Application-Aware Consistency: An Application to Social Network *

Lewis Tseng[1], Alec Benzer[1] and Nitin Vaidya[2]

[1] Department of Computer Science,
[2] Department of Electrical and Computer Engineering,
University of Illinois at Urbana-Champaign

Email: {ltseng3, benzer2, nhv}@illinois.edu

Feb. 15, 2015

Abstract

This work weakens well-known consistency models using graphs that capture applications' characteristics. The weakened models not only respect application semantic, but also yield a performance benefit. We introduce a notion of *dependency graphs*, which specify *relations* between events that are important with respect to application semantic, and then weaken traditional consistency models (e.g., causal consistency) using these graphs. Particularly, we consider two types of graphs: *intra-process* and *inter-process* dependency graphs, where intra-process dependency graphs specify how events in a single process are related, and inter-process dependency graphs specify how events across multiple processes are related. Then, based on these two types of graphs, we define new consistency model, namely *application-aware* consistency. We also discuss why such application-aware consistency can be useful in social network applications.

This is a work in progress, and we present early ideas regarding application-aware consistency here.

*This research is supported in part by National Science Foundation awards 1409416. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies or the U.S. government.

1 Introduction

Geo-replicated storage is commonly used to store data nowadays [14, 1, 15]. Replication brings with it the problem of maintaining *consistency* across the replicas. Strong notions of consistency can be *expensive* to achieve, often resulting in a non-trivial increase in latency in performing the operations. The difficulties of implementing replicated storage were captured in the CAP theorem [13, 19], which essentially says that strong consistency, availability, and partition tolerance cannot *all* be achieved simultaneously in a replicated system. As observed in [2, 20], partition tolerance is necessary in many practical systems, and thus, cloud service providers have often chosen to support replication under *weaker* notions of consistency (e.g., [1, 16, 29, 27, 35]).

Theses *weaker* consistency models – although motivated by applications' need for partition-tolerance – are often agnostic of other application characteristics or application semantics. For instance, the *causal consistency* model [4] ensures that causal order [25] is enforced on events observed by any user, regardless of the *relations* between events. We will use the terms events and operations interchangeably. In many applications, particularly social networking, the causal order may be too strong, and thus cause unnecessary latency [9, 7]. To reduce such unnecessary latency, it is possible to weaken causal consistency without compromising on user-perceived "quality" of the information, by taking into account the applications' semantics. The idea is using *dependency graphs* that describe important application-specific ordering constraint, and relax those ordering constraints (specified by the existing consistency model) that are not important, i.e., the ones not presented in the *dependency graphs*. For brevity, we will only focus on weakening causal consistency [4], and addressing why such weakened causal consistency is adequate for social network applications. Appendix B briefly discusses how to relax other consistency models using our approaches. We believe that these relaxed consistency models can be useful in some other applications.

This is a work in progress, and we present early ideas regarding application-aware consistency here.

Application to Social Network: Consider a simplified version of Facebook-like application: each user has a *wall* where users can add new posts or comment on old posts. Later, Section 4.1 will address more complex operations, such as adding or removing friends. The application is implemented on top of a geo-replicated storage system, which supports two operations – read and write, and stores data on multiple geographically distributed replicas. The users access each other's posts from the closest available replica. Similarly, a user's posts (or writes) are first propagated to one of the replicas, which in turn, will propagate them to other replicas. The *consistency model* being supported determines how the posts are propagated to the replicas, and when the posts become *visible* to users.

Facebook-like applications are usually implemented on top of systems ensuring *eventual consistency* [1, 24] or *causal consistency* [29, 10] due to smaller latency and the ability to work in partitioned network [32]. However, these two models have their own drawbacks. On one hand, as discussed in [29, 30], eventual consistency does not respect application semantic. Consequently, users may observe undesirable outcomes. On the other hand, causal consistency model may result into unnecessary latency for some events due to its restrictive ordering constraint [7, 9]. Appendix A elaborates on the limitations of eventual and causal consistency.

In short, we need a new consistency model for social network applications. Our solution is *application-aware* causal consistency, which weakens the causal ordering constraint based on *dependency graphs* that specify applications' characteristics. First, *application-aware* causal consistency

is based on causal consistency, and hence, unlike eventual consistency, it will respect the necessary application semantic. Second, by weakening the causal ordering, we *reduce* the number of dependent operations for each operation without compromising on user-perceived "quality" of the information. This weakening technique can result in a *reduced latency* in completing operations. In particular, the delay before an operation can be made visible to a client can be smaller, because dependency of a *smaller subset of events* must be enforced (i.e., the operation needs to wait on a smaller number of prior operations before it can be propagated). Waiting for fewer operations (or messages) implies that the expected delay is typically smaller, due to delay variability.

In the discussion below, we will use an example in Figures 1 and 2 to illustrate our relaxed causal consistency models. The example is adapted from an example presented in [30]. Consider three users, Alice, Bob and Calvin, who interact with each other via the social network application – assume that all three of them can indeed access each other's wall because they are all "friends" in the social network. The posts (by Alice) and comments (by Bob) on Alice's wall and the corresponding timestamps are shown in Figure 1. In our discussion, we will use the term *post* to refer to the text appearing first for a certain *topic*, and the term *comment* for the text ensuing some *posts*. For example, Alice's update on 9:00 is a post, and Bob's update on 9:05 is an ensuing comment. For the example in Figure 1, we will use "**lost**" to represent Alice's post at 9:00, "**no**" for Bob's comment at 9:05, "**found**" for Alice's post at 9:30, and "**glad**" for Bob's comment at 9:40, respectively.

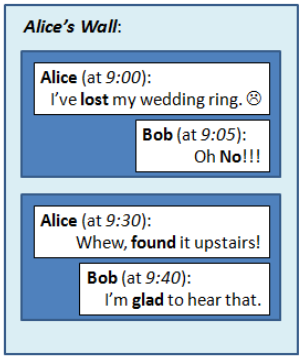

Figure 1: *In the discussion, we use "***lost***" to represent Alice's post at 9:00, "***no***" for Bob's comment at 9:05, "***found***" for Alice's post at 9:30, and "***glad***" for Bob's comment at 9:40, respectively.*

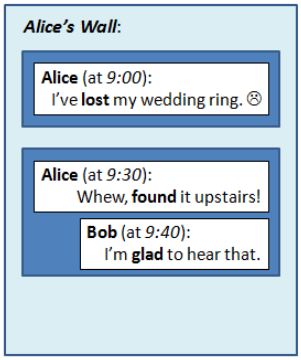

Figure 2: *This example illustrate that missing comments do not affect users' understanding.*

Weakening Techniques: We propose two orthogonal methods to define *application-aware causal consistency*. The core idea is to reduce the number of dependency operations (defined by the causal consistency) for each operation without violating application semantic. As addressed above, this weakening will likely result in a reduced latency.

- *The first approach considers how the events in a single process are related:*

  This is motivated by the following observation: *the ordering observed by a single user may not necessarily reflect the application semantic; thus, the sequential ordering for a user's operation (program order) does not need to be respected at all time.* Consider the example in Figure 1. From Alice's perspective, "**found**" does not causally depend on "**no**". In other words,

even though Alice reads "**no**" before posting "**found**", "**no**" does not *cause* Alice to post "**found**". Consequently, it is fine if Calvin observes "**found**" before observing "**no**". That is, even if Calvin observes Alice's wall as shown in Figure 2, Calvin can still follow the whole story. Such a weakening may be desirable if the propagation delay of "**no**" is too large. In this case, even when Calvin's replica already had received "**found**", the replica cannot make "**found**" visible to Calvin (for satisfying causal order [25]). As a result, Calvin has to wait for the long delay of "**no**", which is an unnecessary delay.

In short, we want a consistency model that achieves two following goals: (i) allowing Calvin to observe "**found**" before "**no**" to shorten latency, and (ii) requiring Calvin to observe "**found**" before "**glad**" to enforce application semantics. It is obvious that these goals cannot be simultaneously satisfied by enforcing either causal or eventual consistency. As a remedy, we propose *application-specific* consistency based on the notion of *intra-process dependency graph*, which is a directed acyclic graph describing how each operation is *related* to other operations at a single process. For instance, in Calvin's scenario, the dependency graph will specify that there is no relation between "**found**" and "**no**". Thus, Calvin's replica can make "**found**" visible to Calvin even if Calvin has not yet seen "**no**". Effectively, the intra-process dependency graph allows us to relax *program order* without violating important application semantic. Section 4 discusses this method in details. In particular, it shows that the new causal consistency model based on the intra-process dependency graph achieves the two aforementioned goals.

- *The second approach considers how the events across multiple processes are related:*

  We use an *inter-process dependency graph*, which explicitly describes how each read operation at a process is *related* to other processes' write operations. One special form of inter-process dependency graph is the "friends" graph, in which two users are connected if they are each other's friends in the social network. Intuitively, each user C must observe operations performed by users within distance d in the "friends" graph in the causal order *induced only by operations performed by users within distance* d. Operations performed by users farther than distance d may be observed by C in an arbitrary order. We elaborate on the motivation using an example below.

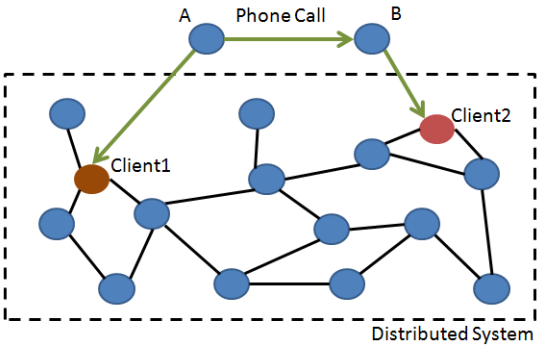

Figure 3: *The view of universe in traditional causal consistency.*

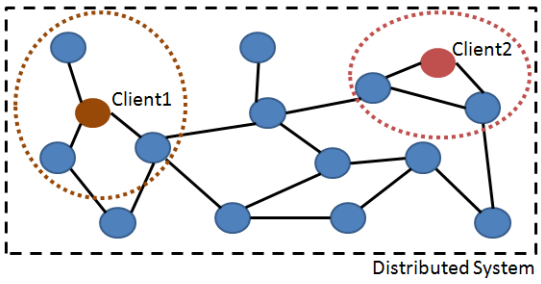

Figure 4: *The view of universe in weakened causal consistency.*

Figure 3 shows a social network, with the graph in the figure depicting the friends relation. In this system, it is possible that agents A and B that are external to the social network may send messages (such as e-mails) to Client1 and Client2 (who are in the social network)

as shown in the figure. In fact, the message from agent B may follow a phone call from agent A to agent B. Thus, there is a potential causal dependence between messages sent by A and B. However, the consistency protocol used for the social network has no way to take into account this causal relationship. The clients find it acceptable that causality in the universe "external" to the social network is not always reflected in the behavior they observe. The friends graph-based consistency model generalizes on the notion of external universe. In particular, as shown in Figure 4, the notion of external universe is defined *with respect to each client*. For Client1 and Client2, everyone outside d-hop distance from them in the friends graph is viewed as external to their universe, respectively. Thus, everyone outside the dotted circle around Client1 is external to Client1's universe for the purpose of enforcing causal delivery (and similarly for Client2).

We believe that such weakening of consistency is likely to be acceptable in social networking contexts, provided that it yields a performance benefit. For instance, the friends graph-based causal consistency model with $d = 1$ in the example in Figure 1 will still result in Calvin observing "**found**" before "**glad**", the desired behavior. Section 5 discusses this method. Note that this method shares some similarity to Fisheye consistency [17] and other consistency models using distance as a metric [34, 23]. We discuss the differences in Section 2.

For brevity, we will only focus on relaxing causal consistency using *dependency graphs*, and discussing why our relaxed model is suitable for Facebook-like applications. Appendix B discusses how to weaken other well-known consistency models.

2 Related Work

There is a long history of research on various weak consistency models for parallel computers and for distributed shared memory systems, with significant activity on this topic in the 1990s [3, 18, 22]. Recently, the weak consistency models have received renewed attention in the context of replicated storage systems, which aims to reduce access latency via *geo-replication*. With the users located across the globe, it has become important to keep the data close to all the users who share it, in order to reduce access latency. This has motivated *geo-replication*, or replication of data across geographically distributed replicas. Such *geo-replicated* storage systems include Windows Azure [14], Cassandra [1], Amazon's Dynamo [16] and Google's Spanner [15] and Megastore [11]. Large latencies can still be incurred if the system insists on supporting *strong notions of consistency*, since that effectively requires a total ordering of requests (or operations) executed by the geographically distributed replicas. This has forced the system designers to address the trade-off anticipated by the CAP theorem, which is initially conjectured by Brewer [13], and later formally proved by Gilbert and Lynch [19]. Designers of many geo-replicated systems determined that *availability* and *partition-tolerance* are sometimes more important than *strong consistency* [2, 8], and chose to relax the consistency guarantees (e.g., see [16]). Many weak versions of consistency have been supported, such as eventual [1, 16], causal+ [29], RedBlue [27], update [33] and multi-level consistency (Pileus system) [35]. This line of work focuses more on availability-consistency trade-off in the face of partitioned network, and does not elaborate on frameworks to weaken existing consistency models. We discuss papers that share similarity with our work below.

The idea of relaxing *program order* at a process is not new. Prior work has also proposed similar ideas, e.g., eventual [16, 24], Hybrid [5], update [33], and RedBlue [27] consistency. Among this work, eventual [16, 24] and update [33] consistency did not consider application characteristics, whereas [5, 27] took into account application semantic. [5, 27] divided operations into two categories:

strong and weak, and the consistency models require strong operations to follow some stronger ordering constraint (e.g., sequential consistency [26]) while weak operations may follow some more relaxed ordering constraint (e.g., PRAM [28] or eventual consistency). However, to the best of our knowledge, the relaxation based on intra- and inter-process dependency graphs in our work has never been studied before, and we believe that such weakening approaches would yield models that may be better suited for the emerging class of applications.

Prior work also attempts to relax ordering constraint of events across multiple processes. There is work using *distance* as a metric to define consistency models. [23, 34] proposed geographical-distance-based consistency models for games, in which users observe nearby events in a strong ordering and far-away events in a weaker ordering. However, [23, 34] did not consider usage of graphs, and it is not clear whether their consistency models can be easily extended to the case when distance becomes a virtual notion like the one in friends graphs. Closest to our second approach (inter-process dependency graphs) is the recent work of Friedman, Raynal and Taïani on *Fisheye consistency*, which also incorporates the idea of using graphs to relax traditional consistency models [17]. Intuitively, Friedman et al. use a proximity graph (, which is an *undirected graph*) to describe important ordering constraints. While our approach and the work on *Fisheye* consistency share the property of using graphs, there are some important limitations to this early work. The key shortcoming is that Friedman et al. only present one concrete use of graphs, namely to support a version of the *sequential* consistency model for operations performed by nodes near each other (neighboring nodes in the proximity graph), and causal consistency for operations performed by other nodes (non-neighboring nodes in the proximity graph). They do not identify how to extend their approach to weaken other useful consistency models using proximity graphs. Most importantly, we allow the inter-process dependency graphs to be *directed* graphs, while the proximity graph in [17] is assumed to be undirected. Later in Section 4.2, we show that using *directed* graphs allow us to capture richer application semantic. The other main difference is that Friedman et al. do not consider how the operations in a single process are related (we use *intra-process dependency graphs* to capture the relation of events occur at a process).

Bailis et al. also identified the scalability issue of ensuring causality [7, 9]. [9] proposed tracking application-specific causal dependency, which results in higher write throughput due to faster check of causality dependency and smaller metadata. In his *blog* [7], Bailis also discussed several other methods to make causality cheaper, including sacrificing availability for more efficiently tracking causality. However, Bailis et al. did not specify how to track the dependency, nor did them provide a systematic way to relax existing consistency models. Particularly, they did not consider usage of intra- and inter-process dependency graphs, and did not provide a formal definition of the relaxed consistency model in [7, 9]. We share their motivation of improving performance by weakening the consistency requirements. Beyond [7, 9], we propose using dependency graphs that describe application-specific ordering constraints, and formally define relaxed consistency models based on the dependency graphs.

3 System Model and Terminology

System Model: We consider a *system* consisting of n processes, $\mathcal{P} = \{p_1, p_2, \cdots, p_n\}$. The geo-replicated storage is modeled as a shared memory of a finite set of objects . We will use lower case italic letters to denote the objects. Each process interacts with the shared memory through a series of read and write operations over a *reliable* channel. The system is assumed to be asynchronous. We further make the following assumptions to simplify our discussion:

- No value is written more than once to the same object. This assumption simplifies the definition of causal order – this can be relaxed by generalizing the reads-from order (presented below).
- Each process is *sequential*, i.e., no two operations are executed at the same time. We do note that our model can be generalized to the case of multi-thread process. We will address this extension in Section 5.1.
- Each operation o consists of an invocation event $inv(o)$ and a corresponding response event $resp(o)$. Moreover, each operation can access one object *atomically* at a time. Generalizing our model to support transactions is a future research direction.

Terminology: We introduce the standard terminology of consistency models in shared memory, e.g., [4, 6, 31]. For each process $p_i \in \mathcal{P}$, the local history L_i of process i is a sequence of read and write operations. If operation o_1 precedes o_2 in L_i, i.e., response event of o_1 occurs before invocation event of o_2 in L_i, we denote it by $o_1 \xrightarrow{L_i} o_2$. A history $H = \{L_1, \cdots, L_n\}$ is the collection of local histories. A serialization S of the history H is a *linear* sequence of all operations in H in which each read operation on an object x returns its *most recent* preceding write operation on the object x (or $\perp$ if there is no preceding write in S). The serialization S respects an order $\rightarrow$, if for any operation o_1 and o_2 in S, $o_1 \rightarrow o_2$ implies o_1 precedes o_2 in S.

Given a history, the reads-from order $\xrightarrow[\textbf{read}]{}$ is defined as follows: if o_r is a read that returns the value written by o_w, then $o_w \xrightarrow[\textbf{read}]{} o_r$. Recall that, for simplicity, we assume that no value is written more than once to the same object. This can be easily relaxed by generalizing reads-from order. Then, we define the causal order [25]. Given a history, $o_1 \xrightarrow[\textbf{CC}]{} o_2$, where $\xrightarrow[\textbf{CC}]{}$ represents the causal order, if any of the following holds [25]:

- **Program-order**: $o_1 \xrightarrow{L_i} o_2$ for some p_i (o_1 precedes o_2 in L_i),
- **Reads-from**: $o_1 \xrightarrow[\textbf{read}]{} o_2$ (o_2 returns the value written by o_1), or
- **Transitivity**: there is some other operation o' such that $o_1 \xrightarrow[\textbf{CC}]{} o' \xrightarrow[\textbf{CC}]{} o_2$.

Let $H|(p_i + W)$ denote the set of all operations in the local history L_i and all write operations in the history H.

Definition 1 (Causal Consistency [4]) *A history is* causally consistent *if for each $p_i \in \mathcal{P}$, there exists a serialization S_i of $H|(p_i + W)$ that respects the causal order $\xrightarrow[\textbf{CC}]{}$.*

We will discuss different methods to relax causal consistency. In particular, Section 4 relaxes the program-order rule, and Section 5 relaxes the reads-from rule. Appendix B discusses how to relax sequential consistency [26], linearizability [21], and PRAM consistency [28].

4 Intra-Process Dependency

Our first approach addresses using intra-process dependency graphs to relax consistency models. More precisely, we will generalize the program-order rule discussed in Section 3. Most consistency

models, e.g., sequential [26], PRAM [28], and causal consistency [4], linearizability [21], enforce the existence of serialization(s) that respect the *program order* at each process. However, as discussed in Section 1, such requirement may not be necessary in some applications, since the ordering constraints defined by the program order may not reflect the real relations among operations. We first introduce the notion of *intra-process dependency graphs* that characterize important ordering constraints based on the applications' semantic. Then, we weaken causal consistency based on the intra-process dependency graphs, thus the name – *intra-causal consistency*. Appendix B discusses how to use dependency graphs to relax other well-known consistency models.

Intra-process dependency graph captures the ordering of events at each process that are important and should be respected. For now, assume that the dependency graphs are given. Later in the section, we will briefly discuss how to generate intra-process dependency graphs for a Facebook-like applications.

Definition 2 (Intra-Process Dependency Graph) *Given a history H, a graph for a process i, $D_i(\mathcal{V}_i, \mathcal{E}_i)$, is an intra-process dependency graph if it satisfies the following properties: (i) Each node in $\mathcal{V}_i$ corresponds to an unique operation in the local history of process i, L_i, and (ii) D_i is a directed acyclic graph.*

With a slight abuse of terminology, we will use the operation name to refer to the node in D_i, the intra-process dependency graph at process i. Whether we are referring to the operation or the corresponding node will be clear from the context.

In essence, intra-process dependency graph for a process i induces an *intra-program order* (denoted as $\xrightarrow[\textbf{intra}]{L_i}$) – which should be respected in consistency models. For brevity, we ignore the dependency graphs in the notation.

Definition 3 (Intra-Program Order) *Consider two operations o_1 and o_2 in the local history at process i, L_i, if node o_1 has a directed edge to node o_2 in D_i, then $o_1 \xrightarrow[\textbf{intra}]{L_i} o_2$.*

If D_i is a chain of operations that follow the linear order of L_i, then $\xrightarrow[\textbf{intra}]{L_i}$ is identical to the program order $\xrightarrow{L_i}$.

Here, we show how to use intra-process dependency graphs to transform the traditional causal consistency into a new model – *intra-causal consistency*. Appendix B discusses how to use intra-process dependency graphs to relax other well-known consistency models.

Given intra-dependency graphs, a history H induces an *intra-causal order*, denoted as $\xrightarrow[\textbf{intra}-\textbf{CC}]{H}$. For simplicity, we ignore the intra-dependency graphs in the notation. $o_1 \xrightarrow[\textbf{intra}-\textbf{CC}]{} o_2$ if any of the following holds:

- **Intra-program-order**: $o_1 \xrightarrow[\textbf{intra}]{L_i} o_2$ for some p_i (node o_1 has a directed edge to node o_2 in D_i),
- **Reads-from**: $o_1 \xrightarrow[\textbf{read}]{} o_2$ (o_2 returns the value written by o_1), or
- **Transitivity**: there is some other operation o' such that $o_1 \xrightarrow[\textbf{intra}-\textbf{CC}]{} o' \xrightarrow[\textbf{intra}-\textbf{CC}]{} o_2$.

Recall that $H|(p_i + W)$ denotes the set of all operations in the local history L_i and all write operations in the history H.

Definition 4 (Intra-Causal Consistency) *A history H is* intra-causally consistent *if for each $p_i \in \mathcal{P}$, there exists a serialization S_i of $H|(p_i + W)$ that respects $\xrightarrow[\mathbf{intra-CC}]{}$.*

Note that the difference between causal consistency and intra-causal consistency is in the first rule above. Whereas causal consistency is based on program order, we use *intr-program order* for intra-causal consistency, which relaxes program order using the intra-dependency graph.

4.1 Application to Social Network

We use the scenario in Figures 1 and 2 to illustrate why intra-causal consistency model is useful in social network applications. In particular, we discuss how intra-causal consistency achieves the following two goals: (i) allowing *some* user to observe "**found**" before "**no**", and (ii) requiring *each* user to observe "**found**" before "**glad**". As discussed in Section 1, the first goal may reduce the latency observed by users, while the second goal ensures that ordering of events follows application semantic. However, these two goals cannot be achieved simultaneous by either eventual or causal consistency as addressed in Appendix A. In the discussion below, we model each user of the Facebook-like application as a process. Thus, we will often use the terms user and process interchangeably.

We assume that intra-process dependency graphs are given, which are shown in Figure 5. In the figure, the boxes with solid and dashed lines represent write and read operations, respectively. For simplicity, users and objects are not shown in Figure 5. Section 5.1 addresses how to construct the intra-process dependency graphs in an online fashion. We first discuss briefly how Facebook-like application is implemented. A write operation by an user will write a value to a *distinct* object, and propagate the value to all the replicas. A read operation can be related to reading snapshot of parts of the shared memory space, e.g., shared memory space storing data of Alice's wall in our example. Intuitively, each read returns the "difference" between snapshot returned by the previous read and the current snapshot (or empty snapshot if there is no previous read). We also borrow some standard notations [4]:

- $w_i(x_{msg})$**msg** denotes that user i writes value **msg** to some distinct object x_{msg}. For example, $w_A(x_{found})$**found** means that Alice writes message "**found**" ("Whew, **found** it upstairs!") to a distinct object x_{found}.
- $r_i(x_{Alice\_wall})$**msg** denotes that user i reads the value **msg** – which is the difference between snapshots of shared memory space that contains data of Alic's wall. In the figures below, we will use the returned values of the read to represent read operations. For instance, $r_A(x_{Alice\_wall})$**No** means that the read operations returns the value "**No**" ("Oh No!!!") at object x_{No}. This is a *difference* between snapshots of Alice's wall, since after Alice wrote "**found**", the snapshot at Alice's replica already contains the value "**found**". Note that in the following figures, such a read operation is represented by a box with dashed line and text "No".

Now, we discuss why *intra-causal consistency* achieves two aforementioned goals below.

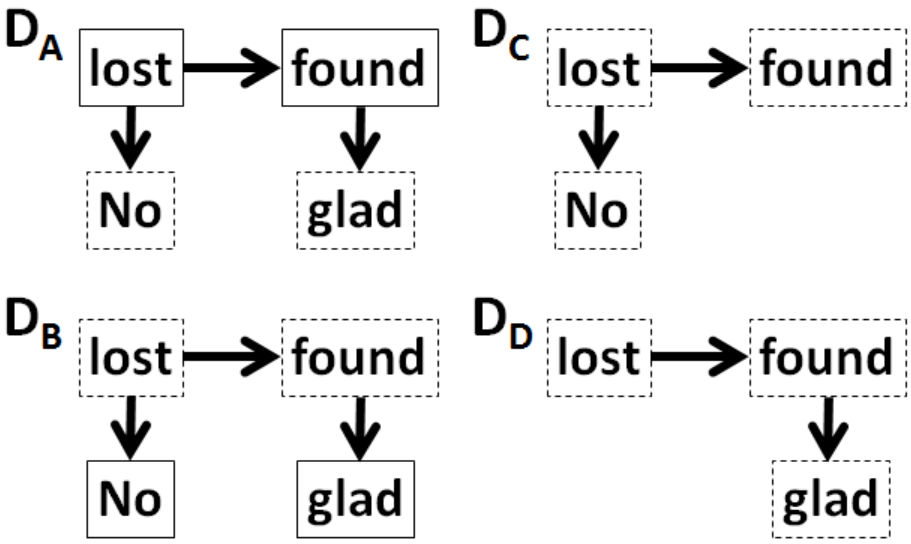

Figure 5: *Intra-process dependency graph for each user. Solid and dashed boxes represent write and read operations, respectively. For example, in the figure, the solid box of the text* ***lost*** *in* D_A *represents the operation* $w_A(x_{lost})$**lost** *– Alice writes message "***lost***" ("I've* **lost** *my wedding ring.") to a distinct object* x_{lost}.

- Allowing some user to observe "**found**" before "**no**":

  Consider the real time interaction of the scenario shown in Figure 6. In the figure, solid and dashed boxes represent write and read operations, respectively. Arrows denote the read-from relation. For simplicity, some operations are not shown in these figures.

  Alice: "I've **lost** my wedding ring."
  Bob: "Oh **No**!!!"
  Alice: "Whew, **found** it upstairs!"
  Bob: "I'm **glad** to hear that."

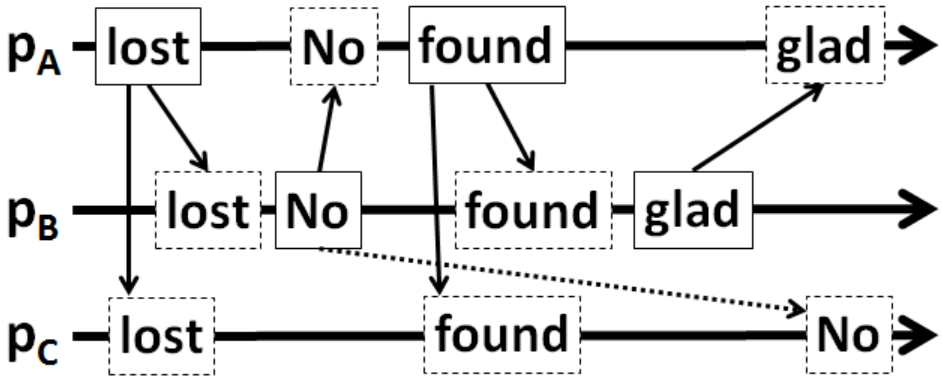

Figure 6: *Real time interaction between Alice, Bob and Calvin. In the figure, solid and dashed boxes represent write and read operations, respectively. Arrows denote the read-from relation. For simplicity, some operations are not shown in these figures.*

Figure 6 shows a scenario that satisfies intra-causal consistency (the sequence of events shown in Figure 6 for each user is the desired serialization). Note that in the scenario, Calvin observes the posts in the order of "**lost**", "**found**", and "**No**". This violates traditional causal consistency (Definition 1), because if we look at the interaction of Alice and Bob, then $w_B(x_{No})$**No** $\xrightarrow[\textbf{CC}]{}$ $w_A x_{found}$**found**; however, Calvin observes "**found**" before observing "**No**". On the contrary, such a scenario is allowed under intra-causal consistency, since $w_A(x_{found})$**found** does not *intra-causally depend* on $w_B(x_{No})$**No**, i.e., we do not have $w_A(x_{found})$**found** $\xrightarrow[\textbf{intra-CC}]{}$ $w_B(x_{No})$**No** using three rules defined for intra-causal consistency (Definition 4).

- Requiring each user to observe "**found**" before "**glad**":

Alice: "I've **lost** my wedding ring."
Bob: "Oh **No**!!!"
Alice: "Whew, **found** it upstairs!"
Bob: "I'm **glad** to hear that."

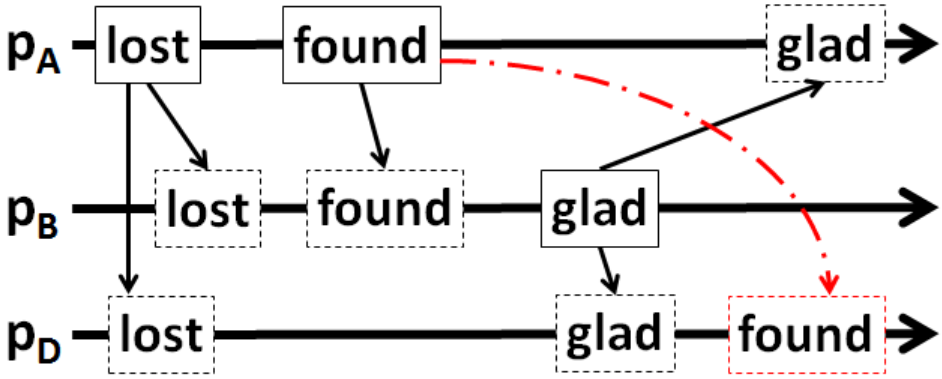

Figure 7: *Real time interaction between Alice, Bob and Darren. In the figure, solid and dashed boxes represent write and read operations, respectively. Arrows denote the read-from relation. For simplicity, some operations are not shown in these figures.*

In this example, we assume that Darren is also Alice's friend, and can read all posts on Alice's wall. Figure 7 shows an undesired result where Darren observes the posts in the order of "**lost**", "**glad**", and "**found**". As addressed in Section 1, this scenario is possible if the system only enforces eventual consistency. On the contrary, a system supporting intra-causal consistency would *prevent* this scenario from happening. Let H be the history shown in Figure 7. Observe that intra-process dependency graph D_D, which shows that $r_D(Alice\_wall)$**found** $\xrightarrow[\textbf{intra-CC}]{}$ $r_D(x_{Alice\_wall})$**glad** – this is due to the intra-program order specified by D_D. However, the timeline of Darren (in Figure 7) shows that Darren observes "**glad**" before observing "**found**". This violates the intra-causal order, and hence, the scenario shown in Figure 7 should not occur in a system ensuring intra-causal consistency.

4.2 Discussion

Construction of Intra-Process Dependency Graphs: In the discussion above, we have assumed that intra-process dependency graphs are given. However, in the real implementation, we

need to specify how to construct the intra-process dependency graphs in an online fashion, i.e., when an operation o is performed at a process i, which nodes should have edges to node o in D_i? Generally, there are two approaches:

- Application at a process i explicitly adds edges to node o from some of the nodes corresponding to previous operations, since the application has knowledge of the semantic
- We can also have a generic set of rules for adding edges for the given application. We will discuss some reasonable rules for Facebook-like application below. A complete set of rules is a future research direction. In the discussion below, let a *topic* depict a set of a single post and all the comments ensuing the post. Recall that posts and comments are defined in Section 1. For example, in Figure 1, "**lost**" is a post, and "**No**" is its comment. Moreover, "**lost**" and "**No**" form a topic, and "**found**" and "**glad**" form another topic. We also assume that given a history, we can infer the topic for each post or comment. In practice, this can be easily achieved by including a field representing the topic in the propagating message. Let us call operations corresponding to *posts* in the local history L_i as *post-operations*. Similarly, operations corresponding to *comments* is called *comment-operations*. Then, Facebook-like application should have some of the following rules:
  - If operation o is a comment-operation, then node o should have an edge from some other node corresponding to an operation (either comment- or post-operation) that belongs to the same topic.
  - If operation o is a post-operation, then node o should have an edge from some other node corresponding to some post-operation, unless o has no preceding operation.

  The first rule effectively says that for a comment, users care about the order between the comment and other post or comments that belong to the same topic, whereas the second rule says that users care about the order among post-operations, but not necessarily the order from comment-operation to post-operation if they do not belong to the same topic.

  Now, let us consider more complex application behavior: *add or remove friends*. Suppose in the social network application, only friends can access the posts or comments on a user's wall. This is typically implemented via a separate checking mechanism that implements *access control* at each replica. Suppose that user j wants to read the most recent update from other user i, then checking mechanism at user j's replica will check the friends graph (which is stored in each replica, as well), and propagate user i's updates to user j if users i and j are friends. We assume that friends graph is an undirected graph, i.e., friend relation is a mutual relation.

  When the add/remove friend operations are interleaved with read and write operations, we need to carefully design the dependency graph to reflect the intended semantic. Consider a simple example: *Alice first removes her boss Bob, and then posts that she wants a new job.* To ensure the expected application behavior, all later posts by Alice should not be observed by Bob after Alice removes Bob. This can be achieved by treating the "remove-Bob operation" as a *post-operation* addressed above. Due to the way that we constructed the intra-process dependency graph D_A, Alice's post looking for a job has a directed edge from "remove-Bob operation". Furthermore, by the *transitivity rule*, *intra-causal order* ensures that Bob would not be able to see any of Alice's later posts. This is because by the time when Bob's replica tries to make Alice's updates visible to Bob, the replica would have already received

"remove-Bob operation" due to the enforced intra-causal order and updated the friends graph accordingly. Therefore, since Bob is not Alice's friend anymore, the checking mechanism will disallow Bob to read Alice's updates. Add friend operation can be treated similarly.

Potential Implementation: We briefly discuss how to implement a geo-replicated storage system that supports *intra-causal consistency*. Our potential implementation is based on COPS [29, 30], which implements a scalable causal memory for geo-replication. COPS has the following two key components:

- When an user issues a write operation o_w, the user-side library propagate the value along with a *dependency tree* T_w that keeps track of all operations that precedes o_w defined by *causal order* to the closest available replica.
- When an user issues a read operation, the closest available replica returns the value of the most recent write o_w at the replica's local storage such that all of the operations in T_w have already been observed by the user.

Please refer to [29, 30] for other details regarding COPS, e.g., how to maintain or propagate the dependency tree efficiently.

One modification we need for our system is to construct the *dependency tree* according to *intra-causal order*. More precisely, for a write operation o_w, the *dependency tree* in our system will track all those operation that precedes o_w defined by *intra-causal order*.

5 Inter-Process Dependency

Our second approach takes into account **where** the events take place. The notion is related to the ideas proposed in prior work [17, 34, 23] – observe events "closer" to you in a more consistent way, and observe "farther" events in a less consistent way. Below, we use the notion of *inter-process dependency graph* to relax causal consistency to yield a model that we call *inter-causal consistency*. Note that such relaxation may be categorized as (CC, EC)-fisheye consistency (Causal Consistency and Eventual Consistency) in [17], but such a model is not discussed in [17]. Also, as we will show later, there are multiple different but reasonable ways to define *inter-causal consistency*. Moreover, we allow inter-process dependency graphs to be directed graphs, whereas [17] only considers undirected graphs. Thus, it is not clear how Friedman et al. [17] would define (CC, EC)-fisheye consistency.

Definition 5 (Inter-Process Dependency Graph) *A graph $G(\mathcal{V}, \mathcal{E})$ is an inter-process dependency graph if each node in $\mathcal{V}$ corresponds to an unique process in $\mathcal{P}$.*

Intuitively, given an application-specific parameter d, an inter-process dependency graph $G(\mathcal{V}, \mathcal{E})$ implies that the ordering of events at node $i \in \mathcal{V}$ is important to node $j \in \mathcal{V}$ if there exists a directed path of at most d-hop from i to j in G. Formally, an inter-process dependency graph G, a parameter d, and a history H together induce the *inter-causal order* (denoted $\xrightarrow[\mathbf{inter-CC}]{}$). For simplicity, we ignore the inter-dependency graphs and d in the notation. $o_1 \xrightarrow[\mathbf{inter-CC}]{} o_2$ if any of the following holds:

- **Program-order**: $o_1 \xrightarrow{L_i} o_2$ for some p_i (o_1 precedes o_2 in L_i),
- **Inter-reads-from**: $o_1 \xrightarrow[\textbf{read}]{} o_2$ **and** there exists a directed path with length at most d from $user(o_1)$ to $user(o_2)$ in the inter-process dependency graph G, where $user(o)$ denotes the user performing operation o (o_2 returns the value written by o_1 and $user(o_1)$ is d hops away from $user(o_2)$ in G), or
- **Transitivity**: there is some other operation o' such that $o_1 \xrightarrow[\textbf{intra-CC}]{} o' \xrightarrow[\textbf{intra-CC}]{} o_2$.

Definition 6 (Inter-Causal Consistency) *A history H is* inter-causally consistent *if for each $p_i \in \mathcal{P}$, there exists a serialization S_i of $H|(p_i + W)$ that respects $\xrightarrow[\textbf{inter-CC}]{H}$.*

5.1 Discussion

Other Inter-Process Graphs: Facebook-like application's semantic implies that if two users i, j are friends, then user i would want to get an update from user j, and vice versa. Thus, it is reasonable to use friend's graph – an undirected graph – as the inter-process dependency graph. However, for other kinds of social network applications like Twitter or Pinterest, the semantic is different. Twitter-like or Pinterest-like application adopts a subscription-based semantic: user i *follows* or *subscribes* user j if user i is interested in learning updates from j; however, user j may not necessarily want to learn user i's update if user j does not subscribe user i. Thus, it is more intuitive to use a directed graph to model such a subscription-based semantic. For Twitter-like application, we may use the *subscription graph* – which describes the subscription relations – as the intra-process dependency graph.

Other Definitions of Intra-Causal Order: We only discuss one way of using intra-process dependency graph above. We propose two other methods that may be useful in some applications:

- Different types of operations performed by the users may be assigned a different distance parameter d. Thus, for some type of operations, a causal ordering in a "larger universe" may be enforced, as compared to other operations. For instance, we may assign a larger universe size (i.e., larger d) to operations that change the friend relationships.
- We can also exploit the multiplicity of the directed paths. Denote by m the given multiplicity parameter. For example, consider the friends graph in Facebook-like application, and let $d = 1$. Then, it may be reasonable to define the new *reads-from rule* as follows:

  (i) $o_1 \xrightarrow[\textbf{read}]{} o_2$, (ii) $user(o_1)$ and $user(o_2)$ are friends or $user(o_1)$ and $user(o_2)$ share at least m common friends.

  This kind of rule is reasonable in Facebook-like application – even if $user(o_1)$ is not a neighbor of $user(o_2)$, $user(o_2)$ is still interested in learning $user(o_1)$'s update because they share enough common friends.

Potential Implementation: Our potential implementation is based on the implementation of causal shared memory presented in [4], which relies on using vector timestamps [25] to decide the most recent values. The algorithm is presented in [12]. How to make the implementation scalable for geo-replicated storage system is a future research direction.

6 Summary

This paper presents two methods to systematically weaken well-known consistency models. In particular, we discuss how to use two different kinds of *dependency graphs* to weaken causal consistency, and why such relaxed models yield performance benefit while still being useful for social network applications.

References

[1] Cassandra. `http://cassandra.apache.org/`.

[2] D. J. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. IEEE Computer, 45(2), 2012.

[3] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, Dec. 1996.

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

[5] H. Attiya, S. Chaudhuri, R. Friedman, and J. L. Welch. Shared memory consistency conditions for nonsequential execution: Definitions and programming strategies. *SIAM Journal on Computing*, 1:65–89, 1988.

[6] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley Series on Parallel and Distributed Computing, 2004.

[7] P. Bailis. Causality is expensive (and what to do about it), 05 Feb 2014. `http://www.bailis.org/blog/causality-is-expensive-and-what-to-do-about-it/`.

[8] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly Available Transactions: virtues and limitations. In *40th International Conference on Very Large Data Bases (VLDB)*, 2014.

[9] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 22:1–22:7, New York, NY, USA, 2012. ACM.

[10] 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*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[11] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[12] A. Benzer. Graph-based causal consistency. Bachelor thesis, University of Illinois at Urbana-Champaign, 2014.

[13] E. Brewer. A certain freedom: Thoughts on the CAP theorem. In *Proc. ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 335–335, 2010.

[14] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[15] 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. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[17] R. Friedman, M. Raynal, and F. Taïani. Fisheye consistency: Keeping data in synch in a georeplicated world. *CoRR*, abs/1411.6478, 2014.

[18] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 18(2SI):15–26, May 1990.

[19] S. Gilbert and N. A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In *SIGACT News*, volume 33, pages 51–59, 2002.

[20] C. Hale. You cannot sacrifice partition-tolerance, 07 October 2010. `http://codahale.com/you-cant-sacrifice-partition-tolerance/`.

[21] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[22] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. *SIGARCH Comput. Archit. News*, 20(2):13–21, Apr. 1992.

[23] B. Kemme, A. Schiper, G. Ramalingam, and M. Shapiro. Dagstuhl seminar review: Consistency in distributed systems. *SIGACT News*, 45(1):67–89, Mar. 2014.

[24] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[26] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979.

[27] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[28] R. Lipton and J. Sandberg. Pram: A scalable shared memory. Technical report, Princeton University, 1988.

[29] 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*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[30] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual consistency. *Queue*, 12(3):30:30–30:45, Mar. 2014.

[31] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[32] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. Technical report, Department of Computer Science, The University of Texas at Austin, May 2011.

[33] M. Perrin, A. Mostefaoui, and C. Jard. Update consistency for wait-free concurrent objects. *CoRR*, abs/1501.02165, 2015.

[34] N. Santos, L. Veiga, and P. Ferreira. Vector-field consistency for ad-hoc gaming. In *Middleware 2007, ACM/IFIP/USENIX 8th International Middleware Conference, Newport Beach, CA, USA, November 26-30, 2007, Proceedings*, pages 80–100, 2007.

[35] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *ACM Symposium on Operating Systems Principles*, 2013.

A Limitations of Eventual and Causal Consistency

This section discusses some limitations of eventual and causal consistency in the context of social network applications. We will refer to an example in Figure 1 in Section 1. Recall that in the scenario, there are three users, Alice, Bob and Calvin. Figure 1 shows Alice's wall, which contains the posts (by Alice) and comments (by Bob) and the corresponding timestamps. In the discussion, we will use the term *post* to refer to the text appearing first for a certain *topic*, and the term *comment* for the text ensuing some *posts*. For example, Alice's update on 9:00 is a post, and Bob's update on 9:05 is an ensuing comment. For the example in Figure 1, we will use "**lost**" to represent Alice's post at 9:00, "**no**" for Bob's comment at 9:05, "**found**" for Alice's post at 9:30, and "**glad**" for Bob's comment at 9:40, respectively.

If the system only enforces *eventual consistency* [1, 24], then due to variable communication delays, Calvin may observe "**lost**" and then "**glad**" before having observed "**found**". As a result, it will appear to Calvin that Bob is glad to hear that Alice lost her wedding ring! This may occur if delay in propagating "**found**" is large, as illustrated in Figure 8. In this figure, we ignore the

propagation of "**no**" for brevity. Note that, as discussed above, the users interact with each other through the replicas of the geo-replicated storage. For simplicity, these replicas are not shown in the figure, and we only show the outcome of the interactions. For instance, the reason Calvin may observe "**found**" before "**glad**" may be that the delay in propagating "**found**" to the replica accessed by Calvin is much larger than the delay in propagating "**glad**" to that replica. Recall that the geo-replicated storage system is built upon an asynchronous communication network, so such a scenario may occur.

If the system enforces *causal consistency* [4], then this unfortunate situation would never occur. Instead, as depicted in Figure 9, Calvin will not observe "**glad**" until he is able to observe "**found**" due to the enforced causal order [25]. The implementation of causally consistent storage system [29, 10] essentially achieves this desired outcome by requiring the replica accessed by Calvin to *delay* propagating "**glad**" to Calvin until the replica receives the update corresponding to "**found**" as shown in Figure 9. Consequently, this may result into unnecessary latency for some events due to the restrictive ordering constraint of causal consistency model. The discussion of first method in Section 1 illustrate this drawback using an example.

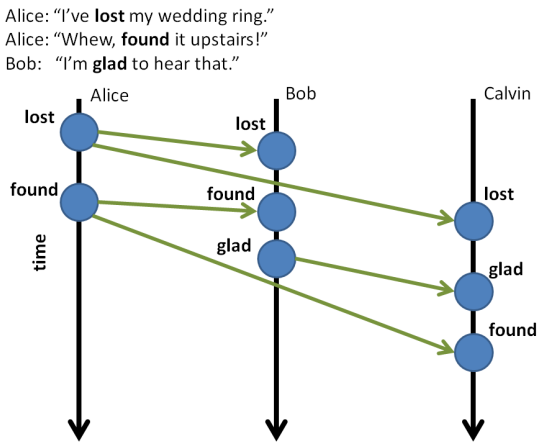

Figure 8: *Unfortunate outcome with eventual consistency [30].*

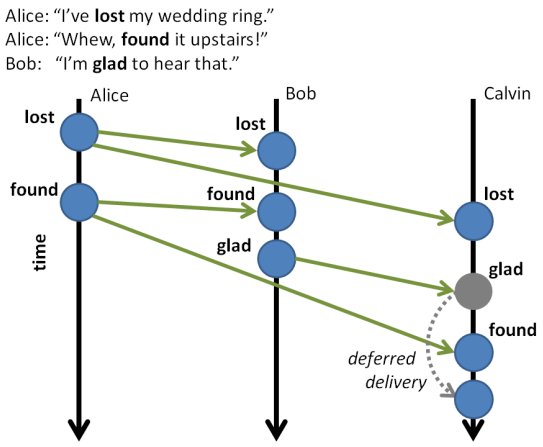

Figure 9: *Causal consistency achieves desired outcome [30].*

B Intra-Consistency Models

This section presents well-known consistency models and weakens them using intra-process dependency graphs. The weakened models are called intra-consistency models.

Total Order: We start with consistency models that require the existence of a *total order* among all processes.

- *Sequential Consistency*:

  Definition 7 (Sequential Consistency[26]) *A history H is* sequentially consistent *if there is a serialization S of H that respects the program order $\xrightarrow{L_i}$ for each $p_i \in \mathcal{P}$.*

Recall that we have defined intra-program order $\xrightarrow[\textbf{intra}]{L_i}$ in Section 4.

Definition 8 (Intra-Sequential Consistency) *Given an intra-process dependency graph, a history H is* intra-sequentially consistent *if there is a serialization S of H that respects the intra-program order $\xrightarrow[\textbf{intra}]{L_i}$ for each $p_i \in \mathcal{P}$.*

- *Linearizability*: The second model is linearizability, which impose a real time constraint. A history H induces a real time partial order $\xrightarrow[\textbf{RT}]{}$ as follows: whenever in H, the response of an operation o_1 *occurs before* the invocation of an operation o_2 in real time, then $o_1 \xrightarrow[\textbf{RT}]{} o_2$.

  Definition 9 (Linearizability [21]) *A history H is* linearizable *if there is a serialization S of H such that (i) S respects the program order $\xrightarrow{L_i}$ for each $p_i \in \mathcal{P}$; and (ii) S respects the real time partial order $\xrightarrow[\textbf{RT}]{}$.*

  Similarly, we can define the intra-real-time order $\xrightarrow[\textbf{intra}-\textbf{RT}]{}$ as follows:

  - For o_1 and o_2 that occur at two different processes in H, if the response of an operation o_1 *occurs before* the invocation of an operation o_2 in real time, $o_1 \xrightarrow[\textbf{intra}-\textbf{RT}]{} o_2$.

  Definition 10 (Intra-Linearizability) *Given an intra-process dependency graph, a history H is* intra-linearizable *if there is a serialization S of H such that (i) S respects $\xrightarrow[\textbf{intra}]{L_i}$ for each $p_i \in \mathcal{P}$; and (ii) S respects the intra-real-time order $\xrightarrow[\textbf{intra}-\textbf{RT}]{}$.*

  Intuitively, intra-linearizability requires (i) operations within the same process respect the intra-program order, and (ii) operations across different processes respect the real time partial order, whereas traditional linearizability requires all operations to respect the real time partial order. We believe such weakening is useful for the scenario of multi-thread executions at a single process.

Partial Order: Now, we consider models that only require a *partial order*. Note that causal consistency is one model that requires only partial order.

- *PRAM Consistency*:

  Definition 11 (PRAM Consistency [28]) *A history H is* PRAM consistent *if for each $p_i \in \mathcal{P}$, there exists a serialization S_i of $H|(p_i + W)$ that respects the program order $\xrightarrow{L_i}$.*

  Definition 12 (Intra-PRAM Consistency) *Given an intra-process dependency graph, a history H is* Intra-PRAM consistent *if for each $p_i \in \mathcal{P}$, there exists a serialization S_i of $H|(p_i + W)$ that respects $\xrightarrow[\textbf{intra}]{L_i}$.*