CaDRoP: Cost Optimized Convergent Causal Consistency in Social Network Systems

Ta-Yuan Hsu
University of Illinois at Chicago
thsu4@uic.edu

Ajay D. Kshemkalyani
University of Illinois at Chicago
ajay@uic.edu

***Abstract*—Asynchronous geo-replication for data resources is used to provide high availability and lower latency in modern cloud store systems. Convergent causal consistency is the cornerstone to provide useful semantics for online human interaction services. Compared to full replication, partial replication has potential benefit of lower message counts in social network systems. However, static replication is ineffective for time-varying workloads. We propose a causal+ consistency protocol, CaDRoP, to support dynamic replication, and ensure the convergence property for all comments following a post and the causal ordering between posts with explicit causality. We evaluate CaDRoP protocol with realistic workloads by different PUT rates in terms of the practical price of Amazon AWS. The results show that CaDRoP incurs much lower cost than the statically replicated data store in another causal+ algorithm. We further evaluate CaDRoP by comparing it with a clairvoyant optimal replication solution. The findings indicate that with cache, CaDRoP incurs only around 6% $\sim$ 16% extra cost. Without cache, CaDRoP brings around 2% $\sim$ 4.5% extra cost in steady states.**

***Index Terms*—causal consistency, partial replication, social networks**

I. Introduction

Modern cloud storage services are hugely popular and increasingly used by businesses and enterprises to manage their data, including mission-driven services such as database queries or resource usage [1]–[5]. Data geo-replication is a critical component of these services and a widely adopted technique to improve the availability and performance for massive scale. It is the process of maintaining copies of data at geographically dispersed stores closer to the users. Thus, the latency between end-users and the store servers can be effectively reduced, in addition to offering improvements in system scalability.

In data replication strategies, partial replication is an effective measure to avoid propagating unnecessary resources to improve storage utilization and reduce network transmission costs. Data objects only replicate to a subset of the system store nodes and their updates are propagated to fewer replicas with respect to full replication. Thus, this allows replicas of different data objects to handle independent parts of the workload.

Replication brings about the problem of data consistency across different replicas. While linearizability is the strongest consistency and the most desirable property from users' perspective, several known cloud store services are satisfied with weaker consistency models to provide lower latency [5]–[8]. Causal consistency (CC) has gained significant attention as an attractive consistency for geo-replicated cloud storage systems [5]–[7], [9]–[20], since it supports the ordering of operations with respect to program and read-from order across data store nodes. Furthermore, it not only avoids the unpredictable execution status allowed by weaker consistency (e.g., eventual consistency), but provides lower latency than strong consistency models, such as linearizability. CC preserves intuitive causal ascription, crucial in social networks (e.g., privacy policies). It improves user experience because, with it, events appear to each user in the correct order. For example, this stream of comments under a landscape image: (c1) "My parents have been living there for 20 years." (c2) "It's too long." Without CC, the temporal coherence degrades if only (c2) below the image shows up on someone else's screen.

Moreover, reputed cloud storage providers, such as Amazon Web Service (AWS), offer a variety of storage classes and charge customers for use of their storage and network resources in different prices. The diversity of the storage and network prices reflects the performance appraisal like availability, utilization, etc. Thus, the monetary cost optimization on cloud-based storage services is a critical factor for application providers.

Contributions:. This paper presents CaDRoP (Causal Consistency under Dynamic Replication Protocol) [21], a new cost-optimized protocol that ensures causal+ consistency (CC+) in a partially geo-replicated platform. CC+ protocol requires that data replicas converge to the same state under concurrent updates. Existing approaches [5], [9]–[19] maintain CC+ in standard key-value storage configuration. Most of them are based on full replication, whereas some CC+ protocols [11], [16], [18], [19] support partial replication. There are some limitations when applying the current CC+ protocols to social media platforms. When users have access to a post (e.g., an image), all the replying comments return. Each comment corresponds to an update operation to a post. The existing CC+ protocols treat the post and its following comments as values to a variable. However, none of these CC+ approaches can achieve the convergence property for the values corresponding to the same post. Since they use the last-writer-wins reconciliation [22], only the value from the latest writer is kept around. Moreover, the current CC+ protocols rely on static underlying replication (i.e., data replication placement is predetermined). However, static replication of

978-1-7281-9586-5/21/$31.00 ©2021 IEEE
DOI 10.1109/CCGrid51090.2021.00052

data resources in dynamic environments with time-varying workloads is ineffective for cost management. CaDRoP is the first protocol to achieve CC+ for all replying comments (update operations) corresponding to an object (a post) with a unique key or for different objects with explicit happens-before relationships in social applications based on a key-values store system. Users from different replica stores can observe the same global causal ordering of all the text replies to a post. CaDRoP is adapted to dynamic data replication. CaDRoP also integrates CC+ across storage layer replicas and caches to reduce network transmission costs.

We conduct an evaluation of the cost-effectiveness of the CaDRoP algorithm via trace-driven CloudSim simulator toolkit and realistic workload traces from Twitter in terms of the prices set on AWS as of 2019. Results show that the total system cost can be highly reduced by CaDRoP in a dynamic replication strategy [23] in comparison to the same protocol without caches and CoCaCo [16] in different static replication models. We further evaluate CaDRoP by comparing it with a clairvoyant optimal replication solution. The findings indicate that with cache, CaDRoP incurs only around $6\% \sim 16\%$ extra cost. Without cache, CaDRoP brings around $2\% \sim 4.5\%$ extra cost in steady states.

This paper is organized as follows. Section II gives the design model of CaDRoP. Section III describes our proposed approach along with the details of CaDRoP algorithm. Section IV reports the simulation experiments along with the cost effectiveness evaluation of our approach. It also evaluates CaDRoP with respect to the same algorithm run in the clairvoyant optimal placement strategy and illustrates the trade-off between them. Section V summarizes our work.

II. Definitions and System Model

A. Causal consistency (CC)

A CC system requires that clients observe the results returned from the data repository servers, consistent with the causality order. Causality is the happen-before relationship between two events [24], [25]. The two events must be visible to all clients in the same order, when they are causally related. In other words, when users in client A observe that event M1 happens before M2, other users in client B can perceive that the effects of M1 occurring are visible to M2. Otherwise, a (potential) causality violation has occurred. When a series of access operations occur on a single thread, they are serialized as a local history h. The set of local histories from all threads forms the global history H. For potential causality [24], if there are two operations o_1 and o_2 in O_H, we say that o_2 causally depends on o_1, denoted as $o_1 \prec_{co} o_2$, if and only if one of the following conditions holds:

1) o_1 precedes another local operation o_2 in a single thread of execution (program order).
2) o_1 is a $write$ operation and o_2 is a $read$ operation that returns a value written by o_1, even if o_1 and o_2 are performed at distinct threads (read-from order).
3) there is some other operation o_3 in O_H such that $o_1 \prec_{co} o_3$ and $o_3 \prec_{co} o_2$ (transitive closure).

Especially, the causality order defines a strict partial order on the set of operations O_H. For a CC system, all the write operations that can be related by the potential causality have to be observed by each thread in the order defined by the causality order.

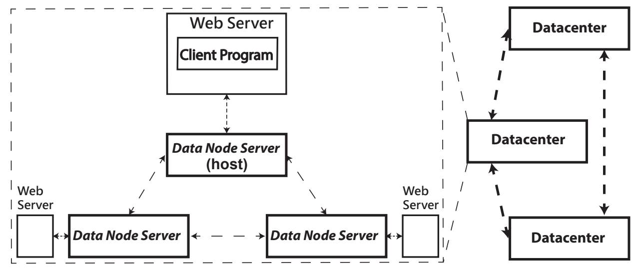

Fig. 1. The system architecture.

B. System Design

CaDRoP runs in a distributed key-[values] data store that manages a set of data objects. Thus, CaDRoP implements a multiversion data store in social networks. [values] is a list of values corresponding to an item key. In our system, one post, such as a picture on Instagram, is viewed as an object and is assigned a global unique number as the item key. The post object is saved in the head of [values], denoted as v_0. Afterwards, when a comment (e.g., a list of strings) is posted out under a post, this comment text, denoted as v_i ($i > 0$; i is the index of [values]), will be inserted to [values]. CaDRoP treats the value of each update operation as an immutable version of the access object. When users request access to a data object, [values] (i.e., a list of version values) is the result returned. Each entry in [values] corresponds to one update operation. In order to track causality, each version value needs to be associated with some metadata. [values] is also a causal list. For example, consider two entries v_i and v_j in [values] and $i < j$. Assume that v_i and v_j are created by update operations o_a and o_b, respectively. CaDRoP can guarantee that $o_b \not\prec_{co} o_a$. Although the potential causality allows to prevent any causal anomalies, it leads to higher costs to maintain many dependencies among different posts without any semantic coherency in social networks. For example, there is a cute dog photo posted in the morning and a blue sky image uploaded at noon. Tracking explicit causal order offers a more flexible solution. Under explicit causality, each application can have its own happens-before relationships between operations [26]. Because it tracks only customized relevant dependencies, explicit causality decreases the number of dependencies per modification and lowers metadata overhead. We have modeled a hybrid causality based on a column-based model. Our system maintains two types of columns:

- key columns: they are used to store data item keys.
- value columns: each value column contains a [values] corresponding to a data item key.

CaDRoP supports the explicit causality in key columns and implements the potential causality for each value column. Explicit causality can be captured through application user interface. For example, user Bob can click @ symbol on Facebook to post an image content to reply a post done by user Alice before. Thus, the client program can capture the causal dependency between the two posts, even if they are realized by different users. Otherwise, the causal relationship between different object keys will be ignored in CaDRoP.

The whole framework is a hierarchical geo-distributed cloud store system composed of multiple geographical DCs (see Fig. 1). All the DCs are fully connected by WANs with higher network access cost. They are deployed and dispersed across the world. In each DC, there are multiple web servers, each of which serves the data access demands from one geographical region and connects to its own data node server, which is called the host server of that connected web server. Data can be replicated asynchronously between different data servers within the same DC or in different DCs. When a data server s_r stores an object with key k, s_r is called a $replica$ server of object o_k. Otherwise, s_r is a $non\text{-}replica$ server. When a DC_r includes at least one replica server of object o_k, DC_r is called a $replica\ DC$ of object o_k. Otherwise, DC_r is a $non\text{-}replica\ DC$. CaDRoP supports partial replication of data. Each data object is replicated in a subset of DCs.

CaDRoP consists of the client layer and the data store layer. They communicate with each other through the client library. The client layer implemented in web servers is responsible for storing or retrieving information to or from data node servers and presenting information to the application users. Note that the client layer has to wait for the corresponding response to the current request before sending the next access request. The underlying store layer controls the physical storage in data store servers and the data propagation between them. CaDRoP provides the following operations to the clients:

- POST(key, object): A POST operation assigns an object item o_k (e.g., a picture or a clip) with an item key.
- PUT(key, value): A PUT operation assigns a text value (string) to an item key. Then, a new version value will be created. Note that if an object is visible to clients, the corresponding key always exists, unless the data object of an item key is removed from the whole system.
- $[\,\text{values}] \leftarrow$ GET(key): The GET operation returns [values] corresponding to an item key in causality order.

C. Convergent conflict handling

CC does not establish a global order for operations in O_H. Therefore, there exist some causally independent operations, which are characterized as concurrent. Formally, two operations o_1 and o_2 in O_H are concurrent if $o_1 \not\prec_{co} o_2$ and $o_2 \not\prec_{co} o_1$. Concurrent write operations applied to the same data object very likely lead to inconsistent data states. Those are said to be in "$conflict$". Essentially, conflicts do not result in causal violation. However, when different concurrent versions of a data object are replicated to remote stores, this potentially leads to divergent undesired results to clients. Multiple concurrent versions of an object could be present in the system at the same time. In this work, CaDRoP uses the timestamp and the local data node identification to order the list of version values. This can achieve a global consistent state for different data replica nodes. Thus, CaDRoP can provide causal consistency with the convergence property.

TABLE I
Definition of symbols and parameters used in the model.

Term	Meaning
s_i	The data node server i
DC	A $datacenter$ including multiple data node servers
dm_c	Dependency meta-data $depm$ set at client c
o_k	An object with a unique key k
$cvl\langle k\rangle$	A causal version list of data object k (o_k)
$IOset$	The invisible object set
TS	the local Lamport timestamp for update operations
d	An item tuple $\langle\ k,\ v,\ dm\ \rangle$
$Dests$	A set of replica store servers
VV_i	The version vector of data node s_i
$V(k)$	The size of data object k
Δt	Time slot interval
t_h	The h-th time slot
$GN_{t_h}[k][i]$	Number of $Gets$ for o_k from s_i in time slot t_h
$PN_{t_h}[k][i]$	Number of $Puts$ for o_k from s_i in time slot t_h
$AN_{t_h}[k][i]$	The sum of $GN_{t_h}[k][i]$ and $PN_{t_h}[k][i]$

III. Algorithm

CaDRoP is adapted from Opt-Track protocol [20], [27], which aims at reducing the dependency metadata size and storage cost for causal ordering in a partially replicated shared memory system. Though Opt-Track achieves CC with non-full replication across geo-distributed servers, it does not support DC-level partial replication and storage cache. We now give the formal CaDRoP algorithm in Algorithms $1 \sim 5$. CaDRoP is designed to achieve CC+ within and across DCs.

A. The client layer

The client library maintains for its session a dependency metadata, denoted as dm_c. dm_c consists of a set of $\langle rid, TS, Dests\rangle$ tuples, each of which indicates an update operation (POST or PUT) initiated by data node server rid at clock time TS in the causal past. $Dests$ includes replica data node servers for that update operation. Only necessary replica node information is stored.

When PUT() or POST() is invoked, the client library retrieves the local dm_c and assigns POSTREQ or PUTREQ attribute to propagate a new object or a new value with dm_c to its host data node server. The host server is in charge of distributing requests to other replica node servers, handling responses from others, and returning feedback to the client. Although PUT and POST operations are very similar in the client layer, their corresponding functions in the storage layer are different. POST needs to implement CC for different objects, whereas PUT needs to enforce CC+ for the comments to an object. When GET() is invoked, the client library assigns GETREQ attribute to propagate an access request to its host data node. Function MERGE() in Algorithms 1 and 4 merges the piggybacked dependency metadata of the corresponding updates to an object key with the local client dm_c. Function PURGE() in Algorithms 1 and 4 removes old records with empty $Dests$, based on Implicit Tracking in Opt-Track protocol [20], [27]. In this function, some new additional

Algorithm 1: Client operations at client c_i

POST($object\_key\ k, object\ o_k, dep\ dm_c$):
1 send $\langle$POSTREQ $k, o_k, dm_c\rangle$ to host data server s_i;
2 receive $\langle$POSTREPLY $dmr\rangle$;
3 $dm_c \leftarrow dmr$;
4 insert k into object name list;

PUT($object\_key\ k, text\ v, dep\ dm_c$):
5 send $\langle$PUTREQ $k, v, dm_c\rangle$ to data server s_i;
6 receive $\langle$PUTREPLY $dmr\rangle$;
7 $dm_c \leftarrow dmr$;

GET($object\_name\ k$):
8 send $\langle$GETREQ $k\rangle$ to host data server s_i;
9 receive $\langle$GETREPLY $cvl\langle k\rangle\rangle$;
10 **for** *each* $d \in cvl\langle k\rangle$ **do**
11   MERGE($DM_c, d.dm_d$);
12 $DM_c \leftarrow$ PURGE(DM_c);
13 return $cvl\langle k\rangle.values$;

Upon receive $f(k)$**:**
14 insert k into the object booking table;

dependencies get added to dm_c and some old existing dependencies in dm_c are deleted. The merging process implements the optimality techniques in terms of Implicit Tracking in Opt-Track protocol and makes the client aware of the necessary causal dependency information of update operations. When the client receives $f(k)$, it updates the object booking table to make users aware of what posts exist in a social network.

B. The storage layer

The data storage layer is composed of multiple data node servers. Each data object can be replicated to one or more data node servers. As mentioned before, the CaDRoP data store layer exposes three main functions to the client library:

- $\langle$POSTREPLY $dmr\rangle \leftarrow \langle$POSTREQ $k, o_k, dm_c\rangle$.
- $\langle$PUTREPLY $dmr\rangle \leftarrow \langle$PUTREQ k, v, $dm_c\rangle$
- $\langle$GETREPLY $cvl\langle k\rangle\rangle \leftarrow \langle$GETREQ $k\rangle$

Note that dm denotes a dependency meatadata set and dmr indicates a returned dm. In Algorithm 2, for a POSTREQ operation in a host data node server, it needs to update the local Lamport timestamp TS (line 1). Then, the metadata per data node server is tailored by REDUCE() in line 3 to minimize its space overhead. It is denoted as dm_s. If s_j is a replica node server, four elements (dm_s, the set of replicas, TS, o_k) are encapsulated into a package d. Line 4 propagates d to each other replica server s_j. If s_j is not a replica server, o_k is replaced with the key id k. The four elements are encapsulated into a package f. Line 5 propagates f to each non-replica server.

Lines 12-13 prune the $Dests$ information, based on the propagation condition in Opt-Track protocol. In the PURGE(), entries with empty $Dests$ are kept as long as they are the most recent update from the source node server. In CaDRoP, we assume that the host server for the client initiating a post o_k is always a replica of object o_k. Lines 14-16 store the source server (rid) and the timestamp(TS) of a POST operation as an entry (denoted as $dm(h)$) at the head of dm and create a data element d to save o_k and the associated metadata dm. Then, d is inserted to the head of $cvl\langle k\rangle$. Line 18 updates the version vector for the host server s_i. Line 19 updates the booking information for object o_k.

Social network systems have access to data objects with much larger space overheads. Thus, CaDRoP implements a relay mechanism to reduce the data communication cost across different DCs. If there are multiple replica servers in a remote datacenter DC_x, lines 7-9 will select a relay replica server s_r and propagate a package d to s_r. Once s_r receives d, it invokes REDUCE() to modify the dm_c from the source data node and then relays an updated d to other data nodes servers in the same DC_x. If there is no replica server in a remote datacenter DC_y, lines 10-11 implements the similar process as lines 5,8-9 to propagate a package f to a non-replica server s_{nr}.

Lines 37-48 handle the process, when d for a POST operation is received by a replica server. The determination *ATP*() realizes an activation predicate of a safe protocol to stop the visibility of any update operation that arrives out of order with respect to $\prec_{co}$. Lines 49-55 deal with the process, when f for a POST operation is received by a non-replica server. After receiving d in a replica server, a copy of the object posted and the replica placement list ($replicas$) are stored. When receiving f in a non-replica server, it only needs to save $replicas$. Line 50 is required to maintain an explicit dependency between two POST operations.

The function for a PUTREQ operation is similar to that for a POST operation. Instead of replicating an object, PUTREQ propagates a text value in the package d. Note that when a data node server implementing function PUTREQ is a non-replica server, d would not be saved.

In Algorithm 3, lines 1-17 run in the case when a replica server in a remote DC_x receives a package d for a POST operation. Lines 18-28 handle the process when a non-replica server in a remote DC_y receives a package f for a POST operation. Lines 29-37 or 38-50 deal with the case when a replica server within the same DC or in a remote DC_x receives a package d for a PUT operation. When a data server s_i receives a d (for an object o_k or a text value v) or a f (for an object notification), *ATP*() is used to check if the d or f is visible to clients. If the received item is not visible, it will be temporarily stored in $IOset$ until the *ATP*() test becomes true.

For a GETREQ operation to a key k in s_i, if s_i is a replica server of object o_k, $cvl\langle k\rangle$ returns to the client. If s_i is a non-replica server of object o_k, s_i needs to fetch $cvl\langle k\rangle$ from a replica server. However, if $cvl\langle k\rangle$ is fetched from a different data server, CaDRoP uses *ATP*() to check if each value is causally visible. `LINK()` is used to insert a d with an updated value into $cvl\langle k\rangle$ in causality order, when an update value is visible. Since $cvl\langle k\rangle$ is a causal list of values, the following condition must be satisfied:

$$\forall d' \in cvl\langle k\rangle : (d'.dm(h).rid, d'.dm(h).TS) \neq (d.dm(h).rid, d.dm(h).TS) \quad (1)$$

However, some entries in $cvl\langle k\rangle$ are concurrent with d. CaDRoP can sort those concurrent entries by their TS and rid, in ascending order. Thus, the text values of $cvl\langle k\rangle$

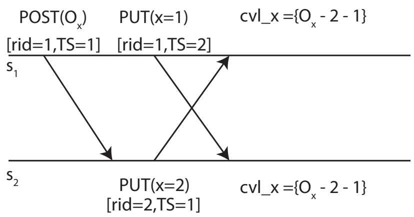

Fig. 2. An example with the convergence property.

saved in different data servers can be present in the same convergent order. As shown in Figure 2, when two users retrieve the $cvl\langle k\rangle$ from s_1 and s_2, respectively, they can obtain a consistent result in causality order.

C. Dynamic Replication Model

The space overheads of data objects in geo-replicated storage systems are composed of the size of dependency metadata (dm) and that of payload data (V). In social network systems, V is substantially larger than dm [20]. Therefore, the replication model in cloud-based data store systems plays a vital role in the cost optimization. Most of the existing CC protocols are based on static replication models in geo-replicated data stores. In other words, the numbers of replicas for a variety of data objects are predetermined. All replication decisions are made before the system is operational and replica configuration is not changed during operation. However, static replication of data resources in dynamic environments hosting time-varying workloads is obviously ineffective for optimizing system utilization, especially in social network systems. Dynamic replication strategies have been widely used as means of increasing the data availability of large-scale cloud store systems. **CORP** model, a proactive dynamic data replication strategy, has been proposed in [23] to effectively improve the total system cost in a social network system. According to the current data resource allocation and historical changes in workload patterns, **CORP** employs the autoregressive integrated moving average (ARIMA) model to predict data object access frequency in the near future. In order to optimize system cost, we incorporate **CORP** model as the underlying replication mechanism into CaDRoP protocol. Based on the requirement of **CORP**, a time slot system is required to realize the data migration process in CaDRoP. Each data server is equipped with a physical clock, which generates monotonically increasing timestamps. Physical clocks are synchronized by a time synchronization protocol, such as NTP. The correctness of the CaDRoP is independent of the synchronization precision.

CORP strategy runs at the end of each time slot and outputs a set of replicas for each data object. Then, the home server for that object triggers the migration process, based on the replica placement at the current time slot and that at the next time slot. It is noted that the regular CORP runs the ARIMA prediction model by an equal time interval. At runtime, the prediction is constantly updated. When new access requests arrive in the current time slot, they are getting involved into the time series and the information in the oldest time slot is removed from the time series. However, when a data object is created, there is not sufficient data in the time series initially (i.e., the training data set is not enough). Therefore, CaDRoP adopts cache mechanism, based on a PUSH model, to reduce the

Algorithm 2: Operations at data node s_i in DC_i (part1)

Upon receive$\langle$**POSTREQ** o_k**,** $dm_c\rangle$
$TS \leftarrow$ LamportTimestamp.increaseAndGet();
for *each data node* s_j *in the local* DC_i **do**
  $dm_s \leftarrow$ REDUCE($dm_c.clone,\ L,\ s_j$);
  if $s_j \in k.replicas$ **then**
    send $d(o_k, k.replicas, TS, dm_s)$ to s_j;
  else send $f(k, k.replicas, TS, dm_s)$ to s_j;
for *each* $DC_j \neq DC_i$ **do**
  if DC_j *is a replica DC of* o_k **then**
    select a replica server s_r in DC_j;
    send $d(o_k, k.replicas, TS, dm_c)$ to s_r;
  else
    select a data node s_{nr} in DC_j;
    send $f(k, k.replicas, TS, dm_c)$ to s_{nr};
for *each* $o \in dm_c$ **do**
  $o.Dests := \backslash L$;
$dm_c := \cup\{\langle rid = s_i, TS, L\backslash\{s_i\}\rangle\}$;
$dm \leftarrow$ PURGE(dm_c);
create $d(o_k, dm)$ and $cvl\langle k\rangle$;
LINK($cvl\langle k\rangle$,d) : insert d to $cvl\langle k\rangle$;
$VV_i[i]$.increment;
OBJTABLEUPDATE(k,$replicas$);
return $\langle$POSTREPLY $dmr = dm\rangle$ to the request client;

Upon receive$\langle$**PUTREQ** k**,** v**,** $dm_c\rangle$
$TS \leftarrow$ LamportTimestamp.increaseAndGet();
for *each* $s_j \in k.replicas$*, in the local* DC_i **do**
  $dm_s \leftarrow$ REDUCE($dm_c.clone,\ o_k.replicas,\ s_j$);
  send $d(k = v, rid = s_i, TS, dm_s)$ to s_j;
for *each replica* $DC_j \neq DC_i$ **do**
  select a replica server s_r in DC_j;
  send $d(k = v, rid = s_i, TS, dm_c)$ to s_r;
for *each* $o \in dm_c$ **do**
  $o.Dests := \backslash k.replicas$;
$dm_c := \cup\{\langle rid = s_i, TS, k.replicas\backslash\{s_i\}\rangle\}$;
$dm \leftarrow$ PURGE(dm_c);
if $s_i \in o_k.replicas$ **then**
  create $d(k = v, dm)$;
  LINK($cvl\langle k\rangle$,d) : insert d to $cvl\langle k\rangle$;
  $VV_i[i]$.increment;
return $\langle$PUTREPLY $dmr = dm\rangle$ to the request client;

Upon receive $d(o_k, replicas, TS, dm_s)$ **from** DC_i
$rid \leftarrow replicas.getFirst()$;
if ATP(dm_s,VV_i,s_i)=**true then**
  $dm_s := \cup\{\langle rid, TS, replicas\rangle\}$;
  for *each* $o \in dm_s$ **do**
    $o.Dests := \backslash s_i$;
  create $d'(o_k, dm_s)$ and $cvl\langle k\rangle$;
  insert d' to $cvl\langle k\rangle$;
  $VV_i[rid] \leftarrow TS$;
  update $IOset$;
  send $f(k)$ to the local client c_i;
else insert d into $IOset$;
OBJTABLEUPDATE(k,$replicas$);

Upon receive $f(k, replicas, TS, dm_s)$ **from** DC_i
$rid \leftarrow replicas.getFirst()$;
if ATP(dm_s, VV_i, s_i)=*true* **then**
  $VV_i[rid] \leftarrow TS$;
  update $IOset$;
  send $f(k)$ to the local client c_i;
else insert d into $IOset$;
OBJTABLEUPDATE(k,$replicas$);

Algorithm 3: Operations at data node s_i in DC_i (part2)

Upon receive $d(o_k, replicas, TS, dm_c)$ **from** $DCs \neq DC_i$

for *each data node* $s_j (\neq s_i)$ *in* DC_i **do**
  $dm_s \leftarrow$ REDUCE($dm_c.clone,\ replicas,\ s_j$);
  if s_j *is a replica node of* o_k **then** send $d(o_k, k.replicas, TS, dm_s)$ to s_j;
  else send $f(k, k.replicas, TS, dm_s)$ to s_j;
REDUCE($dm_c,\ replicas,\ s_i$);
$rid \leftarrow replicas.getFirst()$;
if *ATP*(dm_c,VV_i,s_i)=**true then**
  $dm_c := \cup\{\langle rid, TS, replicas\rangle\}$;
  for *each* $o \in dm_c$ **do**
    $o.Dests := \setminus s_i$;
  create $d'(o_k, dm_c)$ and $cvl\langle k\rangle$;
  insert d' to $cvl\langle k\rangle$;
  $VV_i[rid] \leftarrow TS$;
  update $IOset$;
  send $f(k)$ to the local client c_i;
else insert d into $IOset$;
OBJTABLEUPDATE(k,$replicas$);

Upon receive $f(k, replicas, TS, dm_c)$ **from** $DCs \neq DC_i$

for *each data node* $s_j (\neq s_i)$ *in* DC_i **do**
  $dm_s \leftarrow$ REDUCE($dm_c.clone,\ replicas,\ s_j$);
  send $f(k = v, replicas$,TS$, dm_s)$ to s_j;
REDUCE($dm_c,\ L,\ s_i$);
$rid \leftarrow replicas.getFirst()$;
if *ATP*(dm_c, VV_i, s_i)=**true then**
  $VV_i[rid] \leftarrow TS$;
  update $IOset$;
  send $f(k)$ to the local client c_i;
else insert d into invisible object list;
OBJTABLEUPDATE(k,$replicas$);

Upon receive $d(k = v, rid = s_j, TS, dm_s)$ **from** DC_i

if *ATP*(dm_s, VV_i, s_i)=**true then**
  $dm_s := \cup\{\langle rid, TS, replicas\rangle\}$;
  for *each* $o \in dm_s$ **do**
    $o.Dests := \setminus s_i$;
  create $d'(k = v, dm_s)$;
  insert d' to $cvl\langle k\rangle$;
  $VV_i[rid] \leftarrow TS$;
  update $IOset$;
else insert d into $IOset$;

Upon receive $d(k = v, rid, TS, dm_c)$ **from** $DCs \neq DC_i$

for *each replica data node* $s_j (\neq s_i)$ *in* DC_i **do**
  $dm_s \leftarrow$REDUCE($dm_c.clone,\ replicas,\ s_j$);
  send $d(o_k, k.replicas, TS, dm_s)$ to s_j;
REDUCE($dm_c,\ replicas,\ s_i$);
if *ATP*(dm_c, s_i)=**true then**
  $dm_c := \cup\{\langle rid, TS, replicas\rangle\}$;
  for *each* $o \in dm_c$ **do**
    $o.Dests := \setminus s_i$;
  create $d'(k = v, dm_c)$;
  insert d' to $cvl\langle k\rangle$;
  $VV_i[rid] \leftarrow TS$;
  update $IOset$;
else insert d into $IOset$;

Upon receive⟨GETREQ k**⟩**

if $s_i \notin k.replicas$ **then**
  send⟨REQUEST k⟩ to a replica node in DC_i or a remote DC;
  receive ⟨RREQ $cvl\langle k\rangle$⟩;
  for *each* $d \in cvl\langle k\rangle$ **do**
    if *(*$ATP(dm_d, VV_i, s_i)$=**false***)* **then**
      remove d from $cvl\langle k\rangle$;
else fetch $cvl\langle k\rangle$;
send⟨GETREPLY $cvl\langle k\rangle$⟩;

Upon receive⟨REQUEST k**⟩**

fetch $cvl\langle k\rangle$ and return ⟨PREQ $cvl\langle k\rangle$⟩;

Algorithm 4: Functions used in Algorithm 1, 2, and 3

boolean ATP($depm\ dm, int[]\ VV_{s_i}, node\ s_i$):

for *each* $o \in dm$ **do**
  if $s_i \in o_{z,ts}.Dests$ **then**
    if $ts > VV_i[z]$ **then** return false;
return true;

REDUCE($depm\ dm, node\_list\ replicas, node\ s_n$):

for *each* $o \in dm$ **do**
  if $s_n \in o.Dests$ **then** $o.Dests = \setminus replicas$;
  else $o.Dests := \setminus replicas \cup s_n$;
  if $o_z.Dests = \emptyset \wedge (\exists o'_z \in dm | o_z.ts < o'_z.ts)$ **then** $dm \setminus o_z$;
return dm;

PURGE(dm):

for *each* $o \in dm$ **do**
  if $o_z.Dests = \emptyset \wedge (\exists o'_z \in dm | o_z.ts < o'_z.ts)$ **then** $dm \setminus o_z$;
return dm_c;

MERGE(dm_c, dm_d):

for *all* $o_{z,tz} \in dm_d$ *and* $o_{s,ts} \in dm_c$ *and* $s = z$ **do**
  if $tz < ts \wedge o_{s,tz} \notin dm_c$ **then** mark $o_{z,tz}$;
  if $ts < tz \wedge o_{z,ts} \notin dm_d$ **then** mark $o_{s,ts}$;
  delete marked entries;
  if $tz = ts$ **then**
    $o_{s,ts}.Dests := \cap o_{z,ts}.Dests$;
    delete $o_{z,t}$ from dm_d;
$dm_c := dm_c \cup dm_d$;

OBJTABLEUPDATE($object\_id\ k, node\_list\ replicas$):

ObjectTable$\langle k\rangle = replicas$;

Algorithm 5: Cache operations at data server s_i

Upon receive⟨PUTREQ k**,** v**,** dm_c**⟩ in a replica master node**

1 **for** *each slave caching node* s_a *of object* o_k **do**
2   fetch seq by object key k and node id s_a;
3   seq.increase();
4   send ⟨CACHE $d(k = v, seq, dm_r)$⟩ to s_a;

Upon receive⟨CACHE $d(k = v, seq, dm_r)$**⟩ in a cache node**

5 **wait until** ($d.seq$ = $k.seq + 1$);
6 $k.seq$.increase();
7 insert $d(k = v, dm_r)$ to $cvl\langle k\rangle$;

Upon receive⟨REQUEST k**⟩ from a non-replica node** s_j

8 insert $\langle s_j$,seq=0$\rangle$ to a cache seq map for object key k;
9 fetch $cvl\langle k\rangle$ and return ⟨PREQ $cvl\langle k\rangle$⟩;

Upon receive⟨GETREQ k**⟩ in a non-replica node**

10 send⟨REQUEST k⟩ to a replica node in DC_i or a remote DC;
11 receive ⟨RREQ $cvl\langle k\rangle$⟩;
12 **for** *each* $d \in cvl\langle k\rangle$ **do**
13   **if** $ATP(dm_d, VV_i, s_i)$=**false then**
14   move d from $cvl\langle k\rangle$ to invisible list of object k;
15 save $cvl\langle k\rangle$ in s_i and set $k.seq$ to '0';
16 send⟨GETREPLY $cvl\langle k\rangle$⟩;

network transmission cost, especially in the initial time slot(s). Algorithm 5 presents the cache functions used in CaDRoP. When a non-replica s_i receives a requesting data package with key k by fetching cvl from another replica server s_r, $cvl\langle k\rangle$ may be cached in s_i (line 16) with a sequence number seq assigned by s_r. For object o_k, s_i becomes a slave server of s_r. Afterwards, whenever s_r receives an update value (lines 1-4), s_r relays the update value to s_i with a seq (increasing by one per PUT). Based on the seq, s_i can maintain a visible $cvl\langle k\rangle$ in causality order. Algorithm 6 presents the migration processes in CaDRoP. When the migration process initiates, **CORP** outputs a new set of replicas of a key k (denoted as $k.replicas'$) for the next time slot t_h to the home server s_i. Based on different replica distributions, s_i will send the $replicas'$ (lines 2-7) or replicate $cvl\langle k\rangle$ + $replicas'$ (line 8) to the other servers within the same DC. Similar to POST or PUT operations, the migration process utilizes the relay mechanism to reduce the network transmission cost across DCs. The home s_i may just send $k.replicas'$ to DC_j in the following three cases: 1) DC_j is not a replica DC in t_h (lines 10-12). 2) DC_j was a replica DC or included a cache server in t_{h-1}, and is a replica DC in t_h (lines 13-18). 3) DC_j was not a replica DC in t_{h-1}, but will be a replica DC in t_h (lines 19-21). After receiving $k.replicas'$ or $k.replicas$ + $cvl\langle k\rangle$ from other DCs, it needs to update the replica placement and store $cvl\langle k\rangle$ (if received), and then to relay them to other servers within the same DC (lines 25-32 and 37-43).

Algorithm 6: Migration operations at s_i for o_k in DC_i at t_{h-1}

1 **for** *each* $s_j (\neq s_i)$ *in* DC_i **do**
2 **if** $s_j \notin k.replicas'$ **then**
3 send $f(k, k.replicas')$ to s_j;
4 **else if** $s_j \in k.replicas'$ *and* $s_j \in k.replicas$ **then**
5 send $f(k, k.replicas')$ to s_j;
6 **else if** $s_j \in k.replicas'$ *and* s_j *is a caching server* **then**
7 send $f(k, k.replicas')$ to s_j;
8 **else** send$\langle$MIGR $k, k.replicas', cvl\langle k\rangle\rangle$ to s_j ;
9 **for** *each* $DC_j \neq DC_i$ **do**
10 **if** $R(DC_j)$ = *false in* t_h **then**
11 select s_j with the largest AN_{t_h} from DC_j;
12 send $f(k, k.replicas')$ to s_j;
13 **else if** $R(DC_j)$ = *true in* t_{h-1} **then**
14 select a replica s_j from DC_j;
15 send $f(k, k.replicas')$ to s_j;
16 **else if** DC_j *includes one caching server in* t_{h-1} **then**
17 select a caching server s_j from DC_j;
18 send $f(k, k.replicas')$ to s_j;
19 **else**
20 select s_j with the largest AN from DC_j;
21 send$\langle$MIGRB $k, k.replicas', cvl\langle k\rangle\rangle$ to s_j;

Upon receive $f(k, k.replicas')$ **from** DC_i
22 **if** $s_i \notin k.replicas'$ & $s_i \in k.replicas$ **then**
23 remove $cvl\langle k\rangle$;
24 OBJTABLEUPDATE(k,$replicas'$);

Upon receive $f(k, k.replicas')$ **from** DC_j $(j \neq i)$
25 **for** *each* $s_j (\neq s_i)$ *in* DC_i **do**
26 **if** $s_j \notin k.replicas'$ **then**
27 send $f(k, k.replicas')$ to s_j;
28 **else**
fetch $cvl\langle k\rangle\rangle$;
29 send$\langle$MIGR $k, k.replicas', cvl\langle k\rangle\rangle$ to s_j
30 **if** $s_i \notin k.replicas'$ & $s_i \in k.replicas$ **then**
31 remove $cvl\langle k\rangle$;
32 OBJTABLEUPDATE(k,$replicas'$);

Upon receive $\langle$**MIGR** $k, k.replicas', cvl\langle k\rangle\rangle$
33 **for** *each* $d \in cvl\langle k\rangle$ **do**
34 **if** *(*$ATP(dm_d, VV_i, s_i)$=**false***)* **then**
35 move d from $cvl\langle k\rangle$ to the invisible list of o_k;
36 OBJTABLEUPDATE(k,$replicas'$);

Upon receive $\langle$**MIGRB** $k, k.replicas', cvl\langle k\rangle\rangle$ **from** DC_j $(j \neq i)$
37 **for** *each* $s_j (\neq s_i)$ *in* DC_i **do**
38 **if** $s_j \in k.replicas'$ **then**
39 send$\langle$MIGR $k, k.replicas', cvl\langle k\rangle\rangle$ to s_j
40 **for** *each* $d \in cvl\langle k\rangle$ **do**
41 **if** *(*$ATP(dm_d, VV_i, s_i)$=**false***)* **then**
42 move d from $cvl\langle k\rangle$ to the invisible list of o_k;
43 OBJTABLEUPDATE(k,$replicas'$);

IV. Performance Evaluation

We evaluate the proposed **CaDRoP** protocol by real traces of requests to the web servers from Twitter workload [28] and the CloudSim discrete event simulator [29]. These realistic traces contain a mixture of temporal and spatial information for each http request. The number of http requests received for each of the target data objects (e.g., photo images) is aggregated in 1000-secs intervals based on the dataset used in [23]. By implementing our approaches on the Amazon cloud provider, it allows us to evaluate the cost-effectiveness of request transaction, data store, and network transmission, and to explore the impact of workload characteristics. We also evaluate CaDRoP by a clairvoyant Optimal Placement (OPT) Solution, proposed in [23], based on the time slot system and object access patterns known in advance.

A. Data Object Workload

Our work focuses on the data store framework on image-based sharing in social media networks, where applications have geographically dispersed users who PUT and GET data, and fit straightforwardly into a key-[values] model. We use actual Twitter traces as a representation of the real world. PUT or POST, denoted as Put, to a timeline occurs when users post a tweet, retweet, or reply messages. We crawl the real Twitter traces as the evaluation input data. Since the Twitter traces do not contain information of reading the tweets (i.e., the records of $Gets$), we set five different ratios of Put/Get (P_{rate}: Put rate), where the patterns of $Gets$ on the workloads follow Longtail distribution model [30]. The simulation workload contains several Tweet objects. The volume V of each target tweet in the workload is 2 MB. The simulation is performed for a period of three weeks. The results for each object show that they have similar tendency.

The experiment has been performed via simulation using the CloudSim toolkit [29] to evaluate the proposed system. CloudSim is a JAVA-based toolkit that contains a discrete event simulator and classes that allow users to model distributed cloud environments, from providers and their system resources (e.g., physical machines and networking) to customers and access requests. CloudSim can be easily developed by extending the classes, with customized changes to the CloudSim core. We figure out our own classes for simulation of the proposed framework and model 9 DCs in CloudSim simulator. Each DC is composed of 4 pairs of web servers and data servers. Each data server incorporates a 50GB storage space and each web server is in charge of user's query processing from one (or a few) states in US or one country in Asia and in Europe. The price of the storage classes and network services are set

TABLE II
Cost improvement rates in different Put rates and RF values.

P_{rate}	0.05	0.1	0.2	0.5	0.8
RF=9	3.46%	2.87%	5.24%	4.41%	4.16%
RF=5	72.62%	58.88%	55.05%	21.35%	6.74%
RF=2	79.46%	69.49%	56.33%	29.08%	11.95%

in terms of Amazon Web Service (AWS) as of 2019.

B. Results and Discussion

The performance metrics we use are based on the monetary cost and the cost improvement rates under varying P_{rate}. In order to evaluate our proposed algorithm, we compare it to different replication factors (RF). RF is the number of replica DC, where it is randomly pre-selected and each replica DC includes one replica data server. More specifically, when RF is constant and the replica placement for each key is predetermined, CaDRoP is simplified to 'CaS', which proceeds only by Algorithms 1 $\sim$ 5, without CORP. Cost is represented by the total system cost, which is composed of transaction cost (TC), network transmission cost (NTC), and storage cost (SC). We use the term 'transaction' to denote data query operations, such as Put or Get. NTC depends on the size of the packet (e.g., a d packet) transmitted. SC includes the costs of storing data items (including the dm data) and the bookkeeping management of data replication information.

1) CaS′ Vs. CaS: To evaluate the effectiveness of the cache component, we examine the system performance with the comparisons between CaS′ (CaS w/o cache) and CaS on cost improvement rate with respect to different RF, which is defined as:

$$\frac{cost(CaS') - cost(CaS)}{cost(CaS')} \tag{2}$$

Table II shows the cache effectiveness of different RF modes for different Put rates increases as RF decreases. As Put rate decreases, the cost improvement of CaS becomes higher except for full DC replication (RF=9).

2) CaS Vs. CaDRoP: We now evaluate the cost effectiveness of CaDRoP by comparing it with CaS. By running the same workloads as before, Figure 3 presents the TCs of various RF models in different Put rates. Lowering the number of transactions to fetch objects from remote data servers increases throughput in cloud environments, while an increased number of transactions would lead to an over-utilization of the underlying systems. Thus, the total TC is completely subject to the number of transactions. The results show that CaDRoP can achieve the best performance for TC under the same cache capacity, although it needs to bring additional transactions for the migration process. Figure 4 presents the NTC of CaDRoP in comparison with various RF models in different Put rates. The smaller the NTC, the lower the network bandwidth consumption. Although NTC of CaDRoP is slightly higher than that of the full DC replication, it is much lower than others' NTCs. Figure 5 shows the results of SC of CaDRoP in comparison with other alternatives. It is noteworthy that the SC of CaDRoP falls in between the SCs of the replication models with RF=9 and RF=2. This implies that the proper number of replicas for CaDRoP is able to decrease TC and NTC. Figure 6 presents the total system costs (TSC) for CaDRoP and CaS+cache in different RF values. It illustrates that CaDRoP can reduce TC and NTC at the slight cost of SC.

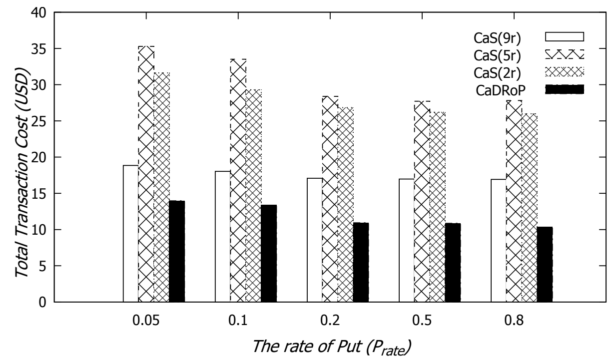

Fig. 3. The Transaction Cost

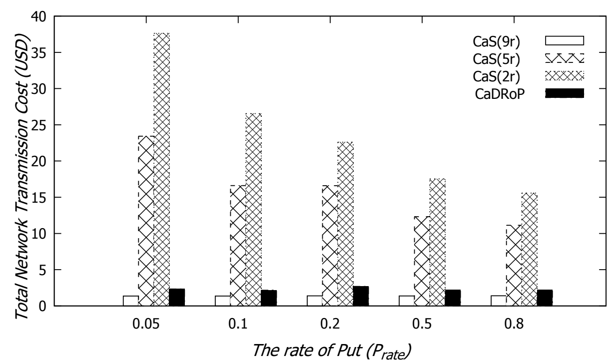

Fig. 4. The Network Transmission Cost

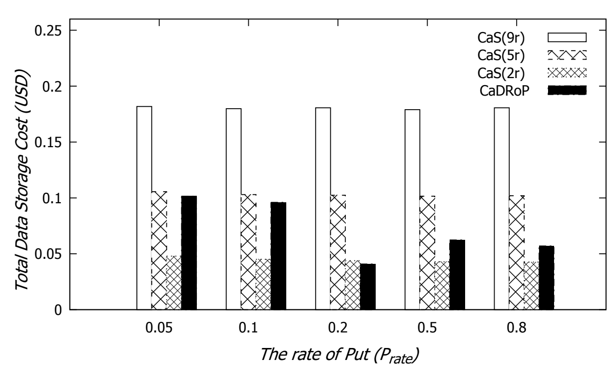

Fig. 5. The Storage Space Cost

3) CaDRoP VS. CaDRoP′ (w/o cache): CaDRoP integrates cache functionality to improve the system costs. Thus, in this section we present experiments aimed at evaluating how the total costs are improved by CaDRoP against CaDRoP′. Table III presents the results of the cost saving ratio (Δ_{saving}) for different Put rates. Δ_{saving} is defined as

$$\frac{cost(CaDRoP') - cost(CaDRoP)}{cost(CaDRoP')} \tag{3}$$

Since the evaluation data come from the social network, each individual data object brings a lot of requests in the initial

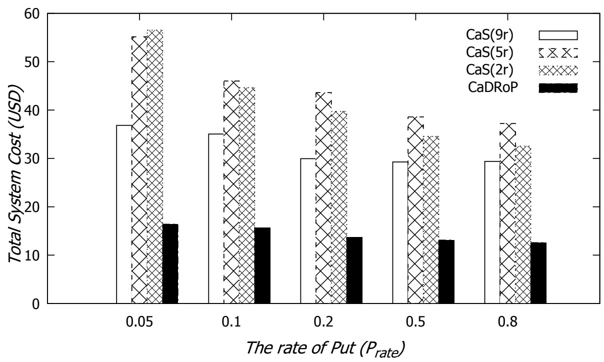

Fig. 6. The Total System Cost

TABLE III
Δ_{saving}: The cost improvement results for different Put rates show that caching has taken an important step to improve the total system costs. Δ_{inc}: The performance evaluation of CaDRoP compared to CaDRoP+OPT. $\Delta_{inc'}$: The performance evaluation of CaDRoP′ compared to CaDRoP′+OPT′ in steady states.

P_{rate}	0.05	0.1	0.2	0.5	0.8
Δ_{saving}	91.95%	85.01%	75.14%	57.84%	49.02%
Δ_{inc}	16.08%	13.17%	10.21%	9.02%	6.17%
$\Delta_{inc'}$	1.72%	2.62%	1.6%	4.51%	3.31%

time slots. It can be observed that the results indicate that the lower the P_{rate} (Get-intensive), the better the Δ_{saving} is.

4) CaDRoP evaluation: In order to evaluate the effectiveness of CaDRoP, we also implemented the Optimal Placement Solution (OPT) proposed in [23] as the clairvoyant replication strategy. As mentioned in Sec. III-C, CORP runs on the underlying replication layer of CaDRoP. Compared to CORP, OPT knows the exact temporal and spatial data object access patterns. OPT can figure out the optimal object placement for each time slot. CaDRoP+OPT means that the underlying layer of CaDRoP implements OPT rather than CORP. Δ_{inc} is defined as

$$\frac{cost(CaDRoP) - cost(CaDRoP + OPT)}{cost(CaDRoP)} \tag{4}$$

Δ_{inc} in Table III presents the comparisons between CaDRoP and CaDRoP+OPT for different Put rates. It is evident that CaDRoP only increases $6\% \sim 16\%$ of total system cost compared to CaDRoP+OPT.

In order to measure the cost effectiveness of CaDRoP in steady states (including enough training time slots), we also compare the cost of CaDRoP′ (w/o cache) to that of CaDRoP′+OPT′ (w/o cache) in steady states. $\Delta_{inc'}$ in Table III gives the cost increase ratios ($\Delta_{inc'}$) of CORP compared to OPT for different Put rates. We notice that $\Delta_{inc'}$ rates are around $2\% \sim 4.5\%$. $\Delta_{inc'}$ is defined as

$$\frac{cost(CaDRoP') - cost(CaDRoP' + OPT')}{cost(CaDRoP')} \tag{5}$$

5) CoCaCo VS. CaDRoP: In order to empirically evaluate the effectiveness of our approach, we compare it to another CC+ protocol, CoCaCo proposed in [16], for the following reasons. 1) It can be applied to partially replicated systems. 2) It also realizes multi-version storage systems to preserve all the updated values. 3) The architecture of CoCaCo is highly similar to that of CaDRoP. 4) CoCaCo implements CC+ both within and across DCs. Note that CoCaCo cannot achieve the convergence property for all replying comments (update operations) corresponding to an object post. Table IV demonstrates the simulation results for CoCaCo and CaDRoP by running the workloads used in the above experiments in various Put rates. As the RF value decreases, the overheads of storing dm decrease in terms of the SC results, but the volume of transmitting dm over networks increases in terms of the NTC results. For CoCaCo, the TC costs are apparently higher that those of CaDRoP, since CoCaCo invokes more acknowledgement messages and implements access requests. The SC costs of CoCaCo are lower than those of CaDRoP in the lower RF values, while CoCaCo's SC is higher in the higher RF value.

V. Conclusion

We proposed CaDRoP to ensure CC+ between posts and for the comments under each post in social network systems. CaDRoP is adapted to a proposed dynamic replication algorithm CORP, which proactively deploys required data replicas in geo-replicated datastores. We presented an evaluation of the effect of CaDRoP in terms of cost improvement via trace-driven CloudSim toolkit and realistic workload traces from Twitter. Simulations show that, with caching, as RF increases, the TSC decreases. CaDRoP is around $55 \sim 70\%$ lower than CaS+cache in different predetermined RF models. We compared CaDRoP to an OPT replication solution based on known temporal and spatial access patterns. CaDRoP increases only $6 \sim 16\%$ of TSC of CaDRoP+OPT. Without cache, simulation results show that the TSC of CaDRoP′ is slighly higher than that of CaDRoP′+OPT′ in a steady state. In other words, by proactively allocating data resources where and when users need them, our approach is capable of being effective in cost saving, even without cache. The simulation results also showed that the TSC of CaDRoP is usually improved better in lower P_{rate}. It implies that CaDRoP is cost-effective for most social applications with Get-intensive workloads.

References

[1] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, p. 1277–1288, Aug. 2008.

[2] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, Aug. 2013.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, Oct. 2007.

[4] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[5] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11, 2011, pp. 401–416.

TABLE IV
The price cost comparisons between CoCaCo and CaDRoP in different Put rates and RF models. 'SC' includes two costs: (i) storing data objects + (ii) storing dm data. Similarly, 'NTC' includes two costs: (i) transmitting data objects + (ii) transmitting dm data.

		CoCaCo				CaDRoP			
		SC	TC	NTC	TSC	SC	TC	NTC	TSC
P_{rate}=0.05	RF=9	0.155+0.029	19.57	1.3+0.256	21.3	0.099+0.002	13.97	2.273+0.06	16.4
	RF=5	0.086+0.018	27.525	186.1+14.23	228				
	RF=2	0.035+0.007	25.07	329.7+24.77	380				
P_{rate}=0.1	RF=9	0.155+0.028	18.594	1.3+0.250	20.3	0.094+0.002	13.36	2.112+0.051	15.62
	RF=5	0.086+0.017	25.503	97.33+6.805	130				
	RF=2	0.035+0.007	24.072	160.2+12.47	196				
P_{rate}=0.2	RF=9	0.155+0.026	18.279	1.3+0.246	20	0.039+0.002	10.93	2.607+0.085	13.67
	RF=5	0.086+0.017	25.091	78.002+4.362	107				
	RF=2	0.035+0.006	22.683	90.62+6.136	119				
P_{rate}=0.5	RF=9	0.155+0.026	17.797	1.3+0.245	19.5	0.061+0.001	10.86	2.113+0.07	13.11
	RF=5	0.086+0.017	24.704	26.837+2.197	54				
	RF=2	0.035+0.006	22.290	42.10+3.024	67				
P_{rate}=0.8	RF=9	0.155+0.025	17.731	1.3+0.235	19.4	0.055+0.002	10.33	2.112+0.078	12.58
	RF=5	0.086+0.016	22.628	19.516+1.890	44				
	RF=2	0.035+0.006	20.216	29.296+2.602	52				

[6] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "Gentlerain: Cheap and scalable causal consistency with physical clocks," in *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 03 - 05, 2014*, 2014, pp. 4:1–4:13.

[7] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13, 2013, pp. 11:1–11:14.

[8] H. Attiya, F. Ellen, and A. Morrison, "Limitations of highly-available eventually-consistent data stores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 141–155, 2017.

[9] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: 313–328.

[10] S. Almeida, J. a. Leitão, and L. Rodrigues, "Chainreaction: A causal+ consistent datastore based on chain replication," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 85–98.

[11] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro, "Write fast, read in the past: Causal consistency for client-side applications," in *Proceedings of the 16th Annual Middleware Conference*, ser. Middleware '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 75–87.

[12] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: Strong semantics meets high availability and low latency," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016, pp. 405–414.

[13] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd, "I can't believe it's not causal! scalable causal consistency with no slowdown cascades," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 453–468.

[14] M. Roohitavaf, M. Demirbas, and S. Kulkarni, "Causalspartan: Causal consistency for distributed data stores using hybrid logical clocks," in *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, 2017, pp. 184–193.

[15] K. Spirovska, D. Didona, and W. Zwaenepoel, "Optimistic causal consistency for geo-replicated key-value stores," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 2626–2629.

[16] Y. Tang, H. Sun, X. Wang, and X. Liu, "Achieving convergent causal consistency and high availability for cloud storage," *Future Generation Computer Systems*, vol. 74, pp. 20 – 31, 2017.

[17] D. Didona, R. Guerraoui, J. Wang, and W. Zwaenepoel, "Causal consistency and latency optimality: Friend or foe?" *Proc. VLDB Endow.*, vol. 11, no. 11, p. 1618–1632, Jul. 2018.

[18] T. Mahmood, S. PN, S. Rao, T. Vijaykumar, and M. Thottethodi, "Karma: Cost-effective geo-replicated cloud storage with dynamic enforcement of causal consistency," *IEEE Transactions on Cloud Computing*, pp. 1–1, 06 2018.

[19] K. Spirovska, D. Didona, and W. Zwaenepoel, "Paris: Causally consistent transactions with non-blocking reads and partial replication," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, July 2019, pp. 304–316.

[20] T.-Y. Hsu, A. Kshemkalyani, and M. Shen, "Causal consistency algorithms for partially replicated and fully replicated systems," *Future Generation Computer Systems*, vol. 86, pp. 1118 – 1133, 2018.

[21] T.-Y. Hsu and A. D. Kshemkalyani, "Convergent causal consistency for social media posts," in *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*, ser. PaPoC '21. New York, NY, USA: Association for Computing Machinery, 2021.

[22] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Trans. Database Syst.*, vol. 4, no. 2, p. 180–209, Jun. 1979.

[23] T.-Y. Hsu and A. D. Kshemkalyani, "A proactive, cost-aware, optimized data replication strategy in geo-distributed cloud datastores," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, ser. UCC'19. New York, NY, USA: Association for Computing Machinery, 2019, p. 143–153.

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

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

[26] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 761–772.

[27] M. Shen, A. D. Kshemkalyani, and T. Y. Hsu, "Causal consistency for geo-replicated cloud storage under partial replication." in *IPDPS Workshops*. IEEE, 2015, pp. 509–518.

[28] R. Li, S. Wang, H. Deng, R. Wang, and K. C.-C. Chang, "Towards social user profiling: unified and discriminative influence model for inferring home locations," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '12*, 2012, pp. 1023–1031.

[29] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw. Pract. Exper.*, vol. 41, no. 1, pp. 23–50, Jan. 2011.

[30] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: Facebook's photo storage," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10, 2010, pp. 47–60.