Causal Consistency for Geo-Replicated Cloud Storage under Partial Replication

Min Shen, Ajay D. Kshemkalyani, TaYuan Hsu
University of Illinois at Chicago
Outline

1 Introduction
2 System Model
3 Algorithms
   - Full-Track Algorithm: partially replicated memory
   - Opt-Track Algorithm: partially replicated memory
   - Opt-Track-CRP: fully replicated memory
4 Complexity measures of causal memory algorithms
5 Conclusion
6 References
Introduction

- Data Replication - a technique for fault tolerance in distributed systems
  Reduces access latency in the cloud and geo-replicated systems.

- Consistency of data - a core issue in the distributed shared memory

- Consistency Models
  Represent a trade-off (cost V.S. convenient semantics)
  - linearizability (the strongest)
  - sequential consistency
  - causal consistency [1]
  - pipelined RAM
  - slow memory
  - eventual consistency (the weakest)

- Industry interest
  For example, Google, Amazon, Microsoft, Facebook, LinkedIn
Geo-Replicated Cloud Storage Features

- CAP theorem (Brewer, 2000) cannot provide all 3 features in the same system
  - Consistency of Replicas
  - Availability of Writes
  - Partition Tolerance
- low Latency
- high Scalability
Causal consistency in distributed shared memory systems (Ahamad et al.)

Causal consistency has been studied (by Baldoni et al., Mahajan et al., Belaramani et al., Petersen et al.).

In the past four years,
- ChainReaction (S. Almeida et al.)
- Bolt-on causal consistency (P. Bailis et al.)
- Orbe and GentleRain (J. Du et al.)
- Wide-Area Replicated Storage (K. Lady et al.)
- COPS, Eiger (W. Lloyd et al.)

The above works assume full replication.
Partial Replication

Figure: Case for Partial Replication.
Partial replication is more natural for some applications. As shown in the previous case, ...

- With $p$ replicas places at some $p$ of the total of $n$ DCs, each write operation that would have triggered an update broadcast to the $n$ DCs now becomes a multicast to just $p$ of the $n$ DCs.

For write-intensive workloads, partial replication gives a direct savings in the number of messages.

- Allowing flexibility in the number of DCs required in causally consistent replication remains an interesting aspect of future work.

- The supposedly higher cost of tracking dependency metadata is relatively small for applications such as Facebook.
A system with $n$ application processes - $ap_1$, $ap_2$, ..., $ap_n$ - interacting through a shared memory $Q$ composed of $q$ variables $x_1, x_2, ..., x_q$

Each $ap_i$ can perform either a read or a write operation on any of the $q$ variables.

- $r_i(x_j)v$: a read operation performed by $ap_i$ on variable $x_j$ which returns value $v$
- $w_i(x_j)v$: a write operation performed by $ap_i$ on variable $x_j$ which writes value $v$

Each variable has an initial value $\bot$.

- local history $h_i$: a series of read and write operations generated by process $ap_i$
- global history $H$: the set of local histories $h_i$ from all $n$ application processes
Causally Consistent Memory [1]

- **Program Order**: under which a local operation $o_1$ precedes another operation $o_2$, denoted as $o_1 \prec_{po} o_2$

- **Read-from Order**: there are variable $x$ and value $v$ such that read operation $o_2 = r(x)v$ retrieves the value $v$ written by the write operation $o_1 = w(x)v$ from a distinct process, denoted as $o_1 \prec_{ro} o_2$
  - 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 no operation $o_1$ such that $o_1 \prec_{ro} o_2$, then $v = \bot$

- **Causality Order**: 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:
  - $\exists ap_i \text{ s.t. } o_1 \prec_{po} o_2$ (program order)
  - $\exists ap_i, ap_j \text{ s.t. } o_1 \prec_{ro} o_2$ (read-from order)
  - $\exists o_3 \in O_H \text{ s.t. } o_1 \prec_{co} o_3 \text{ and } o_3 \prec_{co} o_2$ (transitive closure)
The shared memory abstraction and its causal consistency model is implemented on top of the distributed message passing system.

With $n$ sites (connected by FIFO channels), each site $s_i$ hosts an application process $ap_i$ and holds only a subset of variables $x_h \in Q$.

When an application process $ap_i$ performs a write operation $w(x_1)v$, it invokes the $\text{Multicast}(m)$ to deliver the message $m$ containing $w(x_1)v$ to all sites replicating $x_1$.

When an application process $ap_i$ performs a read operation $r(x_2)v$, it invokes the $\text{RemoteFetch}(m)$ to deliver the message $m$ containing $r(x_2)v$ to a pre-designated site replicating $x_2$ to fetch its value.
System Model

Events Generated at Each Site

- **Send event.** Multicast($m$) by $ap_i$ generates event $send_i(m)$.
- **Fetch event.** RemoteFetch($m$) by $ap_i$ generates event $fetch_i(m)$.
- **Message receipt event.** The receipt of a message $m$ at site $s_i$ generates event $receipt_i(m)$.
- **Apply event.** Applying the value written by $w_j(x_h)v$ to $x_h$’s local replica at $ap_i$, an event $apply_i(w_j(x_h)v)$ is generated.
- **Remote return event.** After the occurrence of $receipt_i(m)$ corresponding to the remote $r_j(x_h)u$ performed by $ap_j$, an event $remote\_return_i(r_j(x_h)u)$ is generated to transmit $x_h$’s value $u$ to site $s_j$.
- **Return event.** Event $return_i(x_h, v)$ corresponding to the return of $x_h$’s value $v$ either through a previous $fetch_i(f)$ event or read from the local replica.
Baldoni et al. [2] defined a new relation, $\rightarrow_{co}$, on send events.

Let $w(x)a$ and $w(y)b$ be two write operations in $O_H$. For their corresponding send events, $send_i(m_w(x)a) \rightarrow_{co} send_j(m_w(y)b)$ 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. $\exists send_k(m_{w(z)}c)$, s.t. $send_i(m_w(x)a) \rightarrow_{co} send_k(m_w(z)c) \rightarrow_{co} send_j(m_w(y)b)$

$\rightarrow_{co} \subseteq \rightarrow$ (Lamport’s happened before relation)

With the $\rightarrow_{co}$ relation, an optimal activation predicate is shown:

$$A_{OPT}(m_w, e) \equiv \exists m'_w : (send_j(m'_w) \rightarrow_{co} send_k(m_w) \land apply_i(w') \notin E_i | e)$$

(1)

It is optimal because the moment this $A_{OPT}(m_w, e)$ becomes true is the earliest instant that the update $m_w$ can be applied.
Two algorithms implement causal consistency in a partially replicated distributed shared memory system.

- **Full-Track**
- **Opt-Track** (a message and space optimal algorithm)

Adopt the optimal activation predicate $A_{OPT}$

A special case of **Opt-Track** - for full replication.

- **Opt-Track-CRP** (optimal) : a lower message size, time, space complexity than the Baldoni et al. algorithm [2]
Algorithm 1: Full-Track

- Algorithm 1 is for a **non-fully replicated** system.
- Each application process performing write operation will only write to a subset of all the sites.
- Each site $s_i$ needs to track the number of write operations performed by every $ap_j$ to every site $s_k$, denoted as $Write_{i}[j][k]$.
- The `Write` clock piggybacked with messages generated by the `Multicast(m)` should not be merged with the local `Write` clock at the message reception, but only at a later read operation reading the value that comes with the message.
  - optimal in terms of the activation predicate
Algorithm 1: Full-Track

Data structures

1. $Write_i$ - the $Write$ clock
   $Write_i[j, k]$: the number of updates sent by application process $ap_j$ to site $s_k$ that causally happened before under the $\rightarrow_{co}$ relation.

2. $Apply_i$ - an array of integers
   $Apply_i[j]$: the total number of updates written by application process $ap_j$ that have been applied at site $s_i$.

3. $LastWriteOn_i\langle$variable id, $Write\rangle$ - a hash map of $Write$ clocks
   $LastWriteOn_i\langle h \rangle$: the $Write$ clock value associated with the last write operation on variable $x_h$ locally replicated at site $s_i$. 
Algorithm 1: Full-Track

```
WRITE\((x_h, v)\):
1 for all sites \(s_j\) that replicate \(x_h\) do
2 \[ Write_i[i, j] +=; \]
3 Multicast \([m(x_h, v, Write_i)]\) to all sites \(s_j\) \((j \neq i)\) that replicate \(x_h\);
4 if \(x_h\) is locally replicated then
5 \[ x_h := v; \]
6 \[ Apply_i[i] +=; \]
7 \[ LastWriteOn_i(h) := Write_i; \]

READ\((x_h)\):
8 if \(x_h\) is not locally replicated then
9 \[ RemoteFetch[f(x_h)]\) from predesignated site \(s_j\) that replicates \(x_h\) to get \(x_h\) and \(LastWriteOn_j(h)\);
10 \(\forall k, l \in [1 \ldots n], Write_i[k, l] := \max(Write_i[k, l], LastWriteOn_j(h).Write[k, l])\);
11 else
12 \(\forall k, l \in [1 \ldots n], Write_i[k, l] := \max(Write_i[k, l], LastWriteOn_i(h).Write[k, l])\);
13 return \(x_h\);
```
Algorithm 1: Full-Track

The activation predicate $A_{OPT}$ is implemented.

```plaintext
On receiving $m(x_h, v, W)$ from site $s_j$:
14    wait until
        $(\forall k \neq j, Apply_i[k] \geq W[k, i] \wedge Apply_i[j] = W[j, i] - 1)$;
15    $x_h := v$;
16    $Apply_i[j] ++$;
17    $LastWriteOn_i(h) := W$;

On receiving $f(x_h)$ from site $s_j$:
18    return $x_h$ and $LastWriteOn_i(h)$ to $s_j$;
```
Algorithm 2: Opt-Track

- Each message corresponding to a write operation piggybacks an $O(n^2)$ matrix in Algorithm 1.
- **Algorithm 2** further reduces the message size and storage cost.
  - Exploits the transitive dependency of causal deliveries of messages as given by the KS algorithm [3][4]
- Each site keeps a record of the most recently received message from each other site (along with the list of destinations of the message).
  - optimal in terms of the activation predicate
  - optimal in terms of log space and message space overhead
  - achieve another optimality that no redundant destination information is recorded.
Two Situations for Destination Information to be Redundant

Figure: $s_2$ is a destination of $M$. The causal future of the relevant message delivery events are shown in dotted lines.
Two Situations for Destination Information to be Redundant

Figure: $s_2$ is a destination of $m$. The causal future of the relevant `apply` and `return` events are shown in dotted lines.
If the Destination List Becomes $\emptyset$, then ...

**Figure**: Illustration of why it is important to keep a record even if its destination list becomes empty.
Algorithm 2: Opt-Track

Data Structures

1. \textit{clock}_i
   local counter at site \textit{s}_i for write operations performed by application process \textit{ap}_i.

2. \textit{Apply}_i - an array of integers
   \textit{Apply}_i[j] : the total number of updates written by application process \textit{ap}_j that have been applied at site \textit{s}_i.

3. \textit{LOG}_i = \{\langle j, \text{clock}_j, \text{Dest}_s \rangle \} - the local log
   Each entry indicates a write operation in the causal past.

4. \textit{LastWriteOn}_i \langle \text{variable id, } \text{LOG} \rangle - a hash map of \textit{LOGs}
   \textit{LastWriteOn}_i \langle h \rangle : the piggybacked \textit{LOG} from the most recent update applied at site \textit{s}_i for locally replicated variable \textit{x}_h.
Algorithm 2: Opt-Track

```plaintext
WRITE(x_h, v):
1   clock_i ++;
2   for all sites s_j (j ≠ i) that replicate x_h do
3       L_w := LOG_i;
4       for all o ∈ L_w do
5           if s_j ∉ o.Dests then
6               o.Dests := o.Dests \ x_h.replicas;
7           else o.Dests := o.Dests \ x_h.replicas ∪ {s_j};
8       for all o_{z, clock_z} ∈ L_w do
9           if o_{z, clock_z}.Dest = ∅ ∧ (∃ o'_{z, clock'_z} ∈ L_w | clock_z <
10             clock'_z) then remove o_{z, clock_z} from L_w;
11       send m(x_h, v, i, clock_i, x_h.replicas, L_w) to site s_j;
12   for all l ∈ LOG_i do
13       l.Dests := l.Dests \ x_h.replicas;
14   PURGE;
15   LOG_i := LOG_i ∪ {{i, clock_i, x_h.replicas \ {s_i}}};
16   if x_h is locally replicated then
17       x_h := v;
18       Apply_i[i] ++;
19   LastWriteOn_i(\langle h \rangle) := LOG_i;
```

Figure: Write process at site s_i.
Algorithm 2: Opt-Track

```
READ(x_h):
18 if x_h is not locally replicated then
19   RemoteFetch[f(x_h)] from predesignated site s_j that
20   replicates x_h to get x_h and LastWriteOn_j{h};
21 else MERGE(LOG_i, LastWriteOn_j{h});
22 PURGE;
23 return x_h;

On receiving m(x_h, v, j, clock_j, x_h.replicas, L_w) from site s_j:
24 for all o_z, clock_z ∈ L_w do
25   if s_i ∈ o_z, clock_z.Dests then wait until
26     clock_z ≤ Apply_i[z];
27 x_h := v;
28 Apply_i[j] := clock_j;
29 L_w := L_w ∪ {(j, clock_j, x_h.replicas)};
30 for all o_z, clock_z ∈ L_w do
31   o_z, clock_z.Dests := o_z, clock_z.Dests \ {s_i};
32 LastWriteOn_i{h} := L_w;
33 On receiving f(x_h) from site s_j:
34 return x_h and LastWriteOn_i{h} to s_j;
```

Figure: Read, receiving processes at site s_i
Procedures used in Opt-Track

**PURGE:**
1. for all \( l_{z,t} \in LOG_i \) do
2. if \( l_{z,t} \neq \emptyset \land (\exists l_{z,t'} \in LOG_i \rightarrow t < t') \) then
   3. remove \( l_{z,t} \) from \( LOG_i \);

**MERGE(\( LOG_i, L_w \)):**
4. for all \( o_{z,t} \in L_w \) and \( l_{s,t'} \in LOG_i \) such that \( s = z \) do
5. if \( t < t' \land l_{s,t} \notin LOG_i \) then mark \( o_{z,t} \) for deletion;
6. if \( t' < t \land o_{z,t'} \notin L_w \) then mark \( l_{s,t'} \) for deletion;
7. delete marked entries;
8. if \( t = t' \)
9. \( l_{s,t'} \cdot Dests := l_{s,t'} \cdot Dests \cap o_{z,t} \cdot Dests; \)
10. delete \( o_{z,t} \) from \( L_w \);
11. \( LOG_i := LOG_i \cup L_w \);

Figure: PURGE and MERGE functions at site \( s_i \)
Algorithm 3: Opt-Track-CRP

- Special case of Algorithm 2 Opt-Track for full replication.
- Same optimizations as for Algorithm Opt-Track.
- Since in the full replication case, every write operation will be sent to exactly the same set of sites, there is no need to keep a list of the destination information with each write operation.
- Each time a write operation is sent, all the write operations it piggybacks as its dependency will share the same set of destinations as the one being sent, thus their destination list will be pruned.
- When a write operation is received, all the write operations it piggybacks also have the receiver as part of their destination.
- We represent each individual write operation using only a 2-tuple \( \langle i, clock_i \rangle \) at site \( s_i \).
- the cost of a write operation from \( O(n) \) down to \( O(1) \).
Further Improved Scalability

- In Algorithm 2, keeping entries with empty destination list is important.
- In the fully replicated case, we can also decrease this cost.

Figure: In fully replicated systems, the local log will be reset after each write operation.
Algorithm 3: Opt-Track-CRP

```plaintext
WRITE\( (x_h, v) \):
1 \( \text{clock} \_i \) ++;
2 send \( m(x_h, v, i, \text{clock}_i, LOG_i) \) to all sites other than \( s_i \);
3 \( \text{LOG}_i := \{(i, \text{clock}_i)\} \);
4 \( x_h := v \);
5 \( \text{Apply}_i[i] := \text{clock}_i \);
6 \( \text{LastWriteOn}_i(h) := \{(i, \text{clock}_i)\} \)

READ\( (x_h) \):
7 \( \text{MERGE}(\text{LOG}_i, \text{LastWriteOn}_i(h)) \);
8 return \( x_h \);

On receiving \( m(x_h, v, j, \text{clock}_j, L_{w}) \) from site \( s_j \):
9 for all \( o_{z, \text{clock}_z} \in L_{w} \) do
10 \( \text{wait until } \text{clock}_z \leq \text{Apply}_i[z] \)
11 \( x_h := v \);
12 \( \text{Apply}_i[j] := \text{clock}_j \);
13 \( \text{LastWriteOn}_i(h) := (j, \text{clock}_j) \)
14 \( \text{MERGE}(\text{LOG}_i, (j, \text{clock}_j)) \):
15 for all \( l_{s, t} \in \text{LOG}_i \) such that \( s = j \) do
16 \( \text{if } t < \text{clock}_j \text{ then delete } l_{s, t} \text{ from } \text{LOG}_i ; \)
17 \( \text{LOG}_i := \text{LOG}_i \cup \{(j, \text{clock}_j)\} \);
```

Figure: There is no need to maintain the destination list for each write operation in the local log.
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 where 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
- \( d \): the number of write operations stored in local log (used only in Opt-Track-CRP algorithm)
## Complexity

<table>
<thead>
<tr>
<th>Metric</th>
<th>Full-Track</th>
<th>Opt-Track</th>
<th>Opt-Track-CRP</th>
<th>OptP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message count</td>
<td>$pw + 2r\frac{(n-p)}{n}$</td>
<td>$pw + 2r\frac{(n-p)}{n}$</td>
<td>$nw$</td>
<td>$nw$</td>
</tr>
<tr>
<td>Message size</td>
<td>$O(n^2pw + nr (n-p))$</td>
<td>$O(n^2pw + nr (n-p))$</td>
<td>$O(nwd)$</td>
<td>$O(n^2w)$</td>
</tr>
<tr>
<td></td>
<td>ammorted $O(npw + r(n-p))$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time Complexity</td>
<td>write $O(n^2)$</td>
<td>write $O(n^2p)$</td>
<td>write $O(n)$</td>
<td>write $O(n)$</td>
</tr>
<tr>
<td></td>
<td>read $O(n^2)$</td>
<td>read $O(n^2)$</td>
<td>read $O(1)$</td>
<td>read $O(n)$</td>
</tr>
<tr>
<td>Space Complexity</td>
<td>$O(npq)$</td>
<td>$O(npq)$</td>
<td>$O(max(n, q))$</td>
<td>$O(nq)$</td>
</tr>
<tr>
<td></td>
<td>ammorted $O(pq)$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure:** Complexity measures of causal memory algorithms for fully-replicated memory. *OptP:* Optimal propagation-based protocol proposed by Baldoni et al. [2].
Comparisons

- Compared with $OptP$, our algorithms also adopt the optimal activation predicate $A_{OPT}$ but incur a lower cost in the message size, space, and time (for read and write operations) complexities.

- Compared with other causal consistency algorithms, our algorithms have the additional ability to implement causal consistency in partially replicated distributed shared memory systems.

- Our algorithms provide scalability without using a form of log serialization and exchange to implement causal consistency.
The Benefit of Partial Replication V.S. Full Replication

- Reduces the number of messages sent with each write operation. The overall number of messages can be lower if the replication factor is low and readers tend to read variables from the local replica instead of remote one (e.g., Hadoop HDFS and MapReduce).
- Also reduces the total size of messages transmitted with the system (Consider the size of the data that is actually being replicated). Modern social multimedia networks are such examples.
- Decreases the cost brought by full replication in the write-intensive workload.
Message Count as a Function of $w_{rate}$

- Message count is the most important metric.
- Partial replication gives a lower message count than full replication if

$$pw + 2r \frac{(n - p)}{n} < nw \Rightarrow w > 2 \frac{r}{n} \quad (2)$$

$$w_{rate} = \frac{w}{w + r} \Rightarrow w_{rate} > \frac{2}{2 + n} \quad (3)$$
Partial Replication versus Full Replication

Figure: The graph illustrates message count for partial replication vs. full replication, by plotting message count as a function of $w_{rate}$. 
Simulation Parameters of KS algorithm

- $n$: Number of processes
- MIMT - Mean intermessage time: the average period of time between two message sending events at any process
- M/T - Multicast frequency: the ratio of the number of send events at which data is multicast to more than one process ($M$) to the total number of message send events ($T$)
- MTT - Mean transmission time: the transmission time of a message usually refers to the message size/bandwidth + propagation delay.
- B/T: the fraction of send events that broadcast messages
- Baseline is $n^2$ matrix size of RST algorithm.
Average message space overhead as a function of $n$

Figure: The simulations were performed for (MTT, MIMT, M/T).
Average Message Space Overhead as a Function of $MTT$

**Figure**: The simulations were performed for $(MIMT, M/T, n)$. 
Average Message Space Overhead as a Function of MIMT

Figure: The simulations were performed for (MTT, M/T, n).
Average Message Space Overhead as a Function of $M/T$

**Figure**: The simulations were performed for $(MTT, MIMT, n)$. 
Average Message Space Overhead as a Function of $|\text{Dests}|/n$

![Message overhead for increasing $|\text{Dests}|/n$](image)

**Figure**: The simulations were performed for (MTT, MIMT, $n$).
Average Message Space Overhead as a Function of $B/T$

Figure: The simulations were performed for (MTT, MIMT, $n$).
Future Work

- For some applications where the data size is small (e.g., wall posts in Facebook), the size of the meta-data can be a problem.
  - quadratic in $n$ in the worst case, even for Algorithm Opt-Track
- Future work aims to reduce the size of the meta-data for maintaining causal consistency in partially replicated systems.
Notion of Credits

**Figure**: 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 (*credit*).
Instantiation of Credits

- Integrate the notion of credits into the Opt-Track algorithm, to give an algorithm that can fine-tune the amount of causal consistency by trading off the size of meta-data overhead.
- Give three instantiations of the notion of credits (hop count, time-to-live, and metric distance)
Conclusions

- A suite of algorithms implementing causal consistency in large-scale geo-replicated storage under **partial replication**.
- For the **partially replicated** scenario, adopted the optimal activation predicate in the sense that each update is applied at the earliest instant while removing false causality:
  - **Full-Track Algorithm**
- The second algorithm further minimizes the size of meta-information carried on messages and stored in local logs.
  - **Opt-Track Algorithm**: partially replicated scenario
  - Provides less overhead (transmission and storage) than the full replication case.
- A derived optimized algorithm of the second one reduces the **message overhead**, the **processing time**, and the **local storage cost** at each site in the fully replicated scenario.
  - **Opt-Track-CRP Algorithm**
References


