Deterministic Threshold Queries of Distributed Data Structures

Lindsey Kuper and Ryan R. Newton

Indiana University
{lkuper, rrnewton}@cs.indiana.edu

Abstract. Convergent replicated data types, or CvRDTs, are lattice-based data structures for enforcing the eventual consistency of replicated objects in a distributed system. Although CvRDTs are provably eventually consistent, queries of CvRDTs nevertheless allow inconsistent intermediate states of replicas to be observed; and although in practice, many systems allow a mix of eventually consistent and strongly consistent queries, CvRDTs only support the former. Taking inspiration from our previous work on *LVars* for deterministic parallel programming, we show how to extend CvRDTs to support deterministic, strongly consistent queries using a mechanism called *threshold queries*. The threshold query technique generalizes to any lattice, and hence any CvRDT, and allows deterministic observations to be made of replicated objects before the replicas' states have converged.

Keywords: Replication, eventual consistency, strong consistency, lattices, determinism

1 Introduction

Distributed systems typically involve *replication* of data objects across a number of physical locations. Replication is of fundamental importance in such systems: it makes them more robust to data loss and allows for good data locality. But the well-known *CAP theorem* [6,1] of distributed computing imposes a trade-off between *consistency*, in which every replica sees the same data, and *availability*, in which all data is available for both reading and writing by all replicas. *Highly available* distributed systems, such as Amazon's Dynamo key-value store [5], relax strong consistency in favor of *eventual consistency* [15], in which replicas need not agree at all times. Instead, updates execute at a particular replica and are sent to other replicas later. All updates eventually reach all replicas, albeit possibly in different orders. Informally speaking, eventual consistency says that if updates stop arriving, all replicas will *eventually* come to agree.

Although giving up on strong consistency makes it possible for a distributed system to offer high availability, even an eventually consistent system must have some way of resolving conflicts between replicas that differ. One approach is to try to determine which replica was written most recently, then declare that replica the winner. But, even in the presence of a way to reliably synchronize

clocks between replicas and hence reliably determine which replica was written most recently, having the last write win might not make sense from a *semantic* point of view. For instance, if a replicated object represents a set, then, depending on the application, the appropriate way to resolve a conflict between two replicas could be to take the set union of the replicas' contents. Such a conflict resolution policy might be more appropriate than a "last write wins" policy for, say, a object representing the contents of customer shopping carts for an online store [5].

1.1 Convergent replicated data types and eventual consistency

Implementing application-specific conflict resolution policies in an ad-hoc way for every application is tedious and error-prone.[1] Fortunately, we need not implement them in an ad-hoc way. Shapiro *et al.*'s *convergent replicated data types* (CvRDTs) [13,12] provide a simple mathematical framework for reasoning about and enforcing the eventual consistency of replicated objects, based on viewing replica states as elements of a lattice and replica conflict resolution as the lattice's join operation.

In a CvRDT, the contents of a replica can only grow over time—that is, updates must be *inflationary* with respect to the given lattice—and replicas merge with remote replicas by taking the join of the remote state and the local state. CvRDTs offer a simple and theoretically-sound approach to eventual consistency. Still, even with CvRDTs, it is always possible to observe inconsistent *intermediate* states of replicated shared objects, and high availability requires that reads return a value immediately, even if that value is stale.

1.2 Strong consistency at the query level

In practice, applications call for both strong consistency and high availability at different times [14], and increasingly, they support consistency choices at the granularity of individual queries, not that of the entire system. For example, the Amazon SimpleDB database service gives customers the choice between eventually consistent and strongly consistent read operations on a per-read basis [16].

Ordinarily, strong consistency is a global property: all replicas agree on the data. When we make consistency choices at a *per-query* granularity, though, a global strong consistency property need not hold. We define a *strongly consistent query* to be one that, if it returns a result x when executed at a replica i:

- will always return x on subsequent executions at i, and
- will *eventually* return x when executed at *any* replica, and will *block* until it does so.

That is, a strongly consistent query of a distributed data structure, if it returns, will return a result that is a *deterministic* function of all updates to the data structure in the entire distributed execution, regardless of when the query executes or which replica it occurs on.

[1] Indeed, as the developers of Dynamo have noted [5], Amazon's shopping cart presents an anomaly whereby removed items may re-appear in the cart!

1.3 Our contribution: bringing threshold queries to CvRDTs

As they are today, CvRDTs only support eventually consistent queries. We could get strong consistency by waiting until all replicas agree before allowing a query to return—but in practice, such agreement may never happen. In this paper, we offer an alternative approach to supporting strongly consistent queries that takes advantage of the existing lattice structure of CvRDTs and does not require waiting for all replicas to agree.

To do so we take inspiration from our previous work [7,10,9] on *LVars*, or lattice-based data structures for shared-memory deterministic parallelism. Like CvRDTs, LVars are data structures whose states are elements of an application-specific lattice, and whose contents can only grow with respect to the given lattice. Unlike CvRDTs, though, LVars make it impossible to observe the *order* of updates to their state. This is because LVar read operations are *threshold* reads: an attempt to read will block until the data structure's contents reach or surpass a particular "threshold" (which we explain in more detail in Section 2), and then return a deterministic result. The combination of inflationary writes and threshold reads allow LVars to serve as the basis for a *deterministic-by-construction* shared-memory parallel programming model: concurrent programs in which all shared data structures are LVars are guaranteed to produce a deterministic outcome on every run, regardless of parallel execution and schedule nondeterminism.

Although LVars and CvRDTs were developed independently, both models leverage the mathematical properties of join-semilattices to ensure that a property of the model holds—determinism in the case of LVars; eventual consistency in the case of CvRDTs. Our contribution in this paper is to bring LVar-style *threshold queries* to CvRDTs and show that threshold queries of CvRDTs are strongly consistent queries, according to the criteria given above. After reviewing the fundamentals of threshold queries (Section 2) and CvRDTs (Section 3), we introduce CvRDTs extended with threshold queries (Section 4) and prove that threshold queries in our extended model are strongly consistent queries (Section 5). That is, we show that a threshold query that returns an answer when executed on a replica will return the same answer every subsequent time that it is executed on that replica, and that executing that threshold query on a different replica will eventually return the same answer, and will block until it does so. It is therefore impossible to observe different results from the same threshold query, whether at different times on the same replica, or whether on different replicas.

A preliminary version of some of the material in this paper appeared in a non-archival workshop [8].

2 How threshold queries work

A *threshold query* is a way of reading the contents of a lattice-based data structure that allows only limited observations of the state of the data structure. It

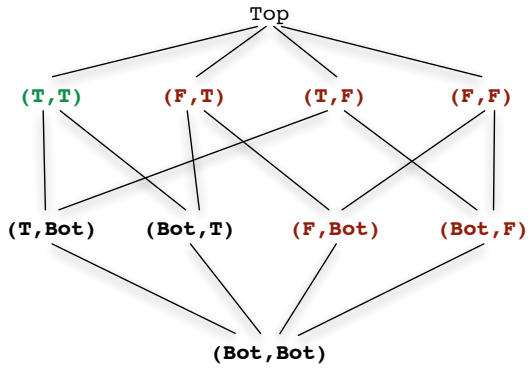

Fig. 1. Lattice of states that an `AndLV` can take on. The five red states in the lattice correspond to a false result, and the one green to a true one.

only returns a value when the data structure's state meets a certain (monotonic) criterion, or "threshold", and the value returned is the same regardless of how far above the threshold the data structure's state goes.

By a *lattice-based data structure*, we mean a data structure whose possible states are elements of a lattice[2], and that can only change over time in a way that is inflationary with respect to the lattice. Both CvRDTs [13,12] and LVars [7,10] are examples of lattice-based data structures. In this section, we illustrate threshold queries with an example before formally describing them in the context of CvRDTs.

2.1 An example: parallel "and"

Consider a lattice-based data structure that stores the result of a parallel logical "and" operation. We will call this data structure an *AndLV*. For this example, we assume two inputs, called "left" and "right", each of which may be either `T` or `F`. To understand this example, it is sufficient to consider a single replica; in Section 4 we will go on to discuss replication and communication between replicas.

We can represent the states an `AndLV` can take on as pairs `(x,y)`, where each of `x` and `y` are `T`, `F`, or `Bot`. The `Bot` value, short for "bottom", means that no input has yet been received, and so `(Bot,Bot)` is the least element of the lattice of states that our `AndLV` can take on, shown in Figure 1. An additional state, which

[2] We use "lattice" as a shorthand; formally, the "lattice" of states is a 4-tuple $(D, \sqsubseteq, \bot, \top)$, where D is a set, $\sqsubseteq$ is a partial order on D, $\bot$ is D's least element according to $\sqsubseteq$, $\top$ is D's greatest element, and every two elements in D have a least upper bound. Hence $(D, \sqsubseteq, \bot, \top)$ is really a *bounded join-semilattice* with a designated greatest element of $\top$.

we call `Top`, is the greatest element of the lattice; it represents the situation in which an error has occurred—if, for instance, one of the inputs writes `T` and then later changes its mind to `F`.

The result of the parallel "and" computation—if it completes successfully—will be either `True` or `False`, but it might also block indefinitely if not enough writes occur (say, if the left input is `(T,Bot)` and the right input never arrives), or it could end in the error state, `Top`, if conflicting writes from the same input occur. (Each update to the lattice takes the form of a complete state, such as `(T,Bot)` to write the left input.)

The lattice induces a *join*, or least upper bound, operation on pairs of states; for instance, the join of `(T,Bot)` and `(Bot,F)` is `(T,F)`, and the join of `(T,Bot)` and `(F,Bot)` is `Top` since the overlapping `T` and `F` values conflict. Importantly, whenever a write occurs, it updates the `AndLV`'s state to the join of the incoming state and the current state. (With CvRDTs, this join is computed when replicas merge. In this example, though, we are dealing with only one replica, and updating to the join of the new state and the current state is sufficient to ensure that all writes are inflationary.)

2.2 Threshold queries of an `AndLV`

Given that we want the result of a threshold query to be a deterministic function of the writes to a data structure—and not of the particular moment in time the query is made—what sorts of observations are safe to make of an `AndLV`? We cannot, for instance, test whether one or both inputs have been written at a particular point in time, because the result will depend on how the test is interleaved with the writes.

Instead, we can describe a *partial function* of the data structure's state that is undefined for all states except those that are at or above a certain *threshold* in the lattice. One way to describe such a function is to define a *threshold set* of sets of "activation" states. In the case of our `AndLV`, one of these sets of activation states is the set of states containing an `F`—that is, the set {`(F,Bot)`, `(Bot,F)`, `(F,T)`, `(T,F)`, `(F,F)`}. The elements of this set are shown in red in Figure 1. The other set of activation states is the singleton set {`(T,T)`}, shown in green in Figure 1. Therefore the entire threshold set is

$$\{\{\texttt{(F,Bot)}, \texttt{(Bot,F)}, \texttt{(F,T)}, \texttt{(T,F)}, \texttt{(F,F)}\}, \{\texttt{(T,T)}\}\}$$

The semantics of a threshold query is as follows: if a data structure's state reaches (or surpasses) any state or states in a particular set of activation states, the threshold query returns *the entire set* of activation states, regardless of which of those activation states was reached. If no state in any set of activation states has yet been reached, the threshold query will *block*; blocking corresponds to states on which the partial function is undefined.

In the case of our `AndLV`, as soon as either input writes an `F`, our threshold query will unblock and return the first set of activation states, that is, {`(F,Bot)`, `(Bot,F)`, `(F,T)`, `(T,F)`, `(F,F)`}. Hence `AndLV` has the expected "short-circuit" behavior and does not have to wait for a second input if the first input

is `F`. If, on the other hand, both inputs write a `T`, the threshold query will unblock and return {`(T,T)`}.

In a real implementation, of course, the value returned from the query could be more meaningful to the client—for instance, we could return `False` instead of returning the set of activation states. However, doing so would only be a convenience, and the translation from {`(F,Bot)`, `(Bot,F)`, `(F,T)`, `(T,F)`, `(F,F)`} to `False` could just as easily take place on the client side. In either case, the result returned from the threshold query is the same regardless of *which* of the activation states caused it to unblock, and it is impossible for the client to tell whether the actual state of the lattice is, say, `(T,F)` or `(F,F)` or some other state containing `F`.

2.3 Pairwise incompatibility

In order for the behavior of a threshold query to be deterministic, it must be unblocked by a *unique* set of activation states in the threshold set. We ensure this by requiring that elements in a set of activation states must be *pairwise incompatible* with elements in every other set of activation states. That is, for all distinct sets of activation states Q and R in a given threshold set: $\forall q \in Q.\ \forall r \in R.\ q \sqcup r = \top$. In our `AndLV` example, there are two distinct sets of activation states, so if we let $Q = \{$`(T,T)`$\}$ and $R = \{$`(F,Bot)`, `(Bot,F)`, `(F,T)`, `(T,F)`, `(F,F)`$\}$, the least upper bound of `(T,T)` and r must be `Top`, where r is any element of R. We can easily verify that this is the case. Furthermore, since the lattice of states that an `AndLV` can take on is finite, the join function can be verified to compute a least upper bound.

Why is pairwise incompatibility necessary? Consider the following (illegal) "threshold set" that does not meet the pairwise incompatibility criterion:

$$\{\{\texttt{(F,Bot)}, \texttt{(Bot,F)}\}, \{\texttt{(T,Bot)}, \texttt{(Bot,T)}\}\}$$

A threshold query corresponding to this so-called threshold set will unblock and return {`(F,Bot)`, `(Bot,F)`} as soon as a state containing an `F` is reached, and {`(T,Bot)`, `(Bot,T)`} as soon as a state containing a `T` is reached. The trouble with such a threshold query is that there exist states, such as `(F,T)` and `(T,F)`, that could unblock either set of activation states. If the left input writes `F` and the right input writes `T`, and these writes occur in arbitrary order, then the threshold query will return a nondeterministic result, depending on the order in which the two writes arrive. But with the original, pairwise-incompatible threshold set we showed, the threshold query would deterministically return `False`—although if the `T` arrived first, the query would have to block until the `F` arrived, whereas if the `F` arrived first, it could unblock immediately. Hence threshold queries enforce consistency at the expense of availability, but it is still possible to do a "short-circuit" computation that unblocks as soon as an `F` is written.

The threshold set mechanism we describe in this section is part of the LVars programming model discussed in section 1; in fact, our `AndLV` example is precisely an LVar [9]. But the utility of threshold queries is not limited to LVars. In the

following sections, we review the basics of the CvRDT model from the work of Shapiro *et al.*, then show how to add threshold queries to the CvRDT model, and prove that they are strongly consistent queries.

2.4 Discussion: the model versus reality

The use of explicit threshold sets should be understood as a mathematical modeling technique, *not* an implementation approach or practical API. To put it another way, read operations on a data structure exposed as an LVar, or queries of a CvRDT extended with threshold reads, must have the *semantic effect* of a threshold, but threshold sets need not be visible to clients, or even written explicitly in the code. Rather, the authors of data structure libraries can use unsafe (but efficient and scalable) operations (such as those provided by our own LVars library implementation [10]) to implement their own library internals, and then make functions that perform threshold queries available as a safe interface for application writers, implicitly baking in the particular threshold sets that make sense for a given data structure without ever explicitly constructing them. Any such data structure API that has the semantics of a threshold CvRDT (which we describe in detail in Section 4) is guaranteed to provide the determinism property that we state and prove in Section 5.

Furthermore, although our `AndLV` example shows a finite threshold set for querying a finite lattice, since threshold queries (and lattices themselves) are a semantic modeling tool with no runtime representation, our model accommodates infinite threshold sets (for querying infinite lattices) as easily as it accommodates finite ones; indeed, infinite threshold sets and infinite lattices (for instance, representing the states that an integer counter can take on, or a shopping cart) are typical in practice.

3 Background: CvRDTs and eventual consistency

Shapiro *et al.* [13,12] define an *eventually consistent* object as one that meets three conditions. One of these conditions is the property of *convergence*: all correct replicas of an object at which the same updates have been delivered eventually have equivalent state. The other two conditions are *eventual delivery*, meaning that all replicas receive all update messages, and *termination*, meaning that all method executions terminate (we discuss methods in more detail below).

Shapiro *et al.* further define a *strongly eventually consistent* (SEC) object as one that is eventually consistent and, in addition to being merely convergent, is *strongly convergent*, meaning that correct replicas at which the same updates have been delivered have equivalent state.[3] A *conflict-free replicated data type* (CRDT), then, is a data type (*i.e.*, a specification for an object) satisfying certain

[3] Strong eventual consistency is not to be confused with strong consistency: it is the combination of eventual consistency and strong convergence. Contrast with ordinary convergence, in which replicas only *eventually* have equivalent state. In a strongly convergent object, knowing that the same updates have been delivered to all correct

conditions that are sufficient to guarantee that the object is SEC. (The term "CRDT" is used interchangeably to mean a specification for an object, or an object meeting that specification.)

There are two "styles" of specifying a CRDT: *state-based*, also known as *convergent*[4]; or *operation-based* (or "op-based"), also known as *commutative*. CRDTs specified in the state-based style are called *convergent replicated data types*, abbreviated *CvRDTs*, while those specified in the op-based style are called *commutative replicated data types*, abbreviated *CmRDTs*. Of the two styles, we focus on the CvRDT style in this paper because CvRDTs are lattice-based data structures and therefore amenable to threshold queries described in Section 2—although, as Shapiro *et al.* show, CmRDTs can emulate CvRDTs and vice versa.

3.1 State-based objects

The Shapiro *et al.* model specifies a *state-based object* as a tuple (S, s^0, q, u, m), where S is a set of states, s^0 is the initial state, q is the *query method*, u is the *update method*, and m is the *merge method*. Objects are replicated across some finite number of processes, with one replica at each process, and each replica begins in the initial state s^0. The state of a local replica may be queried via the method q and updated via the method u. Methods execute locally, at a single replica, but the merge method m can merge the state from a remote replica with the local replica. The model assumes that each replica sends its state to the other replicas infinitely often, and that eventually every update reaches every replica, whether directly or indirectly.

The assumption that replicas send their state to one another "infinitely often" refers not to the *frequency* of these state transmissions; rather; it says that, regardless of what event (such as an update, via the u method) occurs at a replica, a state transmission is guaranteed to occur after that event. We can therefore conclude that all updates eventually reach all replicas in a state-based object, meeting the "eventual delivery" condition discussed above. However, we still have no guarantee of strong convergence or even convergence. This is where Shapiro *et al.*'s notion of a CvRDT comes in: a state-based object that meets the criteria for a CvRDT is guaranteed to have the strong-convergence property.

A *state-based* or *convergent* replicated data type (CvRDT) is a state-based object equipped with a partial order $\leq$, written as a tuple $(S, \leq, s^0, q, u, m)$, that has the following properties:

- S forms a join-semilattice ordered by $\leq$.
- The merge method m computes the join of two states with respect to $\leq$.
- State is *inflationary* across updates: if u updates a state s to s', then $s \leq s'$.

replicas is sufficient to ensure that those replicas have equivalent state, whereas in an object that is merely convergent, there might be some further delay before all replicas agree.

[4] There is a potentially misleading terminology overlap here: the definitions of convergence and strong convergence above pertain not only to CvRDTs (where the C stands for "Convergent"), but to *all* CRDTs.

Shapiro *et al.* show that a state-based object that meets the criteria for a CvRDT is strongly convergent. Therefore, given the eventual delivery guarantee that all state-based objects have, and given an additional assumption that all method executions terminate, a state-based object that meets the criteria for a CvRDT is SEC [13].

3.2 Discussion: the need for inflationary updates

Although CvRDT updates are required to be inflationary, we note that it is not clear that inflationary updates are necessarily required for convergence. Consider, for example, a scenario in which replicas 1 and 2 both have the state $\{a, b\}$. Replica 1 updates its state to $\{a\}$, a non-inflationary update, and then sends its updated state to replica 2. Replica 2 merges the received state $\{a\}$ with $\{a, b\}$, and its state remains $\{a, b\}$. Then replica 2 sends its state back to replica 1; replica 1 merges $\{a, b\}$ with $\{a\}$, and its state becomes $\{a, b\}$. The non-inflationary update has been lost, and was, perhaps, nonsensical—but the replicas are nevertheless convergent.

However, once we introduce threshold queries of CvRDTs, as we will do in the following section, inflationary updates become *necessary* for the determinism of threshold queries. This is because a non-inflationary update could cause a threshold query that had been unblocked to block again, and so arbitrary interleaving of non-inflationary writes and threshold queries would lead to nondeterministic behavior. Therefore the requirement that updates be inflationary will not only be sensible, but actually crucial.

4 Adding threshold queries to CvRDTs

In Shapiro *et al.*'s CvRDT model, the query operation q reads the exact contents of its local replica, and therefore different replicas may see different states at the same time, if not all updates have been propagated yet. That is, it is possible to observe intermediate states of a CvRDT replica. Such intermediate observations are not possible with threshold queries. In this section, we show how to extend the CvRDT model to accommodate threshold queries.

4.1 Objects with threshold queries

Definition 1 extends Shapiro *et al.*'s definition of a state-based object with a threshold query method t:

Definition 1 (state-based object with threshold queries). *A* state-based object with threshold queries *(henceforth* object*) is a tuple* (S, s^0, q, t, u, m)*, where* S *is a set of states,* $s^0 \in S$ *is the initial state,* q *is a* query method*,* t *is a* threshold query method*,* u *is an* update method*, and* m *is a* merge method*.*

In order to give a semantics to the threshold query method t, we need to formally define the notion of a threshold set described in Section 2:

Definition 2 (threshold set). *A* threshold set with respect to a lattice $(S, \leq)$ *is a set* $\mathcal{S} = \{S_a, S_b, \dots\}$ *of one or more sets of* activation states*, where each set of activation states is a subset of* S*, the set of lattice elements, and where the following* pairwise incompatibility *property holds:*

For all $S_a, S_b \in \mathcal{S}$*, if* $S_a \neq S_b$*, then for all activation states* $s_a \in S_a$ *and for all activation states* $s_b \in S_b$*,* $s_a \sqcup s_b = \top$*, where* $\sqcup$ *is the join operation induced by* $\leq$ *and* $\top$ *is the greatest element of* $(S, \leq)$*.*

In our model, we assume a finite set of n processes $p_1, \dots, p_n$, and consider a single replicated object with one replica at each process, with replica i at process p_i. Processes may crash silently; we say that a non-crashed process is *correct*.

Every replica has initial state s^0. Methods execute at individual replicas, possibly updating that replica's state. The kth method execution at replica i is written $f_i^k(a)$, where k is ≥ 1 and f is either q, t, u, or m, and a is the arguments to f, if any. Methods execute sequentially at each replica. The state of replica i after the kth method execution at i is s_i^k. We say that states s and s' are equivalent, written $s \equiv s'$, if $q(s) = q(s')$.

4.2 Causal histories

An object's *causal history* is a record of all the updates that have happened at all replicas. The causal history does not track the order in which updates happened, merely that they did happen. The *causal history at replica i after execution k* is the set of all updates that have happened at replica i after execution k. Definition 3 updates Shapiro *et al.*'s definition of causal history for a state-based object to account for t (a trivial change, since execution of t does not change a replica's causal history):

Definition 3 (causal history). *A* causal history *is a sequence* $[c_1, \dots, c_n]$*, where* c_i *is a set of the updates that have occurred at replica* i*. Each* c_i *is initially* $\emptyset$*. If the* k*th method execution at replica* i *is:*

- *a query* q *or a threshold query* t*, then the causal history at replica* i *after execution* k *does not change:* $c_i^k = c_i^{k-1}$*.*
- *an update* $u_i^k(a)$*: then the causal history at replica* i *after execution* k *is* $c_i^k = c_i^{k-1} \cup u_i^k(a)$*.*
- *a merge* $m_i^k(s_{i'}^{k'})$*: then the causal history at replica* i *after execution* k *is the union of the local and remote histories:* $c_i^k = c_i^{k-1} \cup c_{i'}^{k'}$*.*

We say that an update is *delivered at replica i* if it is in the causal history at replica i.

4.3 Threshold CvRDTs and the semantics of blocking

With the previous definitions in place, we can give the definition of a CvRDT that supports threshold queries:

Definition 4 (CvRDT with threshold queries). *A* convergent replicated data type with threshold queries *(henceforth* threshold CvRDT*) is an object equipped with a partial order $\leq$, written $(S, \leq, s^0, q, t, u, m)$, that has the following properties:*

- *S forms a join-semilattice ordered by $\leq$.*
- *S has a greatest element $\top$ according to $\leq$.*
- *The query method q takes no arguments and returns the local state.*
- *The threshold query method t takes a threshold set $\mathcal{S}$ as its argument, and has the following semantics: let $t_i^{k+1}(\mathcal{S})$ be the $k+1$th method execution at replica i, where $k \geq 0$. If, for some activation state s_a in some (unique) set of activation states $S_a \in \mathcal{S}$, the condition $s_a \leq s_i^k$ is met, $t_i^{k+1}(\mathcal{S})$ returns the set of activation states S_a. Otherwise, $t_i^{k+1}(\mathcal{S})$ returns the distinguished value* block*.*
- *The update method u takes a state as argument and updates the local state to it.*
- *State is* inflationary *across updates: if u updates a state s to s', then $s \leq s'$.*
- *The merge method m takes a remote state as its argument, computes the join of the remote state and the local state with respect to $\leq$, and updates the local state to the result.*

and the q, t, u, and m methods have no side effects other than those listed above.

We use the block return value to model t's "blocking" behavior as a mathematical function with no intrinsic notion of running duration. When we say that a call to t "blocks", we mean that it immediately returns block, and when we say that a call to t "unblocks", we mean that it returns a set of activation states S_a.

Modeling blocking as a distinguished value introduces a new complication: we lose determinism, because a call to t at a particular replica may return either block or a set of activation states S_a, depending on the replica's state at the time it is called. However, we can conceal this nondeterminism with an additional layer over the nondeterministic API exposed by t. This additional layer simply *polls* t, calling it repeatedly until it returns a value other than block. Calls to t at a replica that are made by this "polling layer" count as method executions at that replica, and are arbitrarily interleaved with other method executions at the replica, including updates and merges. The polling layer itself need not do any computation other than checking to see whether t returns block or something else; in particular, the polling layer does not need to compare activation states to replica states, since that comparison is done by t itself.

The set of activation states S_a that a call to t returns when it unblocks is unique because of the pairwise incompatibility property described in Definition 2. The intuition behind pairwise incompatibility is that described in Section 2.3: without it, different orderings of updates could allow the same threshold query to unblock in different ways, introducing nondeterminism that would be observable beyond the polling layer.

4.4 Threshold CvRDTs are strongly eventually consistent

We can define eventual consistency and strong eventual consistency exactly as Shapiro *et al.* do in their model. In the following definitions, a *correct replica* is a replica at a correct process, and the symbol $\Diamond$ means "eventually":

Definition 5 (eventual consistency (EC)). *An object is* eventually consistent *(EC) if the following three conditions hold:*

- Eventual delivery*: An update delivered at some correct replica is eventually delivered at all correct replicas:* $\forall i, j : f \in c_i \implies \Diamond f \in c_j$.
- Convergence*: Correct replicas at which the same updates have been delivered eventually have equivalent state:* $\forall i, j : c_i = c_j \implies \Diamond s_i \equiv s_j$.
- Termination*: All method executions halt.*

Definition 6 (strong eventual consistency (SEC)). *An object is* strongly eventually consistent *(SEC) if it is eventually consistent and the following condition holds:*

- Strong convergence*: Correct replicas at which the same updates have been delivered have equivalent state:* $\forall i, j : c_i = c_j \implies s_i \equiv s_j$.

Since we model blocking threshold queries with block, we need not be concerned with threshold queries not necessarily terminating. Determinism does *not* rule out queries that return block every time they are called (and would therefore cause the polling layer to block forever). However, we guarantee that if a threshold query returns block every time it is called during a complete run of the system, it will do so on *every* run of the system, regardless of scheduling. That is, it is not possible for a query to cause the polling layer to block forever on some runs, but not on others.

Finally, we can directly leverage Shapiro *et al.*'s SEC result for CvRDTs to show that a threshold CvRDT is SEC:

Theorem 1 (strong eventual consistency of threshold CvRDTs). *Assuming eventual delivery and termination, an object that meets the criteria for a threshold CvRDT is SEC.*

Proof. From Shapiro *et al.*, we have that an object that meets the criteria for a CvRDT is SEC [13]. Shapiro *et al.*'s proof also assumes that eventual delivery and termination hold for the object, and proves that strong convergence holds — that is, that given causal histories c_i and c_j for respective replicas i and j, that their states s_i and s_j are equivalent. The proof relies on the commutativity of the least upper bound operation. Since, according to our Definition 3, threshold queries do not affect causal history, we can leverage Shapiro *et al.*'s result to say that a threshold CvRDT is also SEC. □

5 Determinism of threshold queries

Neither eventual consistency nor strong eventual consistency imply that *intermediate* results of the same query q on different replicas of a threshold CvRDT will be deterministic. For deterministic intermediate results, we must use the threshold query method t. We can show that t is deterministic *without* requiring that the same updates have been delivered at the replicas in question at the time that t runs.

In our previous work on LVars [7,10], we showed that a threshold read of the contents of a shared-memory lattice-based data structure returns a deterministic result regardless of how that query is interleaved with updates to the data structure. Theorem 2 establishes that the same is true for threshold queries of *distributed, replicated* lattice-based data structures—namely, threshold CvRDTs.

Theorem 2 (determinism of threshold queries). *Suppose a given threshold query t on a given threshold CvRDT returns a set of activation states S_a when executed at a replica i. Then, assuming eventual delivery and that no replica's state is ever $\top$ at any point in the execution:*

1. *t will always return S_a on subsequent executions at i, and*
2. *t will* eventually *return S_a when executed at* any *replica, and will* block *until it does so.*

Proof. The proof relies on transitivity of $\leq$ and eventual delivery of updates. See Appendix A for the complete proof.

Although Theorem 2 must assume eventual delivery, it does *not* need to assume strong convergence or even ordinary convergence. It so happens that we have strong convergence as part of strong eventual consistency of threshold CvRDTs (by Theorem 1), but we do not need it to prove Theorem 2. In particular, there is no need for replicas to have the same state in order to return the same result from a particular threshold query. The replicas merely both need to be above an activation state from a unique set of activation states in the query's threshold set. Indeed, the replicas' states may in fact trigger *different* activation states from the same set of activation states.

Theorem 2's requirement that no replica's state is ever $\top$ rules out situations in which replicas disagree in a way that cannot be resolved normally. This would happen if, for instance, updates at two different replicas took their states to activation states from two distinct sets of activation states from the same threshold set. The join of the two replicas' states would be $\top$, which by eventual delivery would become the state of both replicas, but in the meantime, threshold queries of the two replicas would return different results. We rule out this situation by assuming that no replica's state goes to $\top$. (In a replicated version of the parallel "and" example from Section 2, this would happen if, for instance, one replica received a write of `T` on its left input while another received a write of `F`, also on its left input—a disagreement for which the only resolution is the error state, `Top`).

6 Related work

The extended CvRDT model we present in this paper is based on Shapiro *et al.*'s work on conflict-free replicated data types [13,12], discussed in Section 3. Various other authors [2,3,4] have used lattices as a framework for establishing formal guarantees about eventually consistent systems and distributed programs. Burckhardt *et al.* [2] propose a formalism for eventual consistency based on graphs called *revision diagrams*. Burckhardt and Leijen [3] show that revision diagrams are semilattices, and Leijen, Burckhardt, and Fahndrich apply the revision diagrams approach to guaranteed-deterministic concurrent functional programming [11]. Conway *et al.*'s Bloom^L language for distributed programming leverages the lattice-based semantics of CvRDTs to guarantee confluence [4]. The concept of threshold queries comes from our previous work on the LVars model for lattice-based deterministic parallel programming [7,10,9].

As mentioned in Section 1, database services such as Amazon's SimpleDB [16] allow for both eventually consistent and strongly consistent reads, chosen at a per-query granularity. Terry *et al.*'s Pileus key-value store [14] takes the idea of mixing consistency levels further: instead of requiring the application developer to choose the consistency level of a particular query at development time, the system allows the developer to specify a service-level agreement that can dynamically adapt to changing network conditions, for instance. However, we are not aware of previous work on using lattice-based data structures as a foundation for both eventually consistent and strongly consistent queries.

7 Conclusion

In this paper we show how to extend CvRDTs with support for deterministic queries. Borrowing from our previous work on LVars for deterministic parallel programming, we propose *threshold queries*, which, rather than returning the exact contents of a CvRDT, only reveal whether the contents have crossed a given "threshold" in the CvRDT's lattice, blocking until the threshold is reached and then returning a deterministic result. We prove that a threshold query that returns a given result is guaranteed to return that result on subsequent queries of the same replica, regardless of the replica's state. Moreover, executions of the same query on other replicas are guaranteed to eventually return the same answer, and to block until they do so. The technique generalizes to any lattice, and hence any CvRDT, allowing CvRDTs to support both eventually consistent and strongly consistent queries without waiting for all replicas to agree.

Since real applications call for a combination of strongly consistent and eventually consistent queries, threshold queries offer a way to support a mix of consistency choices within a single, lattice-based reasoning framework. Furthermore, since threshold queries behave deterministically regardless of whether all replicas agree, they suggest a way to save on synchronization costs: existing operations that require all replicas to agree could be done with threshold queries instead, and retain behavior that is *observably* strongly consistent while avoiding unnecessary synchronization.

References

1. Brewer, E.: CAP twelve years later: How the "rules" have changed. http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed (2012)
2. Burckhardt, S., Fahndrich, M., Leijen, D., Sagiv, M.: Eventually consistent transactions. In: ESOP (2012)
3. Burckhardt, S., Leijen, D.: Semantics of concurrent revisions. In: ESOP (2011)
4. Conway, N., Marczak, W., Alvaro, P., Hellerstein, J.M., Maier, D.: Logic and lattices for distributed programming. In: SOCC (2012)
5. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: SOSP (2007)
6. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2) (Jun 2002)
7. Kuper, L., Newton, R.R.: LVars: lattice-based data structures for deterministic parallelism. In: FHPC (2013)
8. Kuper, L., Newton, R.R.: Joining forces: Toward a unified account of LVars and convergent replicated data types. In: Workshop on Deterministic and Correctness in Parallel Programming (WoDet'14) (2014)
9. Kuper, L., Todd, A., Tobin-Hochstadt, S., Newton, R.R.: Taming the parallel effect zoo: Extensible deterministic parallelism with LVish. In: PLDI (2014)
10. Kuper, L., Turon, A., Krishnaswami, N.R., Newton, R.R.: Freeze after writing: Quasi-deterministic parallel programming with LVars. In: POPL (2014)
11. Leijen, D., Fahndrich, M., Burckhardt, S.: Prettier concurrency: purely functional concurrent revisions. In: Haskell (2011)
12. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Tech. Rep. RR-7506, INRIA (Jan 2011)
13. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: SSS (2011)
14. Terry, D.B., Prabhakaran, V., Kotla, R., Balakrishnan, M., Aguilera, M.K., Abu-Libdeh, H.: Consistency-based service level agreements for cloud storage. In: SOSP (2013)
15. Vogels, W.: Eventually consistent. Commun. ACM 52(1) (Jan 2009)
16. Vogels, W.: Choosing consistency. http://www.allthingsdistributed.com/2010/02/strong_consistency_simpledb.html (2010)

A Proof of Theorem 2

Theorem 2 (determinism of threshold queries). *Suppose a given threshold query t on a given threshold CvRDT returns a set of activation states S_a when executed at a replica i. Then, assuming eventual delivery and that no replica's state is ever $\top$ at any point in the execution:*

1. *t will always return S_a on subsequent executions at i, and*
2. *t will* eventually *return S_a when executed at* any *replica, and will* block *until it does so.*

Proof. Consider replica i of a threshold CvRDT $(S, \leq, s^0, q, t, u, m)$. Let $\mathcal{S}$ be a threshold set with respect to $(S, \leq)$. Consider a method execution $t_i^{k+1}(\mathcal{S})$ (*i.e.*, a threshold query that is the $k+1$th method execution on replica i, with threshold set $\mathcal{S}$ as its argument) that returns some set of activation states $S_a \in \mathcal{S}$.

For part 1 of the theorem, we have to show that threshold queries with $\mathcal{S}$ as their argument will always return S_a on subsequent executions at i. That is, we have to show that, for all $k' > (k+1)$, the threshold query $t_i^{k'}(\mathcal{S})$ on i returns S_a.

Since $t_i^{k+1}(\mathcal{S})$ returns S_a, from Definition 4 we have that for some activation state $s_a \in S_a$, the condition $s_a \leq s_i^k$ holds. Consider arbitrary $k' > (k+1)$. Since state is inflationary across updates, we know that the state $s_i^{k'}$ after method execution k' is at least s_i^k. That is, $s_i^k \leq s_i^{k'}$. By transitivity of $\leq$, then, $s_a \leq s_i^{k'}$. Hence, by Definition 4, $t_i^{k'}(\mathcal{S})$ returns S_a.

For part 2 of the theorem, consider some replica j of $(S, \leq, s^0, q, t, u, m)$, located at process p_j. We are required to show that, for all $x \geq 0$, the threshold query $t_j^{x+1}(\mathcal{S})$ returns S_a eventually, and blocks until it does.[5] That is, we must show that, for all $x \geq 0$, there exists some finite $n \geq 0$ such that

- for all i in the range $0 \leq i \leq n-1$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns block, and
- for all $i \geq n$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns S_a.

Consider arbitrary $x \geq 0$. Recall that s_j^x is the state of replica j after the xth method execution, and therefore s_j^x is also the state of j when $t_j^{x+1}(\mathcal{S})$ runs. We have three cases to consider:

- $s_i^k \leq s_j^x$. (That is, replica i's state after the kth method execution on i is *at or below* replica j's state after the xth method execution on j.) Choose $n = 0$. We have to show that, for all $i \geq n$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns S_a. Since $t_i^{k+1}(\mathcal{S})$ returns S_a, we know that there exists an $s_a \in S_a$ such that $s_a \leq s_i^k$. Since $s_i^k \leq s_j^x$, we have by transitivity of $\leq$ that $s_a \leq s_j^x$. Therefore, by Definition 4, $t_j^{x+1}(\mathcal{S})$ returns S_a. Then, by part 1 of the theorem, we have

[5] The occurrences of $k+1$ and $x+1$ in this proof are an artifact of how we index method executions starting from 1, but states starting from 0. The initial state (of every replica) is s^0, and so s_i^k is the state of replica i after method execution k has completed at i.

that subsequent executions $t_j^{x+1+i}(\mathcal{S})$ at replica j will also return S_a, and so the case holds. (Note that this case includes the possibility $s_i^k \equiv s^0$, in which no updates have executed at replica i.)

- $s_i^k > s_j^x$. (That is, replica i's state after the kth method execution on i is *above* replica j's state after the xth method execution on j.)
  We have two subcases:
  - There exists some activation state $s'_a \in S_a$ for which $s'_a \leq s_j^x$. In this case, we choose $n = 0$. We have to show that, for all $i \geq n$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns S_a. Since $s'_a \leq s_j^x$, by Definition 4, $t_j^{x+1}(\mathcal{S})$ returns S_a. Then, by part 1 of the theorem, we have that subsequent executions $t_j^{x+1+i}(\mathcal{S})$ at replica j will also return S_a, and so the case holds.
  - There is no activation state $s'_a \in S_a$ for which $s'_a \leq s_j^x$. Since $t_i^{k+1}(\mathcal{S})$ returns S_a, we know that there is some update $u_i^{k'}(a)$ in i's causal history, for some $k' < (k+1)$, that updates i from a state at or below s_j^x to s_i^k.[6] By eventual delivery, $u_i^{k'}(a)$ is eventually delivered at j. Hence some update or updates that will increase j's state from s_j^x to a state at or above some s'_a must reach replica j.[7]
    Let the $x+1+r$th method execution on j be the first update on j that updates its state to some $s_j^{x+1+r} \geq s'_a$, for some activation state $s'_a \in S_a$. Choose $n = r+1$. We have to show that, for all i in the range $0 \leq i \leq r$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns block, and that for all $i \geq r+1$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns S_a.
    For the former, since the $x+1+r$th method execution on j is the first one that updates its state to $s_j^{x+1+r} \geq s'_a$, we have by Definition 4 that for all i in the range $0 \leq i \leq r$, the threshold query $t_j^{x+1+i}(\mathcal{S})$ returns block.
    For the latter, since $s_j^{x+1+r} \geq s'_a$, by Definition 4 we have that $t_j^{x+1+r+1}(\mathcal{S})$ returns S_a, and by part 1 of the theorem, we have that for $i \geq r+1$, subsequent executions $t_j^{x+1+i}(\mathcal{S})$ at replica j will also return S_a, and so the case holds.
- $s_i^k \not\leq s_j^x$ and $s_j^x \not\leq s_i^k$. (That is, replica i's state after the kth method execution on i is *not comparable* to replica j's state after the xth method execution on j.) Similar to the previous case.

□

[6] We know that i's state was once at or below s_j^x, because i and j started at the same state s^0 and can both only grow. Hence the least that s_j^x can be is s^0, and we know that i was originally s^0 as well.

[7] We say "some update or updates" because the exact update $u_i^{k'}(a)$ may not be the update that causes the threshold query at j to unblock; a different update or updates could do it. Nevertheless, the existence of $u_i^{k'}(a)$ means that there is at least one update that will suffice to unblock the threshold query.