# Causal Consistency for Distributed Data Stores and Applications as They are

Kazuyuki Shudo, and Takashi Yaguchi
Tokyo Institute of Technology

***Abstract*—There have been proposed protocols to achieve causal consistency with a distributed data store that does not make safety guarantees. Such a protocol works 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 specify explicitly data dependency to be managed. 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. It trades performance for the merit to some extent. Throughput declines from a bare data store were 21% in the best case and 78% in the worst case.**

***Keywords*—*distributed database; consistency model; causal consistency; middleware; concurrency problem***

## I. Introduction

Geo-replication is one of primary features of distributed data stores whereby a client of a data store can access data with small latencies by choosing nearby replicas. But geo-replication usually trades stronger consistency models for its merits [1][2] and then such distributed data stores maintain a consistency model such as eventual consistency [3][4][5], 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 it, data store approach and 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 large dependency graphs to be managed and then the existing protocol taking the middleware approach [6] reduces the size of the graphs by making data store clients specify explicitly 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 (Figure 1).

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

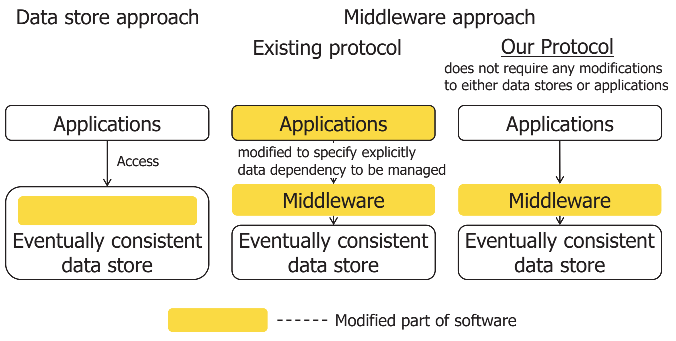
Data store approach
Middleware approach
Existing protocol
Our Protocol
does not require any modifications to either data stores or applications
Applications
Access
Eventually consistent data store
Applications
modified to specify explicitly data dependency to be managed
Middleware
Eventually consistent data store
Applications
Middleware
Eventually consistent data store
Modified part of software

Fig. 1. Approaches to achieving causal consistency.

## II. Background

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

### A. 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 [7][8]. Consider a social networking site as an example of a real-world application. User A posts a photograph on the site. User B sees the photograph and posts a comment on it. All the users who see the comment expect to see the photograph as well. Accordingly, a causally consistent data store guarantees that any user that can observe the photograph if the user observes the comment.

Figure 2 shows an example of operation sequences performed by client 1, 2 and 3. $W(x_i)$ and $R(x_i)$ mean a write operation and a read operation of $x_i$. $x_i$ indicates the version $i$ of the key $x$. Arrows represent causality relationships.

It is a graph of operations and it can be transformed to a graph of data items, strictly a graph of versions of keys. Figure 2 results in Figure 3, that 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 $i+1$ vertexes. In Figure 3, $v_3$ is the level 0 vertex. $x_1$, $y_2$ and $z_1$ are level 1 vertexes. $u_4$ is a level 2 vertex.

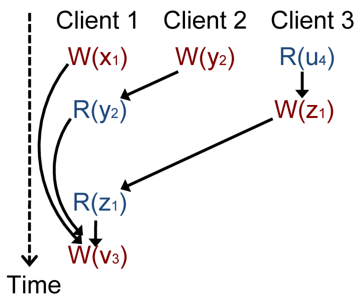

Fig. 2. Example of write and read sequences, and causality relationships between operations.

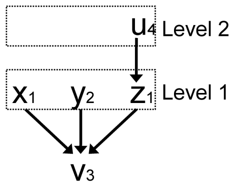

Fig. 3. Dependency graph in case of Figure 2.

If the source of the dependencies (level 0 vertex, $v_3$ in Figure 3) is a target of a write operation, level 1 vertexes, that the source directly depends on, are one of the followings. Our protocol utilizes this fact as described in Section III-B.

1) The last write — a version of a key written just before the source by the same client ($x_1$ in Figure 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 Figure 3)

*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 an 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 [6] requires such explicit specification to reduce the amount of causality relationships it handles.

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

## B. Eventual consistency

There have been emerging distributed data stores whose goals are scalability on numbers of servers while keeping their availability. In compensation for scalability and availability, they loosed requirements on data consistency and their major target has been eventual consistency [3][4][5]. 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 safety guarantees, that causal consistency makes.

## C. Existing protocols and related work

This section describes existing protocols to maintain 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, *data store approach* and *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 [9], Eiger [10], ChainReaction [11] and Orbe [12]. Bolt-on Causal Consistency [6] 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 keeps 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. In detail, it has to 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. [6]. If a protocol lacks an adequate treatment for the problem, part of a dependency graph can be overwritten and lost though the entire graph is still required. In Figure 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.

To address the problem of overwritten dependency graph, the existing protocol taking the middleware approach [6] keeps an entire dependency graph for each data item. In Figure 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 has to keep. Accordingly, The existing protocol reduces the amount of dependency information to be kept by explicit specification described in Section II-A1. 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.

## III. 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

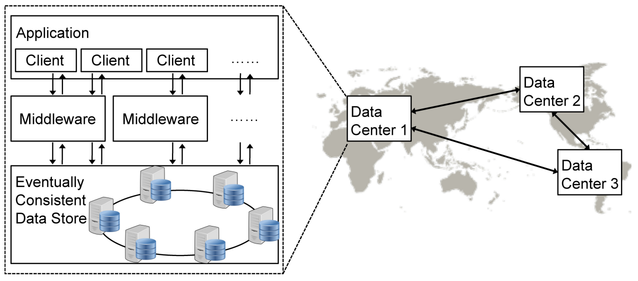

Fig. 4. Supposed system model.

implicit causality relationships. Therefore, it does not require any modifications to either data stores or applications (Figure 1). It works with them as they are.

Section III-A shows the system model the following description of our protocol supposes. In succeeding Sections III-B, III-C and III-D, we propose the protocol.

## A. *System model*

Figure 4 depicts a system model the following description of our protocol supposes. Application instances, middleware instances and a cluster running a data store are located at the same site and they 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. Note that middleware instances do not communicate each other unless a protocol requires. Our protocol performs mutual exclusion between middleware instances in a local set (Section III-D), 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 “last-write-wins”, in which a value with the largest time stamp is chosen as the last value.

## B. *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 II-C.

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 $1$ vertexes of a dependency graph. In Figure 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 keep level 0 and 1 vertexes for all the keys it has. As described in Section II-A, level $1$ vertexes 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 $z_1$). They are just level $1$ vertexes and the middleware embeds them.

In our protocol, a middleware instance keeps only level $0$ and $1$ vertexes for each key unlike the existing protocol [6] that requires an entire graph. By limiting levels it keeps, a middleware instance can handle all the implicit dependency naturally resulting from data accesses.

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, that 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 vertexes, reads values of level $i$ keys to obtain level $i+1$ vertexes, 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 III-A).

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

A middleware instance marks a version of a key in case dependency resolution of it once succeeds. The marks prevent repeated resolution.

## C. *Problem of overwritten dependency graph*

The base protocol does not treat the problem of overwritten dependency graph depicted in Section II-C. 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 to be accompanied by dependency information for multiple versions of the key. In Figure 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 [6], that keeps an entire dependency graph for each key. The existing protocol keeps the same dependency information for multiple keys duplicatedly. In Figure 3, the graph for $v_3$ includes dependency information 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 that $z_1$ depends on $u_4$ appears once in a data store in a local set.

Old dependency information should be wiped out after it becomes unnecessary. A mechanism like garbage collection in programming systems works. There is a trade-off between garbage collecting techniques such as mark and sweep and reference counting. Comparison between them is one of open problems, although they should have little effect on access performance because they run in the background.

### D. Concurrent overwrites by multiple clients

Assuming multiple clients accessing a data item concurrently, required dependency information can be lost even with the 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. Dependency information of $z_2$ is lost.

There is a variety of options to this kind of concurrency problem. We takes a write-time solution in a local set (Section III-A) 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 take any technique for mutual exclusion such as locking and it works. Our current implementation takes an optimistic technique, that is compare-and-swap (CAS). The implementation utilizes 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 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 to the version for the local set it belongs to. Overwriting a key for other local sets does not take place.

All the versions for all the local sets are replicated to all the local sets by replication feature of an underlying data store (Section III-A). The local sets LS1 and LS2 eventually have updated dependency information of $z\_LS1$ and $z\_LS2$. When reading, a protocol has to determine which is the last one between distinct versions for local sets, for example, between $z\_LS1$ and $z\_LS2$. We choose to maintain causal consistency here and use vector clocks [8] for the purpose. Dynamo [3] and Riak [4] adopt the same policy and technique. After a 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.

A whole picture of our protocol is as follows. By vector clocks, a middleware instance captures causality relationships between distinct versions of a key for each local set such as $z\_LS1$ and $z\_LS2$. By dependency graphs, it captures causality relationships between different keys such as $x$, $y$ and $z$. There is a trade-off between mutual exclusion and distinct versions. It is part of future work to investigate the best boundary between them.

TABLE I. YCSB WORKLOADS USED IN SECTION IV.

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

TABLE II. SERVER CONFIGURATION.

| | |
|---|---|
| OS | Ubuntu 12.04.3 with Linux 3.2.0 |
| CPU | 2.40 GHz Xeon E5620 $\times$ 2 |
| Memory | 32 GiB RAM |
| Java Virtual Machine | Java SE 7 Update 4 |

## IV. PERFORMANCE EVALUATION

The contribution of our work is a protocol that maintains causal consistency with no modification to either applications or a data store. Though performance is not the center of our interest, the amount of performance overheads should be acceptable in exchange for the merit. It depends on an application but anyway this section shows experimental results of performance measurement.

### A. Implementation and benchmark conditions

Our implementation of the proposed protocol described in Section III 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 implementation is based on Apache Cassandra 2.1.0 [13][14], that is a production-level and widely deployed distributed data store. Cassandra conforms the system model the protocol supposes described in Section III-A as follows. It provides a function to place replicas of a data item on 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 III-D). Cassandra provides the feature. We implemented the protocol as a library for a client as the same as the existing protocol taking the middleware approach [6] though it is possible to implement as software serving clients via a network.

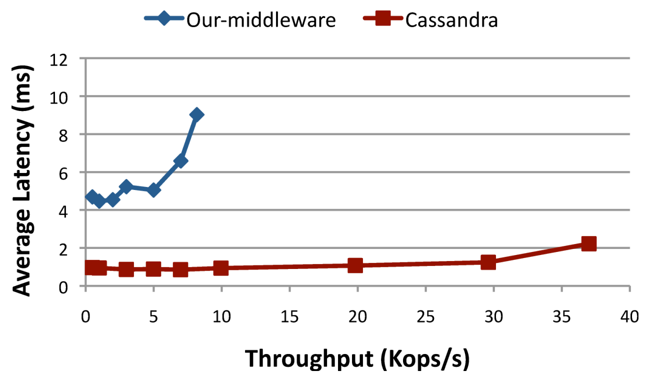

Fig. 5. Write latencies with write-heavy workload.

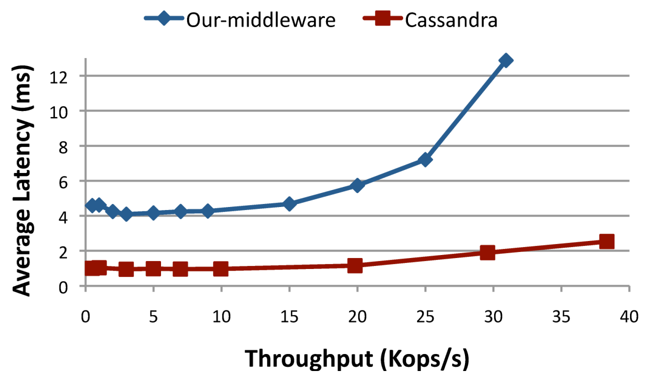

Fig. 7. Write latencies with read-heavy workload.

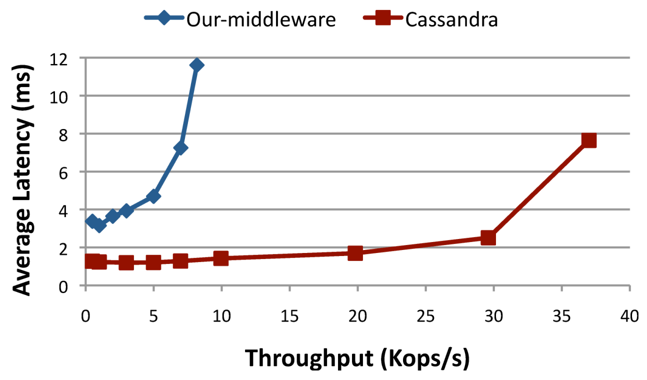

Fig. 6. Read latencies with write-heavy workload.

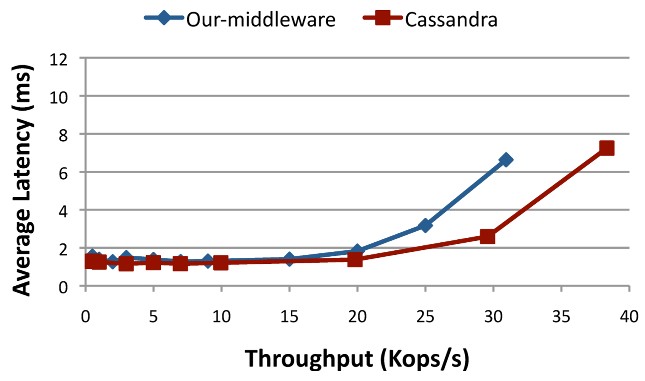

Fig. 8. Read latencies with read-heavy workload.

We use Yahoo! Cloud Serving Benchmark (YCSB) [15] to measure performance of the implementation. It has been widely used by a variety of researches on cloud storage [16]. YCSB issues write and read queries to a target data store continuously and measures access latency. An 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 I shows parameters of the two workloads.

9 servers emulate a data center and and two sets of it 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 II 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 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 the situation in which each of the two data centers hosts a local set.

### *B. Write and read 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 the bare Cassandra. It is interesting to examine performance of the existing protocol taking the middleware approach [6] at first sight. But the existing protocol is designed to be able to handle only explicitly-specified dependency, not all the implicit dependency, that

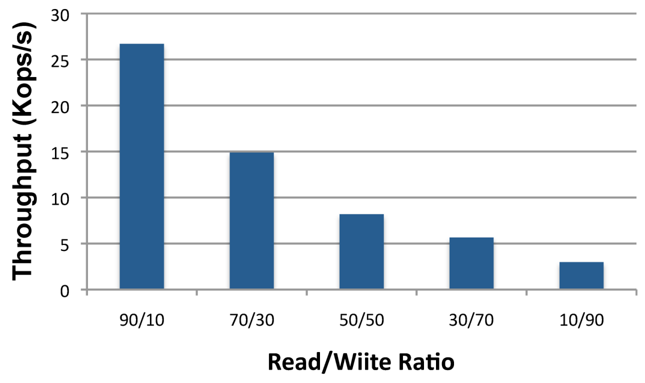

Fig. 9. Maximum throughput with each read / write ratio.

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.

Figure 5 and 6 show access latencies with the write-heavy workload. At 3 and 7 Kbps of throughput, with our implementation, 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 the bare Cassandra.

Figure 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 the bare Cassandra.

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.

Dependency resolution should contribute to the overheads much because it involves multiple accesses to a data store. An access to a data store is one of the heaviest processes because it involves communication over a network. The read-heavy workload requires larger number of dependency resolution than the write-heavy workload because our protocol performs resolution when reading. But if a version of a key has been marked as resolved once, further traversal of a dependency graph is not required as described in Section III-B. 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 YCSB produces. It is interesting to investigate other distributions and find the rational for the results. They are part of future work.

## V. Conclusion

We presented a protocol Letting-It-Be to maintain causal consistency over an existing production-level eventually consistent data stores. Our protocol is unique in that it handles all the implicit dependency naturally resulting from data accesses though it is implemented as middleware. That is, 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 a workload. Throughput declines from bare Cassandra were 21% in the best case and 78% in the worst case.

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 III-B and it is worthwhile to investigate how workload properties affect the forms.

## Acknowledgments

This work is supported by JSPS KAKENHI Grant Numbers 25700008 and 26540161.

## References

[1] E. Brewer, "Towards robust distributed systems," in *Keynote Address, ACM PODC 2000*, Jul. 2000.

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

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proc ACM SOSP 2007*, Oct. 2007.

[4] Basho Technologies, Inc., "Riak," http://www.basho.com.

[5] MongoDB, Inc., "MongoDB," http://www.mongodb.org/.

[6] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *Proc. ACM SIGMOD 2013*, Jun. 2013, pp. 761–772.

[7] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, , and P. W. Hutto, "Causal memory: definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995.

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

[9] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *Proc. SOSP 2011*, Oct. 2011, pp. 401–416.

[10] ——, "Stronger semantics for low-latency geo-replicated storage," in *Proc. NSDI'13*, Apr. 2013, pp. 313–328.

[11] S. Almeida, J. Leitao, and L. Rodrigues, "ChainReaction: a causal+ consistent datastore based on chain replication," in *Proc. EuroSys 2013*, Apr. 2013, pp. 85–98.

[12] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in *Proc. ACM SOCC 2013*, Oct. 2013.

[13] A. Lakshman and P. Malik, "Cassandra - a decentralized structured storage system," in *Proc. LADIS 2009*, Oct. 2009.

[14] Apache Software Foundation, "Apache Cassandra," http://cassandra.apache.org/.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. ACM SOCC 2010*, Jun. 2010, pp. 143–154.

[16] S. Nakamura and K. Shudo, "MyCassandra: A cloud storage supporting both read heavy and write heavy workloads," in *Proc. SYSTOR 2012*, Jun. 2012.