d, v)
28:   else if time_t < time_d then
29:     // situation R4b. Update the time in the WT store
30:     TS.put(k, time_d)
31:   end if
32:   return v
33: end procedure

34: procedure RESOLVE(DD store DS, WT store TS, key k, time time_t,
    time time_d, value v)
35:   {times_reread, level_reread} ← userDefineConfig()
36:   loop times_reread times
37:     {v_reread, time_reread} ← DS.get(k, level_reread)
38:     if time_t < time_reread then
39:       // update the stale time in the WT store
40:       TS.put(k, time_reread)
41:       v = v_reread
42:       return
43:     else if time_t == time_reread then
44:       v = v_reread
45:       return
46:     end if
47:     if time_d < time_reread then
48:       v = v_reread
49:     end if
50:     end loop
51:     return
51: end procedure
```

they have not been stored successfully as a result of W_1–W_3. In these cases, the agent will return *null*, either to maintain high availability without waiting for the successful message, or just return the obtained *null* value from DD store. Lines 13–15 in the Algorithm 1 show the process.

For R_2, $r_d(k)$ obtains data from the DD store while $r_t(k)$ does not. The reasons why $r_t(k)$ does not get time from WT store are the same as those for situation R_1. As $r_t(k)$ is failed, there is no evidence to suggest whether the observed data is stale. Therefore, the agent could return *null* to decrease the number of stale reads. However, this solution is unacceptable, because when the WT store becomes unavailable while the DD store works well, the agent will also return null: failures in WT store may result in clients' reading nothing—that is not what HARP desires. Actually, returning the obtained data from the DD store will not increase the possibility of stale reads compared to original reads on the DD store, because the data is just obtained from the DD store. Therefore, as an optimistic solution, returning the possible stale data is acceptable. Besides, if $r_t(k)$ returns *null* value instead of catching any exceptions, the agent needs to update the timestamp in the WT store, because the timestamp has not been written into the WT store in the last write. Lines 19–25 in Algorithm 1 show the process.

For R_3, $r_t(k)$ obtains a timestamp from the WT store while $r_d(k)$ does not get the corresponding item. The phenomenon means there was a write operation that has tried to insert the item, but $r_d(k)$ does not observe the item because of accessing a stale replica or failures. Therefore, an agent needs to deal with this situation to exclude possible stale reads. Lines 16–17 in Algorithm 1 show the process.

For R_{4a}, both $r_d(k)$ and $r_t(k)$ return timestamps from the underlying data stores, and $time_d < time_t$. Though W_1, W_2 and W_3 can result in the situation, considering W_4's high probability as we assume failures are rare, there is a high probability that the obtained data is stale. This is a situation when the agent needs to deal with the probable staleness. If a R_{4a} is actually "result in" effect of a W_3, which means $r_d(k)$ may not be stale, treating the $r_d(k)$ as a stale read and issuing a second-round read will not cause side effects. Lines 26–27 and 32 in Algorithm 1 show the process.

For R_{4b}, the timestamp obtained from the DD store is larger than that obtained from the WT store. The most probable reason is that the timestamp $time_t$ of the latest write has not been written into the WT store yet. Therefore, the agent also needs to update the timestamp in the WT store to $time_d$. Lines 28–30 and 32 in Algorithm 1 show the process.

Ideally, R_{4c} means the handling read is a recency read. However, in the real world where uninvited failures exist, R_{4c} also cannot guarantee the obtained item is not stale. Though the obtained time from the DD store and WT store are equal, the data is also possible to be stale: as a result of W_2, writes to

the DD store are successful, while the corresponding timestamps cannot be stored into the WT store; then a following read obtains the timestamp from the WT store and stale timestamp from the DD store, and the two obtained timestamps are equal by coincidence. Nonetheless, the possibility of that situation is low and detecting the possible stale reads in such a situation is costly. Therefore, in this situation, the agent can return the obtained data to clients. Line 32 in Algorithm 1 shows the process.

As above, R_3 and R_{4a} are the situations where HARP needs to deal with possible stale reads. The key to resolve staleness is obtaining a data item with the specified time no less than $time_t$. As the agent and WT store do not persist data, the agent has to issue a second-round read to get the item from the DD store. Even though the required item has been stored successfully, with an eventually consistent configuration, the DD store may still return stale data again. However, some DD stores provide a tunable consistency level for client requests. For example, for some quorum-based data stores, the number of replicas a read attempts to access is tunable during runtime[9, 10] , which means clients can tune the consistency level of reads from basically eventual to some stronger ones, e.g., "read your writes", by increasing the number of replicas that need to access. Therefore, a reread operation can try to access more replicas to increase the possibility of obtaining the most recent item. For data stores whose consistency levels are not tunable during runtime, the agents may retry reread operations for more than one time to increase the probability of obtaining the most recent data, and the retry times depends on user's concern about stale reads. Because of the differences between various DD stores, how to resolve staleness is subject to system implementation. The RESOLVE procedure in Algorithm 1 shows the process.

We have analyzed how to deal with six possible situations that reads may encounter when all write operations are finished. Actually, the algorithm is also applicable when reads observe ongoing writes, because no matter what the execution results of the ongoing writes are, the execution results of the concurrent reads also fall in the six situations.

According to Algorithm 1 described above, network or node failures in the WT store will not affect the availability of HARP. Moreover, if the WT store is available, HARP can carry on the staleness detecting and handling process. To reduce introduced overheads, in most cases, HARP will directly return data obtained from the DD store; while for R_3 and R_{4a}, HARP will force a reread operation to attempt to get newer data. Therefore, HARP will not increase the possibility of stale reads compared with the native eventually consistent DD store. On the contrary, HARP can increase the proportion of recency reads.

3.3 Additional design

3.3.1 Handling practical messages

In practical distributed systems, messages can be delayed, reordered, or dropped. We mainly take update request messages into account, as data states are more sensitive to abnormality of update request messages. Firstly, we present how to handle reordered and dropped messages, and then show how to deal with delayed messages. To simplify presentation, we do not consider node failures in this part. Reordered, or dropped update requests may make states in the data store become inconsistent. Reordered update requests may cause data states to "draw back", and Fig. 3 presents two possible situations that may lead to the anomaly: Figs. 3(a) and 3(b) demonstrate "draw back" anomalies caused by reordered requests occured in the DD store and DT store, respectively. A "draw back" anomaly may make an older value replace the most recent value, which should be prohibited in HARP. As each update request attaches the timestamp that represents the moment of issuing, we can leverage the timestamp to avoid "draw back" anomalies: when a server receives an update request, it compares the attaching timestamp of the request with the timestamp of the local item: if the updating item does not exist in the local server or the timestamp of the request is larger, the server performs the update; otherwise, abandon the update request or make the effect of the update invisible. In this way, servers can free from "draw back" anomalies. For example, in Fig. 3(b), as $t_a < t_b$, the server abandons the w_0 request and the item value will not be changed from t_b to t_a. For dropped messages, their senders will catch a timeout exception and can resend the messages for several times until receiving the corresponding acknowledgements.

A delayed update request message may lead to three undesirable consequences: the message is considered to be

(1) *reordered*: the message is delayed such that it comes after a later message in the server (as in the Fig. 3);

(2) *dropped*: the message is delayed such that its sender cannot receive a response within a timeout bound;

(3) both *reordered* and *dropped*: both (1) and (2) may happen simultaneously as they are not mutual exclusive.

[9] http://www.datastax.com/documentation/cassandra/1.2/cassandra/dml/dml_config_consistency_c.html
[10] https://github.com/basho/riak-java-client/wiki/Fetching-Data-from-Riak

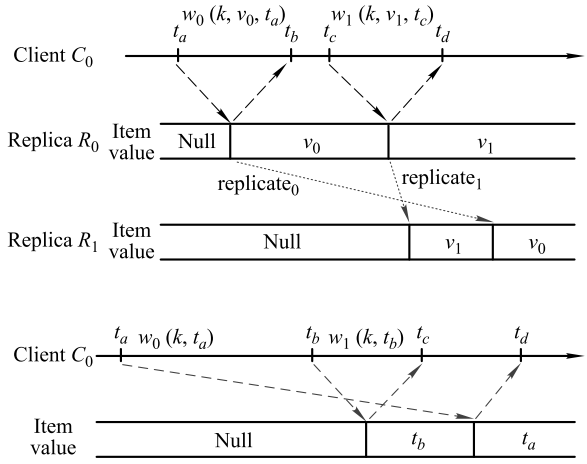

Fig. 3 "Draw-back" anomalies caused by reordered message (The horizontal direction represents time; crossbars and the specified local timestamp t_x correspond to the start and end time for each operations, which are demonstrated between crossbars). (a) DD store; (b) WT store

The solution of reordered messages described ahead in this subsubsection can deal with the situation (1). However, employing the solution of dropped messages to deal with the situation (2) will lead to duplicate messages, because delayed messages are not really dropped. Nonetheless, as operations in HARP are all idempotent, duplicate messages will not result in undesirable consequence. The combination of the solutions of (1) and (2) can deal with the situation.

In summary, our solution guarantees that replicas in both DD store and WT store will eventually converge to the state updated by the writes that possess the largest timestamps.

3.3.2 Handling clock drift

Clock drift is inevitable in distributed systems, and it can reverse the relation "happened before" of the events in a distributed system [16]. For example, as shown in Fig. 4, if $t_e < t_a$, which is possible considering clock drift, then the item value will still be v_0 after the global time t exceeds t_0, because w_1 is ignored according to Section 3.3.1. That means a causally later update may be ignored by the data store due to clock drift. The concept "*causally later*" in this paper means for any operations a and b the following conditions should be satisfied:

(1) If a and b are operations issued by the same client, and a happens before b, then b is a causally later operation (compared with a).

(2) If a is a write operation and b is a read operation that observes the execute result of a, then b is a causally later operation (compared with a).

(3) If c is a causally later operation compared with a, and b is a causally later operation compared with c, then b is a causally later operation compared with a.

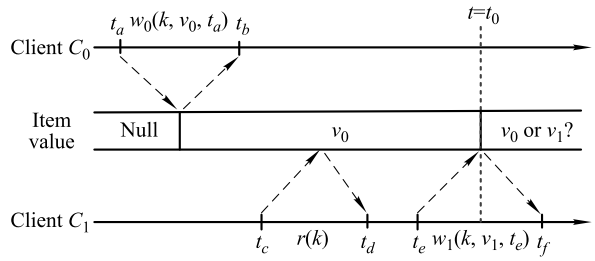

Fig. 4 An example of an anomaly caused by clock drift (The vertical dashed lines represents the global time, but clocks in clients differ due to clock drift. To simplify the presentation, the example only considers the scene where there is one replica in the system)

To eliminate the anomaly, we employ a strategy similar to the Lamport clock [16] in HARP. The strategy includes two main steps:

(1) When a client issues a write, the local timestamp should be attached to the write request and be stored in the data store along with the item value;

(2) When a client observes an item from the data store, it also observes the corresponding timestamp stored during writing. The client needs to compare the observed timestamp with the local timestamp: if the previous value is larger, the client should update the local timestamp to the observed timestamp.

The first step is already contained in the procedures of HARP, and HARP needs to add the second step when handling write operations. The second step only needs clients to execute a compare and set operation when handling each read. Therefore, the strategy will not introduce too much performance overheads. By employing the strategy, causally later writes will not be ignored by the data store, because they possess larger timestamp, unless some newer writes have been performed earlier. As illustrated in Fig. 4, if the strategy is applied, the client C_0 will attach the local timestamp t_a to the write request. Afterwards, the client C_1 observes the item and the timestamp t_a, and then C_1 updates the local timestamp to t_a if $t_a > t_d$. Therefore, $t_e > t_a$ holds and the item value will be updated to v_1 after w_1 is performed. In summary, the strategy guarantees that a causally later update has a larger timestamp.

3.3.3 Multiple datacenters

The previous description is based on the assumption where all servers are deployed in a single datacenter. Nonetheless, HARP can also be applied to multiple datacenters with some

changes about recency enhancement.

In the scene of multiple datacenters, data replicas are deployed in different datacenters to provide the disaster tolerance property or reduce latency for worldwide users. Clients usually communicate with the local datacenter first to reduce user-aware latency. In this situation, if HARP still maintains a globally single replica for each item in the WT store, then an agent may wait more time to communicate with the item's recency metadata placed in the remote datacenter, causing considerable performance overheads. However, if HARP employs one replica for recency metadata in each datacenter, the consistency model of the WT store deserves attention: if strong consistency model like linearizability is adopted, the performance of the whole system will be largely influenced, considering that the DD store is eventually consistent. Therefore, the WT store also adopts eventual consistency to keep pace with the DD store to chase high performance.

Nevertheless, for each datacenter, HARP is often able to make corresponding clients observe updates issued by the clients that communicate with the same datacenter. Such property of observing the latest local updates handled within each datacenter is termed *local data recency* in this work. That means, HARP may not guarantee observing the globally latest updates among different datacenters especially when users' read requests are scheduled to remote datacenters by some scheduler strategies. However, scheduling requests to different datacenters will incur considerable overhead due to wide-area network delay. In practice, most multiple datacenters are deployed across different regions and user requests from a same region are usually delivered to the same nearest datacenter. Local data recency is very useful for this situation when the popularity of each item differs across different datacenters, e.g., geo-distributed social networkings, where end users are more interested in knowing fresh events nearby and user requests are always sent to the nearest datacenter to improve user experience.

3.4 Implementation

As the design described above, there are three primary modules: an eventually consistent DD store, an in-memory WT store and an agent layer. For the DD store, we use Cassandra, a widely deployed quorum-based Dynamo [1]-style NoSQL store. We do not modify the source code of Cassandra and only change some configuration parameters as recommended by official documentation[11]. Cassandra provides two kinds of consistency configurations. One is the strict quorum configuration, which can provide strong consistency guarantee of reads and writes to replicas by ensuring read and write replica sets overlap: given N replicas, together with read and write quorum sizes R and W respectively, $R+W > N$ must be guaranteed. Using the strict quorum configuration, stale reads can be eliminated. The other is the partial (or non-strict) configuration, which provides eventual consistency, for $R + W \leqslant N$. There are also some works [7, 17], which are aiming at quantifying consistency, availability or latency based on quorum systems, helping users determine the value of N, W, R.

For the WT store, we use Redis[12], an open source key-value main memory WT store. Redis is able to save data into disk and supports master-slaver replication, so a node failure of Redis will not result in a loss of data. As data volume can be very large and user requests can be frequently, a single Redis node may not be able to handle that situation, so we set up a Redis cluster. We modify the Redis source code to implement the strategy described in Section 3.3.1 and deal with the "draw back" anomaly in the WT store. While we do not modify the Cassandra source code, because Cassandra can avoid "draw back" anomalies in a similar way: a Cassandra data node can maintain multiple versions for each data item and will return the version that possesses the largest timestamp during handling reads.

We implement the HARP module in Java. The HARP module consists of multiple HARP agents, and the strategies described in Sections 3.1, 3.2 and 3.3.2 are implemented in each HARP agent. HARP agents are placed in every client machines, and the number of HARP agents will increase as the number of client machines does. Therefore, HARP will not introduce much performance overhead with the increasing of client machines.

By default, an HARP agent sets the consistency level that is used to access the underlying Cassandra as $W = 1$ and $R = 1$[13], and sets $W = N - 1$ for reread operations and the maximum retry times for each reread is 3.

4 Evaluation

In this section, we experimentally show that HARP can enhance data recency by detecting and resolving stale reads when there are failures or not. Besides, we compare the performance of HARP with an eventually consistent configuration and a stronger "read your writes" configuration of

[11] http://www.datastax.com/documentation/cassandra/1.2/cassandra/initialize/initializeSingleDS.html
[12] http://redis.io/
[13] https://github.com/apache/cassandra/blob/cassandra-1.2/interface/cassandra.thrift

Cassandra. Furthermore, we also demonstrate that practical anomalies can happen in a distributed system and HARP can deal with them in an efficient way.

4.1 Experimental setup

We evaluate the system with the YCSB (Yahoo! Cloud Serving Benchmark) [10]. The agent layer is placed between YCSB and underlying data stores Cassandra and Redis. In the experiment, Cassandra runs on a cluster of 5 nodes. Each node has 2.0GHz eight-core Processor, 16 GB memory and 200 GB HDD. Redis runs on a cluster of 2 nodes, and each node also has 2.0GHz eight-core Processor, 16 GB memory. All nodes are connected by 1Gbps Ethernet. Each node uses 64bit Debian Linux 3.2.0. By default, YCSB uses 20-byte row keys, while the average size of each value is 1 byte, which adversely affects recency metadata overheads in HARP, and the replica factor N is 3 [1]. For client loads, we choose uniform distribution or Zipfian distribution to simulate user behaviors.

4.2 Data recency

The general reason that data recency cannot be guaranteed for an eventually consistent data store, is that the latest update on a data item cannot be propagated to all replicas in time, and thus a following read may only access a replica that possess a stale item, resulting in a stale read. However, the detail reasons can be classified into two kinds according to whether failures are absent or not (as described in Section 1). Therefore, we check whether HARP really can enhance data recency by eliminating stale reads in two cases: failures are 1) absent, as well as 2) present. In this experiment, we enable the default failure recovery mechanisms in Cassandra, including read repair and hinted handoff.

When failures are absent, Fig. 5 shows the numbers of detected stale reads with and without HARP applied, represented as Cassandra and HARP respectively. In this experiment, the read proportion of the workload is 80%, and each experimental instance runs continuously for 150 seconds. Client requests obey a Zipfian distribution and the Zipfian factor is set to 0.88, which makes 80% of the requests concentrate on 20% of ten million data items. If a client thread observes (or updates) a data item, and after that, a read operation launched by any client thread in the same client machine observes the same item but older versions, then the read operation is considered as a stale read. As Fig. 5 demonstrates, HARP can eliminate stale reads by more than two orders of magnitude compared with eventually consistent Cassandra.

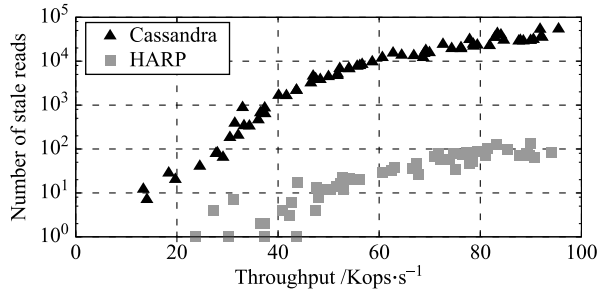

Fig. 5 Detected stale reads when failures are absent

We also check whether HARP can really decrease the possibility of stale reads when a node recovers from failures. In this experiment, we insert ten million items to the data store at the beginning. Then we terminate the Cassandra thread on one server, update two million data items on other available nodes, and restart the terminated thread. After the Cassandra thread restarts, we wait for a specified period of time, launch 100 clients to issue random reads for 60 seconds, and record the number of stale reads detected by HARP.

The experimental procedures simulate a single server's temporary failure and a following recovery. In this scenario, we updates two million items during the server failure; after the failed server recoveries, clients issue randomly reads on all servers, which may result in observing stale data. HARP is applied to detect and try to fix stale reads that are not eliminated by the failure recovery mechanisms in Cassandra, and stale reads in the Fig. 6 are all detected by HARP. After an experimental process is finished, the Cassandra database is clean up, and the experimental procedures are repeated 30 times for each specified waiting period of time.

During the experiment, no exception or error is catched, and there is enough time to apply every write in the WT store, so a situation of R_{4b} means the read is definitely a stale read, while R_{4c} means the read is not stale. The nodes and error bars of single failure in Fig. 6 shows the experimental results. As the waiting time increases, the number of stale reads decreases. That is because failure recovery mechanisms in Cassandra works and stale items are gradually updated. However, failure recovery mechanisms in Cassandra cannot deal with some stale reads in time (5 minutes in the experimental scenario), while HARP is able to repair all of them.

There are some situations where hinted handoff in Cassandra does not work. Repeat the procedures in previous experiment with one modification: before restarting the terminated Cassandra process, we terminate a Cassandra process on another server. The experimental procedures simulate a scenario where a new failure happens on another server before the server that suffers from temporary failure recovers. The

nodes and error bars of consecutive failures in Fig. 6 show the experimental results. As the waiting time increases, stale reads cannot be completely eliminated, because the stored "hint" [1] becomes unavailable along with the second failed server. While HARP can deal with the stale reads on this occasion. However, HARP cannot take place of hinted handoff and read repair, because the latter mechanisms work at the inner data store and can update stale replicas, while HARP works outside and does its best to make clients observe fresh items, instead of updating stale replica states.

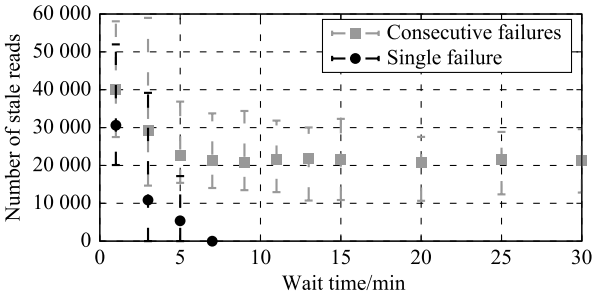

Fig. 6 Detected stale reads caused by failure recovery

4.3 Performance

In this subsection, we evaluate the performance of different systems, including HARP and two different configurations of Cassandra: eventually consistent Cassandra (or shorted as Cassandra-Eventual) and strongly consistent Cassandra (or shorted as Cassandra-Strong). Cassandra-Eventual and Cassandra-Strong are different in the read and write quorum sizes: both W and R are set to 1 in Cassandra-Eventual, while both W and R are set to $\lfloor N/2 \rfloor + 1$ in Cassandra-Strong. Therefore, for any $N \geqslant 3$, Cassandra-Strong can tolerant one replica's failure. To expose the behavior of an originally eventually consistent data store, by default, we disable the failure recovery mechanisms of Cassandra, including read repair and hinted handoff.

Figure 7 demonstrates how the proportions of reads affect peak throughput. As shown in Fig. 7(a), compared with the eventually consistent configuration, the overhead brought by HARP is less than 5.7%. Meanwhile, HARP outperforms the Cassandra-Strong configuration for all workloads. Especially for read-most workloads, whose read proportion is larger than 50%, HARP achieves 69.5%–88.0% higher performance. The throughput gap between Cassandra-Strong and Cassandra-Eventual (or HARP) is enlarged as read proportion increases. The reason is that the number of requests that a coordinator needs to issue is different during reading: for Cassandra-Eventual and a first-round read in HARP, a coordinator only needs to send one (as $R = 1$) message to replicas; while in Cassandra-Strong, a coordinator has to send two (as R=$\lfloor N/2 \rfloor + 1$) messages. Though a second-round read in HARP also needs to send two (as $R = N-1$) messages, but the possibility of incurring a second-round read is small in this experiment. The main reason why the throughput decreases as read proportion increases, is that Cassandra nodes append data in CommitLog and Memtable[14] firstly and then flush data into disk asynchronously, while reads usually have to access disks[15].

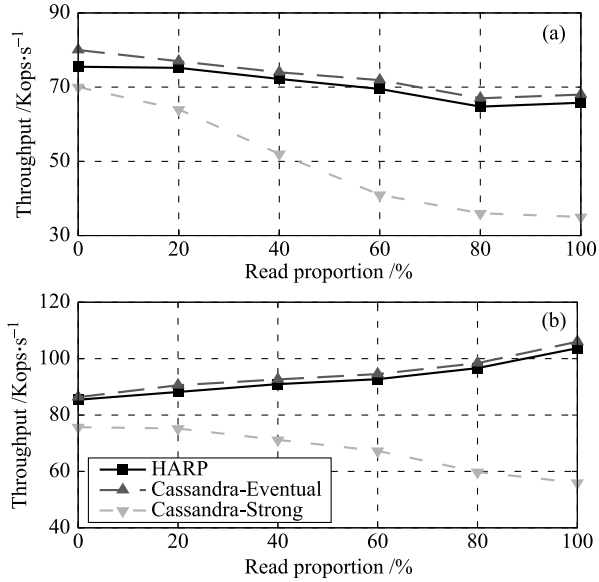

Fig. 7 Peak throughput of different workloads. (a) Uniform distribution; (b) Zipfian distribution

Under the Zipfian distribution, HARP brings in negligible overhead (less than 2.7%) compared to Cassandra-Eventual, and outperforms Cassandra-Strong (37.7%–85.2% for read-most workloads), as shown in Fig. 7(b). Compared with the throughput under the uniform distribution, the reason why throughput is higher under the Zipfian distribution is that more operations only need to access the Memtable (cache) of Cassandra, and thus decrease the number of operations on disks.

We measure the latency for different systems under the similar throughput. Tables 2 and 3 represent the corresponding latency under the uniform distribution and the Zipfian distribution. The read latency of Cassandra-Strong is much higher than that of Cassandra-Eventual and HARP, while read latency of HARP is slightly higher than that of Cassandra-Eventual. The write latency of all three systems are close. As

[14] https://wiki.apache.org/cassandra/MemtableSSTable
[15] http://wiki.apache.org/cassandra/ArchitectureInternals

Table 2 Latency/ms for different systems under the uniform distribution with approximate throughput

Operation	System	Average	50%	90%	95%	99%	99.9%	Throughput/Kops·s^{-1}
Read-only	Cassandra-Eventual	1.20	0.95	1.97	2.57	4.61	9.15	29.8
	HARP	1.35	1.00	2.25	3.04	5.97	13.62	29.6
	Cassandra-Strong	9.48	7.26	15.43	20.33	44.71	266.13	29.5
Write-only	Cassandra-Eventual	0.51	0.43	0.72	0.78	0.97	4.43	29.1
	HARP	0.52	0.43	0.72	0.79	1.02	5.58	30.5
	Cassandra-Strong	0.80	0.68	0.85	0.92	1.22	6.52	29.8

Table 3 Latency/ms for different systems under the Zipfian distribution with approximate throughput

Operation	System	Average	50%	90%	95%	99%	99.9%	Throughput/Kops·s^{-1}
Read-only	Cassandra-Eventual	0.79	0.66	1.27	1.62	2.62	5.76	30.2
	HARP	0.84	0.73	1.31	1.68	2.69	6.02	29.8
	Cassandra-Strong	3.90	2.97	6.80	8.64	14.02	34.62	29.7
Write-only	Cassandra-Eventual	0.49	0.41	0.70	0.76	0.92	3.31	30.5
	HARP	0.49	0.42	0.71	0.76	0.91	3.41	30.1
	Cassandra-Strong	0.71	0.67	0.82	0.88	1.06	3.90	29.2

the explanation described before, a coordinator in Cassandra-Strong needs to send more messages to replicas than a coordinator in other two systems when handling a read operation, while a coordinator in three systems will send the same number of messages to replicas when handling a write operation.

Figure 8 displays the throughput results for different Zipfian factors in the workload whose read proportion is 80%. When the Zipfian factor is no larger than 0.83, the performance of HARP is nearly the same as that of Cassandra-Eventual, and the introduced overhead is no more than 1.9%. However, when the Zipfian factor is no less than 0.88, the overhead is slightly enlarged but still less than 6.9%. The larger the Zipfian factor is, the more concentrated requests are. Therefore, hot items are updated more frequently, which leads to a larger possibility that their replicas are inconsistent, and meanwhile, hot items are also read more frequently. That makes agents observe more stale states and incurs more second-round reads, which eventually leads to more overheads. Nonetheless, HARP greatly outperforms Cassandra-Strong and obtains 57.1%–76.5% higher throughput.

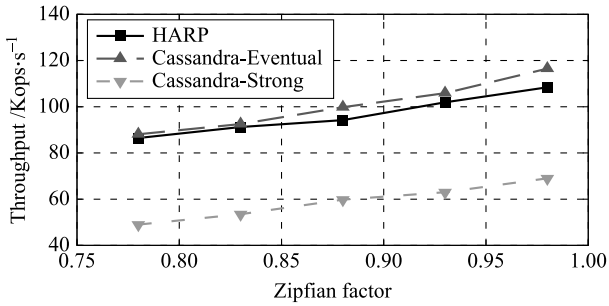

Fig. 8 Peak throughput versus different Zipfian factors

Though HARP cannot guarantee eliminating stale reads as Cassandra-Strong does, it provides much better performance than Cassandra-Strong. Moreover, our experimental results demonstrate that HARP can effectively eliminate stale reads no matter when failures are absent or present.

4.4 Handling practical anomalies

We demonstrate that anomalies can happen in a practical system without applying the strategies introduced in Section 3.3, and show that our strategies can handle practical anomalies in this subsection. Figure 9 shows the experimental results of the numbers of the anomalies for different throughputs. The read proportion of the workload is set to 80%, and each experimental instance runs continuously for 150 seconds in this experiment. Client requests obey a Zipfian distribution and the Zipfian factor is set to 0.88.

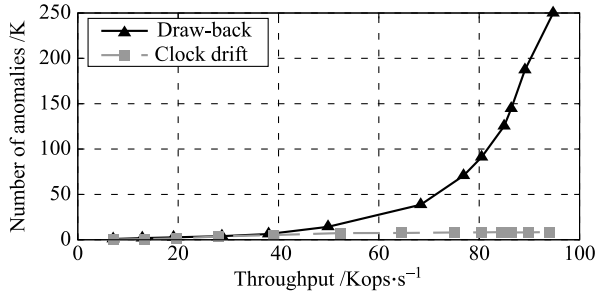

Fig. 9 Observed practical anomalies

• **Clock drift** In this experiment, we manually synchronize the clocks in the system before each experimantal instance. Although clocks in the system do not obviously differ, clock drift can happen in a real system [16] and is also observed in our experiment. We give the number of observed clock drifts

which occur in HARP that does not apply the strategy described in Section 3.3.2. In this experiment, if the timestamp of a response is larger than a client's local timestamp, then the client is considered to have observed a clock drift anomaly. The curve "clock drift" in Fig. 9 represents the numbers of observed clock drifts when HARP does not apply the strategy introduced in Section 3.3.2. As in the curve, the anomaly number slightly increases with the increasing of throughput.

• **Draw-back** In this experiment, if the WT store receives a update request and the request's timestamp is smaller than the existing item's timestamp, then the WT store is considered to have observed a draw-back anomaly. The curve "Draw-back" demonstrates the numbers of handled draw-back anomalies for different throughput after HARP has applied both strategies introduced in Sections 3.3.1 and 3.3.2. As illustrated in the figure, the number of draw-back anomalies remarkably increases with the increasing of throughput. That means messages are more likely to be reordered or delayed as the throughput increases. After observed a draw-back anomaly, the WT store will abandon the corresponding update request according to the strategy described in Section 3.3.1 to prevent the item's state from drawing back.

The performance results of HARP in Section 4.3 are measured after HARP has already applied both strategies introduced in Sections 3.3.1 and 3.3.2. Moreover, we also evaluate the performance overheads caused by those two strategies, and find that they negligibly affect the performance of HARP: the overhead introduced by either strategy is no more than 1.6%, and the introduced overhead is less than 2.2% after applying both strategies.

5 Discussion

In this section, we demonstrate some boundaries of HARP and discuss how an eventually consistent data store working with HARP can solve the practical problems.

5.1 Causal consistency

The strategy described in Section 3.3.2 borrows some idea from causal consistency [18]. Besides, HARP guarantees that the state updated by a write w_1 will not be modified by another write w_2, if w_1 is a causally later write compared with w_2, and that is a property what convergent causal consistency [19] (stronger than causal consistency) intends to guarantee. However, HARP cannot guarantee that a following read should not observe an item that is older than that observed by a earlier read. Therefore, HARP cannot guarantee causal consistency, while causal consistency does not try to make clients observe the newest data items, which is what HARP attempts to do.

5.2 Handling concurrent writes

In our discussion, for two operations, if an operation starts before the other operation ends, then the two operations are called concurrent operations [11]. HARP does not try to provide guarantees on the behavior of writes as Linearizability [11] does, and therefore, concurrent writes on the same item can happen in HARP. Either of the two concurrent writes is not causally later than the other, because they cannot observe the execution result of each other. In general, concurrent writes on the same item will not possess the same timestamp, and replicas will converge to the state modified by the write with the largest timestamp. However, if concurrent writes on the same item hold the same timestamp, the states of replicas in the DD store may be divergent before a newer update comes. In this situation, HARP can present the divergent states to the clients, which can decide the final winner of writes [1, 20]. There is also a method that can avoid the same timestamp of concurrent writes: each client appends its globally unique client ID to the lower bits of each timestamp to make the timestamp globally unique.

5.3 Failure recovery

HARP is able to detect and resolve possible stale reads when failures occur, but it is not designed to be a failure detector or failure recovery mechanism. There are many failure recovery technologies in modern distributed systems. For the widely deployed quorum systems, read repair, hinted handoff and Merkle trees based anti-entropy approaches are typical mechanisms [1]. However, during those mechanisms' proceeding or after recovery is failed, clients may still observe stale data, most of which can actually be avoided if HARP is applied. Moreover, HARP also can remarkably reduce the number of stale reads when failures are absent in an eventually consistent data store.

6 Related work

The consistency property has been long-studied in distributed systems. The CAP theorem highlights the inability to obtain all three properties: strong consistency, high availability and partition tolerance [2, 13]. Many distributed data stores choose weak consistent semantics to maintain available when partitions happen [21], and usually eventual consistency is

the choice [5]. However, there is no time bound to the recency of data returned for eventual consistency. Considering the problem, Bailis et al. [7] prove that partial quorums, which is a widely employed practical method to provide eventually consistent semantics, are often consistent within tens of milliseconds without considering failures. However, failures are unavoidable in real-world distributed systems [22, 23], and stronger semantics are often desired by applications [24].

Meanwhile, there are some works aiming at providing stronger consistent semantics while preserving high availability. COPS [20] defines a causal+ consistency model, and provides an "always-on" property and partition tolerant; Bolt-on [19] designs and implements a shim layer that upgrades the consistent semantics of underlying data stores to provide convergent causal consistency. Other systems [25, 26] also provide convergent causal consistency, which is stronger than eventual consistency. Convergent causal consistency is meaningful as it is achievable with high availability, which is proved by Ref. [14]. Additionally, HATs [27] proves that some "weak isolation" models are achievable without sacrificing availability. However, causal consistency [18] and HATs do not provide recency guarantees, nor try to increase the probability of recency reads.

There are some anti-entropy [1,9] mechanisms designed to deal with inconsistent replicas in eventually consistent systems. Read repair works after the read response has been returned to the caller, only trying to prevent following stale reads, and cannot prohibit on-going stale reads. Moreover, as the read repair process needs to communicate with all available replicas[16], to reduce its influence on the whole performance, Cassandra's default configuration reduces the chance of read repair[17], which makes the mechanism not trustworthy. Hinted handoff is designed to solve temporary node failures or network partitions[18]. While during the recovery process, stale reads also can be possible. Worse still, some continuous failures may make the hinted replicas unavailable, and therefore, a recovered stale replica cannot observe its missing updates until the hinted replica becomes available and finish the hinted handoff process. Merkle trees based anti-entropy [1] is able to handle inconsistency caused by permanent failures. However, as hinted handoff, stale reads can be possible before or during the recovery, which often takes a long time. In summary, those technologies cannot deal with the on-going stale reads during recovery and sometimes, to make matters worse, recovery may fail.

Our earlier work entitled "HARP: towards enhancing data recency for eventually consistent data stores" in ICPADS 2014 [28] presents an approach to enhance data recency for eventually consistent data stores, but it does not consider the practical anomalies in system design, including delayed, reordered and dropped messages and clock drift. In this work, we also show how to extend HARP to multiple datacenters, and evaluate HARP with more extensive experiments.

7 Conclusion

In this paper, we propose a framework HARP, which can increase the probability of data recency for eventually consistent data stores while preserving high performance and availability. Through analyzing all possible cases of HARP caused by failures, exceptions and comparison results of observed timestamps, we find that reread operations can eliminate possible stale reads only when required, which avoids unnecessary reread operations, thus reducing the performance overhead of HARP. We also discuss how to keep the high availability of HARP, and demonstrate how to deal with some practical anomalies including delayed, reordered and dropped messages and clock drift in HARP, as well as how to extend HARP to multiple datacenters. Finally, based on the eventually consistent data store Cassandra and the in-memory data store Redis, we implement HARP and evaluate it with the benchmark YCSB [10]. The experimental results show that, HARP brings in no more than 6.9% overhead compared with eventual consistency, while repairing all stale reads caused by failures and most stale reads when failures are absent.

Acknowledgements This work was supported partly by the National High-tech Research and Development Program (863 Program) of China (2015AA01A202), and partly by the National Natural Science Foundation of China (Grant Nos. 61370057 and 61421003).

References

1. DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W. Dynamo: Amazon's highly available key-value store. In: Proceedings of ACM Symposium on Operating Systems Principles. 2007, 205–220
2. Brewer E A. Towards robust distributed systems. In: Proceedings of the 19th ACM Symposium on Principles of Distributed Computing. 2000
3. Abadi D. Consistency tradeoffs in modern distributed database system design: cap is only part of the story. IEEE Computer, 2012, 45(2): 37–

[16] http://docs.datastax.com/en/cassandra/1.2/cassandra/architecture/architectureClientRequestsRead_c.html
[17] http://www.datastax.com/documentation/cql/3.0/cql/cql_reference/tabProp.html
[18] http://docs.datastax.com/en/cassandra/1.2/cassandra/dml/dml_about_hh_c.html

42

4. Lakshman A, Malik P. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35–40
5. Vogels W. Eventually consistent. Communications of the ACM, 2009, 52(1): 14–19
6. Saito Y, Shapiro M. Optimistic replication. ACM Computing Surveys, 2005, 37(1): 42–81
7. Bailis P, Venkataraman S, Franklin M J, Hellerstein J M, Stoica I. Probabilistically bounded staleness for practical partial quorums. In: Proceedings of International Conference on Very Large Data Bases. 2012, 776–787
8. Bailis P, Venkataraman S, Franklin M J, Hellerstein J M, Stoica I. Quantifying eventual consistency with PBS. VLDB Journal, 2014, 23(2): 279–302
9. Dcmers A, Greene D, Hauser C, Irish W, Larson J, Shenkcr S, Sturgis H, Swinehart D, Terry D. Epidemic algorithms for replicated database maintenance. In: Proceedings of ACM Symposium on Principles of Distributed Computing. 1987, 1–12
10. Cooper B F, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. In: Proceedings of ACM Symposium on Cloud Computing. 2010, 143–154
11. Herlihy M P, Wing J M. Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages & Systems, 1990, 12(3): 463–492
12. Lamport L. On interprocess communication. Distributed Computing, 1986, 1(2): 86–101
13. Gilbert S, Lynch N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. ACM SIGACT News, 2002, 33(2): 51–59
14. Mahajan P, Alvisi L, Dahlin M. Consistency, availability, convergence. Technical Report. 2011
15. Alpern B, Schneider F B. Recognizing safety and liveness. Distributed Computing, 1987, 2(3): 117–126
16. Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 1978, 21(14): 558–565
17. Wang X, Sun H L, Deng T, Huai J D. A quantitative analysis of quorum system availability in data centers. In: Proceedings of the 22nd IEEE International Symposium on Quality of Service. 2014, 99–104
18. Ahamad M, Neiger G, Burns J E, Kohli P, Hutto P W. Causal memory: definitions, implementation, and programming. Distributed Computing, 1995, 9(1): 37–49
19. Bailis P, Ghodsi A, Hellerstein J M, Stoica I. Bolt-on causal consistency. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 2013, 761–772
20. Lloyd W, Freedman M J, Kaminsky M, Andersen D G. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In: Proceedings of ACM Symposium on Operating Systems Principles. 2011, 401–416
21. Davidson S B, Garcia-Molina H, Skeen D. Consistency in a partitioned network. ACM Computing Surveys, 1985, 17(3): 341–370
22. Dean J S. Designs, lessons and advice from building large distributed systems. In: Proceedings of the Workshop on Large-Scale Distributed Systems and Middleware. 2009
23. Gill P, Jain N, Nagappan N. Understanding network failures in data centers: measurement, analysis, and implications. In: Proceedings of ACM International Conference on the Applications, Technologies, Architectures, and Protocols for Computer Communication. 2011, 350–361
24. Bailis P, Ghodsi A. Eventual consistency today: limitations, extensions, and beyond. Queue, 2013, 11(3): 55–63
25. Lloyd W, Freedman M J, Kaminsky M, Andersen D G. Stronger semantics for low-latency geo-replicated storage. In: Proceedings of the USENIX Conference on Networked Systems Design and Implementation. 2013, 313–328
26. Du J, Iorgulescu C, Roy A, Zwaenepoel W. GentleRain: cheap and scalable causal consistency with physical clocks. ACM Symposium on Cloud Computing. 2014, 1–13
27. Bailis P, Davidson A, Fekete A, Ghodsi A, Hellerstein J M, Stoica I. Highly available transactions: virtues and limitations. In: Proceedings of International Conference on Very Large Data Bases. 2013, 181–192
28. Tang Y, Sun H L, Wang X, Liu X D. Harp: towards enhancing data recency for eventually consistent data stores. In: Proceedings of IEEE International Conference on Parallel and Distributed Systems. 2014, 685–692

Yu Tang received the BS degree from Beihang University, China in 2011. Currently, he is working towards the PhD degree in the School of Computer Science and Engineering, Beihang University. His research interests include the areas of distributed systems and availability.

Hailong Sun received the BS degree in computer science from Beijing Jiaotong University, China in 2001. He received the PhD degree in computer software and theory from Beihang University, China in 2008. He is an associate professor in the School of Computer Science and Engineering, Beihang University. His research interests include services computing, cloud computing and distributed systems. He is a member of the IEEE and the ACM.

Xu Wang received the BS degree from Beihang University, China in 2008. He received the PhD degree in computer software and theory from Beihang University in 2015. His research interests include the areas of distributed systems, service computing, replication, and availability.

Xudong Liu is a professor and dean of the School of Computer Science and Engineering, Beihang University, China. Has have leaded several China 863 key projects and e-government projects. He has published more over 30 papers, more than 10 patents. His research interests include software middleware technology, software development methods and tools, large-scale information technology projects and application of research and teaching.