Limitations of Highly-Available Eventually-Consistent Data Stores

Hagit Attiya
Computer Science Department
Technion

Faith Ellen
Department of Computer Science
University of Toronto

Adam Morrison
Computer Science Department
Technion

ABSTRACT

Modern *replicated data stores* aim to provide high availability, by immediately responding to client requests, often by implementing objects that expose concurrency. Such objects, for example, multi-valued registers (MVRs), do not have sequential specifications. This paper explores a recent model for replicated data stores that can be used to precisely specify *causal consistency* for such objects, and liveness properties like *eventual consistency*, without revealing details of the underlying implementation. The model is used to prove the following results:

- An eventually consistent data store implementing MVRs cannot satisfy a consistency model strictly stronger than *observable causal consistency* ($\mathcal{OCC}$). $\mathcal{OCC}$ is a model somewhat stronger than causal consistency, which captures executions in which client observations can use causality to infer concurrency of operations. This result holds under certain assumptions about the data store.
- Under the same assumptions, an eventually consistent and causally consistent replicated data store must send messages of unbounded size: If s objects are supported by n replicas, then, for every $k > 1$, there is an execution in which an $\Omega(\min\{n, s\}k)$-bit message is sent.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Distributed Systems

Keywords

Replicated data store; causal consistency; eventual consistency

1. INTRODUCTION

Distributed data storage provides fault-tolerant access to objects by replicating them over a wide-area network. A *replicated data store* needs to balance between *availability* of data (i.e. accesses to data return without delay), and its *consistency*, while tolerating message delays. The *CAP theorem* [8,18] demonstrates the difficulty of achieving this balance, showing that strong *C*onsistency (i.e. *atomicity*) cannot be satisfied together with high *A*vailability and *P*artition tolerance.

One aspect of data consistency is a safety property restricting the possible values observed by clients accessing different replicas. The set of possible executions is called a *consistency model*. For example, *causal consistency* [2] ensures that the causes of an operation are visible at a replica no later than the operation itself. (A precise definition appears in Section 3.) A smaller set of possible executions means that there is less uncertainty about the data. So, one consistency model is *strictly stronger* than another if its executions are a proper subset of the executions of the other.

Some weak consistency models can be trivially satisfied by never updating the data. Therefore, another aspect of data consistency is a liveness property, ensuring that updates are applied at all replicas. The designers of many systems, e.g., Dynamo [13] and Cassandra [1], opt for a very weak liveness property called, somewhat confusingly, *eventual consistency* [5, 9, 10, 29]. Eventual consistency only ensures that each replica eventually observes all updates to the object, a property also referred to as *update propagation* [11].

Several data stores [4, 6, 16, 21, 22] satisfy causal consistency, motivated by the assumption that it is the strongest consistency model possible with eventual consistency. Towards resolving this assumption, this paper shows that an eventually consistent data store cannot satisfy a consistency model strictly stronger than *observable causal consistency* ($\mathcal{OCC}$). This is done within a precise model of replicated data stores, exposing assumptions and inherent costs.

A challenge in modeling replicated data stores is that many of them implement objects that cannot be specified sequentially. For example, in a *multi-value register* (MVR) [13], a read returns the set of values written by currently conflicting writes, i.e., writes that are not causally related. Similarly, an *observed-remove set* (ORset) [27] specifies that whenever a `remove` and an `add` of the same element are concurrent, the `add` "wins." By exposing concurrency in the specification, such objects are leaking low-level scheduling details to the client.

Specifying these objects independently of their implementation is particularly challenging in the context of causal consistency, whose definition relies on *visibility* of operations. Visibility of operations, in turn, depends on message sending and delivery, which are governed by the implemen-

PODC'15, July 21–23, 2015, Donostia-San Sebastián, Spain.

ACM 978-1-4503-3617-8 /15/07 ...$15.00.
http://dx.doi.org/10.1145/2767386.2767419.

tation. Following [10], we separate the *abstract execution*, as observed by clients of the replicated data store, from the *concrete execution*, which happens "under the hood" of the system.

This paper shows that a consistency model strictly stronger than $\mathcal{OCC}$ cannot be satisfied by an eventually consistent data store implementing MVRs. $\mathcal{OCC}$ captures executions in which client observations can use causality to infer concurrency of operations (see Section 5). This result holds under certain assumptions about the data store: read operations do not modify the state, and messages are generated only following an operation. Without these assumptions, a data store may trivially satisfy a consistency model stronger than $\mathcal{OCC}$ (and causal consistency) by ruling out some causally consistent executions. For example, a data store in which reads may modify replica state can prevent the execution in which replica R writes a value v to an MVR and then another replica R' reads this value, by not exposing v at R' until K additional operations have been applied at R'.

Our result is related to the *CAC theorem* [23,24], showing that natural causal *C*onsistency is the strongest model that can be satisfied together with high *A*vailability and one-way *C*onvergence. (One-way convergence is a stronger liveness property than eventual consistency.) Our proof, and the assumptions it exposes, show that the CAC theorem has an implicit model for the way implementations are structured. Counter-example implementations like the one mentioned above demonstrate that deviating from this structure allows a consistency model stronger than causal consistency to be satisfied. (A detailed comparison appears in Section 5.3.)

We also prove that causally consistent and eventually consistent stores implementing MVRs require messages of unbounded length. Specifically, if s objects are supported by n replicas, then, for every $k > 1$, there is an execution, whose length grows with k, in which an $\Omega(\min\{n, s\} \lg k)$-bit message is sent. This extends a result of Charron-Bost [12], showing that ordering $\Omega(n^2)$ events on n nodes using m-tuples (i.e. vector clocks [17,25]) requires $m \geq n$. In contrast, our result does not assume that messages have a particular format, and we show that, even for a fixed number of replicas and objects, the message length is unbounded.

2. MODEL

We model a highly-available replicated data store as a message-passing system, consisting of *replicas* that handle client operations on the replicated objects, and communicate the changes to each other with messages. We assume replicas *broadcast* their messages to all other replicas; replicas can implement point-to-point communication by ignoring messages for which they are not the intended recipient. Our model is mostly standard, with the following exception: Replicas handle client operations *immediately*, without communicating with other replicas. This models the crucial *high-availability* property of the data store—completion of a client operation must not depend on communication with another replica, so that the data store remains available despite network failures.

Replicas: We model a replica as a state machine $R = (\Sigma, \sigma_0, E, \Delta)$, where Σ is a set of *states*, $\sigma_0 \in \Sigma$ is the *initial state*, E is a set of possible *events*, and $\Delta : \Sigma \times E \to \Sigma$ is a (partial) *transition function*. We say that an event e is enabled in state σ if $\Delta(\sigma, e)$ is defined. A state transition is a local step of a replica, describing how the state of the replica changes in response to an enabled event. Three kinds of events model interactions of the replica with its clients and with other replicas:

- *do(o,op,v)*: a client of the replica invokes operation *op* on replicated object o and immediately receives response v from the replica,
- *send(m)*: the replica broadcasts message m, and
- *receive(m)*: the replica receives message m.

The *action*, $\mathrm{act}(e)$, of an event e specifies which kind of event (*do*, *send*, or *receive*) it is. Each *send(m)* and *receive(m)* event e has message attribute $\mathrm{msg}(e) = m$. Each *do(o,op,v)* event e has attributes $\mathrm{obj}(e) = o$, which is the object the client is operating on, $\mathrm{op}(e) = op$, which is the operation of the client, and $\mathrm{rval}(e) = v$, which is the response the client receives.

We assume that the content of a message sent by a replica is a deterministic function of its state: if $\Delta(\sigma, send(m_1)) = \sigma_1$ and $\Delta(\sigma, send(m_2)) = \sigma_2$, then $m_1 = m_2$ and hence, $\sigma_1 = \sigma_2$.

Executions: An *execution* is a (possibly infinite) sequence of events occurring at the replicas. It is an interleaving of a sequence of events occurring at each replica. We denote by $R(e)$ the replica at which an event e occurs. A (finite or infinite) sequence of events, $e_1, e_2, \ldots$, occurring at a replica $R = (\Sigma, \sigma_0, \Delta)$ is *well-formed* if there is a sequence of states, $\sigma_1, \sigma_2, \ldots$, such that $\sigma_i = \Delta(\sigma_{i-1}, e_i)$ for all $1 \leq i \leq$ the length of the sequence. If this sequence has length n, then σ_n is the *state of R at the end of the sequence.*

We consider only well-formed executions.

Definition 1. *An execution α is* well-formed *if for every replica $R = (\Sigma, \sigma_0, \Delta)$, (1) the subsequence of events at R, denoted $\alpha|_R$, is well-formed, and (2) for every receive(m) event e at R, a send(m) event at replica $R' \neq R$ precedes e in α. If α is finite then* the state of replica R at the end of *α is the state of R at the end of $\alpha|_R$.*

Well-formed executions only prohibit messages from appearing out of thin air or being received before they are sent. Messages may be dropped, delivered out of order, or delivered multiple times.

Recall the happens-before relation [20]:

Definition 2. *Let α be an execution. Event $e \in \alpha$* happens before *event $e' \in \alpha$ (written $e \xrightarrow{hb(\alpha)} e'$, or simply $e \xrightarrow{hb} e'$ if the context is clear) if one of the following conditions holds:*

(1) Thread of execution: *$R(e) = R(e')$ and e precedes e' in α.*

(2) Message delivery: *$e = send(m)$ and $e' = receive(m)$.*

(3) Transitivity: *There is an event $f \in \alpha$, such that $e \xrightarrow{hb} f$ and $f \xrightarrow{hb} e'$.*

Proposition 1. *Let α be an execution, and e an event in α. Then the following sequences of events are well-formed executions:*

1. *β, the subsequence of α consisting of all events e' such that $e \not\xrightarrow{hb} e'$.*

2. γ, the subsequence of α consisting of all events e' such that $e' \xrightarrow{hb} e$.

Further, for any replica R, $\beta|_R$ and $\gamma|_R$ are prefixes of $\alpha|_R$.

Network partitions: *Eventually consistent* data stores require that every operation *eventually* propagates to every other replica. To facilitate eventual consistency, we must require that the network is not partitioned from some point on. We assume that this cannot occur in *infinite* executions. Specifically, we consider only infinite executions in which the network is *sufficiently connected*, as defined below.

We say that a replica *has a message pending in state σ* if some event e with $\text{act}(e) = \textit{send}$ is enabled in σ. We assume that a replica does not have a message pending following a send event. In other words, a send event relays everything the replica has to send. Given an execution α and an event $e \in \alpha$, we say that a replica R *has a message pending in* e if R has a message pending in the state at the end of $\alpha'|_R$, where α' is the prefix of α ending with e.

DEFINITION 3. *The network is* sufficiently connected *in an infinite well-formed execution α if the following conditions hold:*

(1) Eventual transmission: *no replica has a message pending in all events of some (infinite) suffix of α, and*

(2) Eventual delivery: *if replica R sends message m, then every replica $R' \neq R$ eventually receives m.*

Thus, if a replica wants to send a message, i.e. it has a message pending in some state, then either it eventually broadcasts the message *or* it transitions into a state with no message pending without sending the message. This could happen, for example, if a client invokes an operation which cancels a previous operation that had not yet propagated to other replicas.

3. REASONING ABOUT REPLICATED DATA STORES

This section describes a framework for reasoning about highly-available replicated data stores, following Burckhardt et al. [10].

3.1 Specifying Replicated Objects

Replicated data stores are often used to implement objects which have some underlying notion of concurrency, and so cannot be specified sequentially. They generalize standard sequential specifications [19] as follows. A sequential specification of a deterministic object determines the return value of an operation from the sequence of prior operations applied to the object. However, with a replicated object, operations executing at different replicas might have inconsistent notions of "prior operations", e.g., because updates made on one side of a partition do not propagate to the other side. We therefore specify the return value of an operation based on operations that are *visible* to it: prior operations at the same replica, and remote operations whose effects have propagated to the replica through the network.

Visibility is defined in the context of an *abstract execution*. An abstract execution contains only the *histories* of operations invoked by the clients, i.e., only *do* events:

DEFINITION 4. *An* abstract execution *is a tuple (H, vis), where H is a sequence of do events and $vis \subseteq H \times H$ is an acyclic* visibility relation *(where $(e_1, e_2) \in vis$ is denoted by $e_1 \xrightarrow{vis} e_2$) that satisfies (1) if e_1 precedes e_2 in H and $R(e_1) = R(e_2)$, then $e_1 \xrightarrow{vis} e_2$, (2) if $e_1 \xrightarrow{vis} e_2$, e_2 precedes e_3 in H, and $R(e_2) = R(e_3)$, then $e_1 \xrightarrow{vis} e_3$, and (3) if $e_1 \xrightarrow{vis} e_2$ then e_1 precedes e_2 in H.*

We decouple visibility—when an operation becomes *observable* by the client—from the low-level happens-before relation (Definition 2), defined for receiving and sending messages between replicas, because having happens-before imply observability of an operation would place excessively strict demands on the data store. It would mean that receiving a message sent after an operation *op* makes *op* and *all its transitive dependencies* observable. This disallows, for example, buffering received information and not "exposing" it immediately, an implementation technique used in many causally consistent data stores to avoid sending the dependencies with each message [4,6,16,21,22] and to satisfy consistency conditions [11,15,26].

The reason for including both the visibility relation, vis, and a total order of the events, H, is that some specifications need a total order on events to resolve conflicts created by concurrent operations, i.e., those that are not related by visibility.

DEFINITION 5. *Let $A = (H, vis)$ be an abstract execution. An abstract execution $A' = (H', vis')$ is a* prefix *of A if (1) H' is a prefix of H and (2) $vis' = vis \cap (H' \times H')$. A set $\mathcal{S}$ of abstract executions is* prefixed-closed *if $A \in \mathcal{S}$ implies that $A' \in S$ for every prefix A' of A.*

An abstract execution $A = (H, vis)$ is *o-only* if $\text{obj}(e) = o$, for all $e \in H$.

DEFINITION 6. *A* replicated object specification *of an object o is a prefix-closed set $\mathcal{S}(o)$ of o-only abstract executions.*

The specification, $\mathcal{S}(o)$, of an object o is defined using the notion of an *operation context* [10]. The *operation context* of an operation on an object o consists of the prior operations on o that are visible to the operation:

DEFINITION 7. *Let $A = (H, vis)$ be an abstract execution, and $e \in H$. The* operation context of e in A *is the abstract execution $ctxt(A, e) = (H', vis \cap (V_e \times V_e))$, where $V_e = \left\{ e' \in H \mid e' \xrightarrow{vis} e \wedge obj(e') = obj(e) \right\} \cup \{e\}$, and H' is the subsequence of H containing only events in V_e. If $e' \in H$, we say that $e' \in ctxt(A, e)$ iff $e' \in H'$.*

We specify the return value, $rval(e)$, of a *do* event e applying an operation to object o as a function of its operation context, $ctxt(A, e)$:

$$\text{rval}(e) = f_o(ctxt(A, e)).$$

Figure 1 shows examples of such functions, including those of the *multi-value register* (MVR) and the *observed-remove set* (ORset). In an MVR, a read returns the set of values written by currently conflicting writes—i.e. writes that are not causally related. Similarly, an ORset specifies that whenever a `remove` and `add` are concurrent, the `add` "wins"—i.e., `remove` only removes items that it observes.

The specification $\mathcal{S}(o)$ contains abstract executions in which every operation's response is as specified above:

$$f_{\text{r/w reg}}(H', vis', e) = \begin{cases} \texttt{ok} & \text{if } \mathrm{op}(e)=\mathrm{write}(v) \\ v, \text{ where the last write event in } H' \text{ is } \mathrm{write}(v) & \text{if } \mathrm{op}(e)=\mathrm{read} \end{cases}$$

(a) Read/write register

$$f_{\text{MVR}}(H', vis', e) = \begin{cases} \texttt{ok} & \text{if } \mathrm{op}(e)=\mathrm{write}(v) \\ \{v \mid \exists\, e_1 \in H' \ \mathrm{op}(e_1) = \mathrm{write}(v) \ \wedge \neg\exists\, e_2 \in H' \ \mathrm{op}(e_2) = \mathrm{write}(\cdot) \wedge e_1 \xrightarrow{vis'} e_2\} & \text{if } \mathrm{op}(e)=\mathrm{read} \end{cases}$$

(b) Multi-valued register (MVR)

$$f_{\text{ORset}}(H', vis', e) = \begin{cases} \texttt{ok} & \text{if } \mathrm{op}(e)=\mathrm{add}(v) \\ \texttt{ok} & \text{if } \mathrm{op}(e)=\mathrm{remove}(v) \\ \{v \mid \exists\, e_1 \in H' \ \mathrm{op}(e_1) = \mathrm{add}(v) \ \wedge \neg\exists\, e_2 \in H' \ \mathrm{op}(e_2) = \mathrm{remove}(v) \wedge e_1 \xrightarrow{vis'} e_2\} & \text{if } \mathrm{op}(e)=\mathrm{read} \end{cases}$$

(c) Observed-remove set (ORset)

Figure 1: Functions specifying replicated objects: $ctxt(A, e) = (H', vis', e)$.

$\mathcal{S}(o) = \{\, A \mid A = (H, vis)$ is an o-only abstract execution, and $\forall e \in H, \mathrm{rval}(e) = f_o(ctxt(A, e))\,\}$

3.2 Consistency Models

Informally, a *safety property* ensures that nothing "bad" happens in an execution, and hence, the same is true for any of its prefixes. We are interested in safety properties of abstract executions, e.g. *correctness*, which says that all objects conform to their specification.

Definition 8. *An abstract execution $A = (H, vis)$ is* correct *if for every object o,*

$$A|_o = (H|_o, vis \cap (H|_o \times H|_o)) \in \mathcal{S}(o),$$

where $H|_o$ is the subsequence of H consisting of events e with $obj(e) = o$, and $\mathcal{S}(o)$ is the specification of o.

The definition of correctness, like other definitions soon to follow, aims to restrict the responses returned by a data store to high-level object operations, while leaving the low-level details of message transmissions unspecified. We can apply these definitions to a data store execution by considering the abstract execution in which the high-level operations occur in the same order on each replica and return the same responses:

Definition 9. *Execution α* complies with *an abstract execution $A = (H, vis)$ if for every replica R, $H|_R = \alpha|_R^{do}$, where $\alpha|_R^{do}$ denotes the subsequence of do events by replica R.*

Definition 10. *A data store D is* correct *if every execution α of D complies with a correct abstract execution.*

Henceforth, we consider only correct data stores.

We say that abstract executions $A = (H, vis)$ and $A' = (H', vis')$ are *equivalent*, or $A \equiv A'$, if for every replica R, $H|_R = H'|_R$. (Notice that if an execution α complies with A, then any abstract execution A' that α complies with it is equivalent to A.) A *consistency model* is a prefix-closed set of abstract executions that is closed under equivalence. It extends the concept of object specification to handle multiple objects. Analogously, *satisfying a consistency model* extends the concept of correctness for a data store:

Definition 11. *A data store D* satisfies *a consistency model $\mathcal{C}$ if every execution α of D complies with some $A \in \mathcal{C}$.*

Causal consistency guarantees that effects are visible only after their causes. In other words, if e_1 is visible to e_2 and e_2 is visible to e_3, then e_1 is visible to e_3:

Definition 12. *A correct abstract execution $A = (H, vis)$ is* causally consistent *if vis is transitive: if $e_1 \xrightarrow{vis} e_2$ and $e_2 \xrightarrow{vis} e_3$, then $e_1 \xrightarrow{vis} e_3$.*

Causal memory [2] can be seen as a special case of this definition, capturing only read/write registers. It has been widely adopted in distributed key/value stores [4, 6, 14, 16, 21, 22, 30], but extending to general, replicated data types seems to require using a framework like the one used here.

3.3 Eventual Consistency

Informally, a *liveness property* promises that something "good" eventually happens in an execution. The distinguishing feature of liveness is that it cannot be violated by any finite execution [3]—it is always possible that the required "good thing" will occur in the future. In our context, a "good thing" is an operation becoming visible.

Definition 13. *An infinite abstract execution $A = (H, vis)$ is* eventually consistent *if for every event $e \in H$, there are only finitely many events $e' \in H$ such that $obj(e') = obj(e)$ and $e \not\xrightarrow{vis} e'$.*

Observe that eventual consistency is a property of abstract executions because *visibility* is defined for abstract executions only. In other words, while our network model guarantees eventual message delivery, the delivery of a message about some operation does not guarantee that the data store will *choose* to make that operation visible. As with consistency models, we bridge the high-level and low-level worlds using compliance:

Definition 14. *A data store D is* eventually consistent *if, for every infinite execution α of D, every abstract execution that α complies with is eventually consistent.*

3.4 Exposing Concurrency in Specifications

Perrin et al. have shown that a replicated object can be implemented so that client operations appear to be totally

ordered (by breaking ties between concurrent operations in the same manner in every replica). They thus argue that replicated objects should be specified with sequential specifications [26]. Indeed, with a single object, concurrency can be "hidden," in the sense that the client cannot detect that concurrent operations exist and are being arbitrarily ordered.

Imagine that a data store implementing a single MVR object arbitrarily orders concurrent writes and exposes only one of them to the clients—in effect, implementing a read-/write register instead of an MVR. The clients cannot detect that this is happening, because there is always an MVR abstract execution consistent with their observations.

In contrast, with *multiple* objects, the combination of causal and eventual consistency allows clients to infer concurrency of operations even if the data store tries to "hide" it by ordering them. Figure 2 shows how the causal and eventual consistency requirements enable the client to infer concurrency of operations if multiple objects are supported. In the execution depicted, there are three MVRs shown as differently shaded circles. The data store must return $\perp$ as a response to r_z since $w_z \xrightarrow{hb} r_z$ (Proposition 2 in Section 4), and similarly for r_y. Suppose the data store orders w_x^1 before w_x^2, and thus reads from x return w_x^2. The client can still infer w_x^1 and w_x^2 are concurrent, because had w_x^1 been visible at R_2 when w_x^2 was executed, r_y should have returned w_y. (A symmetric argument applies if w_x^2 is ordered before w_x^1.)

Therefore, since we consider multiple objects and causal consistency, we use replicated object specifications that incorporate concurrency.

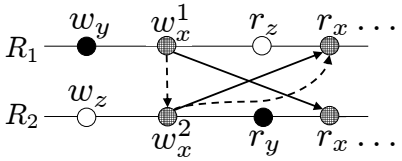


Figure 2: Causal links (dashed arrows) implied by r_x returning only the value of w_x^2 contradict information flow in messages (solid arrows), since they imply that r_y should return w_y's value.

4. WRITE-PROPAGATING DATA STORES

A *write-propagating* data store is an eventually consistent data store whose implementation (i.e., replica state machine) satisfies two properties, *invisible reads* and *op-driven messages*, defined next. Every highly-available replicated data storage system we are aware of [6,10,13,14,16,21,28,30] has these properties, when viewed in our model (i.e., ignoring issues such as timeouts for retransmitting dropped messages).

A data store has *op-driven messages* if it generates messages only as a result of a client operation, and not in response to a received message:

DEFINITION 15. *A data store D has* op-driven messages *if for any replica $R = (\Sigma, \sigma_0, E, \Delta)$, (1) R does not have a message pending in σ_0, and (2) if $\Delta(\sigma_1, receive(m)) = \sigma_2$ and R does not have a message pending in σ_1, then R does not have a message pending in σ_2.*

A data store has *invisible reads* if a client read operation does not change the state of the replica:

DEFINITION 16. *A data store D has* invisible reads *if for any replica $R = (\Sigma, \sigma_0, E, \Delta)$, if $\Delta(\sigma_1, e) = \sigma_2$ and $op(e) = read$, then $\sigma_1 = \sigma_2$.*

In this paper, we concentrate on data stores providing MVRs, as they expose concurrency and are used in real systems [10, 13, 21]. To simplify notation, we begin the names of events invoking write operations with w and begin the names of events invoking read operations with r. We assume that each write event writes a different value, so we do not distinguish between the event $w = do(o, write, v)$ and its value v; e.g., we say that $w \in \text{rval}(r)$, instead of saying that the value written by the event w is in $\text{rval}(r)$.

We proceed to prove several key properties of write-propagating data stores, which serve as building blocks for the proofs of our main results.

A basic property, which holds for *any* data store providing MVRs, is that if read r returns some write w, then w happens before r.

PROPOSITION 2. *Let D be a data store providing MVRs and α be an execution of D. Let $A = (H, vis)$ an abstract execution that α complies with. Let r be a $do(o, read, V)$ event and w be a $do(o, write, v)$ event such that $v \in V$. Then $w \xrightarrow{hb} r$.*

We show (in Corollary 4 below) that write-propagating data stores, i.e. with invisible reads and op-driven messages, satisfy the original definition of eventual consistency [29], i.e., that "if clients stop issuing update requests, then the replicas will eventually reach a consistent state." We first define *quiescent executions*, in which no message is "in flight."

DEFINITION 17. *A finite execution α is* quiescent *if for every replica R, (1) R does not have a message pending after its last event in α, and (2) every message R sends in α is received by every replica $R' \neq R$.*

Now we show that if the execution is quiescent, then all replicas are in a consistent state, in the sense that a client operation receives the same response regardless of the replica that executes it.

LEMMA 3. *Let D be an eventually consistent data store with invisible reads. Let α be a quiescent execution of D. If two reads of the same MVR at different replicas are appended to α, then both reads return the same response.*

Finally, we show that if D has op-driven messages, we can extend a finite execution α of D into a quiescent execution α' by (1) sending any message pending at any replica, and then (2) delivering each message in flight to every replica but its sender. There are no messages pending at the end of α', since D has op-driven messages. Therefore, α' is quiescent. This establishes the following corollary of Lemma 3.

COROLLARY 4. *Let D be an eventually consistent data store with invisible reads and op-driven messages. Let α be a finite execution of D. Then α can be extended to an execution in which, for any object o, a read of o that occurs at the end of the extended execution returns the same response no matter at which replica it is performed.*

In most real data store systems, a replica generates a message following a client write operation to propagate the write

to remote replicas. We show that a write-propagating data store *must*, in fact, do this, if the execution appears quiescent from the replica's perspective. To reason about individual messages that a replica receives in a given execution, we use the notation $e \xrightarrow{rcv} e'$ to denote that the first message sent by $R(e)$ after e is received by $R(e')$ before e'.

LEMMA 5. *Let D be an eventually consistent data store with invisible reads and op-driven messages. Let α be an execution of D whose last event is a write w at replica R. Suppose that in α, $e \xrightarrow{hb} w \implies e \xrightarrow{rcv} w$ for any do event e. Then R has a message pending following w.*

5. LIMITS OF SATISFIABLE CONSISTENCY MODELS

We explore the *strength* of consistency models satisfiable by eventually consistent replicated data stores. A consistency model $\mathcal{C}'$ is *stronger* than consistency model $\mathcal{C}$ if $\mathcal{C}' \subset \mathcal{C}$, that is, $\mathcal{C}'$ prohibits some abstract executions that $\mathcal{C}$ admits, but not vice versa. We show that a strengthening of causal consistency, *observable causal consistency* ($\mathcal{OCC}$), is the strongest consistency model satisfiable by a write-propagating data store implementing MVRs. To prove that no consistency model stronger than $\mathcal{C}$ is satisfied by a data store D, it suffices to show that, for every $A \in \mathcal{C}$, at least one execution α of D complies with A. (This is because any execution A' that α complies with is equivalent to A. Since $A \in \mathcal{C}$ and $\mathcal{C}$ is closed under equivalence, $A' \in \mathcal{C}$. Thus, it cannot be that D satisfies a stronger consistency model $\mathcal{C}' \subset \mathcal{C}$.)

5.1 Observable causal consistency

Intuitively, in an MVR abstract execution $A = (H, vis)$, vis specifies some information flow between replicas that dictates which prior writes are visible to each executed read, which dictates the return value of each read given this available information. Indeed, for write-propagating data stores, one can construct an execution in which, for each (non-transitive) vis edge (w, r) between events in distinct replicas, $R(w)$ transmits a message after w and $R(r)$ receives this message before r (Section 5.2). In other words, we can reconstruct the information flow given in any MVR abstract execution.

A data store wishing to avoid generating an execution that complies with $A = (H, vis)$ (e.g, to implement a stronger consistency model that does not include A) must return different responses for some of the reads in this execution. How can this be achieved? One way, for example, is to ignore the received information (for a while). However, a data store with invisible reads cannot do this (Section 5.3).

Another option is to "hide" concurrency. For example, if read r returns $\{w_0, w_1\}$ in A, the execution we construct will have w_0 and w_1 concurrent according to happens-before. The data store could still have r return just w_1, but it must make sure that the response it generates complies with *some* abstract execution. As Figure 3a shows, this means finding an abstract execution in which these writes are *ordered*. Intuitively, the data store is pretending that w_0 is visible to w_1 even though $w_0 \xrightarrow{hb} w_1$.

This kind of pretense has casuality implications, as demonstrated in Figure 3b. If $w_0 \xrightarrow{vis} w_1$, then w_1' should be visible to r' because of transitivity. However, r' does not know about w_1'. To remain correct, there has to be a write $\hat{w}$ visible to r' which can be artificially ordered after w_1', just like w_0 and w_1 were ordered.

Observable causal consistency ($\mathcal{OCC}$) contains abstract executions whose induced information flow is such that concurrent writes cannot be "hidden" in this way. $\mathcal{OCC}$ contains causally consistent abstract executions that expose concurrency between operations only when the clients can infer this concurrency from their observations, and so it cannot be hidden. $\mathcal{OCC}$ achieves this by requiring that whenever a read returns $\{w_0, w_1\}$, there must exist writes w_0' and w_1' to different objects, such that w_1' is visible to w_0 but not to w_1 (and w_0' is visible to w_1 but not to w_0), and any write $\hat{w}$ to $obj(w_1')$ that is visible to w_1 cannot be concurrent with w_1', i.e., must be visible to w_1' (again, symmetrically for w_0'). This guarantees that w_1 cannot be ordered after w_0 nor w_0 ordered after w_1, as shown in Figure 3c.

DEFINITION 18. *A causally consistent abstract execution $A = (H, vis)$ is* observably causally consistent *($\mathcal{OCC}$) if for any read r of some MVR o for which $rval(r) \supseteq \{w_0, w_1\}$, there exist w_0' and w_1' such that the following conditions hold:*

1. *w_i' is a write that is visible to w_{1-i} but to a different object than o: $w_i' \xrightarrow{vis} w_{i-1}$ and $obj(w_i') \neq o$,*
2. *w_0' and w_1' are writes to different objects: $obj(w_0') \neq obj(w_1')$,*
3. *w_i' is not visible to w_i: $w_i' \not\xrightarrow{vis} w_i$,*
4. *no write to $obj(w_i')$ occurring concurrently with w_i' is visible to w_i: for any $\hat{w}$, if $obj(\hat{w}) = obj(w_i')$ and $\hat{w} \xrightarrow{vis} w_i$, then $\hat{w} \xrightarrow{vis} w_i'$.*

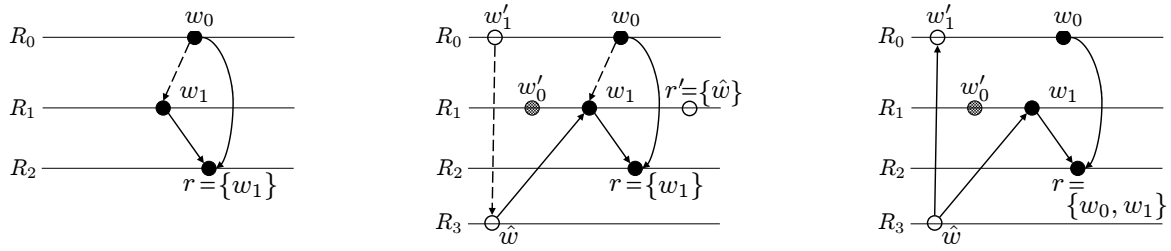
5.2 Impossibility of Stronger Consistency than $\mathcal{OCC}$

THEOREM 6. *Let D be an eventually consistent data store, providing MVRs, with invisible reads and op-driven messages. Then D does not satisfy a consistency model stronger than observable causal consistency ($\mathcal{OCC}$).*

To prove the theorem it suffices to show that for any abstract execution $A \in \mathcal{OCC}$, there is an execution α of D that complies with A. Thus, let $A = (H, vis) \in \mathcal{OCC}$ be an observably causally consistent execution. (In particular, A is causally consistent.) We will construct an execution α that complies with A. We obtain α by a recursive process, in which we deliver messages generated after events to ensure that $hb(\alpha) \subseteq vis$. The crux of the proof is to show that D cannot add "fake" visibility edges, and, hence, every read returns the same response in α as it does in A.

5.2.1 Revealing Executions

An MVR abstract execution $A = (H, vis)$ is *revealing*, if for every write operation $w \in H$, the operation $R(w)$ performs immediately before w is a read operation r_w of the same object and, for every $e \in H$, $r_w \xrightarrow{vis} e \implies w \xrightarrow{vis} e$ if $e \neq w$, and $e \xrightarrow{vis} w \implies e \xrightarrow{vis} r_w$ if $e \neq r_w$. That is, w and r_w are identical with respect to the visibility relation. Consequently, r_w "reveals" the state of the MVR when the write w is applied. This enables us to reason about the write operations that are visible to w, which is otherwise difficult. (In contrast, writes visible to a read can be inferred from the read's response.) The following lemma, proved in the full version, formalizes this intuition:



(a) If r returns only w_1, the data store effectively "pretends" that $w_0 \xrightarrow{vis} w_1$.

(b) $w_0 \xrightarrow{vis} w_1$ implies that w_1' should be visible to r'. Here, the data store can pretend that $w_1' \xrightarrow{vis} \hat{w}$ to make the return value of r' valid.

(c) An $\mathcal{OCC}$ execution prevents pretending that $w_1' \xrightarrow{vis} \hat{w}$. In turn, this prevents pretending that $w_0 \xrightarrow{vis} w_1$. Thus r is required to return $\{w_0, w_1\}$.

Figure 3: **Observable causal consistency:** motivation for definition. Circles of the same color correspond to operations on the same MVR object. Solid arrows represent $\xrightarrow{vis}$ edges induced by information flow in messages (i.e., are also $\xrightarrow{hb}$ edges). Dashed arrows represent additional $\xrightarrow{vis}$ edges required to make the execution comply with the MVR specification.

LEMMA 7. *Let $A = (H, vis)$ be a causally consistent revealing MVR abstract execution, $r \in H$ be a read operation in A, and $A' = (H', vis')$, be the operation context of r in A (Definition 7). Let β be an execution of D such that, for all $w \in H'$, $r_w, w \in \beta$. Then for any causally consistent MVR abstract execution $\hat{A} = (\hat{H}, \hat{vis})$ that β complies with, $w' \xrightarrow{vis} w \implies w' \xrightarrow{\hat{vis}} w$ for any $w', w \in H'$.*

Without loss of generality, we assume the $\mathcal{OCC}$ abstract execution A we consider in our proof is revealing. (Due to invisible reads, we can always obtain a revealing abstract execution A' by adding the appropriate r_w operations and vis edges to A, without affecting the response of existing reads in A. Then, for any execution α' of D that complies with A', the execution α obtained by removing the r_w operations from α' is an execution of D that complies with A.)

5.2.2 Recursive Construction of α

We construct the execution α by induction on the length of H. In the base case, $H = \varepsilon$, we take $\alpha = \varepsilon$. For the recursive step, let $H = H'\,e$, and let $R = R(e)$. That is, $op = \mathrm{op}(e)$ is the last high-level operation invoked in α, and it is invoked at replica R on the MVR $\mathrm{obj}(e)$. Let $A' = (H', vis')$ be a prefix of A. (By Definition 5, if $e_1 \xrightarrow{vis} e_2$, either $e_1 \xrightarrow{vis'} e_2$ or $e_2 = e$.) Let α' be the execution of D our construction returns for A'. We define α by extending α' in three steps:

(1) Message delivery. Consider the events e' such that $e' \xrightarrow{vis} e$, in the order they occur in H. Let $m_{e'}$ be the first message sent by $R(e')$ after e' in α', if such a message exists. If $m_{e'}$ is not delivered to R in α', we deliver it to R in α by appending a $receive(m_{e'})$ event at R.

(2) Invoking op(e). Let $\hat{e}$ be the transition in which $\mathrm{op}(e)$ is invoked at R on $\mathrm{obj}(e)$. We append $\hat{e}$. (The crux of the proof will be to show that $\hat{e} = e$, i.e., that $\mathrm{rval}(\hat{e}) = \mathrm{rval}(e)$.)

(3) Message sending. If R has a message m pending as a result of $\hat{e}$, we append a $send(m)$ event at R.

Before proving that $\hat{e} = e$, we state some properties of α.

PROPOSITION 8. *For any do events $e_0, e_1 \in \alpha$, $e_0 \xrightarrow{hb} e_1 \implies e_0 \xrightarrow{vis} \pi(e_1)$ and, if e_0 and e_1 are at different replicas, then $e_0 \xrightarrow{hb} e_1 \implies e_0 \xrightarrow{rcv} \pi(e_1)$, where $\pi(e_1) = e$ if $e_1 = \hat{e}$, or e_1 otherwise.*

Write operations have an even stronger property: if a write w is visible to some operation e' in A, then $w \xrightarrow{hb} e'$ in α:

PROPOSITION 9. *Let $w, e' \in H$. If $w \xrightarrow{vis} e'$ then $w \xrightarrow{hb} \pi^{-1}(e')$, where $\pi^{-1}(e') = \hat{e}$ if $e' = e$ or e' otherwise.*

5.2.3 α Complies with A

We prove by induction on A that the constructed execution α of D complies with A. The base case, $H = \varepsilon$, is vacuously true. For the inductive step, let $H = H'\,e$, $A' = (H', vis')$ be a prefix of A, and α' the output of our construction on A'. From the induction hypothesis, α' complies with A'. We must therefore prove that $\hat{e}$, the *do* event we invoke at replica R when extending α' to α, returns the same response as the event e in A. If $\mathrm{op}(e) = \mathrm{write}$, then $\mathrm{rval}(e) = \mathrm{ok}$ because the only response the MVR specification allows for a write is `ok`. Thus, because D is correct, it is also the case that $\mathrm{rval}(\hat{e}) = \mathrm{ok}$. The remaining case, when $\mathrm{op}(e) = \mathrm{read}$, follows from the next two lemmas.

LEMMA 10. *$w \in rval(\hat{e}) \implies w \in rval(e)$.*

PROOF. Suppose that $w \in \mathrm{rval}(\hat{e})$ but $w \notin \mathrm{rval}(e)$. Then $w \xrightarrow{hb} \hat{e}$ (Proposition 2) and thus $w \xrightarrow{vis} e$ (Proposition 8). But, as $w \notin \mathrm{rval}(e)$, there exists some $w_1 \in \mathrm{rval}(e)$ such that $w \xrightarrow{vis} w_1 \xrightarrow{vis} e$. Then $w_1 \xrightarrow{hb} \hat{e}$ by Proposition 9.

Let α_0 be the execution obtained by removing from α any event e' such that $e' \not\xrightarrow{hb} \hat{e}$. (By Proposition 1, α_0 is an execution of D.) Let $\alpha_1 = \alpha_0\,\hat{e}$. Then α_1 is an execution of D. R does not have a message pending at the end of α_1 (by construction of α and since D has invisible reads and op-driven messages). If $R' \neq R$ sends a message m in α_1, R receives m in α_1 (by Proposition 8). Let $\beta = \alpha_1\,\alpha_2$ be an execution of D obtained by appending *receive* events to α_1 in some arbitrary order, so that, in β every replica receives

every message sent by another replica in α_1. Then β is a quiescent execution, and R receives no messages in α_2, so its state at the of β is the same as its state at the end of α_1.

Now, consider the execution $\beta_\infty = \beta\,\hat{r}_1\,\hat{r}_2\,\ldots$ in which we append to β a sequence of read operations of the MVR obj($\hat{e}$) at R. Then β_∞ is an execution of D and $\hat{r}_i = \hat{e}$ for all $i \geq 1$ (as D has invisible reads). Since D is eventually consistent, β_∞ complies with some causally consistent MVR abstract execution $\hat{A} = (\hat{H}, \hat{vis})$ such that, for some $j \geq 1$, for any *do* event $e' \in \beta$, $e' \xrightarrow{\hat{vis}} \hat{r}_j$.

By Proposition 9, for any $w \in ctxt(A, e)$, $w \xrightarrow{hb} \hat{e}$, and so $r_w \xrightarrow{hb} \hat{e}$ (since $r_w \xrightarrow{hb} w$). Thus, $r_w, w \in \beta$ (and so $r_w, w \in \beta_\infty$). Since $w \xrightarrow{vis} w_1$, Lemma 7 implies that $w \xrightarrow{\hat{vis}} w_1$. Yet $w \in \text{rval}(\hat{r}_j) = \text{rval}(\hat{e})$, although $w \xrightarrow{\hat{vis}} w_1 \xrightarrow{\hat{vis}} \hat{r}_j$, which contradicts the MVR specification. □

LEMMA 11. $w_0 \in rval(e) \implies w_0 \in rval(\hat{e})$.

PROOF. Suppose that $w_0 \in \text{rval}(e)$ but $w_0 \notin \text{rval}(\hat{e})$. Let $\text{rval}(\hat{e}) = \{w_1, \ldots, w_k\}$. Observe that $w_0, w_1, \ldots, w_k \in ctxt(A, e)$. This follows because (1) for any $w' \in \text{rval}(e)$, $w' \xrightarrow{vis} e$ (MVR specification), (2) $w_0 \in \text{rval}(e)$ by definition, and (3) for $1 \leq i \leq k$, $w_i \in \text{rval}(e)$ by Lemma 10.

Now, for any $1 \leq i \leq k$, $\{w_0, w_i\} \subseteq \text{rval}(e)$, and $A \in \mathcal{OCC}$, so, by Definition 18, there exists a write w_i' by $R(w_0)$ preceding w_0 such that $\text{obj}(w_i') \neq \text{obj}(w_0)$ and $w_i' \not\xrightarrow{vis} w_i$. Notice that $w_i' \in ctxt(A, e)$ since $w_i' \xrightarrow{vis} w_0 \xrightarrow{vis} e$.

Let α_0 be the execution obtained by removing from α any event e' such that $e' \not\xrightarrow{hb} \hat{e}$. (By Proposition 1, α_0 is an execution of D.) Let $\alpha_1 = \alpha_0\,\hat{e}$. Then α_1 is an execution of D. R does not have a message pending at the end of α_1 (by construction of α and since D has invisible reads and op-driven messages). If $R' \neq R$ sends a message m in α_1, R receives m in α_1 (by Proposition 8). Let $\beta = \alpha_1\,\alpha_2$ be an execution of D obtained by appending *receive* events to α_1 in some arbitrary order, so that, in β every replica receives every message sent by another replica in α_1. Then β is a quiescent execution, and R receives no messages in α_2, so its state at the of β is the same as its state at the end of α_1.

Let β' be the execution obtained from β by adding a read r_i' of the MVR obj(w_i') at $R(w_i)$ immediately after w_i. (Since D has invisible reads, β' is a quiescent execution of D.)

Now, consider the execution $\beta_\infty = \beta'\,\hat{r}_1\,\hat{r}_2\,\ldots$ in which we append to β' a sequence of read operations of the MVR obj($\hat{e}$) at R. Then β_∞ is an execution of D and $\hat{r}_i = \hat{e}$ for all $i \geq 1$ (as D has invisible reads). Since D is eventually consistent, β_∞ complies with some causally consistent MVR abstract execution $\hat{A} = (\hat{H}, \hat{vis})$ such that for some $j \geq 1$, for any *do* event $e' \in \beta'$, $e' \xrightarrow{\hat{vis}} \hat{r}_j$.

Notice that for any $w \in ctxt(A, e)$, $w \xrightarrow{hb} \hat{e}$ (by Proposition 9), and so $r_w \xrightarrow{hb} \hat{e}$ (since $r_w \xrightarrow{hb} w$). Thus, $r_w, w \in ctxt(A, e) \implies r_w, w \in \beta \implies r_w, w \in \beta' \implies r_w, w \in \beta_\infty$. We can thus apply Lemma 7 to β_∞.

Since $R(w_i)$ does not receive any messages between w_i and r_i', $w_i' \xrightarrow{hb(\beta')} r_i' \iff w_i' \xrightarrow{hb(\beta)} w_i \iff w_i' \xrightarrow{hb} w_i$. However, $w_i' \in \text{rval}(r_i')$ implies $w_i' \xrightarrow{hb(\beta')} r_i'$ (by Proposition 2), which implies $w_i' \xrightarrow{hb} w_i$ and, thus, $w_i' \xrightarrow{vis} w_i$ (by Proposition 8), which is impossible since $w_i' \not\xrightarrow{vis} w_i$. Thus $w_i' \notin \text{rval}(r_i')$ for all $1 \leq i \leq k$.

Now, since $w_0 \xrightarrow{\hat{vis}} \hat{r}_j$ yet $w_0 \notin \text{rval}(\hat{r}_j) = \text{rval}(\hat{e})$, it must be that $w_0 \xrightarrow{\hat{vis}} w_i \xrightarrow{\hat{vis}} \hat{r}_j$ for some $1 \leq i \leq k$ (by the MVR specification). However, $w_i' \xrightarrow{\hat{vis}} w_0$ (from $w_i' \xrightarrow{vis} w_0$ and Lemma 7). Since $\hat{A}$ is causally consistent and $w_0 \xrightarrow{\hat{vis}} w_i$, $w_i' \xrightarrow{\hat{vis}} w_i \xrightarrow{\hat{vis}} r_i'$. Yet $w_i' \notin \text{rval}(r_i')$. Hence, there must exist a write $\hat{w}$ such that $w_i' \xrightarrow{\hat{vis}} \hat{w} \xrightarrow{\hat{vis}} r_i'$, for some $\hat{w} \in \text{rval}(r_i')$. Thus, $\hat{w} \xrightarrow{hb(\beta')} r_i'$ (Proposition 2), implying $\hat{w} \xrightarrow{hb(\beta)} r_i'$, which in turn implies $\hat{w} \xrightarrow{hb} w_i$ (as $R(w_i)$ receives no messages between w_i and r_i). Thus, $\hat{w} \xrightarrow{vis} w_i$ (Proposition 9).

Then $\hat{w} \xrightarrow{vis} w_i'$ by the definition of $\mathcal{OCC}$ (Definition 18). Lemma 7 implies that $\hat{w} \xrightarrow{\hat{vis}} w_i'$. But $w_i' \xrightarrow{\hat{vis}} \hat{w}$ (by definition of $\hat{w}$ above), which means that $\hat{vis}$ is cyclic—a contradiction. □

This implies that $\hat{e} = e$, proving Theorem 6.

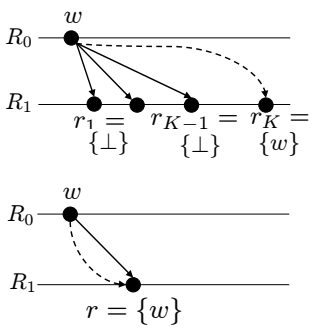
5.3 Comparison to the CAC Theorem

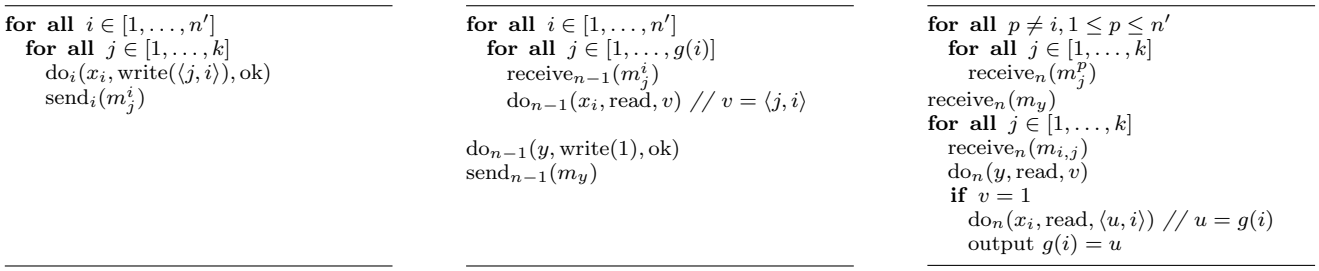
Mahajan, Alvisi and Dahlin's CAC theorem [23,24] shows that a certain class of data stores cannot implement a consistency model stronger than *natural causal consistency*, in which the causal order (i.e., visibility) does not violate the real-time order of operations. Respecting real-time order is a stronger demand than we make: interpreted in our framework, natural causal consistency requires that an execution complies with an abstract executions in which events occur in exactly the same order, as opposed to our requirement for identical *per-replica* order.

Our result applies to eventually consistent data stores, whereas the CAC theorem applies to *one-way convergent* data stores. These satisfy a stronger liveness property, essentially stating that any pair of replicas can converge to the same state with two steps of one-way communication. This difference is relevant in practice, as some systems weaken their liveness guarantee to satisfy stronger consistency than natural causal consistency—e.g., GSP, which globally orders write operations [9,11].

Both proofs construct an execution meant to comply with a specific abstract execution, by delivering messages according to the visibility relation. Both proofs require invisible reads; ours also requires op-driven messages.

Having invisible reads is inherently required for Theorem 6 and the CAC theorem to hold. Without invisible reads, the data store D can avoid producing certain causally consistent executions and thus satisfy a consistency model which is stronger than causal consistency (and $\mathcal{OCC}$). For example, suppose that D does not "expose" a write by returning it in response to a read until K read operations have been applied locally. This is illustrated in the top figure, where a *vis* edge is shown as a dashed arrow and an *hb* edge that is not a *vis* edge is shown as a solid arrow. Then D still satisfies eventual consistency, but no execution of D complies with the causally consistent abstract execution in which





(a) β (b) γ_g (c) d_i

Figure 4: Generating execution $\alpha_g = \beta\,\gamma_g$, which encodes g, and the transition sequence d_i, which decodes $g(i)$. We subscript an event with the id of its replica, e.g., do_i for a *do* event at R_i.

R_0 writes and R_1 immediately reads the value written (illustrated in the bottom figure). In contrast, we do not have an example of a data store without op-driven messages that satisfies a stronger consistency model than $\mathcal{OCC}$. Whether this assumption can be relaxed is an interesting question.

6. LOWER BOUND ON MESSAGE SIZE

We bound the size of messages used by write-propagating eventually consistent data stores. If s MVRs are supported by n replicas, then, for every $k > 1$, there is an execution in which an $\Omega(\min\{n,s\}k)$-bit message is sent. When $s \geq n$, this matches the complexity of the causal consistency algorithm of [2]. Messages in their algorithm contain vector timestamps [17, 25] of n components, each of which is logarithmic in the number of operations in the respective replica, yielding messages of size $O(nk)$, when there are 2^k operations. This leaves open the question of designing a data store using $O(sk)$-bit messages, when $s \in o(n)$.

THEOREM 12. *Let D be a causally consistent and eventually consistent data store with invisible reads and op-driven messages. Suppose that D provides s MVRs and has n replicas. Then for every $k \geq min\{n-2, s-1\}$ there is an execution α_k in which D sends a message containing $min\{n-2, s-1\} \lg k$ bits.*

PROOF. Let $n' = \min\{n-2, s-1\}$. For every $g : [n'] \mapsto [k]$, we construct an execution α_g that ends with a message m_g being sent. We prove that g can be decoded given m_g, which implies that for some g, m_g contains at least $n' \lg k$ bits. The basic idea is that, in α_g, each replica R_i, $1 \leq i \leq n'$, performs k writes, $w_1^i, \ldots, w_k^i$ to the MVR x_i. R_{n-1} performs a write w_y to the MVR y after observing the $g(i)$-th write of R_i, for every $1 \leq i \leq n'$. The message R_{n-1} subsequently broadcasts is m_g. We can decode g using m_g as follows. To output $g(i)$ for some i, we start with all replicas in their initial state, and execute the writes $w_1^i, \ldots, w_k^i$ for all i. (They are independent of g.) We deliver to R_n all messages but those sent by R_i, followed by m_g. Due to causal consistency, R_n cannot return w_y in response to a read of y, since w_y depends on $w_{g(i)}^i$, and R_n does not "know" this value. But if we now deliver R_i's messages to R_n, reading y after each message is received, the result of the read of y will contain w_y after the $g(i)$-th message has been delivered. Thus, we can find $g(i)$ given only m_g.

Now we describe the construction and decoding procedure more formally.

Construction of α_g: For any g, $\alpha_g = \beta\,\gamma_g$, where β is independent of g, and γ_g ends with R_{n-1} sending m_g. In β (Figure 4a), the sequence of events for each replica R_i consists of a write operation w_j^i writing $\langle j,i\rangle$ to MVR x_i followed by broadcasting a message m_i^j, for $1 \leq j \leq k$. (Lemma 5 implies each m_i^j exists.) In γ_g (Figure 4b), R_{n-1} receives, for each R_i, the messages $m_1^i, \ldots, m_{g(i)}^i$, while performing a read r_i^j of x_i after receiving each message. In the full version, we prove that $w_j^i \in \text{rval}(r_j^i)$. Finally, by Lemma 5, R_{n-1} writes 1 to register y and broadcasts m_g.

Decoding g from m_g (Figure 4c): Decoding $g(i)$ given m_g is performed by going through a sequence of transitions at R_n that ends with a $do(x_i, \text{read}, \langle g(i), i\rangle)$, which provides $g(i)$. We begin by having R_n receive the messages $m_1^p, \ldots, m_k^p$ for each $p \neq i$. (These messages can be obtained from β, which does not depend on g, so do not need to be provided explicitly.) We then obtain $g(i)$ by delivering m_j^i for j in increasing order, performing a read of y after each message delivery. If R_n reads 1, we read $\langle u,i\rangle$ from x_i and return $g(i) = u$. Let $d_i = e_1, \ldots, e_l$ be this sequence of transitions at R_n. To show that the execution $\alpha' = \beta\,\gamma_g\,d_i$ is well-formed, we prove in the full version that $\alpha'|R_n = d_i$. Thus, $g(i)$ can be computed given m_g, which implies g can be computed given m_g, and the theorem follows. □

One can prove Proposition 2, Lemma 3, and Lemma 5 for read/write registers (see Section 4), and these imply analogs of Theorem 12 for a data store providing read/write registers, as well as a combination of MVRs and registers.

7. SUMMARY

We present a framework for specifying highly-available replicated data stores, for objects that can expose concurrency. For write-propagating data stores, representing many real systems, we show that an eventually consistent data store cannot satisfy a consistency model strictly stronger than $\mathcal{OCC}$. An important open question is to implement an eventually consistent $\mathcal{OCC}$ data store, which will show that $\mathcal{OCC}$ is the strongest possible consistency model for eventually consistent data stores. We also prove that the data store must send messages whose size grows with the number of replicas and the number of operations.

In the full version of the paper, we also extend the space lower bounds proved by Burckhardt et al. [10] for replicas implementing certain objects, like MVRs and ORsets. While

their results seemingly imply the space optimality of existing implementations of these objects [7,27], they hinge on properties that many networks do not exhibit: *redelivery* and *reordering* of messages. We extend these lower bound proofs so that they do not require message redelivery and reordering. Thus, they hold even for relatively well-behaved networks that only delay or delete messages.

We have focused on MVRs, and left other objects, like read/write (last-writer wins) registers and ORsets, to future work. It would be interesting to determine whether Theorem 12 applies to ORsets, and more importantly, if Theorem 6 holds for other objects besides MVRs.

Acknowledgements: This work is supported by the Israel Science Foundation (grants 1227/10 and 1749/14), by Yad HaNadiv foundation, and by the Natural Science and Engineering Research Council of Canada. Faith Ellen was supported in part at the Technion by a Lady David Fellowship. Adam Morrison was supported in part at the Technion by an Aly Kaufman Fellowship.

8. REFERENCES

[1] *The Apache Cassandra Project.*

[2] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: definitions, implementation, and programming. *Dist. Comp.*, 9(1), 1995.

[3] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21:181–185, October 1985.

[4] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The Potential Dangers of Causal Consistency and an Explicit Solution. In *SoCC '12*, pages 22:1–22:7.

[5] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Commun. ACM*, 56(5):55–63, May 2013.

[6] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on Causal Consistency. In *SIGMOD '13*, pages 761–772.

[7] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An Optimized Conflict-free Replicated Set. Technical Report RR-8083, INRIA, Oct. 2012.

[8] E. A. Brewer. Towards robust distributed systems. In *PODC '00*, page 7.

[9] S. Burckhardt. *Principles of Eventual Consistency.* Found. and Trends in Programming Languages, 2014.

[10] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. In *POPL '14*, pages 271–284.

[11] S. Burckhardt, D. Leijen, J. Protzenko, and M. Fähndrich. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In *ECOOP '15*.

[12] B. Charron-Bost. Concerning the Size of Logical Clocks in Distributed Systems. *Inf. Process. Lett.*, 39(1):11–16, July 1991.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP '07*, pages 205–220.

[14] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *SOCC '13*, pages 11:1–11:14.

[15] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *SoCC '14*, pages 4:1–4:13.

[16] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Closing The Performance Gap between Causal Consistency and Eventual Consistency. In *Proceedings of the 1st Workshop on Principles and Practice of Eventual Consistency*, 2014.

[17] C. J. Fidge. Partial orders for parallel debugging. *ACM SIGPLAN Notices*, 24(1):183–194, 1989.

[18] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[19] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12:463–492, July 1990.

[20] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[21] 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 *SOSP '11*, pages 401–416.

[22] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *NSDI '13*, pages 313–328.

[23] P. Mahajan. *Highly Available Storage with Minimal Trust.* PhD thesis, The University Of Texas at Austin, Austin, TX, USA, May 2012.

[24] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, Availability, and Convergence. Technical Report TR-11-22, The University of Texas at Austin, 2011.

[25] F. Mattern. Virtual time and global states of distributed systems. In M. Consard and P. Quinton, editors, *Parallel and Distributed Algorithm*, pages 215–226. North-Holland, 1989.

[26] M. Perrin, A. Mostéfaoui, and C. Jard. Brief Announcement: Update Consistency in Partitionable Systems. In *DISC '14*, pages 546–547.

[27] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report RR-7506, INRIA, Jan. 2011.

[28] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. In *SSS '11*, pages 386–400.

[29] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009.

[30] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. Preguiça. SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. Technical Report RR-8347, INRIA, Aug. 2013.