Causal Consistency Algorithms for Partially Replicated and Fully Replicated Systems

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

Outline

Motivation for Replication

- Replication – makes copies of data/services on multiple sites.
- Replication improves ...
  - Reliability (by redundancy)
    If primary File Server crashes, standby File Server still works
  - Performance
    Reduce communication delays
  - Scalability
    Prevent overloading a single server (size scalability)
    Avoid communication latencies (geographic scale)
- However, concurrent updates become complex
  - Consistency models define which interleavings of operations are valid (admissible)

Consistency Models

- Spectrum of consistency models trade-off: cost vs. convenient semantics
  - Linearizability (Herlihy and Wing 1990)
    - non-overlapping ops seen in order of occurrence + overlapping ops seen in common order
  - Sequential consistency (Lamport 1979)
    - all processes see the same interleaving of executions
  - Causal consistency (Ahamad et al. 1991)
    - all processes see the same order of causally related writes
  - PRAM consistency (Lipton and Sandberg 1988)
    - pipeline per pair of processes, for updates
  - Slow memory (Hutto et al. 1990)
    - pipeline per variable per pair of processes, for updates
  - Eventual consistency (Johnson et al. 1975)
    - updates are guaranteed to eventually propagate to all replicas

The Motivation of Causal Consistency

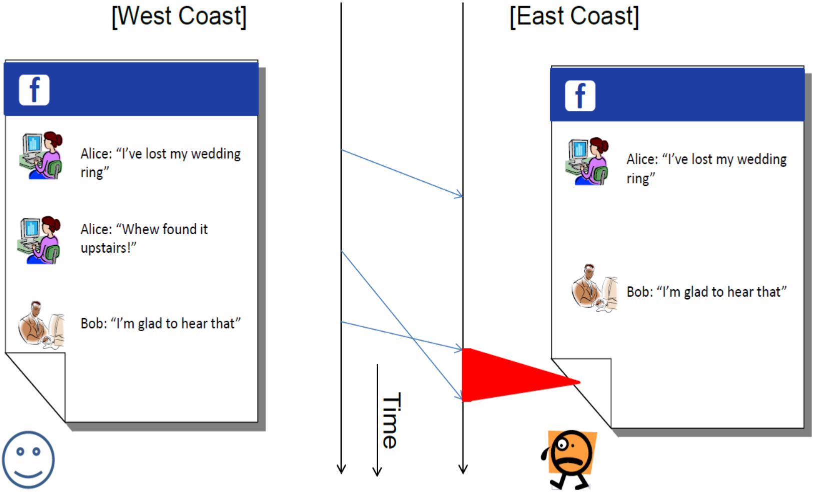


Figure: Causal consistency is good for users in social networks.

Related Works

- **Causal consistency** in distributed shared memory systems (Ahamad et al. [1])
- **Causal consistency** has been studied (by Baldoni et al., Mahajan et al., Belaramani et al., Petersen et al.).
- Since 2011,
  - **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

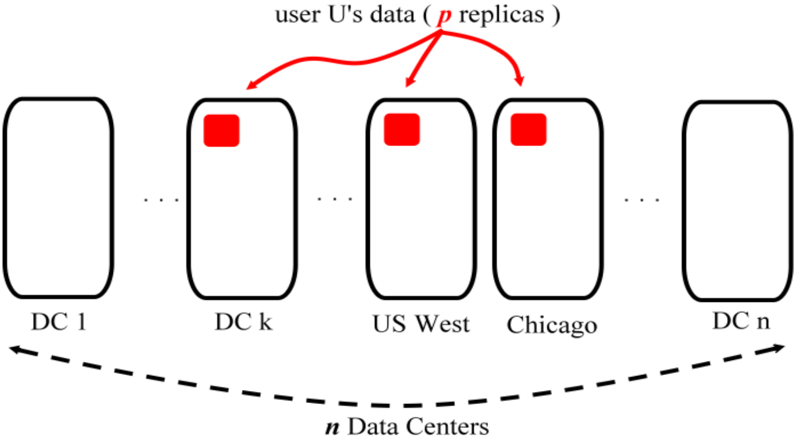


Figure: Case for Partial Replication.

Case for Partial Replication

- Partial replication is more natural for some applications. See previous fig.
  - With p replicas at some p of the total of n DCs, each write operation that would have triggered an update broadcast now becomes a multicast to just p of the n DCs.
- Savings in storage and infrastructure costs
- 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 Instagram.

Causal Consistency

- Causal consistency: writes that are potentially causally related must be seen by all processors in that same order. Concurrent writes may be seen in a different order on different machines.
  - causally related writes: the write comes after a read that returned the value of the other write
- Examples

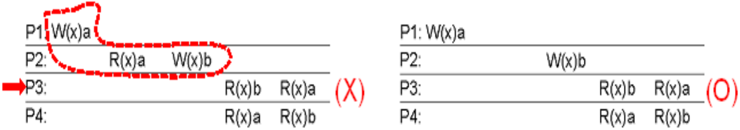


Figure: Should enforce W(x)a < W(x)b ordering?

System Model

- n application processes - ap_1, ap_2,...,ap_n - interacting through a shared memory $\mathcal{Q}$ composed of q variables x_1,x_2,...,x_q
- Each ap_i can perform either a *read* or a *write* on any of the q variables.
  - $r_i(x_j)v$: a **read** operation by ap_i on variable x_j returns value v
  - $w_i(x_j)v$: a **write** operation by ap_i on variable x_j writes value v
- 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 processes

Causality Order

- *Program Order*: a local operation o_1 precedes another local operation o_2, denoted as $o_1 \prec_{po} o_2$
- *Read-from Order*: *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 $o_1 \prec_{ro} o_2$
- *Causality Order*: $o_1 \prec_{co} o_2$ iff one of the following conditions holds:
    - $\exists ap_i$ s.t. $o_1 \prec_{po} o_2$ (program order)
    - $\exists ap_i, ap_j$ s.t. $o_1 \prec_{ro} o_2$ (read-from order)
    - $\exists o_3 \in O_H$ s.t. $o_1 \prec_{co} o_3$ and $o_3 \prec_{co} o_2$ (transitive closure)

Underlying Distributed Communication System

- 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 $x_h \in \mathcal{Q}$.
- When ap_i performs a write operation $w(x_1)v$, it invokes the **Multicast**(m) to deliver the message m containing $w(x_1)v$ to all sites replicating x_1.
    - *send* event, *receive* event, *apply* event in msg-passing layer
- When ap_i performs a read operation $r(x_2)v$, it invokes the **RemoteFetch**(m) to deliver the message m containing $r(x_2)v$ to a pre-designated site replicating x_2 to fetch its value.
    - *fetch* event, *remote_ return* event, *return* event in msg-passing layer

Activation Predicate

- Baldoni et al. [2] defined relation $\rightarrow_{co}$ on *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. $send_i(m_{w(x)a}) \rightarrow_{co} send_k(m_{w(z)c}) \rightarrow_{co} send_j(m_{w(y)b})$
- $\rightarrow_{co} \subset \rightarrow$ (Lamport's **happened before** relation)
- A **write** can be applied when its activation predicate becomes true
  - there is no earlier message (under $\rightarrow_{co}$) which has not been locally applied

Implementing the Activation Predicate

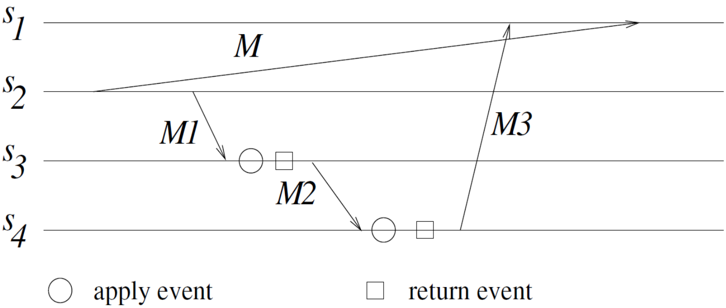


Figure: Can the **write** for $M3$ can be applied at s_1? Only after the earlier **write** for M is applied. The dependency "s_1 is a destination of M" is needed as meta-data on $M3$.

- Meta-data grows as computation progresses
- Objective: to reduce the size of meta-data

Overview of Algorithms

- Two algorithms [3, 4] implement causal consistency in a partially replicated distributed shared memory system.
  - **Full-Track**
  - **Opt-Track** (a message and space optimal algorithm)
- Implement the $\rightarrow_{co}$ relation; adopt the activation predicate
- 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.

Algorithm 1: Full-Track

- Data structures
  1. $Write_i$ - the *Write* clock
     $Write_i[j, k]$: the number of updates sent by 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 number of updates written by 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):

for *all sites* s_j *that replicate* x_h **do**
  $Write_i[i, j] + +$;
Multicast[$m(x_h, v, Write_i)$] to all sites s_j ($j \neq i$) that replicate x_h;
if x_h *is locally replicated* **then**
  $x_h := v$;
  $Apply_i[i] + +$;
  $LastWriteOn_i\langle h\rangle := Write_i$;

READ(x_h):

if x_h *is not locally replicated* **then**
  RemoteFetch[$f(x_h)$] from any site s_j that replicates x_h to get x_h and $LastWriteOn_j\langle h\rangle$;
  $\forall k, l \in [1 \ldots n], Write_i[k, l] := \max(Write_i[k, l], LastWriteOn_j\langle h\rangle.Write[k, l])$;
else
  $\forall k, l \in [1 \ldots n], Write_i[k, l] := \max(Write_i[k, l], LastWriteOn_i\langle h\rangle.Write[k, l])$;
return x_h;

Algorithm 1: Full-Track

The activation predicate is implemented.

On receiving $m(x_h, v, W)$ from site s_j:

wait until
$(\forall k \neq j, Apply_i[k] \geq W[k, i] \wedge Apply_i[j] = W[j, i] - 1)$;
$x_h := v$;
$Apply_i[j] + +$;
$LastWriteOn_i\langle h \rangle := W$;

On receiving $f(x_h)$ from site s_j:

return x_h and $LastWriteOn_i\langle h \rangle$ 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** uses propagation constraints to further reduce the **message size** and **storage cost**.
  - Exploits the transitive dependency of causal deliveries of messages (ideas from the KS algorithm [5]).
- Each site keeps a record of the most recently received message from each other site (along with the list of its destinations).
  - optimal in terms of log space and message space overhead
  - achieve another optimality that no redundant destination information is recorded.

Propagation Constraints: Two Situations for Destination Information to be Redundant

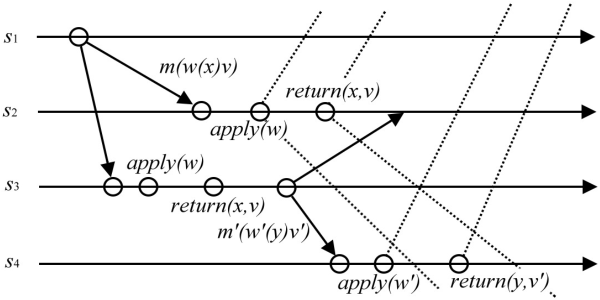


Figure: Meta-data information $\chi =$ "s_2 is a destination of m". The causal future of the relevant *apply* and *return* events are shown in dotted lines.

- χ must not exist in the causal future of
  - *apply(w)*: **(Propagation Constraint 1)**
  - *apply(w')*: **(Propagation Constraint 2)**

Algorithm 2: Opt-Track

- Data Structures
  1. $clock_i$
     local counter at site s_i for write operations performed by ap_i.
  2. $Apply_i$ - an array of integers
     $Apply_i[j]$: the number of updates written by ap_j that have been applied at site s_i.
  3. $LOG_i = \{\langle j, clock_j, Dests\rangle\}$ - the local log
     Each entry indicates a write operation in the causal past.
  4. $LastWriteOn_i\langle$variable id, $LOG\rangle$ - a hash map of $LOGs$
     $LastWriteOn_i\langle h\rangle$: the piggybacked LOG from the most recent update applied at site s_i for locally replicated variable x_h.

Algorithm 2: Opt-Track

<u>WRITE(x_h, v):</u>

$clock_i++$;
for *all sites* $s_j (j \neq i)$ *that replicate* x_h **do**
    $L_w := LOG_i$;
    for *all* $o \in L_w$ **do**
        if $s_j \notin o.Dests$ **then** $o.Dests := o.Dests \setminus x_h.replicas$;
        else $o.Dests := o.Dests \setminus x_h.replicas \cup \{s_j\}$;
    for *all* $o_{z,clock_z} \in L_w$ **do**
        if $o_{z,clock_z}.Dests = \emptyset \wedge (\exists o'_{z,clock'_z} \in L_w | clock_z < clock'_z)$
        then remove $o_{z,clock_z}$ from L_w;
    send $m(x_h, v, i, clock_i, x_h.replicas, L_w)$ to site s_j;
for *all* $l \in LOG_i$ **do**
    $l.Dests := l.Dests \setminus x_h.replicas$;
$LOG_i := LOG_i \cup \{\langle i, clock_i, x_h.replicas \setminus \{s_i\}\rangle\}$;
PURGE;
if x_h *is locally replicated* **then**
    $x_h := v$;
    $Apply_i[i]++$;
    $LastWriteOn_i\langle h\rangle := LOG_i$;

Figure: Write process at site s_i

Algorithm 2: Opt-Track

READ(x_h):

if x_h *is not locally replicated* **then**
- $\mathtt{RemoteFetch}[f(x_h)]$ from any site s_j that replicates x_h to get x_h and $LastWriteOn_j\langle h\rangle$;
- $\mathtt{MERGE}(LOG_i, LastWriteOn_j\langle h\rangle)$;

else $\mathtt{MERGE}(LOG_i, LastWriteOn_i\langle h\rangle)$;

PURGE;

return x_h;

On receiving $m(x_h, v, j, clock_j, x_h.replicas, L_w)$ from site s_j:

for *all* $o_{z,clock_z} \in L_w$ **do**
- **if** $s_i \in o_{z,clock_z}.Dests$ **then** wait until $clock_z \leq Apply_i[z]$;

$x_h := v$;

$Apply_i[j] := clock_j$;

$L_w := L_w \cup \{\langle j, clock_j, x_h.replicas\rangle\}$;

for *all* $o_{z,clock_z} \in L_w$ **do**
- $o_{z,clock_z}.Dests := o_{z,clock_z}.Dests \setminus \{s_i\}$;

$LastWriteOn_i\langle h\rangle := L_w$;

On receiving $f(x_h)$ from site s_j:

return x_h and $LastWriteOn_i\langle h\rangle$ to s_j;

Figure: Read, receiving processes at site s_i

Procedures used in Opt-Track

PURGE:

for *all* $l_{z,t_z} \in LOG_i$ **do**
- **if** $l_{z,t_z}.Dests = \emptyset \wedge (\exists l'_{z,t'_z} \in LOG_i | t_z < t'_z)$ **then**
  - remove l_{z,t_z} from LOG_i;

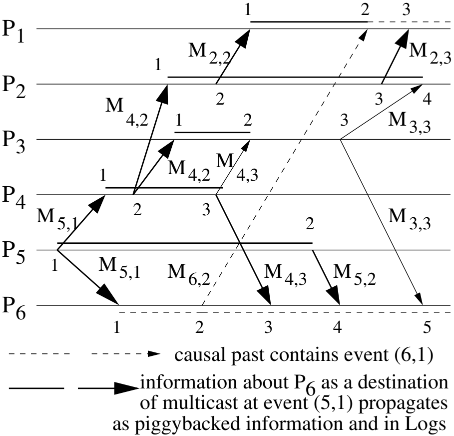
MERGE(LOG_i, L_w):

for *all* $o_{z,t} \in L_w$ *and* $l_{s,t'} \in LOG_i$ *such that* $s = z$ **do**
- **if** $t < t' \wedge l_{s,t} \notin LOG_i$ **then** mark $o_{z,t}$ for deletion;
- **if** $t' < t \wedge o_{z,t'} \notin L_w$ **then** mark $l_{s,t'}$ for deletion;
- delete marked entries;
- **if** $t = t'$ **then**
  - $l_{s,t'}.Dests := l_{s,t'}.Dests \cap o_{z,t}.Dests$;
  - delete $o_{z,t}$ from L_w;

$LOG_i := LOG_i \cup L_w$;

Figure: PURGE and MERGE functions at site s_i

The Propagation Constraints: An example



(a)

Message to dest.	piggybacked $M_{5,1}$.Dests
$M_{5,1}$ to P_4, P_6	$\{P_4, P_6\}$
$M_{4,2}$ to P_3, P_2	$\{P_6\}$
$M_{2,2}$ to P_1	$\{P_6\}$
$M_{6,2}$ to P_1	$\{P_4\}$
$M_{4,3}$ to P_6	$\{P_6\}$
$M_{4,3}$ to P_3	$\{\}$
$M_{5,2}$ to P_6	$\{P_4, P_6\}$
$M_{2,3}$ to P_1	$\{P_6\}$
$M_{3,3}$ to P_2, P_6	$\{\}$

(b)

Figure: Meta-data information χ = "P_6 is a destination of $M_{5,1}$". The propagation of explicit information and the inference of implicit information.

If the Destination List Becomes $\emptyset$, then ...

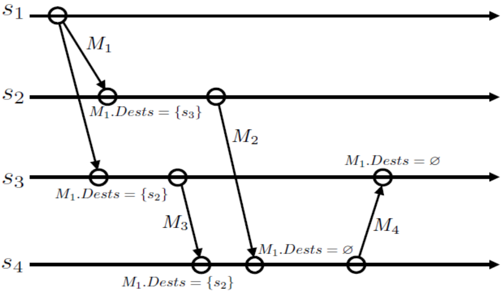


Figure: Illustration of why it is important to keep a record even if its destination list becomes empty.

Algorithm 3: Opt-Track-CRP

- Special case of Algorithm 2 for full replication; same optimizations.
- 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**.
- Represent each **write** operation as $\langle i, clock_i \rangle$ at site s_i.
- the cost of a **write** operation down from $O(n)$ to $O(1)$.
- d entries in local log, where $d =$ no. of write operations in local log
  - Local log always gets reset after each **write**
  - Each **read** will add at most one new entry into the local log

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.

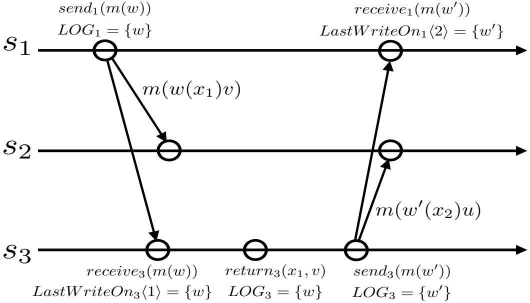


Figure: In fully replicated systems, the local log will be reset after each write.

Algorithm 3: Opt-Track-CRP

WRITE(x_h, v):

$clock_i++$;

send $m(x_h, v, i, clock_i, LOG_i)$ to all sites other than s_i;

$\boxed{LOG_i := \{\langle i, clock_i\rangle\}}$;

$x_h := v$;

$Apply_i[i] := clock_i$;

$LastWriteOn_i\langle h\rangle := \boxed{\langle i, clock_i\rangle}$;

READ(x_h):

MERGE(LOG_i, $LastWriteOn_i\langle h\rangle$);

return x_h;

On receiving $m(x_h, v, j, clock_j, L_w)$ from site s_j:

for *all* $o_{z,clock_z} \in L_w$ **do**

  wait until $clock_z \leq Apply_i[z]$;

$x_h := v$;

$Apply_i[j] := clock_j$;

$LastWriteOn_i\langle h\rangle := \boxed{\langle j, clock_j\rangle}$;

MERGE(LOG_i, $\langle j, clock_j\rangle$):

$unionflag_i := 1$;

for *all* $l_{s,t} \in LOG_i$ *such that* $s = j$ **do**

  if $t < clock_j$ **then**

    delete $l_{s,t}$ from LOG_i;

  else

    $unionflag_i := 0$;

$\boxed{\textbf{if } unionflag_i \textbf{ then } LOG_i := LOG_i \cup \{\langle j, clock_j\rangle\}}$;

Figure: There is no need to maintain the destination list for each write operation in the local log.

Parameters

- n: number of sites in the system
- q: number of variables in the system
- p: replication factor, i.e., the number of sites where each variable is replicated
- w: number of write operations performed in the system
- r: number of read operations performed in the system
- d: number of write operations stored in local log (used only in Opt-Track-CRP algorithm)

Complexity

Table: Complexity measures of causal memory algorithms.

Metric	Full-Track	Opt-Track	Opt-Track-CRP	*optP* [2]
Message Count	$((p-1)+\frac{n-p}{n})w+2r\frac{(n-p)}{n}$	$((p-1)+\frac{n-p}{n})w+2r\frac{(n-p)}{n}$	$(n-1)w$	$(n-1)w$
Message Size	$O(n^2pw+nr(n-p))$	$O(n^2pw+nr(n-p))$ amortized $O(npw+r(n-p))$	$O(nwd)$	$O(n^2w)$
Time Complexity	write $O(n^2)$ read $O(n^2)$	write $O(n^2p)$ read $O(n^2)$	write $O(d)$ read $O(d)$	write $O(n)$ read $O(n)$
Space Complexity	$O(\max(n^2, npq))$	$O(\max(n^2, npq))$ amortized $O(\max(n, pq))$	$O(\max(d, q))$	$O(nq)$

Message Count as a Function of w_{rate}

- Define write rate as $w_{rate} = \frac{w}{w+r}$
- Partial replication gives a lower message count than full replication if

$$pw + 2r\frac{(n-p)}{n} < nw \Rightarrow w > 2\frac{r}{n} \tag{1}$$

$$\Rightarrow w_{rate} > \frac{2}{1+n} \tag{2}$$

Message count: Partial Replication vs. Full Replication

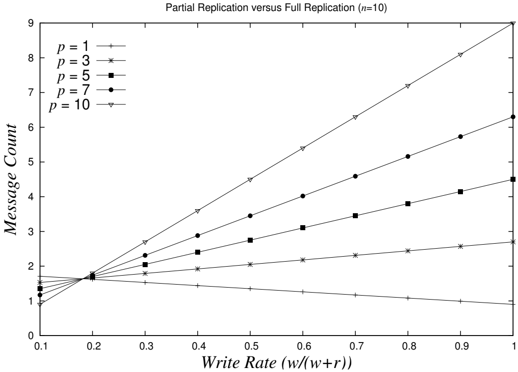


Figure: The graph illustrates message count for partial replication vs. full replication, by plotting message count as a function of w_{rate}.

Message meta-data structures

	Full-Track	Opt-Track
SM (Multicast)	$x_h, v, Write$	$x_h, v, Site_{id}, clock, L_w$
FM (Fetch)	x_h, v	x_h, v
RM (Remote return)	$v, LastWriteOn\langle h\rangle$	$v, LastWriteOn\langle h\rangle$

Figure: Partial Replication

$$m(x_h, v, Site_{id}, clock, LOG)$$

Figure: Full Replication (only SM)

Simulation Methodology

- Time interval T_e between two consecutive events from $5ms \sim 2000ms$.
- Propagation time T_t from $100ms \sim 3000ms$.
- Number of processes n is varied from 5 up to 40.
- w_{rate} is set to be 0.2, 0.5, and 0.8, respectively.
- Replica factor rate $\frac{p}{n}$ for partial replication is defined as 0.3.
- Message meta-data size (m_s): The total size of all the meta-data transmitted over all the processes.
- Each simulation execution runs $600n$ operation events totally.

Meta-Data Space Overhead in Partial Replication

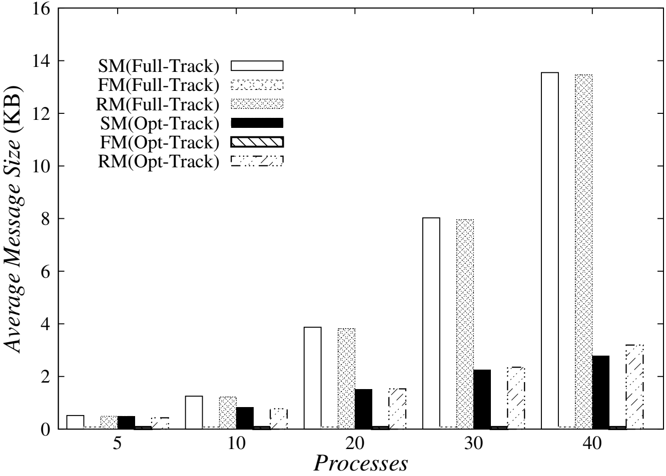


Figure: $w_{rate} = 0.2$.

Meta-Data Space Overhead in Partial Replication

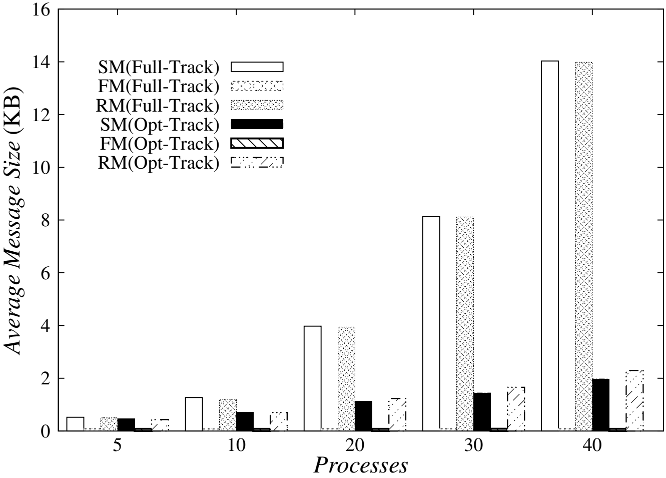


Figure: $w_{rate} = 0.5$.

Meta-Data Space Overhead in Partial Replication

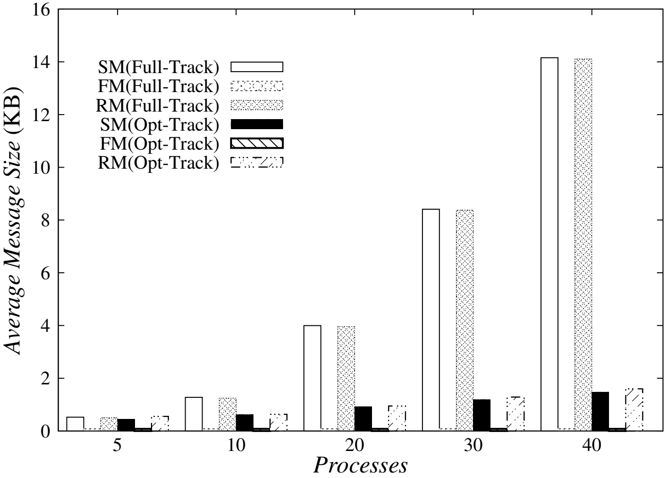


Figure: $w_{rate} = 0.8$.

Ratio of Message Overhead of Opt-Track to Full-Track

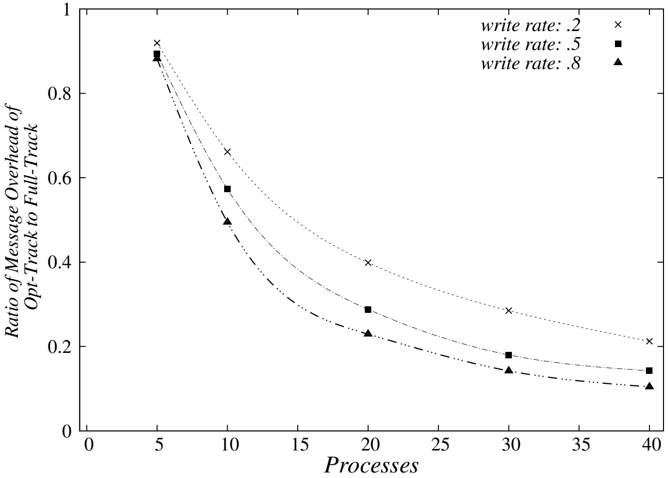


Figure: Total message meta-data space overhead as a function of n and w_{rate} in partial replication protocols.

Meta-Data Size in Partial Replication

- With increasing n, the ratio rapidly decreases. For 40 processes, the Opt-Track overheads are only around 10 $\sim$ 20 % of Full-Track overheads on different write rates.
- In Full-Track protocol, the average message space overheads of SM and RM quadratically increases with n. However, the increasingly lower overhead of SM and RM in Opt-Track are linear in n.

Meta-Data Space Overhead in Full Replication

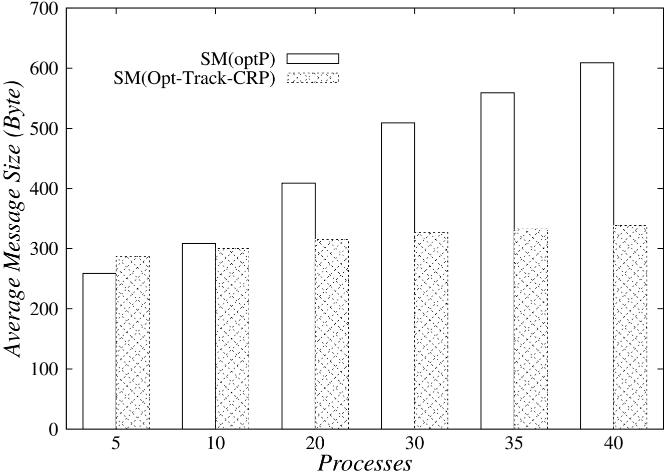


Figure: $w_{rate} = 0.2$.

Meta-Data Space Overhead in Full Replication

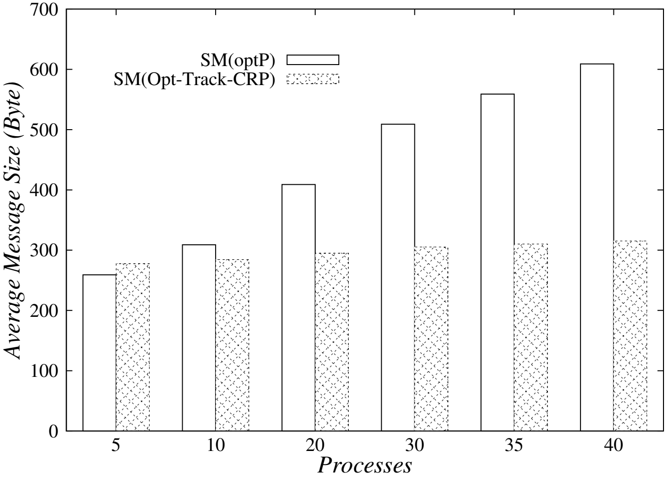


Figure: $w_{rate} = 0.5$.

Meta-Data Space Overhead in full Replication

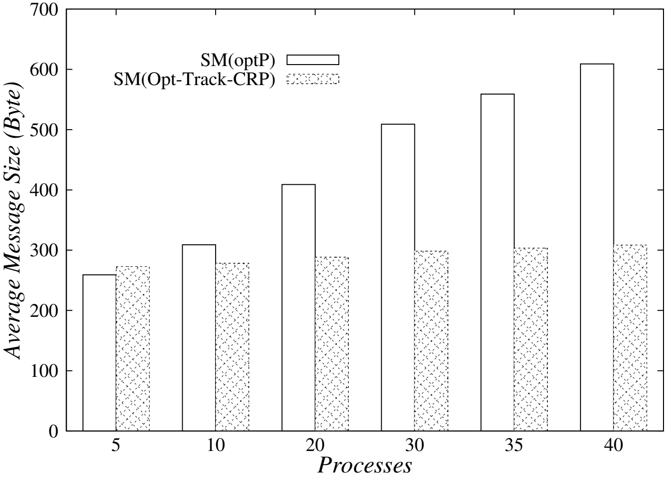


Figure: $w_{rate} = 0.8$.

Ratio of Message Overhead of Opt-Track-CRP to optP

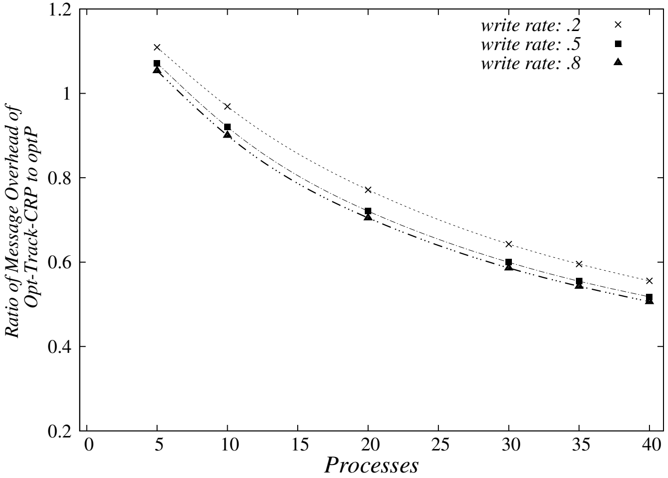


Figure: Total message meta-data space overhead as a function of n and w_{rate} in full replication protocols.

Message Count: Partial Replication vs. Full Replication

- Total message size overhead
  $=$ Total message count $*$ (meta-data size $+$ replicated data size)

✓ meta-data size $\ll$ replicated data size

n	Full replication			Partial replication		
	(0.2)	(0.5)	(0.8)	(0.2)	(0.5)	(0.8)
5	2036	4960	8004	3208	3463	3764
10	8910	22,266	35,892	8297	10,234	12,156
20	38,057	95,114	151,905	22,808	35,668	48,128
30	86,826	217,181	347,304	42,600	75,679	108,810
40	156,156	390,039	624,390	69,405	130,572	192,883

Figure: Total message count for Full Replication (Opt-Track-CRP) VS. Partial Replication (Opt-Track).

Message Size: Partial Replication vs. Full Replication

	$f = 100$ KB	$f = 10$ KB	$f = 1$ KB	$f = 0.1$ KB
Full replication	3900 KB + 6.09 KB	390 KB + 6.09 KB	39 KB + 6.09 KB	3.9 KB + 6.09 KB
Partial replication	1240 KB + 77.5 KB	124 KB + 77.5 KB	12.4 KB + 77.5 KB	1.24 KB + 77.5 KB

(a) when $n = 40$, $p = 12$, $w_{rate} = 0.5$, in the worst case.

	$f = 100$ KB	$f = 10$ KB	$f = 1$ KB	$f = 0.1$ KB
Full replication	3900 KB+0.152d KB	390 KB+0.152d KB	39 KB+0.152d KB	3.9 KB+0.152d KB
Partial replication	1240 KB + 1.94 KB	124 KB + 1.94 KB	12.4 KB + 1.94 KB	1.24 KB + 1.94 KB

(b) when $n = 40$, $p = 12$, $w_{rate} = 0.5$, in the real case.

Figure: Total message size for Full Replication (Opt-Track-CRP) and Partial Replication (Opt-Track).

Approximate Causal Consistency

- 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.
- Can further reduce meta-data overheads at the risk of some (rare) violations of causal consistency.
- As dependencies age, w.h.p. the messages they represent get delivered and the dependencies need not be carried around and stored.
- Amount of violations can be made arbitrarily small by controlling a parameter called *credits*.

Notion of Credits: Case 1

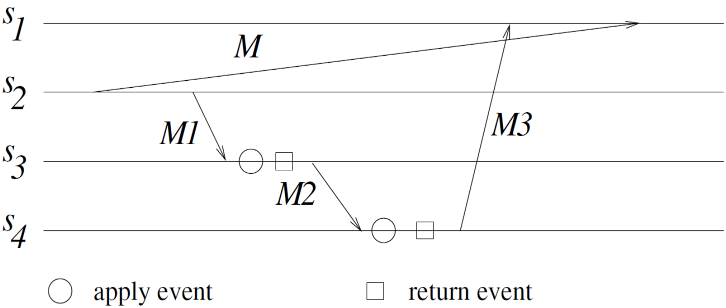


Figure: Reduce the meta-data at the cost of some possible violations of causal consistency. The amount of violations can be made arbitrarily small by controlling a tunable parameter (*credits*).

Notion of Credits: Case 2

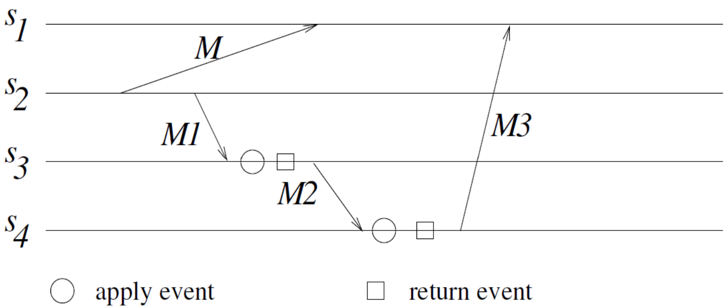


Figure: Illustration of meta-data reduction when credits are exhausted.

Approx-Opt-Track

- Integrate the notion of credits into the **Opt-Track** algorithm, to give an algorithm – **Approx-Opt-Track** [6] – that can fine-tune the amount of causal consistency by trading off the size of meta-data overhead.
- Give three instantiations of credits (*hop count*, *time-to-live*, and *metric distance*)
- Violation Error Rate: $R_e = \frac{n_e}{m_c}$
  - n_e: number of messages applied by the remote replicated sites with violation of causal consistency
  - m_c: total number of transmitted messages
- Meta-Data Saving Rate: R_s = 1- $\frac{m_s(cr \neq \infty)}{m_s(Opt-Track)}$
  - m_s: message meta-data size, the total size of all the meta-data transmitted over all the processes

Simulation Methodology in Approx-Opt-Track

- Time interval T_e between two consecutive events from $5ms \sim 2000ms$.
- Propagation time T_t $100ms \sim 3000ms$.
- Number of processes n is varied from 5 up to 40.
- w_{rate} is set to 0.2, 0.5, and 0.8, respectively.
- Replica factor rate $\frac{p}{n}$ for partial replication defined as 0.3.
- Number of variables q used is 100.
- ($\star$) Hop Count Credit (cr): denotes the hop count available before the entry meta-data ages out and is removed.
- ($\star$) Initial credit cr is specified from one to a critical value cr_0, with which there is no message transmission violating causal consistency.
- Each simulation execution runs $600n$ operation events totally.

Violation Error Rates

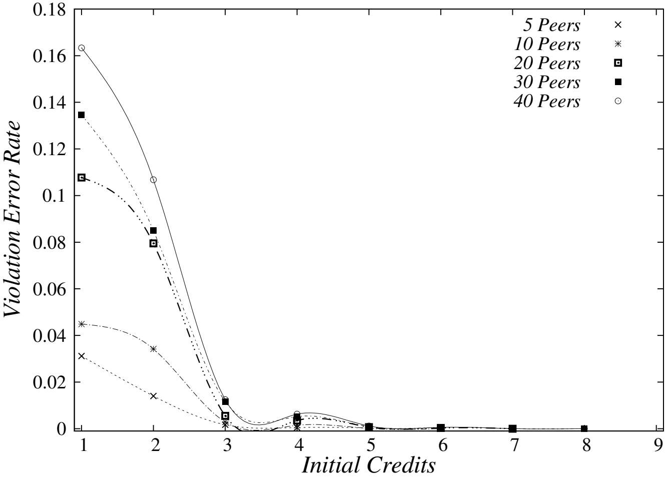


Figure: $w_{rate} = 0.2$.

Violation Error Rates

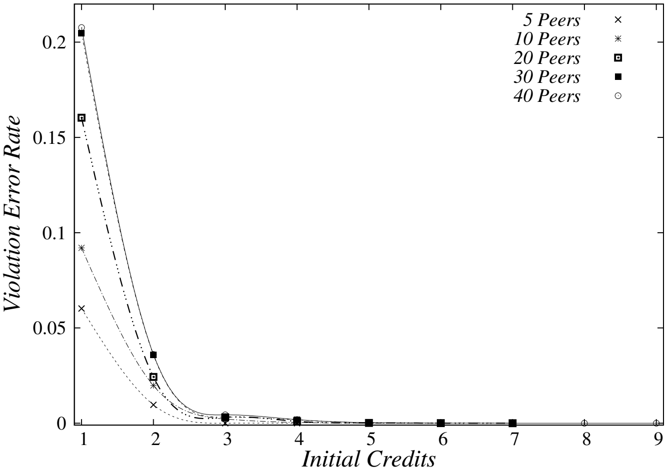


Figure: $w_{rate} = 0.5$.

Violation Error Rates

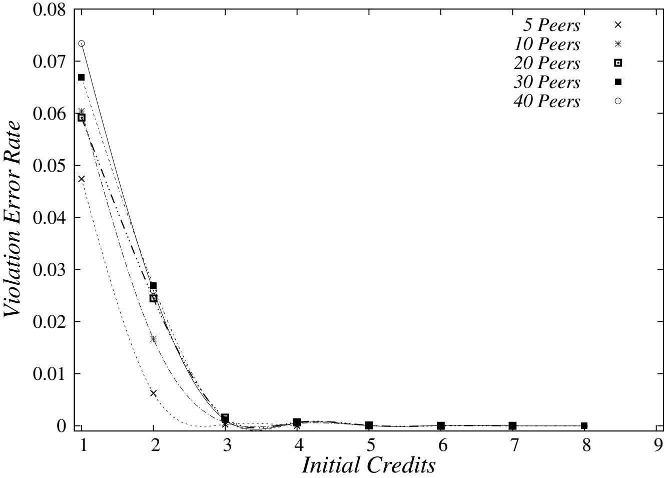


Figure: $w_{rate} = 0.8$.

Critical Initial Credits

Table : Critical Initial Credits.

	w_{rate}	the number of processes				
		5	10	20	30	40
$R_e \sim 0.5\%$	0.2	3	3	3	4	4
	0.5	3	3	3	3	3
	0.8	3	3	4	4	4
$R_e = 0$	0.2	5	6	7	8	8
	0.5	3	5	7	7	9
	0.8	4	5	7	8	8

Figure: Summary of the critical values of cr_0 and $cr_{\sim 0.5\%}$.

Impact of initial cr on R_e

- cr_0 : major critical initial credit ($R_e = 0$)
- $cr_{\sim 0.5\%}$: minor critical initial credit ($R_e \sim 0.5\%$).
- $cr_{\sim 0.5\%}$ seems not to highly increase as n.
- By setting the initial cr to a small finite value but enough, most of the dependencies will become aged and can be removed without violating causal consistency after the associated meta-data is transmitted across a few hops (even for a large number of processes.)
- The correlation coefficients of cr_0 and n are around $0.94 \sim 0.95$. The major critical credit values increase as n to avoid causal violations.

Average Meta-Data Size (KB)

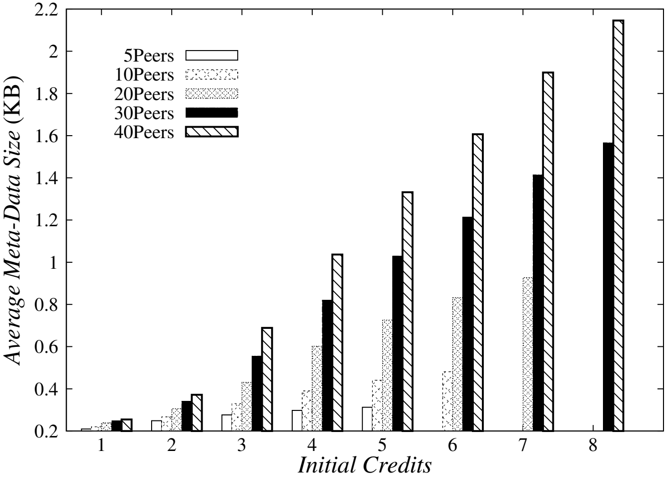


Figure: $w_{rate} = 0.2$.

Average Meta-Data Size (KB)

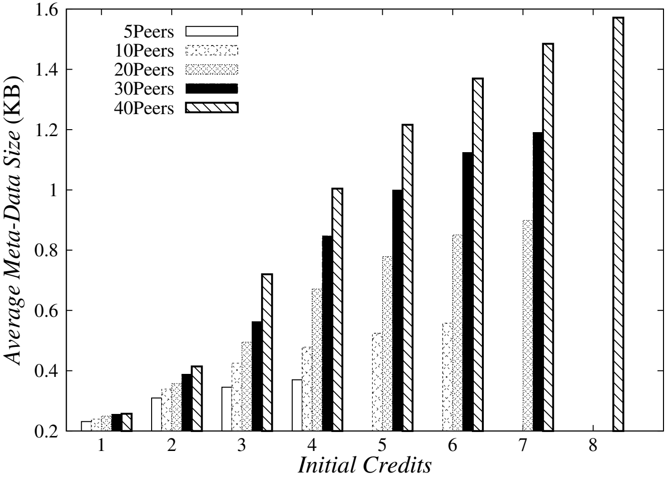


Figure: $w_{rate} = 0.5$.

Average Meta-Data Size (KB)

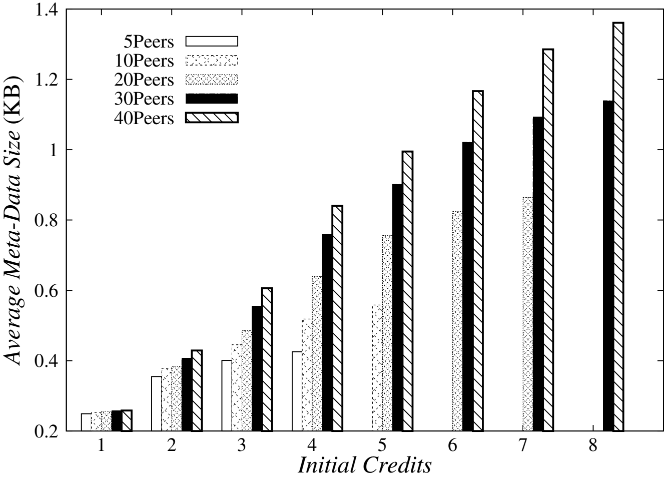


Figure: $w_{rate} = 0.8$.

Critical Average Message Meta-Data Size m_{ave} (KB)

Table : Critical Average Message Meta-Data Size m_{ave} (KB).

R_e	w_{rate}	the number of processes				
		5	10	20	30	40
∼0.5%	0.2	0.277	0.330	0.430	0.820	1.037
	0.5	0.345	0.425	0.495	0.562	0.720
	0.8	0.401	0.445	0.640	0.759	0.840
0	0.2	0.312	0.481	0.927	1.566	2.146
	0.5	0.345	0.524	0.899	1.190	1.572
	0.8	0.426	0.558	0.864	1.140	1.361

Figure: Summary of the critical average message meta-data sizes.

Critical Average Message Meta-Data Size

- With increasing number of processes, m_{ave} linearly increases.
- m_{ave} becomes smaller with a higher w_{rate} for more processes.
  - A read operation will invoke a MERGE function to merge the piggybacked log list. So, a higher read rate may increase the likelihood to make the log size enlarged.
  - Furthermore, although a write operation results in the increase of explicit information, it comes with a PURGE function to delete the redundant information.

Message Meta-Data Saving Rate

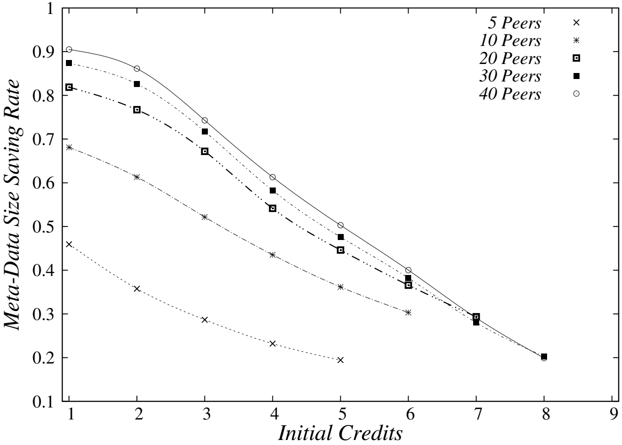


Figure: $w_{rate} = 0.2$.

Message Meta-Data Saving Rate

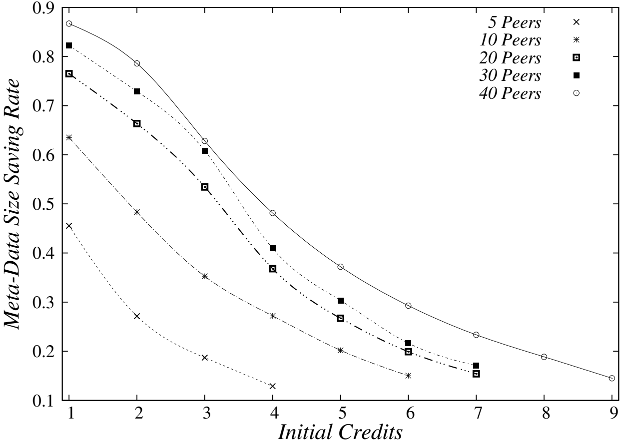


Figure: $w_{rate} = 0.5$.

Message Meta-Data Saving Rate

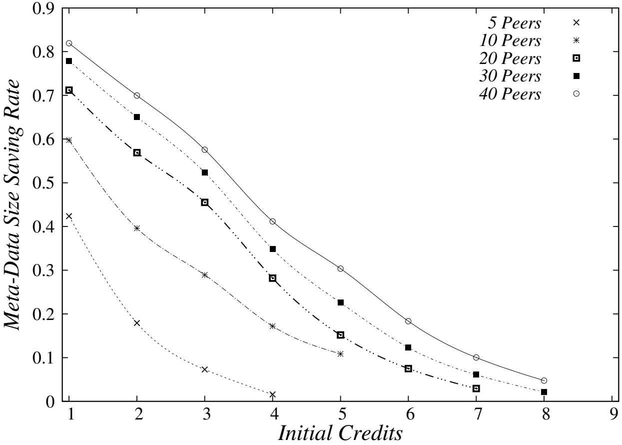


Figure: $w_{rate} = 0.8$.

Critical Message Meta-Data Size Saving Rates R_s

Table : Message Meta-Data Size Saving Rates R_s in R_e close to or equal to zero.

R_e	w_{rate}	the number of processes				
		5	10	20	30	40
∼0.5%	0.2	0.287	0.521	0.672	0.582	0.613
	0.5	0.187	0.352	0.534	0.608	0.628
	0.8	0.073	0.289	0.282	0.348	0.412
0	0.2	0.194	0.303	0.294	0.203	0.198
	0.5	0.187	0.202	0.154	0.171	0.145
	0.8	0.016	0.108	0.029	0.021	0.047

Figure: Summary of the critical message meta-data size saving rates.

Critical Message Meta-Data Size Saving Rate

- Focus on the case of 40 processes:
  - R_s is around 40% $\sim$ 60% at a very slight cost of violating causal consistency.
  - R_s reaches around 5% $\sim$ 20% without violating causality order in different write rates.
- This evidence proves that if the initial credit allocation is just a small digit, when the corresponding meta-data is removed, the associated message would already (very likely) have reached its destination.

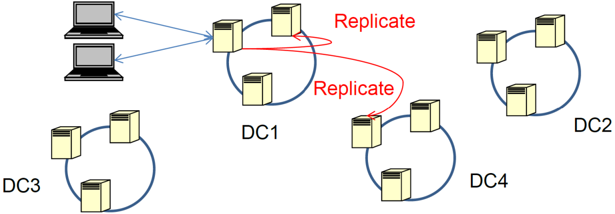
Conclusions

- **Opt-Track** has a better network capacity utilization and better scalability than **Full-Track** for causal consistency in partial replication.
  - The meta-data overhead of **Opt-Track** is linear in n.
  - These improvements increase in higher write-intensive workloads.
  - For the case of 40 processes, the overheads of Opt-Track are only around $10 \sim 20$ % of those of Full-Track for different write rates.
- **Opt-Track-CRP** can perform better than **optP** (Baldoni et. al 2006) in full replication.
  - For 40 processes, the overheads of **Opt-Track-CRP** are only around $50 \sim 55$ % of those of **optP** for different write rates.
- Showed the advantages of partial replication and provided the conditions under which partial replication can provide less overhead than full replication.
- Modification of **Opt-Track**, called **Approx-Opt-Track**, to provide approximate causal consistency by reducing the size of the meta-data.
  - By controlling a parameter cr, we can trade-off the level of potential inaccuracy by the size of meta-data.

Future Work

- Reduce the size of the meta-data for maintaining causal consistency in partially replicated systems.
  - Dynamic and Data-driven Replication Mechanism (Optimize the replication mechanism).
    - Which file?
    - How many replicas?
    - Where?
    - When?

      (i.e., the replica factor rate $\frac{p}{n}$ is a variable.)
  - Hierarchical Causal Consistency Protocol in Partial Replication.
    - A client-cluster model (two-level architecture) for causal consistency under partial replication.



References I

M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto.
Causal memory: Definitions, implementation and programming.
Distributed Computing, 9(1):37–49, 1994.

R. Baldoni, A. Milani, and S. Tucci-piergiovanni.
Optimal propagation based protocols implementing causal memories.
Distributed Computing, 18(6):461–474, 2006.

Min Shen, Ajay D. Kshemkalyani, and Ta Yuan Hsu.
OPCAM: optimal algorithms implementing causal memories in shared memory systems.
In *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, January 4-7, 2015*, pages 16:1–16:4.

References II

Min Shen, Ajay D. Kshemkalyani, and Ta Yuan Hsu.
Causal consistency for geo-replicated cloud storage under partial replication.
In *IPDPS Workshops*, pages 509–518. IEEE, 2015.

A. Kshemkalyani and M. Singhal.
Necessary and sufficient conditions on information for causal message ordering and their optimal implementation.
Distributed Computing, 11(2):91–111, April 1998.

Ta Yuan Hsu and Ajay D. Kshemkalyani.
Performance of approximate causal consistency for partially replicated systems.
In *Workshop on Adaptive Resource Management and Scheduling for Cloud Computing (ARMS-CC)*, pages 7–13. ACM, 2016.

Questions

Thank You!