Journal of Information Processing Vol.25 775–782 (Aug. 2017)

[DOI: 10.2197/ipsjjip.25.775]

Regular Paper

Causal Consistency for Data Stores and Applications as They are

Kazuyuki Shudo[1,a)] Takashi Yaguchi[1,†1]

Received: November 3, 2016, Accepted: May 16, 2017

Abstract: There have been proposed protocols to achieve causal consistency with a distributed data store that does not make safety guarantees. Such protocols work with an unmodified data store if it is implemented as middleware or a shim layer while it can be implemented inside a data store. But the middleware approach has required modifications to applications. Applications have to explicitly specify data dependency to be managed. Our Letting-It-Be protocol to the contrary, handles all implicit dependency naturally resulting from data accesses even though it is implemented as middleware. Our protocol does not require any modifications to either data stores or applications. It works with them as they are. It trades performance for the merit to some extent. Throughput declines from a data store alone were 21% in the best case and 78% in the worst case without multi-level management of dependency graph, which is a performance optimization technique.

Keywords: distributed database, consistency model, causal consistency, middleware, concurrency problem

1. Introduction

Geo-replication is one of the primary features of distributed data stores whereby a client of a data store can access data with small latency by choosing nearby replicas. But geo-replication usually trades stronger consistency models for its merits [6], [10] and then such distributed data stores maintain a consistency model such as eventual consistency [5], [8], [15], that does not make safety guarantees.

There have been attempts to add support for stronger consistency models to such distributed data stores while preserving their merits. Causal consistency has been the target of those attempts. There are two approaches to this: a data store approach and a middleware approach. In the former approach a protocol to achieve causal consistency is implemented in a data store itself. In the latter approach a middleware over a data store implements a protocol.

The middleware approach has an advantage over the data store approach in that it works with an unmodified data store. But the middleware approach involves management of large dependency graphs. The existing protocol taking the middleware approach [4] reduces the size of the graphs by making data store clients explicitly specify the dependency to be managed. This means the middleware approach has required modifications to applications. On the contrary, our Letting-It-Be protocol handles all the implicit dependency naturally resulting from data accesses though it is implemented as middleware. Our protocol does not require any modifications to either data stores or applications (**Fig. 1**).

This paper is an extended version of our previous work [17].

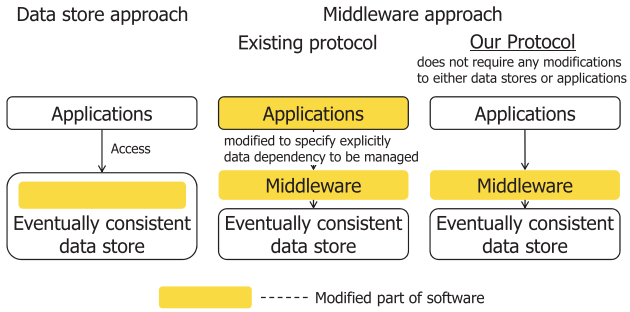

Fig. 1 Approaches to achieving causal consistency.

The differences include a performance optimization technique (Section 3.5) and its evaluation (Section 4.3)

This paper is organized as follows. Section 2 provides prior knowledge by introducing related consistency models and existing protocols. Section 3 describes our protocol. Section 4 shows experimental results of performance measurement and discusses them. In Section 5, we summarize our contributions.

2. Background

This section provides dependency representation for the following section, and existing approaches and protocols.

2.1 Causal Consistency

Causal consistency is a consistency model in which all writes and reads of data items obey causality relationships between them. If a write or read operation influences a subsequent operation, a client that observed the second can always observe the first [1], [12]. Consider a social networking site as an example of a real-world application. Note that the following example has the same structure as an example shown in Bolt-on Causal Consistency paper [4]. Alice posts update A: "My research paper could

[1] Tokyo Institute of Technology, Meguro, Tokyo 152–8552, Japan
[†1] Presently with Industrial Growth Platform, Inc. (IGPI)
[a)] shudo@is.titech.ac.jp

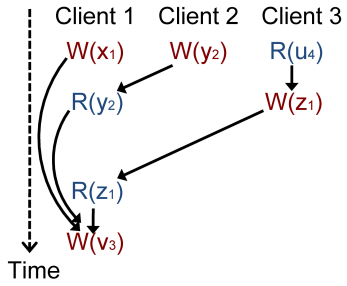

Fig. 2 Example of write and read sequences, and causality relationships between operations.

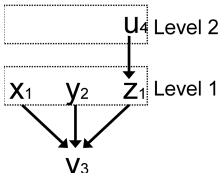

Fig. 3 Dependency graph in case of Fig. 2.

not be accepted." After Alice posts A, she is notified that the notification of rejection was a false alarm. She edits A, resulting $A*$: "My paper has been accepted!." Bob observes $A*$ and posts status B in response: "Congratulations!" If causality is not maintained, another user, Carol, could see A and B but not $A*$. She might then think Bob is pleased to hear of the rejection. Accordingly, a causally consistent data store guarantees that any user that can observe $A*$ if the user observes B.

Causality relationships between write and read operations are defined as follows. Here $W(x_i)$ and $R(x_i)$ signify a write operation and a read operation of x_i. x_i indicates the version i of the key x. A version number increases one by one when a write updates a value of the key. $op(x_i)$ and just op_j signify a write or read operation. Suppose that an operation op_1 influences another operation op_2. The relationship is called op_1 `happens-before` op_2 and denoted by $op_1 \rightarrow op_2$. The following four rules yield a causality relationship between two operations.

Rule 1 A client reads x_i and then writes y_j.
$R(x_i) \rightarrow W(y_j)$ on the same client holds.

Rule 2 A client writes x_i and then writes y_j.
$W(x_i) \rightarrow W(y_j)$ on the same client holds.

Rule 3 A client writes x_i and then the client or another client reads x_i.
$W(x_i) \rightarrow R(x_i)$ on the same client or different clients holds.

Rule 4 If $op(x_i) \rightarrow op(z_k)$ and $op(z_k) \rightarrow op(y_j)$ hold, then $op(x_i) \rightarrow op(y_j)$ holds.

Figure 2 shows an example of operation sequences performed by client 1, 2 and 3. Arrows represent causality relationships resulting from the rule 1, 2 and 3 and they form a graph

It is a graph of operations and can be transformed to a graph of data items, strictly a graph of versions of keys. The transformation takes the rule 4 about transitivity into consideration. Figure 2 results in **Fig. 3**, which is the dependency graph of v_3.

In a dependency graph, we define levels. A version of a key that is the source of the dependencies is the level 0 vertex. Versions of keys that level i vertex directly depend on are level $i + 1$ vertices. In Fig. 3, v_3 is the level 0 vertex. x_1, y_2 and z_1 are level 1 vertices. u_4 is a level 2 vertex.

If the source of the dependencies (level 0 vertex, v_3 in Fig. 3) is a target of a write operation, level 1 vertices, that the source directly depends on, are one of the following. Our protocol utilizes this fact as described in Section 3.2.

(1) The last write which is a version of a key written just before the source by the same client (x_1 in Fig. 3)
(2) Reads following the last write — versions of keys read after the last write by the same client (y_2 and z_1 in Fig. 3)

2.1.1 Explicit Specification of Causality Relationships

Causality relationships occur spontaneously when data items are written and read. But an application may not require all the relationships to be maintained. It depends on the application. In that case the amount of the relationships can be reduced by making the application explicitly specify relationships to be maintained. The existing protocol taking the middleware approach [4] requires such explicit specification to reduce the amount of causality relationships it handles.

The explicit specification works as long as the application can identify and specify causality relationships that the application requires. On the other hand, such a protocol cannot be adopted in cases where an application cannot identify the requisite causality relationships or cannot be modified.

2.2 Eventual Consistency

There have been distributed data stores emerging whose goals are scalability on numbers of servers while keeping their availability. In compensation for scalability and availability, they have looser requirements for data consistency and their major target has been eventual consistency [5], [8], [15]. Eventual consistency is a consistency model in which all replicated data items converge to the same value eventually. It is a liveness guarantee and does not make the safety guarantees, that causal consistency makes.

A data store can be eventually consistent by just guaranteeing that a write will propagate to all the replicas. All the replicas choose their last value along the same predetermined policy and they converge. An example of such a policy is "last-writer-wins" in which a value with the largest time stamp is chosen as the last value. It is not required to consider the order of write operations when propagating and applying them. Such lack of concern for the order violates causal consistency.

2.3 Existing Protocols and Related Work

This section describes existing protocols for maintaining causal consistency with a distributed data store that does not make safety guarantees, thereby presenting our contributions.

Existing protocols took one of the two approaches: a *data store approach* and a *middleware approach*. A protocol taking the data store approach is implemented in a data store itself and then requires modifications to a data store. A protocol taking the middleware approach is implemented as a middleware that works over a bare data store and does not require modifications to the data store itself.

Examples of the data store approach are COPS [13], Eiger [14], ChainReaction [2] and Orbe [9]. Bolt-on Causal Consistency [4] took the middleware approach.

An advantage of the data store approach is that it allows

dependency resolution when writing, that we call *resolution-on-write*. A protocol taking the data store approach works as follows. When receiving a replica update of x_i, before writing x_i, the protocol confirms that the data items that x_i directly depends on have already been written. For each data item, for example x_i, the protocol maintains pointers to other data items that x_i directly depends on. The pointers enable the dependency confirmation.

The middleware approach does *resolution-on-read* because it cannot implement the resolution-on-write. The resolution-on-write requires changes to a replication mechanism inside a data store. More specifically, it must capture all replica updates that happen inside a data store. The middleware approach cannot adopt the resolution-on-write because it does not make changes to a data store itself.

The resolution-on-read involves the problem of *overwritten dependency graph*, that is called overwritten histories by Bailis et al. [4]. If a protocol lacks an adequate treatment for the problem, part of a dependency graph can be overwritten and lost, even though the entire graph is still required. In Fig. 3, if z_1 has been just replaced by z_2, a client that tries to read v_3 cannot find z_1. Resolution cannot finish.

The resolution-on-write does not involve this problem. A protocol taking the resolution-on-write can confirm a version of a key has been resolved by a means such as vector clocks between replicas kept by different local sets. With resolution-on-write, a version has been resolved if it has been written. This fact enables the confirmation by a means such as vector clocks such that the version 1 of z has been already resolved because the version is 2. If resolution finishes in failure, a protocol can defer application of the replica update. For example, a protocol applies a replica update of v_3 after observing z_1.

To address the problem of overwritten dependency graph, the existing protocol taking the middleware approach [4] maintains an entire dependency graph for each data item. In Fig. 3, the existing protocol keeps the entire graph consists of v_3, x_1, y_2, z_1 and u_4 for a data item v_3. This treatment enables the protocol to check the entire graph for v_3 even if z has been updated to z_2.

But this treatment involves large dependency information that a middleware must maintain. Accordingly, the existing protocol reduces the amount of dependency information to be kept by explicit specification described in Section 2.1.1. The explicit specification requires an application to identify and specify causality relationships that it requires. The existing protocol does not work if an application cannot identify the requisite relationships or cannot be modified.

3. Causal Consistency for Distributed Data Stores and Applications as They Are

In contrast to the existing protocols, our Letting-It-Be protocol takes the middleware approach and handles all the implicit causality relationships. Therefore, it does not require any modifications to either data stores or applications (Fig. 1). It works with them as they are.

Section 3.1 shows the system model assumed in the following description of our protocol. In succeeding Sections 3.2, 3.3 and 3.4, we propose the protocol. Section 3.5 introduces an optional performance optimization technique.

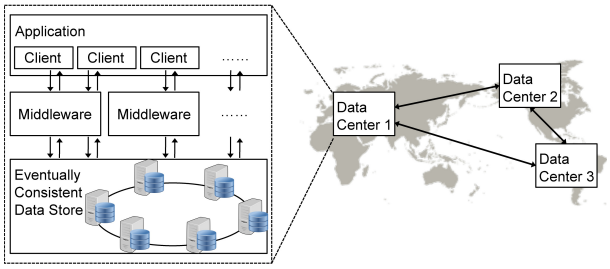

Fig. 4 Supposed system model.

3.1 System Model

Figure 4 depicts a system model assumed in the following description of our protocol. Application instances, middleware instances and a cluster running a data store are located at the same site and these form a local set of servers. There are a number of such local sets providing the same service to users. Such local sets are geographically distributed and usually each set serves nearby users. In the real world, a data center hosts a single or several local sets of servers.

An application instance accesses only its paired local data store cluster. Each cluster holds all the data items, that are available locally in a local set. To achieve it, a data item has to be replicated to all the local sets.

A middleware instance mediates between applications and a data store cluster and maintains causal consistency. A local set can run an arbitrary number of middleware instances. Note that middleware instances do not communicate each other unless a protocol requires this. Our protocol performs mutual exclusion between middleware instances in a local set (Section 3.4), but our current implementation involves no communication between the instances. Mutual exclusion is carried out with a compare-and-swap (CAS) feature of underlying data stores.

A data store is eventually consistent and its policy to choose the last value is the "last-write-wins," in which a value with the largest time stamp is chosen as the last value (Section 2.2).

3.2 The Base Letting-It-Be Protocol

This section describes a simplified version of our Letting-It-Be protocol that handles neither concurrent multiple clients nor the problem of overwritten dependency graph depicted in Section 2.3. The following Section 3.3 shows treatment for the problem and Section 3.4 shows techniques to handle concurrent overwrites by multiple clients.

When a middleware instance receives a write request, that is a pair of a key and a value, from a client, it embeds dependency information of the received key in the received value. The embedded dependency information consists of an updated version number of the received key and a set of versions of keys that the received key directly depends on. They are level 0 and 1 vertices of a dependency graph. In Fig. 3, a middleware instance receiving a write request to v embeds 3 as the version of v, x_1, y_2 and z_1 in the value of v. And then the middleware instance writes the processed value with the requested key to a data store.

This embedding process requires a middleware instance to

maintain level 0 and 1 vertices for all the keys it has. As described in Section 2.1, level 1 vertices are the last write (x_1) just before the write to the source of the dependency graph and reads (y_2 and z_1) that follow the last write. A middleware keeps a history that consists of the last write (x_1) and the following reads (y_2 and z_1). These are just level 1 vertices and the middleware embeds them.

As described in Section 2.3, in the existing protocol [4], a middleware maintains an entire graph for each key. It requires much storage. In contrast to it, in our protocol, a middleware instance maintains only level 0 and 1 vertices for each key. By limiting the levels it keeps, a middleware instance can handle all the implicit dependency naturally resulting from data accesses. The amount of dependency information that a middleware instance keeps is the product of the number of keys it has and the average length of a histories. The average length of histories depends on the rate of writes and reads in a workload. A read-heavy workload yields longer histories because a history starts at the last write. Anyway the length is definitely limited as long as there is a write. If there is no write, then there are no causality relationships and no dependency graph is formed.

When a middleware instance receives a read request, that is a key, from a client, it reads a value corresponding to the key from a data store. The read value is accompanied by dependency information which is a history. The middleware instance starts resolving dependency based on the dependency information. It reads values of level 1 keys from a data store to obtain level 2 vertices, reads values of level i keys to obtain level $i + 1$ vertices, and then traverses the entire graph. An entire graph is available locally in a cluster of servers running a data store because a cluster in a local set, usually located in a data center, has at least one replica of all data items (Section 3.1).

Resolution-on-read finishes in a success if values of all the vertices of the entire graph are available, and the middleware instance replies the value after stripping the dependency information from the value. If a value of a vertex is not available, resolution finishes in failure. In that case a middleware implementation has options. one option is waiting for the entire graph to become available. Another option is returning a previous version of the requested key that has already been resolved. Our current implementation returns an error to a client.

A middleware instance marks a version of a key when the dependency resolution for it succeeds. Making these marks prevent repeated resolution.

3.3 Problem of Overwritten Dependency Graph

The base protocol does not treat the problem of overwritten dependency graph depicted in Section 2.3. The base protocol assumes that there is a single client accessing a data store sequentially but succeeding writes by the same client can overwrite part of a graph even without multiple clients.

Here we extend the base protocol to handle the problem. In the base protocol, a key is accompanied by dependency information for a single version of the key. The extension lets a key be accompanied by dependency information for multiple versions of the key. In Fig. 3, a write to z overwrites z_1 by z_2 in the base protocol. Now the value of z holds dependency information for both versions z_1 and z_2 with the extension. A middleware instance embeds them in the value such that z_1 depends on u_4 and z_2 depends on its dependency destinations. This treatment prevents z_1 from being overwritten.

Even with the extension, the amount of dependency information is smaller than the existing protocol [4], that keeps an entire dependency graph for each key. The existing protocol keeps the same dependency information for multiple keys duplicatedly. In Fig. 3, the graph for v_3 includes dependency information such as that z_1 depends on u_4. Graphs for other keys depending on z_1 also include the same information duplicatedly. In our protocol, dependency information such as that z_1 depends on u_4 appears once in a data store in a local set.

Old dependency information that no key refers to should be wiped out after it becomes unnecessary. A mechanism such as garbage collection in programming systems serves this function. There is a trade-off between garbage collecting techniques such as mark and sweep and reference counting. Comparison between them is still an open problem, although this should have little effect on access performance because they run in the background.

3.4 Concurrent Overwrites by Multiple Clients

Assuming the case where multiple clients concurrently access a data item, the dependency information might be lost even with multiple versions.

In case multiple clients try to update dependency information of a key, it is possible for an update to be overwritten by other updates. Suppose that client A and B concurrently try to update dependency information of a key z. After both clients read dependency information of z_1, they try to write updated dependency information. Client A writes z_2 with dependency information and then client B writes z_2' with different dependency information. So the dependency information of z_2 is lost.

There are a variety of options for this type of concurrency problem. Here, We adopt a write-time solution in a local set (Section 3.1) and a read-time solution between local sets. Mutual exclusion takes place in a local set and multiple versions are maintained for each local set.

A local set can utilize any technique for mutual exclusion such as locking and these are effective. Our current implementation utilizes an optimistic technique, that is compare-and-swap (CAS). The implementation utilizes the CAS feature of an underlying data store.

Mutual exclusion tends to be costly if it is carried out between local sets. It involves communication between local sets in either cases of optimistic techniques or pessimistic techniques such as locking. Communication between local sets can get across boundaries of data centers, that are supposed to host local sets, and it involves a large latency. In our protocol, write and read operations do not involve any communication between local sets. In a data store, a key has distinct versions for each local set. For example, a key z has distinct versions, z_LS1 and z_LS2, for local sets LS1 and LS2. A middleware instance writes only onto the version for the local set it belongs to. Overwriting a key for other local sets does not take place.

All versions for all the local sets are replicated to all the lo-

cal sets by replication feature of an underlying data store (Section 3.1). The local sets LS1 and LS2 eventually have updated dependency information of *z_LS*1 and *z_LS*2. When reading, a protocol has to determine which is the last one written between distinct versions for local sets, for example, between *z_LS*1 and *z_LS*2. We choose to maintain causal consistency here and use vector clocks [12] for the purpose. Dynamo [8] and Riak [5] adopt the same policy and technique. After system trouble involving network partitions, a middleware occasionally finds concurrent conflicting values between the distinct versions. Causal consistency allows it to return any value of them. Our current implementation chooses one of the concurrent values based on identifiers of local sets.

Distinct versions for each local set respectively consume space in a data store. They eliminate communication between local sets but take up much space. There is a trade-off and the best boundary between mutual exclusion and distinct versions for each local set depends on applications and especially the network environment.

The overall picture of our protocol is as follows. By vector clocks, a middleware instance captures the causality relationships between distinct versions of a key for each local set such as *z_LS*1 and *z_LS*2. By using dependency graphs, it captures causality relationships between different keys such as *x*, *y* and *z*.

3.5 Multi-level Management of Dependency Graph — Performance Optimization

While the above sections described the complete protocol, this section introduces a performance optimization technique. It is optional. Section 4.3 shows its effect on performance.

Our protocol traverses an entire dependency graph to accomplish resolution-on-read as described in Section 3.2. A middleware instance reads values of level i keys from a data store to obtain level $i + 1$ vertices. In that case, a middleware can issue in parallel all the read requests for all the level i vertices. Traversal of a dependency graph with up to level n repeats such possibly parallel reads of level i vertices n times. The traversal process possesses parallelism for the number of vertices on the same level and the length of the critical path of the process is the highest level number of a graph, n. Thus a higher graph involves larger processing time due to a longer critical path.

It is possible to shorten the critical path and increase parallelism by embedding multiple levels together in a value. In the base protocol, the version number of the level 0 vertex and level 1 vertices are embedded. If multiple levels up to level k are embedded, this enables possibly parallel reads of all vertices from level 1 to level k and shorten the critical path to $\lceil n/k \rceil$.

This optimization requires a middleware instance to construct and embed a dependency subgraph up to level k in a value. A middleware instance can do these tasks by keeping dependency subgraphs up to level $k - 1$ for all the level 1 vertices. Level 1 vertices are the last write and reads that follow it, and they construct a history (Section 3.2). For this optimization, a history is extended to keep dependency subgraphs up to level $k - 1$ for all the elements in the history in addition to the elements themselves. A middleware instance can obtain the subgraphs at the time of the last write and the following reads. There is no need for additional communication to obtain them.

This optimization can improve performance of dependency resolution but involves dependency management costs due to larger dependency information. Section 4.3 shows results of performance measurement.

4. Performance Evaluation

The contribution our work provides is a protocol that maintains causal consistency with no modification to either applications or a data store. Nevertheless the amount of performance overhead should be acceptable in exchange for the benefits obtained. This depends on the application but anyway this section shows experimental results of performance measurement.

Section 4.2 shows performance overheads of the protocol. Section 4.3 shows performance improvement by multi-level management of dependency graph.

4.1 Implementation and Benchmark Conditions

Our implementation of the proposed protocol described in Section 3 consists of 3,000 lines of Java code. It uses Google's Protocol Buffers 2.5.0 for data serialization and Google's Snappy 1.1.2 for data compression. The target of performance measurement in this paper is this implementation.

The implementation is based on Apache Cassandra 2.1.0 [3], [11], which is a production-level and widely deployed distributed data store. Cassandra is compatible with the system model described in Section 3.1 as follows. It provides a function to place replicas of a data item in every data center (`NetworkTopologyStrategy`). All the replicas converge to the same value because Cassandra adopts eventual consistency.

The current implementation performs mutual exclusion using compare-and-swap (CAS) (Section 3.4). Cassandra provides the feature. We implemented the protocol as a library for a client in the same way as the existing protocol taking the middleware approach [4] though it is possible to implement as software serving clients via a network.

We use Yahoo! Cloud Serving Benchmark (YCSB) [7] to measure performance of the implementation. It is a framework implementation to benchmark distributed data stores and compares them fairly. It has been widely used in a variety of research on cloud storage [16]. YCSB issues write and read queries to a target data store continuously and measures access latency or namely the round trip time, of each write and read operation. A user of YCSB can specify the ratio of write and read operations, distribution of accesses to data items and the target of throughput that is the number of queries in a unit time.

We impose two diverse workloads, write-heavy and read-heavy, on the implementation. **Table 1** shows parameters of the two workloads. Accessed data are chosen along a Zipfian distribution. This is a probability distribution, which means that access frequency of each data item is determined by its popularity and

Table 1 YCSB workloads used in Section 4.

Workload	Write	Read	Access distribution
Write-heavy	50%	50%	Zipfian distribution
Read-heavy	5%	95%	Zipfian distribution

Table 2 Server configuration.

OS	Ubuntu 12.04.3 with Linux 3.2.0
CPU	2.40 GHz Xeon E5620 × 2
Memory	32 GiB RAM
Java Virtual Machine	Java SE 7 Update 4

not by freshness.

Nine servers emulate a data center and and two sets of the servers emulate two data centers. All the 18 servers run Cassandra and compose a cluster of Cassandra. Another server runs YCSB to access other 18 servers. **Table 2** shows the configuration of the servers. All the servers are on the same LAN but communication latency between the data centers is emulated by imposing 50 milliseconds of latency with a tool named `tc`. We configure the Cassandra cluster to have one replica in each emulated data center by setting the replication strategy as `NetworkTopologyStrategy` and consistency level as `ONE`. By that, each of the two emulated data centers has its own replica. These settings correspond to a situation in which each of the two data centers hosts a local set.

4.2 Read and Write Performance

The number of data items is 10,000,000 and the size of a data item is 1 KiB. The total amount of the data items is about 10 GiB or more with their metadata such as schema information. After loading all the data items into the Cassandra cluster, we warm up the cluster with the same workload as the following measurement. And then we measure performance.

We examine overheads imposed by the proposed protocol by performance comparisons with Cassandra alone. It is interesting to examine performance of the existing protocol taking the middleware approach [4]. But the existing protocol is designed to be able to handle only explicitly-specified dependency, not all the implicit dependency, which is our target. Explicit specification of dependency allows the existing protocol to run but it is not our target and our protocol does not support it.

Figures 5 and **6** show access latency with the write-heavy workload. At 3 and 7 Kbps of throughput, with our implementation, the write latencies are 5.2 and 6.6 milliseconds. Read latencies are 3.9 and 7.2 milliseconds. Without our implementation, write latencies are 0.9 and 0.9 milliseconds. Read latencies are 1.2 and 1.4 milliseconds. Thus overheads in write latencies are 4.3 and 5.7 milliseconds, and overheads in read latencies are 2.7 and 5.8 milliseconds. The maximum throughput with our implementation is 78% lower than Cassandra alone.

Figures 7 and **8** show access latencies with the read-heavy workload. At 3 and 7 Kbps of throughput, with our implementation, write latencies are 4.2 and 4.2 milliseconds. Read latencies are 1.4 and 1.4 milliseconds. Without our implementation, write latencies are 1.0 and 1.0 milliseconds. Read latencies are 1.2 and 1.2 milliseconds. Thus overheads in write latencies are 3.2 milliseconds, and overheads in read latencies are 0.2 milliseconds. The maximum throughput with our implementation is 21% lower than Cassandra alone.

The read-heavy workload showed smaller overheads than the write-heavy workload. **Figure 9** shows the maximum throughputs with different ratios of read and write operations. A larger ratio of read operations exhibits better throughput.

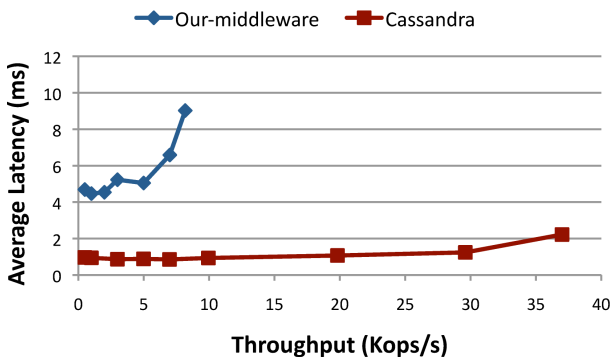

Fig. 5 Write latencies with write-heavy workload.

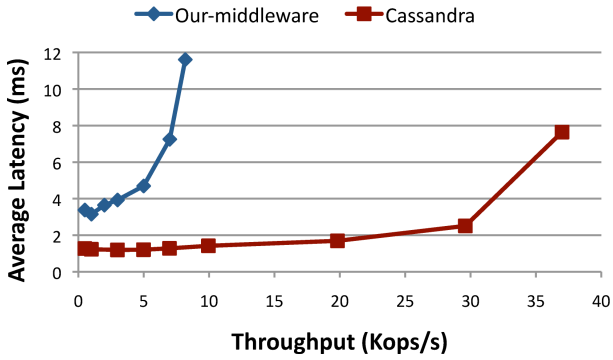

Fig. 6 Read latencies with write-heavy workload.

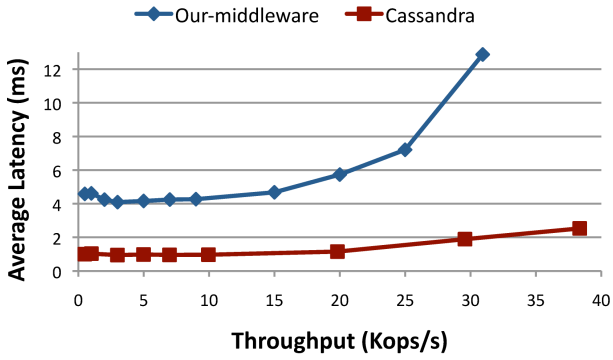

Fig. 7 Write latencies with read-heavy workload.

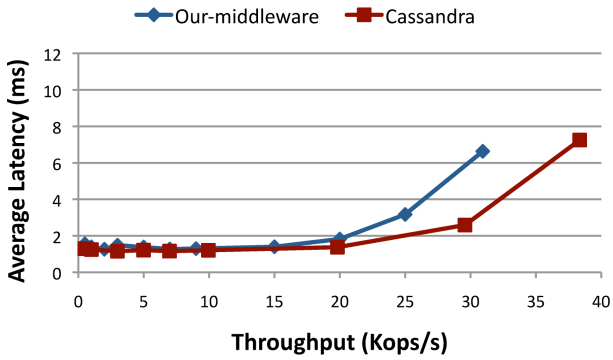

Fig. 8 Read latencies with read-heavy workload.

Dependency resolution should greatly contribute to the overhead because it involves multiple accesses to a data store. An access to a data store involves communication over a network and then involves a large latency. The read-heavy workload re-

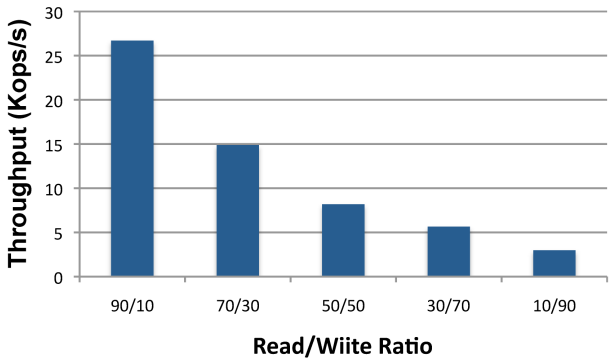

Fig. 9 Maximum throughput with each read/write ratio.

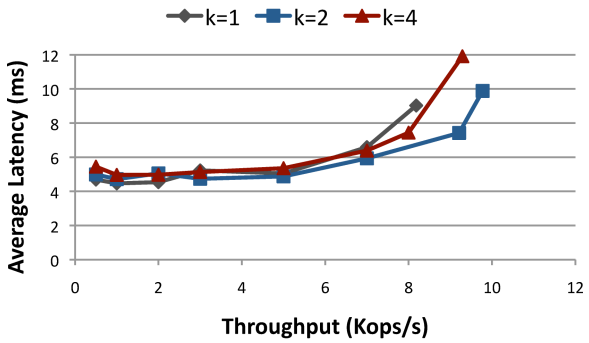

Fig. 10 Write latencies with write-heavy workload with multi-level management.

quires larger number of dependency resolutions than the write-heavy workload because our protocol performs resolution when reading. But if a version of a key has been marked as resolved, then further traversal of a dependency graph is not required as described in Section 3.2. More frequent reads yield more marks and reduce the number of accesses to a data store. In summary, more reads increase the number of dependency resolution but decrease the number of accesses to a data store in dependency resolution. It seems that the latter effect is greater in the Zipfian distribution that YCSB produces.

An application enjoys the merits of the protocol by accepting the performance overhead. Performance overheads heavily depend on a workload. The application developer must judge whether it is acceptable or not. The results here should help make such a judgment because the key property of application workloads is the ratio of write and read operations and it dominates performance.

4.3 Multi-level Management of Dependency Graph

We examine the effect of multi-level management of dependency graph described in Section 3.5. This section shows access latencies with different heights of a subgraph embedded to a value. The examined heights k are 1, 2 and 4.

Figures 10 and **11** show access latencies with the write-heavy workload. Latencies in case $k = 2$ are better than those in case $k = 1$. But a further larger k as 4 did not improve latencies, and even showed slightly worse latencies than $k = 2$ when writing. The effect of the optimization of multi-level management looks to be saturated by $k = 2$.

Figures 12 and **13** show access latencies with the read-heavy workload. Larger k showed worse latencies. As demonstrated

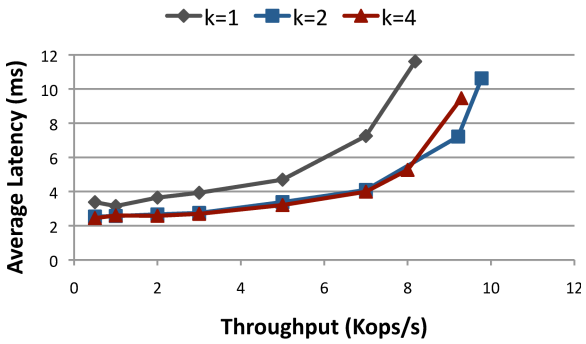

Fig. 11 Read latencies with write-heavy workload with multi-level management.

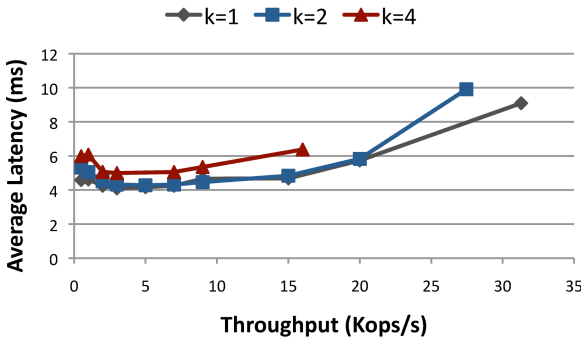

Fig. 12 Write latencies with read-heavy workload with multi-level management.

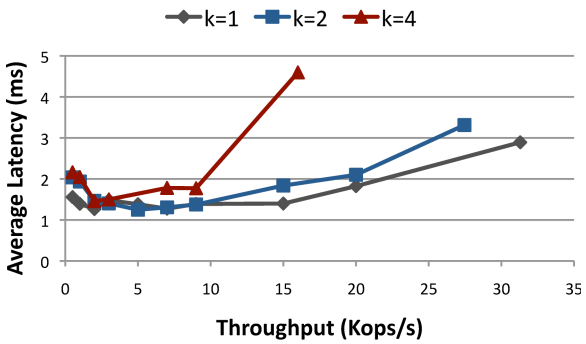

Fig. 13 Read latencies with read-heavy workload with multi-level management.

in Section 4.2, with the read-heavy workload, access reduction effect by marking a resolved version of a key took place very well even in case $k = 1$ without the optimization. The optimization does not take effect for already resolved and marked keys. Moreover, finding the best parameter k for a variety of workloads makes sense for real-world deployment of the optimization.

5. Conclusion

We present a protocol Letting-It-Be for maintaining causal consistency over an existing production-level eventually consistent data store. Our protocol is unique in that it handles all the implicit dependency naturally resulting from data accesses though it is implemented as middleware. Namely, it does not require any modifications to either a data store or applications. It works with them as they are.

Performance overheads of the proposed protocol heavily depend on the workload. Throughput declines from Cassandra alone were 21% in the best case and 78% in the worst case. A

performance optimization technique or namely multi-level management of dependency graphs, proves effective for write-heavy workloads though establishing a method for finding the best parameter is a topic for future study.

Future work includes performance measurement with various workloads including real-world ones though YCSB emulates them. Forms of dependency graphs have an effect on performance of resolution as pointed out in Section 3.2 and it is worthwhile to investigate how workload properties affect the forms.

Investigating the following trade-offs and relationships are open problems.

- A trade-off between the number of middleware instances, and access performance (Section 3.1)
- The relationship between the lengths of histories and workload properties (Section 3.2)
- The best boundary between mutual exclusion and distinct versions for each local set (Section 3.4)
- Other distributions than Zipfian (Section 4.2)

Our protocol expands applicability of such a protocol providing stronger consistency by eliminating the necessity of any modifications to a data store and applications. Causal consistency looks a good target for such a protocol. But it does not deny the possibility of existence of better consistency models for layers of a system including applications, middleware and a data store. A good consistency model involves less modification to each layer, less costs, less and simple interaction between layers, easier extraction of consistency relationships from an application.

Acknowledgments This work is supported by JSPS KAKENHI Grant Numbers 25700008, 26540161 and 16K12406. This work was also supported by the New Energy and Industrial Technology Development Organization (NEDO) especially in a study on performance optimization.

References

[1] Ahamad, M., Neiger, G., Burns, J.E., Kohli, P. and Hutto, P.W.: Causal memory: definitions, implementation, and programming, *Distributed Computing*, Vol.9, No.1, pp.37–49 (1995).
[2] Almeida, S., Leitao, J. and Rodrigues, L.: ChainReaction: A Causal+ Consistent Datastore based on Chain Replication, *Proc. EuroSys 2013*, pp.85–98 (2013).
[3] Apache Software Foundation: Apache Cassandra, available from ⟨http://cassandra.apache.org/⟩
[4] Bailis, P., Ghodsi, A., Hellerstein, J.M. and Stoica, I.: Bolt-on Causal Consistency, *Proc. ACM SIGMOD 2013*, pp.761–772 (2013).
[5] Basho Technologies, Inc.: Riak, available from ⟨http://www.basho.com⟩
[6] Brewer, E.: Towards Robust Distributed Systems, *Keynote Address, ACM PODC 2000* (2000).
[7] Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R. and Sears, R.: Benchmarking Cloud Serving Systems with YCSB, *Proc. ACM SOCC 2010*, pp.143–154 (2010).
[8] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W.: Dynamo: Amazon's Highly Available Key-value Store, *Proc ACM SOSP 2007* (2007).
[9] Du, J., Elnikety, S., Roy, A. and Zwaenepoel, W.: Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks, *Proc. ACM SOCC 2013* (2013).
[10] Gilbert, S. and Lynch, N.: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services, *SIGACT News*, Vol.33, No.2, pp.51–59 (2002).
[11] Lakshman, A. and Malik, P.: Cassandra - A Decentralized Structured Storage System, *Proc. LADIS 2009* (2009).
[12] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System, *Comm. ACM*, Vol.21, No.7, pp.558–565 (1978).
[13] Lloyd, W., Freedman, M.J., Kaminsky, M. and Andersen, D.G.: Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS, *Proc. SOSP 2011*, pp.401–416 (2011).
[14] Lloyd, W., Freedman, M.J., Kaminsky, M. and Andersen, D.G.: Stronger Semantics for Low-Latency Geo-Replicated Storage, *Proc. NSDI'13*, pp.313–328 (2013).
[15] MongoDB, Inc.: MongoDB, available from ⟨http://www.mongodb.org/⟩
[16] Nakamura, S. and Shudo, K.: MyCassandra: A Cloud Storage Supporting both Read Heavy and Write Heavy Workloads, *Proc. SYSTOR 2012* (2012).
[17] Shudo, K. and Yaguchi, T.: Causal Consistency for Distributed Data Stores and Applications as They are, *Proc. IEEE COMPSAC 2016*, pp.602–607 (2016).

Kazuyuki Shudo received B.E. in 1996, M.E. in 1998, and a Ph.D. degree in 2001 all in computer science from Waseda University. He worked as a Research Associate at the same university from 1998 to 2001. He later served as a Research Scientist at National Institute of Advanced Industrial Science and Technology. In 2006, he joined Utagoe Inc. as a Director, Chief Technology Officer. Since December 2008, he currently serves as an Associate Professor at Tokyo Institute of Technology. His research interests include distributed computing, programming language systems and information security. He has received the Best Paper Award at SACSIS 2006, Information Processing Society Japan (IPSJ) Best Paper Award in 2006, the Super Creator certification by Japanese Ministry of Economy Trade and Industry (METI) and Information Technology Promotion Agency (IPA) in 2007, IPSJ Yamashita SIG Research Award in 2008, Funai Prize for Science in 2010, The Young Scientists' Prize, The Commendation for Science and Technology by the Minister of Education, Culture, Sports, and Technology in 2012, and IPSJ Nagao Special Researcher Award in 2013. He is a member of IEEE, IEEE Computer Society, IEEE Communications Society and ACM.

Takashi Yaguchi received B.S. in 2012, and M.S. in 2014 from Tokyo Institute of Technology. His research interests are on database systems and distributed systems.