Approximate Causal Consistency for Partially Replicated Geo-Replicated Cloud Storage

Ajay D. Kshemkalyani
Dept. of Computer Science
University of Illinois at Chicago
Chicago, IL 60607-7053, USA
ajay@uic.edu

Ta-yuan Hsu
Dept. of Electrical and Computer Engineering
University of Illinois at Chicago
Chicago, IL 60607-7053, USA
thsu4@uic.edu

ABSTRACT

In geo-replicated systems and the cloud, data replication provides fault tolerance and low latency. Causal consistency in such systems is an interesting consistency model. Most existing works assume the data is fully replicated because this greatly simplifies the design of the algorithms to implement causal consistency. Recently, we proposed causal consistency under partial replication because it reduces the number of messages used under a wide range of workloads. One drawback of partial replication is that its meta-data tends to be relatively large when the message size is small. In this paper, we propose approximate causal consistency whereby we can reduce the meta-data at the cost of some violations of causal consistency. The amount of violations can be made arbitrarily small by controlling a tunable parameter, that we call *credits*.

CCS Concepts

•**Networks** → **Network algorithms;** •**Computing methodologies** → *Distributed algorithms;*

Keywords

causal consistency; causality; cloud computing; distributed shared memory; partial replication

1. INTRODUCTION

In geo-replicated systems and the cloud, data replication provides fault tolerance and low latency. However, consistency of data in the face of concurrent reads and updates becomes an important problem when data is replicated. There exists a range of consistency models in distributed shared memory systems [20]: eventual consistency (the weakest), slow memory, pipelined RAM, causal consistency, sequential consistency, and linearizability (the strongest). Each of these consistency models represents a different trade-off between cost and convenient semantics for the application programmer.

NDM '15, November 15–20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2832099.2832102

In the context of cloud computing with data centers and geo-replicated storage, consistency models have received much attention with product designs from industry, e.g., Amazon, Microsoft, Google, LinkedIn, and Facebook. In the light of the CAP Theorem [17], most systems such as Amazon's Dynamo [13] chose to implement eventual consistency [7]. Besides the three features (Consistency, Availability, and Partition tolerance) of the CAP Theorem, two other desirable features of large-scale distributed data stores are: low latency and high scalability [23]. Causal consistency is the strongest form of consistency that satisfies low latency [23], defined as the latency less than the maximum wide-area delay between replicas. Causal consistency in distributed shared memory systems was proposed by Ahamad et al. [1] and later studied by several researchers [5, 6, 25, 26]. In the past four years, causal consistency has been widely studied [2, 3, 4, 14, 15, 21, 23, 24]. Quite importantly, all the works assume full replication and do not consider the case of partial replication. This is primarily because full replication makes it easy to implement causal consistency and does not have to deal with tracking dependencies between pairs of processes.

In this paper, we focus on causal consistency under partial replication for geo-replicated cloud storage. Studying partial replication is useful because of the following reasons [28]. (1) Recent researchers have explicitly acknowledged that providing causal consistency under partial replication is a big challenge. Bailis et al. [3] and Lloyd et al. [23] write: "While weaker consistency models are often amenable to partial replication, allowing flexibility in the number of data centers required in causally consistent replication remains an interesting aspect of future work." A preliminary discussion of the challenges involved is given in [12]. (2) Partial replication is more natural for applications for which most of the reads come from specific geographical regions only. It is an overkill to replicate the user's data in data centers outside these regions, and partial replication has very small impact on the overall latency in this scenario. (3) With p replicas scattered across n data centers, each write operation that would have triggered an update broadcast to the n data centers now becomes a multicast to just p of the n data centers. This is a direct savings in the number of messages and p is a tunable parameter. Thus, partial replication reduces the number of messages sent with each write operation. Although the read operation may incur additional messages, the overall number of messages will still be lower than the case of full replication if the replication factor is low and the readers tend to read variables from the local replicas

instead of remote ones. Hadoop HDFS and MapReduce is one such example. The HDFS framework usually chooses a small constant number as the replication factor even for large clusters. Moreover, the MapReduce framework tries its best to satisfy data locality. In such a case, partial replication generates far fewer messages than full replication. (4) For write-intensive workloads, it follows that partial replication gives a direct savings in the number of messages without incurring any delay or latency for reads. (5) Partial replication allows a direct savings in resources for networking hardware and storage. (6) The supposedly higher cost of tracking dependency meta-data, which has deterred prior researchers from considering partial replication, is relatively small for applications such as Facebook, where photos, videos, and large files are uploaded.

Recently, we proposed a causal consistency algorithm Opt-Track under partial replication [27, 28]. The algorithm uses a lesser number of messages for a wide range of workloads (including those that are read-intensive) because messages get multicast to a partial set of sites rather than being broadcast throughout the system on each write operation. Four metrics were used in the complexity analysis:

- message count: count of the total number of messages generated by the algorithm.
- message space overhead: the total size of all the messages generated by the algorithm. It can be formalized as $\sum_i$*(# type i messages * size of type i messages)*.
- time complexity: the time complexity at each site s_i for performing the write and read operations.
- space complexity: the space complexity at each site s_i for storing local logs.

The complexity analysis used the following parameters:

- n: the number of sites in the system
- q: the number of variables in the distributed shared memory system
- p: the replication factor, i.e., the number of sites at which each variable is replicated
- w: the number of write operations performed in the distributed shared memory system
- r: the number of read operations performed in the distributed shared memory system

The complexity of Opt-Track is summarized in Table 1.

Of the four metrics, message count is the most important. Partial replication gives a lower message count than full replication if

$$((p-1)+\frac{n-p}{n})w+2r\frac{(n-p)}{n} < (n-1)w \Longrightarrow w > 2\frac{r}{n-1}.$$

This is equivalently stated as: partial replication has a lower message count if the write rate (defined as $w_{rate} = \frac{w}{w+r}$) is such that $w_{rate} > \frac{2}{1+n}$. In addition, the Opt-Track protocol has relatively low meta-data overheads, viz., low message size. This is because it uses the optimization mechanisms used for causal ordering of messages in message-passing systems [18, 19], which were shown to have low meta-data overheads via extensive simulations [9, 10].

Table 1: Complexity measures of Opt-Track [28].

Metric	Opt-Track
Message count	$((p-1)+\frac{n-p}{n})w + 2r\frac{(n-p)}{n}$
Message space overhead	$O(n^2pw + nr(n-p))$ amortized $O(npw + r(n-p))$
Time Complexity	write $O(n^2p)$ read $O(n^2)$
Space Complexity	$O(max(n^2, npq))$ amortized $O(max(n, pq))$

In modern social networks, multimedia files like images and videos (typically several KB to several MB or more) are frequently shared. These files are much larger than the meta-data control information piggybacked with them. Doing full replication might somewhat improve the latency for accessing these files from different locations when locality is not exploited for the partial replication case, however it also incurs a large overhead on the underlying system for transmitting and storing these files across different sites.

Let f be the size of an image/data being written and let b be the number of bytes in an integer.

Under full replication, the net message payload size for the write multicast is $(n-1)f$, and n^3b for the message meta-data overheads [5]. The read cost is zero.

Under partial replication using Opt-Track, the net message payload size is $((p-1)+\frac{(n-p)}{n})f$ for the write multicast, and n^2pb for the message meta-data overheads. The read cost is $(\frac{r}{w})(f)(\frac{n-p}{n})$ because there are $\frac{r}{w}$ reads per write, and $\frac{n-p}{n}$ of the reads fetch the file from a remote location.

Thus, partial replication has lower message size if

$$(n-1)f + n^2(n-1)b >$$

$$((p-1)+\frac{n-p}{n})f + n^2pb + ((\frac{r}{w})(f)(\frac{n-p}{n}))$$

$$\Longrightarrow (n-p)f[1-\frac{1}{n}-\frac{r}{nw}] > n^2b(p-n+1)$$

The above analysis was for r reads and w writes of the image/data.

Example:

Consider a system with n=10, p=3 and w_{rate} = 0.5. A message of 1000 bytes has to be stored and transmitted in a write operation.

- With full-replication, the message takes 10KB space locally across all data centers. The normalized cost of the total message size for transmitting this file in a Write operation (and subsequently reading it in the system) is 9KB.
- With partial replication using algorithm Opt-Track, the message takes 3KB space locally across all data centers. The normalized cost of the total message size for transmitting this file in a Write operation (and subsequently reading it in the system), even assuming the asymptotic case of the formula for message space overhead, namely $O(n(n-1)pw + nr(n-p))$, is approximately 3KB + (10(9)(3)(1) + 10(1)(7))(# bytes for each entry in the $Write$ clock in Opt-Track) = 3KB + (270 + 70)(4) = 4,360 bytes.

Contributions

For images of the size of tens of KB, the net message space overhead is negligible and Algorithm Opt-Track shows very good performance. However, we recognize that for some applications where the data size is very small, such as wall posts in Facebook or Twitter, the size of the meta-data (quadratic in n in the worst case, even for Algorithm Opt-Track) can be a problem. This paper aims to reduce the size of the meta-data for maintaining causal consistency in partially replicated systems.

1. We propose the concept of *approximate* causal consistency whereby we can reduce the meta-data at the cost of some possible violations of causal consistency. The amount of violations can be made arbitrarily small by controlling a tunable parameter, which we call *credits*.

2. We integrate the notion of credits into the Opt-Track algorithm, to give an algorithm Approx-Opt-Track that can fine-tune the amount of causal consistency by trading off the size of meta-data overhead.

3. We give three instantiations of the notion of credits, namely hop count, time-to-live, and metric distance.

Organization

Section 2 gives the system model of causally consistent memory and the underlying distributed communication system. This section uses the model used previously in [27, 28]. Section 3 presents the algorithm Approx-Opt-Track that implements causal consistency under partial replication. We have integrated the notion of credits into this algorithm, and explain how it leads to approximate causal consistency. Section 4 shows how the notion of credits can be instantiated. Section 5 gives a discussion and concludes.

2. SYSTEM MODEL

2.1 Causally Consistent Memory

We consider a system which consists of n application processes $ap_1, ap_2, \ldots, ap_n$ interacting through a distributed shared memory $\mathcal{Q}$ composed of q variables $x_1, x_2, \ldots, x_q$. Each application process ap_i can perform either a *read* or a *write* operation on any of the q variables. A *read* operation performed by ap_i on variable x_j which returns value v is denoted as $r_i(x_j)v$. Similarly, a *write* operation performed by ap_i on variable x_j which writes the value u is denoted as $w_i(x_j)u$. Each variable has an initial value $\perp$.

By performing a series of *read* and *write* operations, an application process ap_i generates a local history h_i. If a local operation o_1 precedes another operation o_2, we say o_1 precedes o_2 under *program order*, denoted as $o_1 \prec_{po} o_2$. The set of local histories h_i from all n application processes form the global history H. Operations performed at distinct processes can also be related using the *read-from order*, denoted as $\prec_{ro}$. Two operations o_1 and o_2 from distinct processes ap_i and ap_j respectively have the relationship $o_1 \prec_{ro} o_2$ if there are variable x and value v such that $o_1 = w(x)v$ and $o_2 = r(x)v$, meaning that *read* operation o_2 retrieves the value written by the *write* operation o_1. As shown in [1],

- for any operation o_2, there is at most one operation o_1 such that $o_1 \prec_{ro} o_2$;
- if $o_2 = r(x)v$ for some x and there is no operation o_1 such that $o_1 \prec_{ro} o_2$, then $v = \perp$, meaning that a read with no preceding write must read the initial value.

With both the *program order* and *read-from order*, the *causality order*, denoted as $\prec_{co}$, can be defined on the set of operations O_H in a history H. The causality order is the transitive closure of the union of local histories' program order and the read-from order. Formally, for two operations o_1 and o_2 in O_H, $o_1 \prec_{co} o_2$ if and only if one of the following conditions holds:

1. $\exists ap_i$ s.t. $o_1 \prec_{po} o_2$ (program order)
2. $\exists ap_i, ap_j$ s.t. o_1 and o_2 are performed by ap_i and ap_j respectively, and $o_1 \prec_{ro} o_2$ (read-from order)
3. $\exists o_3 \in O_H$ s.t. $o_1 \prec_{co} o_3$ and $o_3 \prec_{co} o_2$ (transitive closure)

Essentially, the causality order defines a partial order on the set of operations O_H. For a shared memory to be causal memory, all the write operations that can be related by the causality order have to be seen by each application process in the order defined by the causality order.

2.2 Underlying Distributed Communication System

The distributed shared memory abstraction and its causal consistency model is implemented on top of the underlying distributed message passing system which also consists of n sites connected by FIFO channels, with each site s_i hosting an application process ap_i. Since we assume a partially replicated system, each site holds only a subset of variables $x_h \in \mathcal{Q}$. For application process ap_i, we denote the subset of variables kept on the site s_i as X_i. If the replication factor of the distributed shared memory system is p and the variables are evenly replicated on all the sites, then the average size of X_i is $\frac{pq}{n}$.

To facilitate the read and write operations in the distributed shared memory abstraction, the underlying message passing system provides several primitives to enable the communication between different sites. For the write operation, each time an application process ap_i performs $w(x_1)v$, it invokes the `Send`(m) primitive to deliver the message m containing $w(x_1)v$ to all sites that replicate the variable x_1. For the read operation, there is a possibility that an application process ap_i performing read operation $r(x_2)u$ needs to read x_2's value from a remote site since x_2 is not locally replicated. In such a case, it invokes the `RemoteFetch`(m) primitive to deliver the message m containing $r(x_2)u$ to a randomly selected site replicating x_2 to fetch its value u. This is a synchronous primitive, i.e., it will block until returning the variable's value. If the variable to be read is locally replicated, then the application process simply returns the local value.

The read and write operations performed by the application processes also generate *events* in the underlying message passing system. The following is a list of events:

- *Send event.* The invocation of `Send`(m) primitive by application process ap_i generates event $send_i(m)$.
- *Fetch event.* The invocation of `RemoteFetch`(m) primitive by application process ap_i generates event $fetch_i(f)$.

- *Message receipt event.* The receipt of a message m at site s_i generates event $receipt_i(m)$. The message m can correspond to either a $send_j(m)$ event or a $fetch_j(f)$ event.
- *Apply event.* When applying the value written by the operation $w_j(x_h)v$ to variable x_h's local replica at application process ap_i, an event $apply_i(w_j(x_h)v)$ is generated.
- *Remote return event.* After the occurrence of event $receipt_i(m)$ corresponding to the remote read operation $r_j(x_h)u$ performed by ap_j, an event $remote\_return_i(r_j(x_h)u)$ is generated which transmits x_h's value u to site s_j.
- *Return event.* Event $return_i(x_h, v)$ corresponds to the return of x_h's value v either fetched remotely through a previous $fetch_i(f)$ event or read from the local replica.

Each time an update message m corresponding to a write operation $w_j(x_h)v$ is received at site s_i, a new thread is spawned to check when to locally apply the update. The condition that the update is ready to be applied locally is called activation predicate in [5]. This predicate, $A(m_{w_j(x_h)v}, e)$, is initially set to $false$ and becomes $true$ only when the update $m_{w_j(x_h)v}$ can be applied after the occurrence of local event e. The thread handling the local application of the update will be blocked until the activation predicate becomes $true$, at which time the thread writes value v to variable x_h's local replica. This will generate the $apply_i(w_j(x_h)v)$ event locally.

2.3 Activation Predicate

Consider the relation, $\rightarrow_{co}$, on *send events* generated in the underlying message passing system [5]. We modify its definition by adding condition (3) to accommodate the partial replication case. Let $w(x)a$ and $w(y)b$ be two write operations in O_H. Then, for their corresponding send events in the underlying message passing system, $send_i(m_{w(x)a}) \rightarrow_{co} send_j(m_{w(y)b})$ iff one of the following conditions holds:

1. $i = j$ and $send_i(m_{w(x)a})$ locally precedes $send_j(m_{w(y)b})$
2. $i \neq j$ and $return_j(x, a)$ locally precedes $send_j(m_{w(y)b})$
3. $i \neq j$ and $apply_l(w(x)a)$ locally precedes $remote\_return_l(r_j(x)a)$, which precedes (as per Lamport's $\rightarrow$ relation [22]) $return_j(x, a)$, which locally precedes $send_j(m_{w(y)b})$
4. $\exists send_k(m_{w(z)c})$, such that $send_i(m_{w(x)a}) \rightarrow_{co} send_k(m_{w(z)c}) \rightarrow_{co} send_j(m_{w(y)b})$

The relation defined by $\rightarrow_{co}$ is a subset of Lamport's "happened before" relation [22], denoted by $\rightarrow$. If two send events are related by $\rightarrow_{co}$, then they are also related by $\rightarrow$. However, the other way is not necessarily true. Even though $send_i(m_{w(x)a}) \rightarrow send_j(m_{w(y)b})$, if there is no return event that occurred and $i \neq j$, these two send events are concurrent under the $\rightarrow_{co}$ relation. The $\rightarrow_{co}$ relation better represents the causality order in the distributed shared memory abstraction as it prunes the "false causality" in the underlying message passing system, where message receipt events may causally relate two send events while their corresponding write operations in the shared memory abstraction are concurrent under the $\prec_{co}$ relation. (False causality was identified by Lamport [22] and later discussed by others [3, 4, 8, 11, 16, 21].) In [5], the authors have shown that $send_i(m_{w(x)a}) \rightarrow_{co} send_j(m_{w(y)b}) \Leftrightarrow w(x)a \prec_{co} w(y)b$.

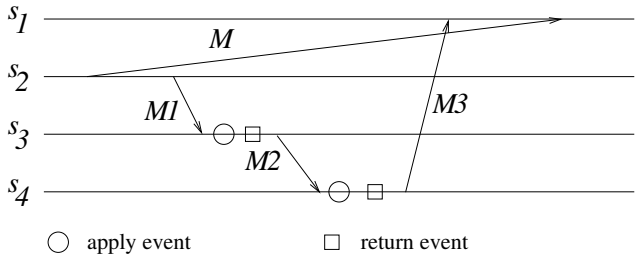

Figure 1: Illustration of causal consistency violation if credits are exhausted.

The optimal activation predicate is as follows:

$$A_{OPT}(m_w, e) \equiv \nexists m_{w'} : (send_j(m_{w'}) \rightarrow_{co} send_k(m_w) \wedge apply_i(w') \notin E_i|_e)$$

where $E_i|_e$ is the set of events that happened at the site s_i up until e (excluding e).

This activation predicate cleanly captures the causal memory's requirement: a write operation shall not be seen by an application process before any causally preceding write operations. It is optimal because the moment this activation predicate $A_{OPT}(m_w, e)$ becomes true is the earliest instant that the update m_w can be applied.

3. ALGORITHM APPROX-OPT-TRACK

Using implicit knowledge about messages delivered and guaranteed to be delivered in causal order, the Opt-Track algorithm [28] automatically prunes the meta-data. In the amortized case, the meta-data is manageable and linear in n, rather than quadratic (See Table 1 [28]) [9, 10, 18, 19]. We can further reduce the size of meta-data by deleting older dependencies rather than carry them around and store them in logs. With very high probability, the older the dependencies are, the more they are likely to be immediately satisfied as the corresponding messages are more likely to be delivered.

We introduce the notion of *credits* associated with each meta-data unit of information. When a dependency is created, it is allocated a certain number of initial credits. For every read and write operation, we decrement the available credits by some used-up credits, and when the available number of credits reaches zero, the dependency becomes "old enough" and can be deleted. By setting the initial credits to ∞, we get the original Opt-Track algorithm. By setting them to a smaller finite value, we can prune meta-data information about older dependencies by risking that those dependencies might not be satisfied, rather than wait for the pruning mechanisms of Opt-Track to prune them. *Credits* is a parameter that lets us approximate causal consistency to the accuracy desired.

Consider the timing diagram in Figure 1. The messages shown indicate those sent due to Write operations to update the remote replica. The causality chain induced by Write operations corresponding to $M1$, $M2$, and $M3$, and the intervening *apply* and *return* events, ends in $M3$ being sent to site s_1. Normally in Opt-Track, the meta-data on $M3$ contains the dependency that "M is sent to s_1", and will

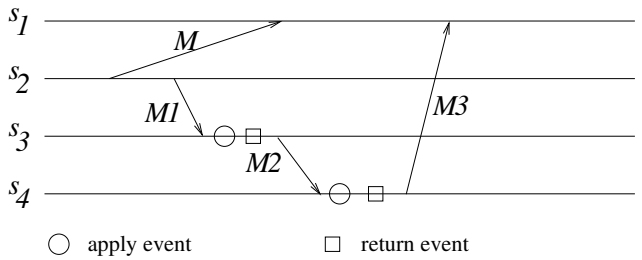

Figure 2: Illustration of meta-data reduction when credits are exhausted.

prevent $M3$ from being delivered to s_1 before M is delivered. However, if the credits get expired along this causality chain, then $M3$ will not carry the meta-data dependency that "M is sent to s_1" and hence $M3$ will be delivered by violating causal consistency at s_1. If credits are decremented slowly enough, then with very high probability, $M3$ will carry the meta-data information about M and causal consistency is not violated.

On the other hand, consider the timing diagram in Figure 2. The scenario is the same as in Figure 1, with the exception that message M is delivered to s_1 within a reasonable (i.e., an expected) amount of time. Assume that the credits about the dependency "M is sent to s_1" get exhausted when $M2$ is delivered to s_4 along the causality chain $\langle M1, M2\rangle$. The dependency is thus deleted at s_4 and is not carried in the meta-data sent along with message $M3$. This results in reduced meta-data on $M3$. This does not cause any violation of causal consistency when the reduced meta-data is delivered to s_1, because M has already been delivered to s_1.

We give the algorithm Approx-Opt-Track in Algorithm 1. The following data structures are maintained at each site.

1. $clock_i$: local counter at site s_i for write operations performed by application process ap_i.

2. $Apply_i[1 \ldots n]$: an array of integers (initially set to 0s). $Apply_i[j] = a$ means that a total number of a updates written by application process ap_j have been applied at site s_i.

3. $LOG_i = \{\langle j, clock_j, Dests, cr\rangle\}$: the local log (initially set to empty). Each entry indicates a write operation in the causal past. $Dests$ is the destination list for that write operation. Only necessary destination information is stored. cr is the remaining amount of credits allowed before the entry ages out.

4. $LastWriteOn_i\langle$variable id, $LOG\rangle$: a hash map of LOGs. $LastWriteOn_i\langle h\rangle$ stores the piggybacked LOG from the most recent update applied at site s_i for locally replicated variable x_h.

The data structures are the same as in algorithm Opt-Track, with the addition of the credits parameter cr in each entry in LOG_i. Algorithm 1 implements the optimality mechanisms described in algorithm Opt-Track [28]. The optimal activation predicate A_{OPT} is implemented in lines (28)-(29), as in algorithm Opt-Track.

Algorithm 2 gives the procedures used by Algorithm Approx-Opt-Track (Algorithm 1). Function `PURGE` removes old records

Algorithm 1: Approx-Opt-Track Algorithm, which is a modification of Algorithm Opt-Track [28] (Code at site s_i)

```
   WRITE(x_h, v):
1  clock_i ++;
2  cr := initial credits;
3  for all l ∈ LOG_i do
4  |  l.cr := l.cr − used credits;
5  |  if l.cr ≤ 0 ∧ l.Dests ≠ ∅ then delete l;
6  for all sites s_j (j ≠ i) that replicate x_h do
7  |  L_w := LOG_i;
8  |  for all o ∈ L_w do
9  |  |  if s_j ∉ o.Dests then
   |  |  o.Dests := o.Dests \ x_h.replicas;
10 |  |  else o.Dests := o.Dests \ x_h.replicas ∪ {s_j};
11 |  for all o_{z,clock_z} ∈ L_w do
12 |  |  if o_{z,clock_z}.Dests = ∅ ∧ (∃ o'_{z,clock'_z} ∈ L_w | clock_z <
   |  |  clock'_z) then remove o_{z,clock_z} from L_w;
13 |  Send m(x_h, v, i, clock_i, x_h.replicas, cr, L_w) to site s_j;
14 for all l ∈ LOG_i do
15 |  l.Dests := l.Dests \ x_h.replicas;
16 PURGE;
17 LOG_i := LOG_i ∪ {⟨i, clock_i, x_h.replicas \ {s_i}, cr⟩};
18 if x_h is locally replicated then
19 |  x_h := v;
20 |  Apply_i[i] ++;
21 |  LastWriteOn_i⟨h⟩ := LOG_i;
   READ(x_h):
22 if x_h is not locally replicated then
23 |  RemoteFetch[f(x_h)] from randomly selected site s_j that
   |  replicates x_h to get x_h and LastWriteOn_j⟨h⟩;
24 |  MERGE(LOG_i, LastWriteOn_j⟨h⟩);
25 else MERGE(LOG_i, LastWriteOn_i⟨h⟩);
26 PURGE;
27 return x_h;
   On receiving m(x_h, v, j, clock_j, x_h.replicas, c, L_w) from site s_j:
28 for all o_{z,clock_z} ∈ L_w do
29 |  if s_i ∈ o_{z,clock_z}.Dests then wait until
   |  clock_z ≤ Apply_i[z];
30 for all o ∈ L_w do
31 |  o.cr := o.cr − used credits;
32 |  if o.cr ≤ 0 ∧ o.Dests ≠ ∅ then delete o;
33 x_h := v;
34 Apply_i[j] := clock_j;
35 L_w := L_w ∪ {⟨j, clock_j, x_h.replicas, c − used credits⟩};
36 for all o_{z,clock_z} ∈ L_w do
37 |  o_{z,clock_z}.Dests := o_{z,clock_z}.Dests \ {s_i};
38 LastWriteOn_i⟨h⟩ := L_w;
   On receiving f(x_h) from site s_j:
39 return x_h and LastWriteOn_i⟨h⟩ to s_j;
```

with ∅ destination lists, per sender process. On a read operation of variable x_h, function `MERGE` merges the piggybacked log of the corresponding write to x_h with the local log LOG_i. In this function, new dependencies get added to LOG_i and existing dependencies in LOG_i are pruned, based on the information in the piggybacked data L_w. The merging implements the optimality mechanisms described in [28].

Notice that in the `PURGE` function, and in lines 11-12 of the WRITE procedure, entries with empty destination list are kept as long as and only as long as they are the most recent update from the sender. This is required for implicit track-

Algorithm 2: Procedures used in Algorithm 1, Approx-Opt-Track Algorithm (Code at site s_i)

```
   PURGE:
1  for all l_{z,t_z} ∈ LOG_i do
2      if l_{z,t_z}.Dests = ∅ ∧ (∃ l'_{z,t'_z} ∈ LOG_i | t_z < t'_z) then
3          remove l_{z,t_z} from LOG_i;

   MERGE(LOG_i, L_w):
4  for all l ∈ LOG_i do
5      l.cr := l.cr − used credits;
6  for all o ∈ L_w do
7      o.cr := o.cr − used credits;
8  for all o_{z,t} ∈ L_w and l_{s,t'} ∈ LOG_i such that s = z do
9      if t < t' ∧ l_{s,t} ∉ LOG_i then mark o_{z,t} for deletion;
10     if t' < t ∧ o_{z,t'} ∉ L_w then mark l_{s,t'} for deletion;
11     delete marked entries;
12     if t = t' then
13         l_{s,t'}.Dests := l_{s,t'}.Dests ∩ o_{z,t}.Dests;
14         l_{s,t'}.cr := min(l_{s,t'}.cr, o_{z,t}.cr);
15         delete o_{z,t} from L_w;
16 LOG_i := LOG_i ∪ L_w;
17 for all l ∈ LOG_i do
18     if l.cr ≤ 0 ∧ l.Dests ≠ ∅ then delete l;
```

ing of messages delivered and guaranteed to be delivered in causal order, as explained in [18, 19, 28]. Such entries should not be deleted even if their credits allocation becomes zero. Thus in line 5, line 32, of the main algorithm, and in line 18 of `MERGE`, we delete an entry with exhausted credits only if its destination list is non-empty.

The main changes to algorithm Opt-Track are as follows.

1. Line 2: initial credit assignment for a new dependency created by a write operation.
2. Lines 3-5: The LOG_i entries are aged by consuming some credits, and deleted if the remaining credits are zero.
3. Line 13: The send message has a new parameter, cr, for the credit allocation of the corresponding dependency.
4. Line 17: The new local LOG_i entry has a new field for the credit allocation of the new dependency.
5. Lines 30-32: Entries in the piggybacked meta-data are aged by consuming some credits, and deleted if the remaining credits are zero.
6. Line 35: The log entry added for the new dependency of the received message has a new parameter for the remaining amount of credits left.
7. Lines 4-7 in `MERGE`: Lines 4-5 decrement the credits for each entry in LOG_i by the amount of credits used. Lines 6-7 decrement the credits for each entry in L_w by the amount of credits used.
8. Lines 14 in `MERGE`: For each pair of records corresponding to the same dependency being merged, we retain the minimum of the available credits. This captures the fact that the maximum amount of credits have been consumed.
9. Lines 17-18 in `MERGE`: After the credits consumption in lines 4-7 and after the credit adjustment in line 14, we delete LOG_i entries if their available credits are zero.

4. CREDIT INSTANTIATIONS

We show three different ways in which the concept of credits can be instantiated.

4.1 Hop Count

Credits of a meta-data entry denote the hop count available before the entry ages out and is deleted. A message is said to traverse one hop when it traverses along a logical channel between any pair of processes (sites). We make some notes about this instantiation of Algorithm Approx-Opt-Track.

1. Line 2: initial assignment of the hop count for a new dependency created by a write operation.
2. Lines 3-5: These lines are no-ops because there is no message transfer.
3. Lines 30-32: In line 31, the hop count is decremented by one for each entry in the piggybacked meta-data received.
4. Line 35: The hop count is decremented by one for the new dependency just formed by the received message.
5. Lines 4-7 in `MERGE`: The entries in LOG_i do not experience any decrease in hop count, while the entries in L_w have the hop count decremented if the data was remotely fetched by the read operation that triggered the `MERGE`.
6. Lines 14 in `MERGE`: The hop count is set to the minimum of the hop counts of the entries being merged.
7. Lines 17-18 in `MERGE`: LOG_i entries whose hop count is zero are deleted.

We expect that if the initial allocation of hop count is made as a high single-digit, by the time the hop count reaches zero and the meta-data entry is deleted, the message about which the meta-data is deleted would already have reached its destination (with very high probability).

4.2 Physical Time Lapse

Credits of a meta-data entry denote the physical time-to-live (TTL) before the entry ages out and is deleted. This instantiation assumes only that physical clocks are loosely synchronized because no catastrophic error results if the clocks are not tightly synchronized. We make some notes about this instantiation of Algorithm Approx-Opt-Track.

1. Line 2: initial assignment of the TTL for a new dependency created by a write operation.
2. Lines 3-5: Decrement the TTL by the physical time that the log entries have been sitting in the local LOG_i.
3. Lines 30-32: In line 31, the TTL is decremented by the propagation time for the message that was received.
4. Line 35: The TTL is decremented by the propagation time for the message that was just received.

5. Lines 4-7 in `MERGE`: The entries in LOG_i and L_w have their TTL decremented by the amount of time lapse since the last time each entry's TTL was updated.

6. Lines 14 in `MERGE`: The TTL is set to the minimum of the TTLs of the entries being merged.

7. Lines 17-18 in `MERGE`: LOG_i entries whose TTL is zero are deleted.

We expect that if the initial allocation of TTL is made as a high single-digit multiple of the average inter-data center message propagation-cum-transmission time, by the time the TTL reaches zero and the meta-data entry is deleted, the message about which the meta-data is deleted would already have reached its destination (with very high probability).

4.3 Metric Distance Traversed

Credits of a meta-data entry denote the physical distance the entry traverses before it ages out. This instantiation of credits assumes that the topology of the network is known by the data centers/sites, viz., the physical distance between each pair of data centers is known.

We make some notes about this instantiation of Algorithm Approx-Opt-Track.

1. Line 2: initial allocation of metric distance traversed for a new dependency created by a write operation.

2. Lines 3-5: These lines are no-ops because there is no message transfer.

3. Lines 30-32: In line 31, the metric distance is decremented by the distance traversed by the message just received.

4. Line 35: The metric distance is decremented by the distance traversed by the message just received.

5. Lines 4-7 in `MERGE`: The entries in LOG_i do not experience any decrease in metric distance, while the entries in L_w have the metric distance decremented if the data was remotely fetched by the read operation that triggered the `MERGE`.

6. Lines 14 in `MERGE`: The metric distance is set to the minimum of the metric distances of the entries being merged.

7. Lines 17-18 in `MERGE`: LOG_i entries whose metric distance is zero are deleted.

We expect that if the initial allocation of metric distance is made as a high single-digit multiple of the average inter-data center message distance, by the time the residual metric distance reaches zero and the meta-data entry is deleted, the message about which the meta-data is deleted would already have reached its destination (with very high probability).

5. DISCUSSION AND CONCLUSIONS

We considered the problem of providing causal consistency in large-scale geo-replicated storage under the assumption of partial replication. The recently proposed algorithm Opt-Track [28] is optimal in the sense that that each update is applied at the earliest instant while removing false causality in the system. It is additionally optimal in the sense that it minimizes the size of meta-information carried on messages and stored in local logs. However, in the worst case, the size of the meta-data may be quadratic in the number of sites/data centers. In this paper, we proposed a modification of Opt-Track, called Approx-Opt-Track, that gives approximate causal consistency by reducing the size of the meta-data. By controlling a parameter called credits, we can trade-off the level of potential inaccuracy by the size of meta-data. We showed three different ways in which credits can be instantiated, namely by hop count, TTL, and metric distance.

As future work, we would like to (i) study the exact nature of the trade-off experimentally for all the three instantiations of credits, (ii) provide transactional semantics, and (iii) provide causal+ consistency.

The introduction of the concept of credits and their manipulation in the algorithm Approx-Opt-Track does not increase the message count complexity, message space complexity, time complexity, or space complexity beyond the corresponding values of algorithm Opt-Track (see Table 1).

6. REFERENCES

[1] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: definitions, implementation and programming. *Distributed Computing, 9, 1*, pages 37–49, 1995.

[2] S. Almeida, J. Leitao, and L. Rodrigues. ChainReaction: a causal+ consistent datastore based on chain replication. *In ACM Eurosys*, pp. 85-98, 2013.

[3] P. Bailis, A. Fekete, A. Ghodsi, J.M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution. *In ACM SOCC*, 2012.

[4] P. Bailis, A. Ghodsi, J.M. Hellerstein, and I. Stoica. Bolt-on causal consistency. *ACM SIGMOD*, pp. 761-772, 2014.

[5] R. Baldoni, A. Milani, and S. Piergiovanni. Optimal propagation-based protocols implementing causal memories. *Distributed Computing, 18, 6*, pages 461–474, 2006.

[6] N. Belaramani, M. Dahlin, L. Gao, A. Venkataramani, P. Yalagandula, and J. Zheng. Practi replication. *In NSDI*, 2006.

[7] P. Bernstein and S. Das. Rethinking eventual consistency. *Proc. of the 2013 ACM SIGMOD International Conf. on Management of Data*, 2013.

[8] K. Birman. A response to Cheriton and Skeen's criticism of causally and totally ordered communication. *In ACM SIGOPS Operating Systems Review*, 28(1): 11-21, 1994.

[9] P. Chandra, P. Gambhire, and A.D. Kshemkalyani. Performance of the optimal causal multicast algorithm: A statistical analysis. *IEEE Transactions on Parallel and Distributed Systems, 15(1)*, pages 40–52, January 2004.

[10] P. Chandra and A.D. Kshemkalyani. Causal multicast in mobile networks. *Proc. of the 12th IEEE/ACM Symposium on Modelling, Analysis, and Simulation of Computer and Communication Systems*, pages 213–220, 2004.

[11] D.R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. *In ACM SOSP*, 1993.

[12] T. Crain and M. Shapiro. Designing a causally consistent protocol for geo-replicated partial replication. *In ACM PaPoC*, 2015.

[13] 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. *Proc. of the 19th ACM SOSP*, pages 205–220, 2007.

[14] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. *In ACM SOCC*, 2013.

[15] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. GentleRain: Cheap and scalable causal consistency with physical clocks. *In ACM SOCC*, 2014.

[16] P. Gambhire and A.D. Kshemkalyani. Reducing false causality in causal message ordering. *Proc. 7th International High Performance Computing Conference (HiPC), LNCS 1970, Springer,* pp 61-72, 2000.

[17] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 2002.

[18] A.D. Kshemkalyani and M. Singhal. An optimal algorithm for generalized causal message ordering. *Proc. of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, page 87, 1996.

[19] A.D. Kshemkalyani and M. Singhal. Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. *Distributed Computing, 11, 2*, pages 91–111, 1998.

[20] A.D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems.* Cambridge University Press, 2008.

[21] K. Lady, M. Kim, and B. Noble. Declared causality in wide-area replicated storage. *In Workshop on Planetary-Scale Distributed Systems*, 2014.

[22] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM, 21*, pages 558-564, 1978.

[23] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. *Proc. of the 23rd ACM SOSP*, pages 401–416, 2011.

[24] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. Stronger semantics for low latency geo-replicated storage. *In NSDI*, 2013.

[25] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. *Tech. Rep. TR-11-22, Univ. Texas at Austin, Dept. Comp. Sci.*, 2011.

[26] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. *In SOSP*, 1997.

[27] M. Shen, A.D. Kshemkalyani, and T. Hsu. OPCAM: Optimal algorithms implementing causal memories in shared memory systems. *In Int. Conference on Distributed Computing and Networking (ICDCN)*, Jan 2015.

[28] M. Shen, A.D. Kshemkalyani, and T. Hsu. Causal consistency for geo-replicated cloud storage under partial replication. *IEEE Int. Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 509-518, May 2015.