Efficient, Consistent Distributed Computation with Predictive Treaties

Tom Magrino
Cornell University
Ithaca, NY, USA
tmagrino@cs.cornell.edu

Jed Liu*
Barefoot Networks
Ithaca, NY, USA
liujed@cs.cornell.edu

Nate Foster
Cornell University
Ithaca, NY, USA
jnfoster@cs.cornell.edu

Johannes Gehrke
Microsoft Corporation
Redmond, WA, USA
johannes@microsoft.com

Andrew C. Myers
Cornell University
Ithaca, NY, USA
andru@cs.cornell.edu

Abstract

To achieve good performance, modern applications often partition their state across multiple geographically distributed nodes. While this approach reduces latency in the common case, it can be challenging for programmers to use correctly, especially in applications that require strong consistency. We introduce *predictive treaties*, a mechanism that can significantly reduce distributed coordination without losing strong consistency. The central insight behind our approach is that many computations can be expressed in terms of predicates over distributed state that can be partitioned and enforced locally. Predictive treaties improve on previous work by allowing the locally enforced predicates to depend on time. Intuitively, by predicting the evolution of system state, coordination can be significantly reduced compared to static approaches. We implemented predictive treaties in a distributed system that exposes them in an intuitive programming model. We evaluate performance on several benchmarks, including TPC-C, showing that predictive treaties can significantly increase performance by orders of magnitude and can even outperform customized algorithms.

ACM Reference Format:
Tom Magrino, Jed Liu, Nate Foster, Johannes Gehrke, and Andrew C. Myers. 2019. Efficient, Consistent Distributed Computation with Predictive Treaties. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3302424.3303987

*Work done while at Cornell

EuroSys '19, March 25–28, 2019, Dresden, Germany
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303987

1 Introduction

In an ideal world, programmers could write high-performance distributed applications using abstractions that provide strong, easy-to-understand consistency guarantees. Unfortunately, programmers today usually choose between performance and strong consistency. Mechanisms that ensure strong consistency, such as serializable transactions, offer a simple programming model but generally require costly synchronization. On the other hand, mechanisms that avoid synchronization, such as conflict-free replicated data types (CRDTs), are limited in the operations they support or the consistency they provide. Hence, they can be difficult to use correctly.

This work develops a new mechanism, *predictive treaties*, that helps move toward the best of both worlds: strong consistency with reduced synchronization. The key insight is that future updates to the results of computations can often be cheaply predicted with reasonable accuracy based on previous updates. Moreover, these predictions can be used to avoid remote data access without harming consistency guarantees that applications rely on.

Predictive treaties capture the predictability of computation using logical predicates over system state. For instance, if a deterministic computation $f(x)$ produces a value y, a corresponding predicate $f(x) = y$ holds. As long as x does not change, the value y can be cached and reused without recomputing $f(x)$, which might otherwise read data x located on a remote node. This essential insight has been exploited by the previous work on warranties [33] and homeostasis [45]. Predictive treaties improve performance through more optimistic, accurate estimates of how long a predicate will hold. The key novelty is the introduction of *metrics*, which measure the expected "distance" to the violation of a predicate and predict how this distance will change. Metrics allow predictive treaties to be *time-dependent*: predictive treaties not only model the current state of the system, but also anticipate how the state will change as a function of time. For example, if $f(x)$ above computes the amount of stock in a warehouse, a predictive treaty might guarantee the inequality $f(x) > 100 - 5t$, where t represents elapsed time in minutes. Such a treaty would allow this inequality (and any other predicate it implies) to be evaluated without any distributed communication. Note that a predictive treaty is not an invariant in the strictest sense, as it is not guaranteed to hold for all time. Even so, our evaluation shows that time dependence allows predictive treaties to reduce synchronization by orders of magnitude for some applications.

Another novelty of predictive treaties is that they can be *hierarchical*: a treaty can be enforced by a set of other predictive treaties when the predicates in the lower-level treaties collectively imply the predicate in the top-level treaty. In cases where lower-level predictive treaties are each defined with respect to the local state at a single node, updates to that local state can be handled without synchronization, provided the update does not invalidate any local treaties. Thus, the higher-level predicate can be enforced locally, without synchronization. By structuring predictive treaties into a well-designed hierarchy, it is possible to build systems in which distributed synchronization tends to involve small subgroups of nodes with good locality.

Of course, predictive treaties are not free: there are costs associated with creating and maintaining run-time objects to represent metrics and predicates. In a naive implementation, these costs could be greater than the costs associated with executing the distributed application itself. To help reduce this overhead, we introduce *stipulated commit*, a mechanism that allows the programmer to propose updates which are applied only if doing so does not violate a treaty. As shown in our evaluation, stipulated commit enables building distributed applications using simple predictive treaties whose performance is competitive with hand-written code.

In summary, this paper explores the design and implementation of predictive treaties as a primitive for distributed programming, and makes the following contributions:

- We introduce predictive treaties and illustrate how their time dependence allows communication to be avoided, by using trends in local updates to predict future state.
- We show that with acceptable cost, *metrics* can be maintained that predict state evolution with sufficient accuracy to support predictive treaties.
- We demonstrate that *hierarchical* predictive treaties can be used to localize and reduce synchronization costs.
- We introduce *stipulated commit* as a programming abstraction that simplifies obtaining good performance from predictive treaties.

2 System Model

Applications are assumed to run on a distributed system comprising multiple geodistributed host nodes that store data objects and perform computation. Each object's current state is stored at a single location (its *store*). A store may be implemented by a machine or by a set of machines, possibly replicated across an availability zone[1] for high availability. We abstract from such implementation details of a store and just treat it as a single node.

Clients can read and modify objects at one or more stores within a distributed transaction. The system uses a commit protocol, such as two-phase commit (2PC) [11], to ensure that transactions are linearizable (strictly serializable) [25, 40]. Thus, unlike some prior work (e.g., [8, 9]) that enforces notions of consistency defined in terms of programmer-specified invariants, we assume that the underlying system offers strong consistency by default.

We refer to the system state as seen by a transaction as the *current system state*; it is the set of object values that result from applying previously committed transactions in the serialization order guaranteed by strict serializability. In a running transaction that has not yet committed, an object may take on a new value that is not yet visible to other transactions, and this object value may be cached at the client(s) performing the transaction. Once the transaction is committed, the new object value is updated at the object's store and becomes part of the current system state.

For transactions that span multiple geodistributed stores, the high communication latency can reduce throughput and increase contention with other transactions. Our goal is to avoid distributed synchronization, where client requests wait for communication over long distances. In particular, we aim to avoid multi-store transactions, whose commit protocols require such synchronization. Background messages sent outside of client-initiated transactions are not considered synchronizations if they are unlikely to delay client requests.

We assume that system nodes maintain loosely synchronized clocks that agree with only limited precision. This assumption is reasonable; the accuracy of clock synchronization offered by older protocols such as NTP [37] and Marzullo's algorithm [35] already suffices for the results presented in this paper. In fact, recent work has shown that clocks can be kept synchronized with much greater precision and with failure rates that are lower than a host of other more serious failures such as bad CPUs [15, 21, 29, 46].

3 Predictive Treaties by Example: Voting

We begin by considering a simple application where predictive treaties prove useful: a voting system that totals the votes for candidates in an election. Votes arrive at one of some number of geographically distributed voting stations and must be tallied to obtain candidate totals. The application keeps track of which candidate is leading—a global property—and makes this information widely available. While accurate up-to-the-minute winner determination is not a feature of current voting systems, it is paradigmatic of a broader class of applications requiring tracking of data aggregates [6, 14, 41].

[1]A group of data centers with low latency between them.

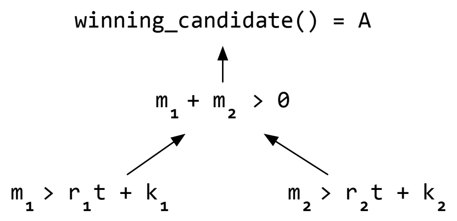

Figure 1. Locally enforceable predictive treaties imply a global predicate. Here $k_2 = -k_1$ and $r_2 = -r_1$.

For simplicity, assume there are two candidates, A and B, and two voting stations, nodes S_1 and S_2. The two nodes process transactions for casting votes, `vote(A)` or `vote(B)`, and query for the current front runner, `winning_candidate()`. Voting increments a station-local vote total for the indicated candidate. At station S_1, the vote totals for the two candidates are a_1 and b_1; at S_2, they are a_2 and b_2. Front runner queries return which candidate has a greater vote total across both nodes, ie. returning A when $a_1 + a_2 > b_1 + b_2$.

Because the current winner is a global property that depends on widely distributed data, any straightforward implementation of `winning_candidate()` using conventional serializable transactions will be slow, even if the winner changes infrequently. Each transaction must check who the new winning candidate is before committing, and this check requires synchronization among all voting stations to ensure that the vote totals observed are consistent with the system state. Fortunately, the state of this application evolves in a predictable way that can be exploited by predictive treaties to avoid synchronization. In particular, each update changes only one total at a single station. Furthermore, we may reasonably assume that the overall voting trend is fairly consistent at each station, over significant time periods.

3.1 Enforcing Predicates with Slack

Suppose A is in the lead, and the application creates a global predicate, `winning_candidate() = A`, to monitor the current winner. This global predicate can be enforced using predictive treaties that track the local vote totals at each station.

With A currently in the lead, the global predicate is equivalent to the predicate $(a_1 - b_1) + (a_2 - b_2) > 0$: the total of the margins must be positive. Defining local margin variables $m_1 = a_1 - b_1$ and $m_2 = a_2 - b_2$, this predicate can be written more simply: $m_1 + m_2 > 0$.

The quantity $m_1 + m_2$ changes by 1 on each vote, so it tracks the minimum number of votes that might invalidate the global predicate. We call this quantity the *slack* of the global predicate, because it measures how far the global predicate is from being invalidated. The global predicate can therefore be enforced by identifying values k_1 and k_2 such that the inequalities $m_1 > k_1$ and $m_2 > k_2$ hold and the global slack $k_1 + k_2$ is nonnegative. As long as the local predicates hold at all of the nodes, no synchronization is required. If an update violates a local predicate, then synchronization among the nodes is required, either to invalidate the global predicate or to establish new local predicates that can then be enforced. However, consistency is not threatened by the falsification of local predicates: the system falls back to synchronization to ensure consistency when predictive treaties no longer hold.

Ignoring questions of how to obtain local predicates and of how to choose values for k_1 and k_2, what we have described thus far corresponds to the approach taken in prior work [45]. We show next how to further improve performance using predictive treaties, which generalize static predicates with mechanisms for predicting the evolution of system state in order to reduce synchronization.

3.2 Time-Dependent Treaties

Assume voting stations are associated with populations that each exhibit their own overall preference for the candidates. Further, for simplicity, assume that voters cast votes independently with different biases at the two sites. In this case, there may be some variation in the local margin observed at each node, but a trend is likely to be observable over a long series of votes. For example, if the population at node S_1 is split 60–40 for candidate A, and an equal-sized population at node S_2 is split 48–52, then the margin for A at S_1 is likely to increase over time, whereas the margin for A at S_2 is likely to (more slowly) decrease over time. In this case, we say that S_1 is a *positively biased* node, because its updates tend to increase the slack of the global predicate. Conversely, S_2 is a *negatively biased* node: its updates tend to decrease the slack of the global predicate. Assuming that voting patterns do not change over time, it is likely that the total bias across all nodes will be positive, meaning that the global predicate is unlikely to be violated soon.

Note that the assumption of independence of voting does not need to hold perfectly. What is key is that there are predictable trends over time periods that are long enough to be useful for reducing synchronization. If votes are correlated with time—for example, if B voters tend to vote later than A voters, then the trend may depend on time, and global bias could become negative as the trend switches.

We can take advantage of these underlying trends by creating time-dependent treaties that automatically track the evolution of system state. Suppose that the nodes have the biases above (the system is positively biased), and for simplicity, that votes arrive at an average rate of one vote per time unit at each site. Then the expected rate of increase in global slack from node S_1 is 0.2 votes per time unit, whereas S_2 is expected to decrease global slack by 0.04 votes per time unit. We can then define local predicates that incorporate these *slack velocities*. For some constants k_1, k_2, r_1, and r_2, node S_1 enforces a local predicate—a predictive treaty—with the form $m_1 > r_1 t + k_1$, and S_2 enforces $m_2 > r_2 t + k_2$. If the sum $k_1 + k_2$ is no larger than the initial global slack and the sum $r_1 + r_2$ is

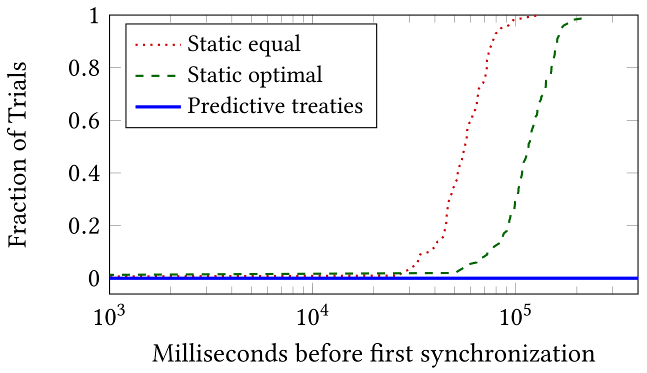

Figure 2. CDF of time until first synchronization under three different slack-allocation strategies. With predictive treaties, less than 1% of runs synchronize.

nonnegative, the conjunction of these local treaties enforces the global treaty, as depicted in Figure 1.

The parameters r_1 and r_2 allow building slack velocities into the local predicates with the effect that slack is continuously transferred between nodes without any synchronization. If $r_1 < 0.2$ and $r_2 < -0.04$ (with $r_1 + r_2 \geq 0$ as before), the local predicates at S_1 and S_2 can remain true indefinitely, despite the negative bias of S_2. Specifically, if we choose $r_1 = 0.12$ and $r_2 = -0.12$, local slack is expected to accumulate, equally fast, at both S_1 and S_2.

Of course, it would be awkward for programmers to have to choose values for parameters like r_i and k_i. Fortunately, they do not need to make this choice. The predictive treaties framework selects them automatically—see Section 4.5.

3.3 Preliminary Evaluation

To see the potential performance benefits of predictive treaties, consider the results from a distributed implementation of the two-station scenario just described, shown in Figure 2. (Section 7 presents the implementation in more detail.) Stations receive 100 votes per second. The program creates a treaty after 30 seconds of voting with the given biases and measures the time until the system must synchronize to create new treaties. The time to this first synchronization is a proxy for the median time between synchronizations. The figure compares three strategies for avoiding synchronization, combining data from 100 trials for each. The dotted red line shows the result when dividing slack equally between the nodes, in a manner similar to that of some previous work [6, 16]. The dashed green line shows the result when using knowledge of the workload to optimally give most of the slack to the negatively biased node; this strategy, which almost doubles the median time to first synchronization, is most similar to the homeostasis approach [45], which uses workload data to approximate the optimal static division of slack. However, predictive treaties reduce synchronization even more dramatically, as shown by the solid blue line in the figure. Synchronization is usually avoided entirely, improving significantly on even the best possible static division of slack. In fact, because predictive treaties adapt to stable trends in the updates, the intervals between synchronizations actually tend to increase over time.

3.4 Hierarchical Treaties

A typical voting system would have more than two voting stations. The approach sketched above can be generalized directly to an arbitrary number of nodes by dividing up slack and slack velocity among all participating nodes. However, this approach does not scale well: the rate of synchronization increases because there are more predictive treaties that can individually fail, and the cost of synchronization also increases because more nodes need to achieve consensus on the new predictive treaties to be enforced subsequently.

A better alternative is to enforce predictive treaties hierarchically. As described above, a predicate of the form $m_1 + m_2 > 0$, where m_1 and m_2 are the margins at the two nodes, can be enforced via predictive treaties of the form $m_1 > r_1 t + k_1$ and $m_2 > -r_1 t - k_1$. But this strategy can be generalized. Assertions of the form $m_i > r_i t + k_i$ can themselves be enforced recursively via lower-level predictive treaties at other nodes.

Hence, we can organize the voting stations into a tree in which connected nodes, especially near the leaves of the tree, are located near each other to reduce communication latency. Each tree node enforces a predictive treaty. Leaf nodes accept votes that update state and that potentially violate the predictive treaty which the node is enforcing. Interior nodes enforce predictive treaties of the same form by negotiating predictive treaties with their child nodes based on the relative bias of those nodes. Slack and velocity are distributed from the root downward in such a way that predicate failures occur infrequently and when they occur, usually do not propagate changes high into the treaty tree.

As the results in Section 7 show, hierarchical predictive treaties allow synchronization to be localized to just part of the system, involving relatively few nodes that are connected with relatively low latency.

4 Predictive Treaties and Metrics

We now present our approach in more formal manner, introducing predictive treaties as well as metrics, which enable efficient and largely automatic enforcement.

4.1 Predictive Treaties

A predictive treaty is a time-limited predicate on the value of a *metric computation* and the current time

$$\overbrace{\phi(\underbrace{m(s)}_{\text{metric}}, t)}^{\text{predicate}} \text{ until } t_{\text{expiry}} \qquad (1)$$

The predicate is guaranteed to be true until the associated expiry time has passed or until the treaty is explicitly retracted. Note that these predicates are *not* necessarily invariants of the system state, and may only hold briefly.

The components of the treaty are as follows:

- $m : S \to \tau$ is a *metric*, which represents a computation on a system state $s \in S$, with a result of type τ.
- $\phi : \tau \times \mathbb{T} \to \mathbb{B}$ is a boolean predicate over the metric's value and the current time t.
- t_{expiry} is the expiry time, after which the predicate ϕ is no longer guaranteed to be true.

The general form of predictive treaty (1) is a predicate over both the state s and time t. Time is left out of the metric, and is therefore treated differently from other parts of system state. This simplification is useful because the system has no control over how time evolves.

The voting example in Section 3 demonstrates *threshold treaties*, in which the predicate ϕ checks a time-dependent vector threshold against a vector-valued metric:

$$\overbrace{\underbrace{\vec{m}(s)}_{\text{metric}} \geq \underbrace{\vec{b}(t)}_{\text{boundary}}}^{\text{predicate } \phi} \text{ until } t_{\text{expiry}} \tag{2}$$

Here, the metric type τ is $\mathbb{R}^n$ for some number of dimensions n. For the treaty to hold, the predicate's inequality $\vec{m}(s) \geq \vec{b}(t)$ must hold componentwise; effectively, the predictive treaty enforces a conjunction of n scalar constraints that share an expiration time.

Threshold treaties model two intuitions from Section 3:

- To enforce distributed assertions with low synchronization, we use local assertions that are "far" from being violated. The metric represents the current system state as a point in $\mathbb{R}^n$ that is compared with a boundary $\vec{b}(t)$. When the metric's location is far from the boundary, the treaty is similarly far from being violated.
- Slack can be implicitly shifted between nodes by having their treaties vary with time. The boundary term $\vec{b}(t)$ shifts over time to either reduce or increase slack.

Threshold treaties have a variety of uses and can be maintained with low synchronization overhead. *Linear* threshold treaties, in which the boundary $\vec{b}(t)$ depends linearly on t, are particularly useful, as in the voting example.

As in prior systems [2, 3, 15, 30, 31, 46], we rely on loosely synchronized clocks. We account for possible skew ε between clocks by enforcing the most conservative interpretation of a predictive treaty—i.e., assume that clocks may differ by ε.

4.2 Enforcing Predictive Treaties

We say that a predictive treaty created at time t_{create} holds if for all times $t \in [t_{\text{create}}, t_{\text{expiry}})$, the predicate $\phi(m(s), t)$ holds. In other words, a predictive treaty holds if, after its creation,

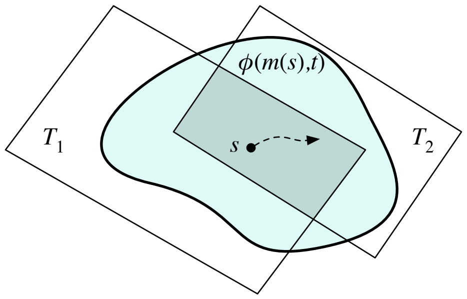

Figure 3. System state s evolves within the intersection of local predictive treaties, enforcing global predicate $\phi(s)$.

and at any time before t_{expiry}, a distributed application can use it in place of a strongly consistent computation that explicitly checks the predicate.

A predicate $\phi(m(s), t)$ that currently holds may be violated (i.e., stop holding) in two ways:

- An update to the system state may change the value of the metric m from value $m(s) = v$ to a new value $m(s') = v'$, such that $\phi(v', t)$ is false.
- As time passes, the predicate may become false due to its dependence on t. In the case of threshold treaties, $\vec{b}(t)$ can grow with time and become larger than the value of $\vec{m}(s)$ in the current system state.

For a given system state, the future time t_{fail} when this second scenario occurs, if any, can be determined from the current value of $m(s)$, assuming $\vec{b}(t)$ is suitably well-behaved. Therefore, to ensure that a predictive treaty holds, the system ensures that $t_{\text{expiry}} \leq t_{\text{fail}}$, if it exists. This may invalidate the treaty if t_{expiry} is set to earlier than the current time.

A direct method for enforcing predictive treaties is to recompute m and update t_{expiry} as needed on each update to the state s supplied to the metric m. However, recomputing m can be expensive and may require synchronization if it reads state from more than one node. A different strategy that avoids this synchronization is to instead enforce multiple local *subtreaties* that in conjunction imply the original, higher-level treaty. The subtreaties are local in the sense that they only depend on local state, so they can be enforced without synchronization. This approach is illustrated in Figure 3, where a global predicate $\phi(m(s), t)$ is enforced using subtreaties T_1 and T_2. As long as system state s stays within the intersection of T_1 and T_2, it also satisfies $\phi(m(s), t)$. Using subtreaties to enforce a predictive treaty, t_{expiry} only needs to be updated for the original predictive treaty if the minimum expiration time of the subtreaties has become earlier than t_{expiry}. Hence, the system requires less synchronization if subtreaties are chosen so that the minimum of their expiration times is unlikely to change, despite changes to the state read by their metrics.

4.3 Metrics

The metric in a predictive treaty is an object that tracks the value of a computation over stored state. The implementation of a metric may also track statistics for modeling how this value evolves over time. These *update statistics* can help predict future changes and, as a result, enable estimation of how long the predicate in the treaty will hold. We discuss specifics of a statistical model used for threshold treaty predictions in Section 4.4.

There are two kinds of metrics, *direct* and *derived*:

Direct metrics. Direct metrics are computed directly from the state of the system, and are updated as the system state evolves. Recall that in the hierarchical voting system example, there is a tree of predictive treaties. Each leaf node in the tree maintains a direct metric for the margin observed at that node. As votes come in, changing the margin, the metric and its estimated statistics are updated.

Derived metrics. Derived metrics are used for hierarchical predictive treaties. They depend on other metrics. In the hierarchical voting system, each interior node maintains derived metrics, which in this case are aggregate margins for the subtree at that point, like the node labeled $m_1 + m_2$ in Figure 1. To avoid synchronization, the state of a derived metric is constructed using the state of its submetrics and is not updated until nodes are otherwise required to synchronize. When a node maintaining a derived metric synchronizes with the nodes maintaining the source metrics, the statistics for the source metrics are combined to construct statistics for the derived metric.

The hierarchy created using derived metrics creates a *metrics tree*, with direct metrics as the leaves and derived metrics as the internal and root nodes. Metrics trees are analogous to abstract syntax trees for representing program structure in compilers or logical plans in databases; their structure guides the system when automatically creating subtreaties.

4.4 Models for Predicting Metrics

Accurate prediction of the system trajectory depends on tracking not only the current value of metrics, but also other attributes. For threshold treaties, the expected metric *velocity* (that is, rate of change in $\mathbb{R}^n$) allows the system to estimate how long it will take for a given predictive treaty to fail. This estimation allows choosing predictive treaties at multiple nodes that allocate slack so that the earliest predictive treaty violation is expected to happen as late as possible.

Of course, system state does not typically exhibit perfectly predictable behavior. To predict the value of a metric, some model of its behavior is needed. If the model is inaccurate, it can harm performance, but not consistency.

We have explored a simple model for numeric metrics, as a random variable M that is the sum of two processes: a predictable linear process and a scaled Brownian process [17] that represents the accumulation of random variation. The variable M is defined as $M = m_0 + vt + \sigma B_t$, where the linear process currently has value m_0 and moves at constant velocity v. The Brownian process σB_t at time t has a Gaussian distribution with standard deviation $\sigma\sqrt{t}$. A numeric metric is therefore characterized by three parameters: m_0, v, σ.

For example, in the voting example of Section 3.2, at site S_1 (where votes are split 60–40), the margin is a Markov chain that is approximated well by parameters $m_0 = 0$, $v = 0.2$, and $\sigma = 0.98$.

If underlying system state changes in an approximately linear way, it is reasonable to use a linear model for the nonrandom component of the metric; we have no evidence that more complex models of metric behavior will be worthwhile. Work in settings with weaker consistency needs have found that linear models often work well in practice, with diminishing returns for more sophisticated models [22]. However, a larger class of functions could be captured by including a transformation in the metric. For example, a quantity expected to vary exponentially over time can be converted into a linear metric by using the logarithm of the quantity as the metric. Quantities expected to vary polynomially could be similarly transformed to more nearly linear behavior.

In our system, numeric direct metrics create estimates of v and σ. These estimates, in addition to the current value m_0, are used by the system when constructing predictive treaties. To accommodate changes in workload over time, we use an exponentially weighted moving average (EWMA) [13, 26, 36]. With a moving average, old behavior of the metric is forgotten over time, at the price of lower accuracy for stable workloads. In the case of a derived metric, estimates are derived from the parameter estimates of the derived metric's submetrics.

4.5 Choosing Treaty Parameters Automatically

When using predictive treaties, programmers are only required to specify the top-level treaties used by their application. Any subtreaties needed to avoid synchronization are automatically chosen by the runtime system. Subtreaties are chosen by a recursive procedure that starts from the top-level treaty metric and works down the metrics tree. At each derived metric m (a parent node in the metrics tree), subtreaties are chosen for the submetrics using a two-step procedure similar to syntax-directed translation in compilers [4] and to query planning in databases [44].

First, our implementation uses templates for the subtreaties, similar to the local treaty templates used in the homeostasis protocol [45]. Templates are predicates with parameters to be filled with specific values. The form of subtreaty templates is determined by the form of the derived metric and the form of its treaty's predicate. For example, if the derived metric is a sum and the treaty's predicate is a threshold, as in the voting example, the system may choose subtreaty templates specifying thresholds on each summand. Thus, in the voting example in Section 3, each subtreaty γ uses a template of the form "$\geq r_\gamma t + k_\gamma$", with parameters r_γ and k_γ. Alternatively, if the derived metric is a minimum of other metrics and the treaty predicate is the strict equality $\min(x, y, z) = 5$, the templates chosen by the system could be an equality for the current minimum metric argument, and thresholds for the other arguments. If x were the current minimum argument, subtreaty templates would take the form $x = x_Y$, $y \geq y_Y$ and $z \geq z_Y$.

Second, the parameters in subtreaty templates are filled with specific values chosen by solving a constrained optimization problem. This problem corresponds to maximizing the predicted time until any subtreaty will become invalid, subject to constraints that ensure the subtreaties imply the original treaty holds. Predictions for how long each subtreaty will hold are based on the predictive model discussed in Section 4.4.[2] This constrained optimization problem ensures the subtreaties chosen are expected to avoid synchronization as long as possible according to the predictive model.

For example, consider the two-station voting scenario in Section 3, where the first and second stations have respective margins m_1 and m_2, with estimated velocities v_1 and v_2, and noise σ_1 and σ_2. With the treaty $m_1 + m_2 \geq 0$ and templates $m_1 \geq r_1 t + k_1$ and $m_2 \geq r_2 t + k_2$, the system solves the optimization problem:[3]

$$\operatorname*{arg\,max}_{r_1, r_2, k_1, k_2} (\min(t_1, t_2))$$

$$\text{where} \quad \begin{array}{ll} m_1 \pm \sigma_1\sqrt{t_1} + v_1 t_1 = r_1 t_1 + k_1 & \quad r_1 + r_2 \geq 0 \\ m_2 \pm \sigma_2\sqrt{t_1} + v_2 t_1 = r_2 t_1 + k_2 & \quad k_1 + k_2 \geq 0 \end{array}$$

In other words, parameters r_1, r_2, k_1, k_2 are solved for to maximize the shorter of two projected expiration times t_1 and t_2. The projected expiration times are times when the model predicts the metric values will coincide with the thresholds, as described by the constraints on the left. The right two constraints ensure that the resulting choices of parameters are such that the original treaty is valid.

For simple cases, our implementation directly solves for parameters; for example, the treaty $min(x, y) \geq 5$ using templates $x \geq a_Y$ and $y \geq b_Y$ has optimal parameter value 5 for both a_Y and b_Y. In more complex cases, our implementation solves for parameters using a numerical optimization library.

4.6 Expiration

The general form of a predictive treaty in Equation (1) includes an expiration time t_{expiry}. This component is useful for enforcing application-level predicates that include an expiration time, as in the case of warranties [33]. However, there is a deeper reason why expiration times are needed.

As discussed in Section 3, a time-dependent predictive treaty can transfer slack continuously from positively biased local treaties to negatively biased ones. A threshold predictive treaty is positively biased at time t if the term $b(t)$ is increasing over time, presumably because the workload updates are also positively biased. In the simple case of a linear bound $b(t) = rt + c$, the sign of r determines the treaty bias.

[2]Of course, alternative predictive models can be used for this process. What matters is that there is a way to predict how long a treaty will hold and a way to set up and efficiently solve the constrained optimization problem.

[3]For clarity, we elide some rewriting of these formulae to reduce the parameter space, such as requiring equality for the final two conditions, and elide checks to ensure that assumptions hold, such as the treaty being currently valid.

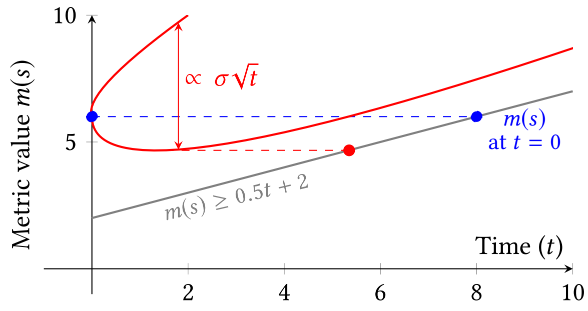

Figure 4. A positively biased predictive treaty must have an expiration time. The red parabolic curve suggests the envelope of likely metric values over time. Hedging the expiration time (the lower dashed line) leaves room for negative updates.

Predictive treaties that are not positively biased remain valid in the absence of updates to the state s because the bound $b(t)$ remains fixed or moves away from the metric value. Therefore, these predictive treaties can be enforced by forcing synchronization when updates arrive that would require them to be retracted. This synchronization may cause the global assertion to fail but its failure will be serializable and observations of state will remain consistent.

Positively biased predictive treaties, however, build in an expectation that incoming updates generate slack in the underlying metric. Consequently, if updates cease, a positively biased treaty becomes invalid simply through the passage of time. In general, there is no way to prevent such invalidations.

For example, Figure 4 shows a predictive treaty with the form $m(s) \geq 0.5t + 2$, corresponding to the gray diagonal line. The metric is required to stay above this line. At time 0, the metric has value 6. Absent any updates, the treaty becomes invalid at time 8, so synchronization-free enforcement of the treaty requires that the value of t_{expiry} be at most 8.

Hedging. It would be safe in this example to use 8 as t_{expiry}, but updates could not be safely accepted by the node if they reduced the metric below its initial value; synchronization would be required. Any reduction below the initial value could make the predictive treaty become invalid before the promised expiration date. To allow some slack-reducing updates to occur without synchronization, the expiration time can be artificially shortened, as suggested by the lower dashed line in the figure. The amount of expiration-time hedging should depend on the noise parameter σ, to balance the cost of making the expiration time too short against the possible cost of synchronization arising from slack-reducing updates.

Asynchronous extensions. As time passes, a node managing a positively biased predictive treaty expects updates which increase the expiration time that it could promise to other nodes. Importantly, these extensions can be communicated *asynchronously*. The managing node can at any time send out *extension messages* to other nodes, increasing the expiration time of the predictive treaty. There is no need for recipients to acknowledge the message or for the sender to wait for a response; a lost or delayed message may lead to unnecessary synchronization but cannot cause inconsistency. Sending extension messages is also discretionary. To avoid gratuitous overhead, the sending node can wait until just before the previously advertised expiration time to send out an extension message. Because such messages do not incur round-trip delays and do not delay client transactions, we do not consider them to be synchronizations.

5 Using Predictive Treaties

Predictive treaties and metrics enable complex, efficient implementations for distributed applications. The API is simple and intuitive, however.

5.1 Programming with Treaties and Metrics

Part of the appeal of predictive treaties is that they support a simple programming interface. To demonstrate this simplicity, Figure 5(b) gives the top-level code for the voting example, using several supporting definitions from Figure 5(a). The function `winner()` defines the top-level computation: it computes the election winner while memoizing the result by generating a treaty that can be used later to check the result. Under the covers, the implementation enforces the underlying predictive treaties to keep this result valid for as long as possible, avoiding recomputation and synchronization.

The method `winner()` computes a `Metric` for the margin between the candidates across a set of voting stations, using a helper method `margin()` that builds a metrics tree by partitioning the voting stations and recursively computing the sub-margins for those stations, combining the results into a single metric using the `plus` operation. Depending on the sign of the resulting margin, `winner()` then uses `getTreaty()` to generate a predictive treaty for either the returned metric or its negation (using the `times(-1)` operation). The parameters to the call to `getTreaty()` represents the lower bound on the value, set to 0 here. Figure 6 shows the tree of metrics used in the voting example. The association between the returned winner and the treaty can be treated as enforcing assertions of the form `f(s) = y`.

Figure 5(a) defines the interfaces for objects of type `Treaty`, `TreatyStatement`, and `Metric`. After a treaty is created, `valid()` returns true until the treaty expires. `Metric` objects provide methods for computing their `value()`, for estimating their `velocity()` and `noise()`, and for obtaining derived metrics. The method `policy(stmt)` automatically creates subtreaties to enforce a predictive treaty enforcing the given statement on the metric's `value()`, using subtreaty templates

```
interface TreatyStatement {
  // For internal use
  boolean check(Metric m);
}
interface Treaty {
  // For client use
  boolean valid();
}
interface Metric {
  // For client use
  double value();
  Metric plus(Metric other); // Makes SumMetric
  Metric times(double scalar); // Makes ScaledMetric
  Metric minus(Metric other); // Same as x+(-1*y)
  Metric min(Metric other); // Makes MinMetric
  Treaty getTreaty(TreatyStatement stmt);
  // For internal use
  double velocity();
  double noise();
  Set<Treaty> policy(TreatyStatement stmt);
}
```

(a) Treaty and Metric interfaces.

```
Pair<String, Treaty> winner(String u, String v) {
  Metric diff = margin(u, v, allStations);
  if (diff.value() >= 0)
    return new Pair<>(u,
      diff.getTreaty(new LowerBound(0));
  return new Pair<>(v,
    diff.times(-1).getTreaty(new LowerBound(0));
}
Metric margin(String u, String v, List<Station> ds){
  int n = ds.size();
  if (n == 1) {
    Station d = ds.get(0);
    return d.votesFor(u).minus(d.votesFor(v));
  }
  int mid = n / 2;
  Metric fst = margin(u, v, ds.subList(0, mid));
  Metric snd = margin(u, v, ds.subList(mid, n));
  return fst.plus(snd);
}
```

(b) Voting example.

Figure 5. Example: memoization with predictive treaties.

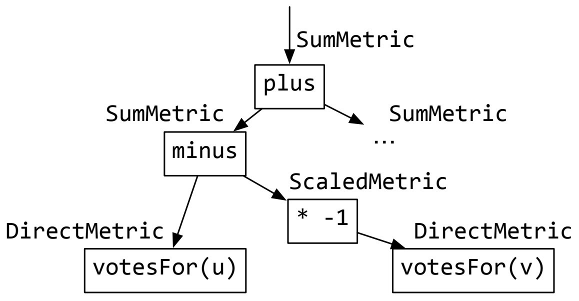

Figure 6. Metrics tree created by example code

and numerical optimization as discussed in Section 4.5. If no subtreaties are returned, the statement is enforced directly.

In our example, votes at an individual station are tracked using metrics that can be updated directly by the application. Votes across stations are tracked using `SumMetrics` and `ScaledMetrics` (produced by the `plus` and `minus` operators), which use the methods `velocity()` and `noise()` to divide up slack proportionally between their sub-metrics.

5.2 Stipulated Commit

To reduce the overhead of creating and maintaining predictive treaties, it can be better to beg forgiveness than to ask permission. The predictive treaties programming interface converts logical conditions that the programmer wants to test into treaties that are monitored and enforced. However, creating and maintaining a treaty ("asking permission") has a cost that is not worth paying if the treaty cannot be reused enough. Unfortunately, programs as written often test logical conditions that are not worth promoting into treaties.

One problematic pattern arises when an application performs certain updates only if a postcondition would hold afterward, but where the success of the update depends on varying input. For example, in a banking application, a withdrawal from an account might be allowed only if the final account balance is nonnegative. Traditionally, a programmer would enforce such a postcondition by checking a sufficiently strong *pre*condition before performing the update. For example, the banking application might guard the withdrawal with code like the following:

```
if (balance - amount >= 0)
    balance -= amount;
```

However, this code does not expose a reusable predicate: first, the guard condition depends on the quantity `amount`, which may vary from request to request; second, when the balance is low, the guard condition may be immediately violated by the update.

To make reusable treaties easier to express, the programming interface allows the specification of *stipulations*, postconditions that must hold after some set of updates is applied. The updates are performed optimistically, but if the resulting state does not satisfy the stipulations, the updates are rolled back and are not committed ("begging forgiveness"). The application then has the opportunity to perform alternative actions. In the banking example, it is enough to check that a treaty of the form `balance` $\geq$ `0` would still be valid *after* the update to the balance. This predicate is reusable because it does not mention the amount being withdrawn, and it is never invalidated by withdrawals.

Actual code for the withdrawal transaction using stipulated commit is shown in Figure 7. In this code, the account balance is sharded across two sites in `balance_us` and `balance_eu`, which may become negative as long as their sum (`balance`) remains nonnegative. The keyword `atomic` starts a nested transaction that aborts and rolls back all of its updates if an exception occurs.

```
Metric balance_us, balance_eu;
Metric balance = balance_us.plus(balance_eu);
int withdraw(int amount) {
  atomic {
    balance.requireStipulation(new LowerBound(0));
    // withdraw from the appropriate shard
    withdraw_locally(amount);
    return amount;
  } catch (StipulationFailure f) {
    return 0;
  }
}
```

Figure 7. Using stipulated commit to withdraw money from a sharded balance.

In cases where an existing treaty already asserts the postcondition, it is enough to check that the enforcement logic, described in Section 4.2, does not determine that the treaty is violated by this transaction. Thus, stipulated commit uses treaties to avoid synchronization using the same underlying mechanisms. On the other hand, if there is no active treaty for a satisfied postcondition, a treaty is automatically created and activated, ensuring that later transactions can avoid synchronization in their postcondition checks.

While stipulated commit relies on a rollback mechanism, it has a subtle difference from previous mechanisms for nested transactions: reads performed when checking the treaty statement postcondition must be treated as part of the parent transaction, even if the postcondition fails. This ensures the serializability of application logic that depends on the failure.

6 Implementation

We implemented predictive treaties and the API described in Section 5 on top of Fabric [32]. Fabric is a persistent programming language that supports nested, distributed transactions. Fabric's security features are not germane to this work, but its support for linearizable multistore transactions and optimistic concurrency control make it a good fit for the geodistributed setting. However, we made a few changes to Fabric to support predictive treaties.

Ignoring comments and blank lines, the implementation added about 5,000 lines of FabIL and Java code to implement the API and changed about 4,000 lines of Java code in the Fabric runtime.

Checking the expiration of predictive treaties in 2PC. Fabric's transactions are strictly serializable, so a transaction's reads and writes behave as if they happen atomically in a single step. Because a predictive treaty's validity depends on the relation between t_{expiry} and the current time, the use of a predictive treaty in a transaction must behave as if it was performed at the time the transaction was committed. Therefore, the transaction protocol must determine for each transaction a *commit time* that respects strict serializability.

Our implementation sets the commit time of a transaction to be the latest time at which the prepare phase finished read- and write-locking the persisted objects at any of the stores. This commit time respects strict serial ordering because it is guaranteed to be within the period of time the transaction appears to have occurred and will be strictly before or after the times other (possibly conflicting) transactions are applied.

We modified Fabric's prepare-phase responses to include the time after all objects at the store have been prepared. The coordinator is modified so that, if there are no failed prepares, the latest of these timestamps is compared against the expiration times of predictive treaties that appear valid to the transaction. If all predictive treaties expiration times are later than the commit time, the coordinator sends out a commit message. Otherwise, an abort message is sent and the coordinator retries the transaction. During the retry, the new attempt will either observe the treaties used as invalid or updated by another transaction with longer expiries.

7 Evaluation

Our evaluation aims to address several questions:

1. Do predictive treaties reduce synchronization? (§3)
2. How does bias difference affect synchronization? (§7.1)
3. How does this scale with the number of sites? (§7.1)
4. What happens when the model's assumption of a stable update trend is violated? (§7.1)
5. Does hierarchy reduce synchronization costs? (§7.1)
6. Do predictive treaties work on realistic workloads? (§7.2)
7. How does performance compare with prior related techniques? (§7.2, 7.3)
8. Does stipulated commit help performance? (§7.3)

The first question is answered by our initial results given in Section 3: using predictive treaties in the voting example can significantly reduce synchronization. We now explore the remaining questions.

7.1 Voting Microbenchmark

In Section 3, we implemented the voting example as a microbenchmark. We make further use of the microbenchmark to investigate questions 2–5.

Behavior with different biases. We ran a series of variations of the voting example in which we varied the voting bias of the pro-A station to see how the results change with the overall bias in the system. In each experiment, the pro-B station is biased with 48% of votes for A, as in the previous experiment, and the pro-A station prefers A at 56% and 60%. As in Section 3, we run votes for 30 seconds and then create a treaty that asserts the current winner is in the lead. We then measure the time that elapses until either a query or voting transaction must synchronize with a remote station, stopping after 400 seconds if there are no synchronizations. This measured time captures how often the voting application would require client transactions to synchronize.

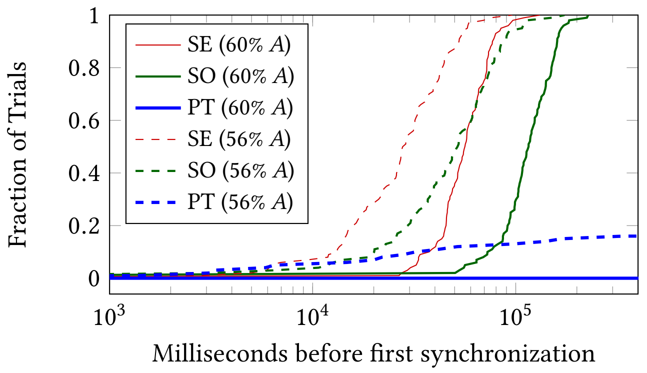

Figure 8. CDF of time until first synchronization under the three strategies static equal (SE), static optimal (SO), and predictive treaties (PT), for slack allocation with varying bias at the first station. Stations received 100 votes per second.

The results are shown in Figure 8. When the overall bias of the system trends toward favoring neither candidate, there is less time between synchronization for all strategies. This occurs because the expected margin between the two candidates is lower and therefore there is less slack to allocate across the two sites. With predictive treaties, synchronization is avoided entirely when trends are stable, even in less biased scenarios. In nearly all cases where the trials hit the cutoff time, the system state was such that the treaties would continue to hold indefinitely; slack was increasing in all stations, reducing the likelihood of synchronization.

Scaling with number of sites. To see how our approach scaled with the number of voting stations, we ran additional comparisons using 4 and 8 voting sites. Each additional pair of voting stations had the same biases and voting rates as the two stations in the previous experiment, ensuring that the overall system bias was the same, 54% votes for candidate A overall, while the voting rates scaled with the number of stations. The measurement is the same as before: the distribution of time from the treaty being created until a client transaction needed to synchronize with a remote station.

The results of this experiment are shown in Figure 9. For static strategies, the time until synchronization is required with either static strategy falls as the number of stations increases. With predictive treaties, few trials ever synchronize even in the largest configuration. This is because the predictive treaties are time-varying and implicitly shift slack.

Adapting to changing update trends. Predictive treaties are extremely effective in avoiding synchronization in scenarios where updates exhibit a stable trend. This is the assumption of our predictive model: past trends in updates can be used to predict future behavior. However, in many realistic workloads this might not be the case: a video may suddenly go viral or

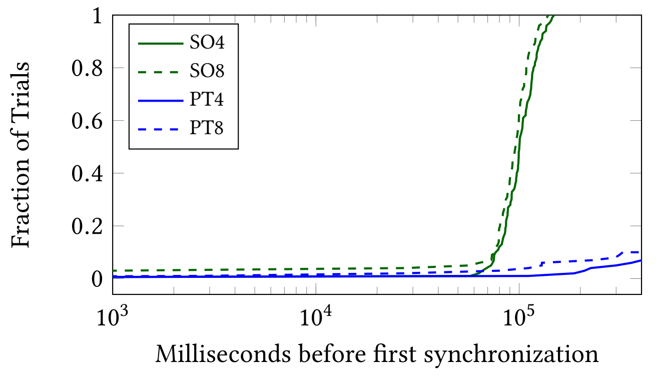

Figure 9. CDF of time until first synchronization in the voting system under static optimal (SO) and predictive treaty (PT) strategies for slack allocation. Each strategy is measured with 4 and 8 stations. With predictive treaties, synchronization is rarely needed.

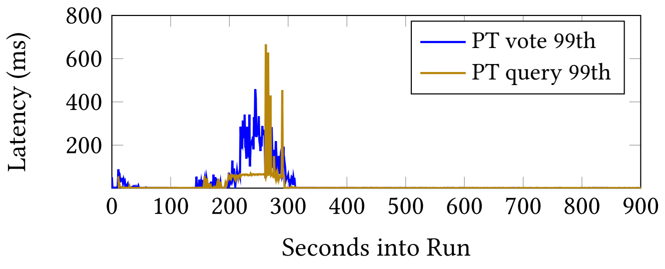

Figure 10. 99th percentile vote and query latencies per second of 2-station adaptivity test.

the underdog candidate's voters show up in strength later in the day. To evaluate how well predictive treaties adapt to a sudden change in update trends, we ran a scenario where the bias in voting suddenly changes after a period. We started the system with 2 stations and an overall bias of 54% pro-A and ran it for 10 seconds of warmup, voting only, followed by 2 minutes with querying. Then, clients flip the voting bias to a new overall bias of 46% pro-A, which we ran for another 13 minutes. During this period, B is expected to pull ahead of A.

In this and following experiments, network latencies between locations are set to simulate round-trip times (RTT) between different Amazon EC2 regions, based on real latency data from Roy et al. [45]. This allows us to validate system behavior under realistic geodistributed conditions. The RTT between each pair of regions ranges between 64 and 372 ms.

In Figure 10, we see the results of this experiment with 2 stations using predictive treaties measuring the 99th percentile latencies of votes and winner queries throughout the run. After the shift in bias, the votes start to tip the margin in favor of B. During this tipping period votes take longer because they must check whether they require retractions for the pre-existing treaties, stating that A is still the winner, and queries take longer as the treaties enforcing the previous result are being retracted. However, eventually the system stabilizes to a new bias and winner, and tail latencies become relatively stable again, with occasional spikes to adjust to new subtreaties. Thus, we see that predictive treaties are able to adapt to changes in bias.

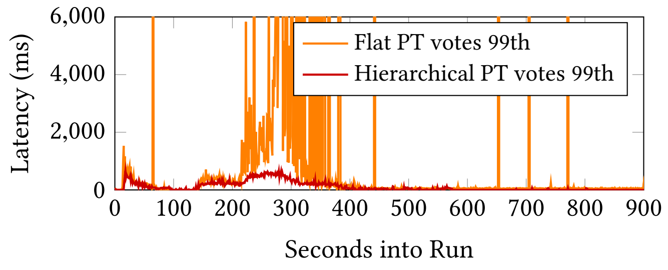

Figure 11. 99th percentile vote and query latencies for 4-station adaptivity test with and without hierarchical treaties. Hierarchy helps avoid large latency spikes.

Benefits of hierarchical structure. Another feature of predictive treaties relevant to adapting to changes in bias is that predictive treaties can be constructed *hierarchically*. Hierarchical structures allows updates to avoid renegotiating the top-level treaty whenever it violates a local treaty, reducing the number of locations the update synchronizes with. To demonstrate the effectiveness of hierarchy, we ran the same experiment but with 4 stations, with station 3 similar to 1 but in Ireland and station 4 similar to 2 but in Singapore.

The result of this comparison, in Figure 11, shows that the maximum 99th percentile latency per second of vote operations rises much higher for the flat organization. In the flat organization, all synchronization occurs across all 4 sites, increasing peak latencies and creating more contention with other transactions. However, the hierarchical organization allows synchronization to sometimes be localized to pairs of sites that are nearby each other, reducing peak latencies.

7.2 Distributed Top-k Monitoring

Babcock and Olston [6] showed how to efficiently monitor the top k items from a set with sharded counts, by tracking the validity of constraints across the distributed system. We implemented a simpler alternative top-k monitoring algorithm, taking advantage of predictive treaties to automatically construct and maintain related constraints.

In this scenario, counts for a set of identifiers are incremented periodically across a number of geodistributed servers; the goal is to be able to quickly query the identities of the top k items in the set. Our algorithm introduces a pseudo-item that we call the *marker*. Its count always lies between that of the k-th and $k+1$-th items. The algorithm maintains global predicates asserting that the items in the current top k all are above the marker, and that the rest of the items are all below. Our framework automatically maintains these predicates.

To evaluate this algorithm, we used a benchmark based on the HTTP request logs for the 1998 FIFA World-Cup website, which was served from 33 servers distributed across 4 regions [5]. This benchmark has been used in related work [6,

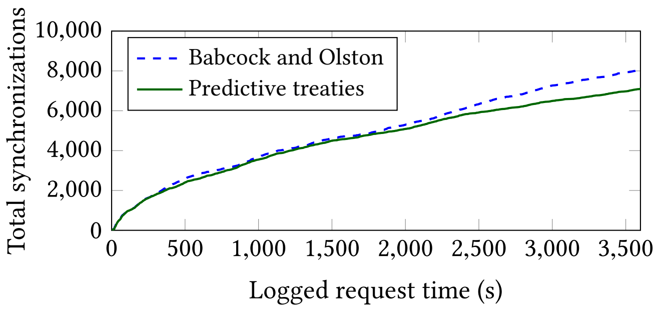

Figure 12. Using synchronizations to compare Babcock and Olston's top-k algorithm with using predictive treaties.

20]. For comparison purposes, we implemented the more complex algorithm of Babcock and Olston in the Fabric system. Unlike prior work, both implementations guarantee strict serializability for updates and queries.

We created a top-level predictive treaty that queried for the 20 most popular pages. As in the original workload, 33 servers served the 4 regions, each logging its received HTTP requests. We ran a one-hour period of the request logs, with one transaction per page hit logged and a total of 84,398 requests processed. A client requested the current top 20 page identifiers every second, and both implementations ensured the top-20 set was up to date at all times.

Figure 12 shows the synchronizations required over time by our algorithm, compared with the Babcock and Olston algorithm. The results show that our algorithm synchronizes less by the end of the experiment, delivering better performance than that of a more complex algorithm specialized to the problem. The treaty implementation was simpler: its update routine was 100 lines of code versus 210 lines of code for our implementation of Babcock and Olston's algorithm.

7.3 Modified TPC-C

The TPC-C benchmark allows us to compare with prior work and to validate that our approach scales to a larger benchmark. TPC-C is an OLTP[4] benchmark that simulates a system for order entry and fulfillment.

For purposes of comparison, our variant of TPC-C is based on the one used by Roy et al. [45]. We similarly shard the database across multiple stores, with each item's stock sharded across the stores. We make the realistic assumption that items are ordered with a nonuniform popularity, skewed across both items and the locations where they are ordered.

As in Roy et al., the database is initialized with 10 warehouses, 10 districts per warehouse, and 100 customers per district. There is an inventory of 1,000 items, for a total of 100,000 Stock objects. Initial stock levels are set randomly between 0 and 100.[5] There are no orders in the initial state. Each store, along with an associated 8 clients, experiences latency simulated to act like one of the EC2 regions.

The workload consists of two of the TPC-C transactions, based on the two potentially distributed operations of the three most frequent transactions in TPC-C. The `NewOrder` transaction orders a random quantity (between 1 and 5) of a random item from a random district at a random warehouse. If there is insufficient stock to meet the order quantity, item stock is first replenished by adding 100 more items before decrementing the stock amount. The `Delivery` transaction enqueues the oldest order at a random warehouse and district for deferred processing. A thread at each warehouse later fulfills the order and charges the appropriate customer. Except when comparing our baseline with the performance reported by Roy et al., we do not include the `Payment` transaction, the remaining of the three most common transactions. `Payment` transactions do not require synchronization; they pad out the workload with operations that don't have read–write conflicts with the other two transactions.

Like Roy et al., we avoid synchronization on every `NewOrder` transaction by relaxing the requirement for globally monotonic order IDs. Instead, they are generated monotonically on a per-shard basis. To determine the oldest order, the `Delivery` transaction requires a total ordering; we obtain one by breaking ties with the shard ID. Like Roy et al., we report `NewOrder` latencies as a distribution plot that captures the core system performance; throughput is directly affected by how long operations take, longer latencies leads to more bottlenecks and contention in the system producing worst throughput. Furthermore, these plots help identify the percentage of orders which coordinated or experienced contention with other transactions, where latency is nontrivial.

Lazy balancing baseline. For a performance baseline, we use a simple algorithm for sharded TPC-C orders that we call *lazy balancing*. It tries to fulfill orders entirely locally, but when the current store lacks enough stock, it synchronizes with the other stores to obtain the missing stock, and divides remaining stock equally among the stores. Lazy balancing does not pay any cost for setting up treaties, and performs especially well when there is no bias across stores, because all stores run out of stock around the same time and hence, treaties do not offer much performance benefit.

To determine whether lazy rebalancing is a competitive baseline, we compared it against the results published for the homeostasis protocol by Roy et al. [45]. When running the same TPC-C workload with the same network configuration they reported, lazy rebalancing achieves a 90th percentile `NewOrder` latency of 130ms and a 99th percentile of 200ms, whereas Roy et al. reported a 90th percentile of roughly 260 ms and a 99th percentile of well over 1 second.

[4]OLTP (Online Transaction Processing) applications handle online transaction requests from external clients.

[5]Random values are drawn from a uniform distribution.

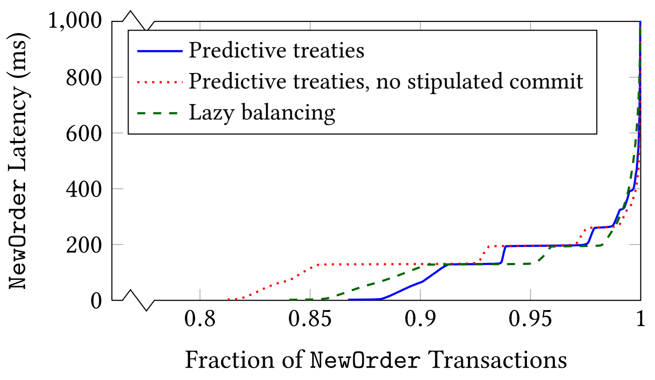

Figure 13. CDF of latencies for TPC-C `NewOrder` transactions, run on two sharded stores with geographic round-trip latency, with 50% of orders going to hot items uniformly across sites. Stipulated commit allows performance of predictive treaties to be comparable to that of lazy balancing.

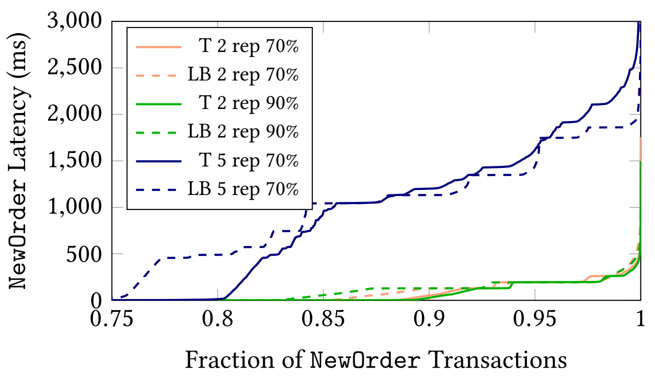

Figure 14. CDF of latencies for TPC-C `NewOrder` transactions with skewed order distribution across 2 and 5 replicas using lazy balancing (LB) and treaties (T).

The next two experiments involve a mix of 95% `NewOrder` and 5% `Delivery` transactions. The system is given 10 minutes of warm-up time so caches heat up and model parameters reach a steady state, followed by 10 minutes of measurement.

Benefits of stipulated commit. To demonstrate the benefits of stipulated commit, we implemented one version of the benchmark using stipulations, similar to the withdrawal example in Section 5.2, and one using treatied preconditions; both enforce the invariant that total stock is positive for all items. These treaties allows clients to remove stock from the local region without synchronizing to ensure there was enough stock to accommodate oversales across the entire database.

We compare the performance of these two implementations with lazy balancing in a scenario with *globally* popular items: 1% of the items are ordered 50% of the time. The symmetry of this scenario makes it favorable to lazy balancing, but Figure 13 shows that in a 2-shard scenario, predictive treaties using stipulated commit perform similarly.[6] Without stipulated commit, the latencies are higher with predictive treaties. Stipulated commit allows the application to avoid creating treaties tailored to specific order amounts.

Skewed popularity. We also evaluated a second scenario to which predictive treaties are particularly well suited: skewed popularity across shards. In these experiments, each replica has a share of "locally hot" items, and a majority of orders on each replica go to its locally hot items. This results in a skewed distribution of updates for each item across its sharded values, with most orders for each item happening at the replica where it is locally hot. We see the results for a skewed order distribution in Figure 14. Predictive treaties allow the system to adapt to uneven popularity across replicas, reducing synchronization over a lazily balanced implementation. With 2 replicas and 70% of orders going to locally hot items at each replica, the lazily balanced implementation synchronizes on 10.5% of orders, while the treaties implementation only synchronizes on 8%. In the highly skewed case where 90% of orders go to locally hot items, this gap in synchronization widens: the treaties implementation synchronizes on only 7.5% of orders, while lazy balancing synchronizes on 12.5%.

[6] To facilitate comparison with prior work [45], the CDF is oriented with probability along the horizontal axis, so the area under the curve is proportional to expected latency.

7.4 Discussion

Predictive treaties are particularly effective when treaties' slack grows over time, as in the voting and top-k benchmarks. In this case, predictive treaties improve on previous techniques by rebalancing slack in the background, avoiding synchronization during client operations.

When, as in TPC-C, global slack does not grow, slack cannot be continuously rebalanced. However, the results show that predictive treaties still have benefits. The predictive model allows the system to *automatically* identify and adapt to the trends and distribution of slack across nodes.

In the unfavorable case where the workload violates the predictive model's assumption of stable trends, the system can compensate by detecting and adapting to new trends. Organizing treaties hierarchically helps reduce overheads in these chaotic scenarios by restricting synchronizations to small local subsets of the nodes when possible.

8 Related Work

Many prior projects have investigated methods for avoiding contention and synchronization in applications by leveraging application-specific semantics. Even in single-store systems, the notion of using higher-level semantics to better manage contention has been proposed, including hierarchical reader-writer locking [23] and predicate locks [19]. Particular attention has been paid to this idea in the distributed application setting, with recent work trying to identify rules for when synchronization is unnecessary [7].

Predictive treaties are designed to improve performance of applications built on top of strongly consistent systems by enforcing application invariants. Some work instead starts with efficient systems with weaker consistency guarantees, such as eventual [53] or causal consistency [34], and introducing techniques such as reservations [43] or CRDTs [9, 47, 48] to enforce stronger guarantees where necessary.

In particular, Indigo [8] allows creating and using reservations to enforce programmer-specified application invariants in a causally consistent setting. Unlike predictive treaties, Indigo's invariants are statically specified at compile time. They are limited to predicates expressible in first-order logic; for example, Indigo's annotations could not enforce a graph connectedness invariant because it is not expressible in first-order logic [18] without further restrictions on the application, such as knowing all vertices in the graph at compile time. Predictive treaties do not have this limitation because they are generated at run time.

Like predictive treaties, some work focuses on monitoring distributed results that may not be invariant for the lifetime of the application, such as results of computation on stored state or the values seen in a stream [16]. Some of this work has examined thresholds on vector values [27, 49] and predictive models [22], but focuses on settings like sensor networks where strong consistency is not required.

Leveraging similar insights to anticipate how remote values will continue to behave, distributed simulations and games use dead reckoning [39, 50, 51]. This technique extrapolates the last known state and behavior of remote objects to perform tasks such as generating visuals. Dead reckoning is useful when immediately computing a inconsistent result is better than blocking the program to ensure a consistent result. In contrast, predictive treaties use a predictive model to make consistency cheaper.

Similar to the goal of metrics for providing a basis of high level strong guarantees on the system state, Conits [54] aims to provide high level consistency guarantees and systems like Pileus [52] offer APIs to directly specify SLA-style guarantees on reads and updates. These guarantees are primarily concerned with consistency of individually read and updated items whereas predictive treaties are intended to construct high-level semantic guarantees.

In settings that require stronger consistency guarantees, problems such as monitoring the top k items in a ranked listing [6], thresholds on a single quantity [38], or thresholds on linear combinations [10, 45] have been studied. Prior work similarly divides a slack-like resource between nodes. MDCC [28] applies similar techniques to provide better concurrency for georeplicated values, processing transactions that commute without determining an explicit ordering. However, this work is focused on either specialized scenarios or guarantees on individual objects and has not leveraged predictive models nor time dependence to shift slack.

Warranties [33], like predictive treaties, allow for compositional predicates built from arbitrary computations. These computations were more general than metrics but assertions are limited to state on a single storage node. Like predictive treaties, warranties have time limits; however, once created, they cannot be revoked before they expire.

Both predictive treaties and warranties leverage compositionality to ensure that enforcement checks recompute only the subcomputations possibly affected by an update. Recomputing only the affected subcomputations to update a result has been explored in work on incremental self-adjusting computation [1, 12]. Incoop [12] applies this technique to Hadoop clusters. RDDs [55] use a similar technique for a more limited class of distributed applications in a cluster. In databases, this technique is used for incremental view maintenance [24], and TxCache [42] offers similar functionality for web applications. However, these techniques were not designed for high-latency, geodistributed settings.

9 Conclusion

Predictive treaties and metrics are new abstractions for building applications that can benefit from the ability to enforce assertions over their geodistributed state. Predictive treaties are defined in terms of metrics that can be maintained locally and computed hierarchically. Coordination across geographic distances can thereby be reduced. Our results show that these new abstractions permit programs to be straightforwardly built in terms of predictive treaties and metrics, with significant performance benefits.

Acknowledgments

Many people have offered useful suggestions on this work or its presentation here, including Matthew Milano, Chinawat Isradisaikul, Isaac Sheff, Natacha Crooks, Matthew Burke, Kai Mast, Ethan Cecchetti, Yizhou Zhang, David Bindel, Christoph Koch, Rolph Recto, Rodrigo Rodrigues, Soumya Basu, Drew Zagieboylo, and the anonymous reviewers from the program committee and shadow program committee.

We acknowledge support from NSF grants 1717554 and 1253165 and a National Defense Science and Engineering Graduate (NDSEG) Fellowship. Foster is also a principal research engineer at Barefoot Networks.

References

[1] Acar, U. A., Ahmed, A., and Blume, M. Imperative self-adjusting computation. In *35th ACM Symp. on Principles of Programming Languages (POPL)* (2008), pp. 309–322.

[2] Adya, A., Gruber, R., Liskov, B., and Maheshwari, U. Efficient optimistic concurrency control using loosely synchronized clocks. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (San Jose, CA, May 1995), pp. 23–34.

[3] Adya, A., and Liskov, B. Lazy consistency using loosely synchronized clocks. In *16th ACM Symp. on Principles of Distributed Computing* (Aug. 1997), PODC '97, pp. 73–82.

[4] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., 2006.

[5] Arlitt, M., and Jin, T. A workload characterization study of the 1998 world cup web site. *IEEE network 14*, 3 (2000), 30–37.

[6] Babcock, B., and Olston, C. Distributed top-k monitoring. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2003), SIGMOD '03, ACM, pp. 28–39.

[7] Bailis, P., Fekete, A., Franklin, M. J., Ghodsi, A., Hellerstein, J. M., and Stoica, I. Coordination avoidance in database systems. *PVLDB 8* (2014), 185–196.

[8] Balegas, V., Duarte, S., Ferreira, C., Rodrigues, R., Preguiça, N., Najafzadeh, M., and Shapiro, M. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 6:1–6:16.

[9] Balegas, V., Serra, D., Duarte, S., Ferreira, C., Shapiro, M., Rodrigues, R., and Preguiça, N. Extending eventually consistent cloud databases for enforcing numeric invariants. In *IEEE Symp. on Reliable Distributed Systems (SRDS)* (Sept. 2015).

[10] Barbará-Millá, D., and Garcia-Molina, H. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal 3*, 3 (July 1994), 325–353.

[11] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[12] Bhatotia, P., Wieder, A., Rodrigues, R., Acar, U. A., and Pasquini, R. Incoop: MapReduce for incremental computations. In *ACM Symp. Cloud Computing* (Oct. 2011).

[13] Brown, R. G. Exponential smoothing for predicting demand. *Operations Research 5*, 1 (1957), 145–145.

[14] Chong, C.-Y., and Kumar, S. P. Sensor networks: Evolution, opportunities, and challenges. *Proceedings of the IEEE 91*, 8 (2003), 1247–1256.

[15] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS) 31*, 3 (2013), 8.

[16] Cormode, G. The continuous distributed monitoring model. *ACM SIGMOD Record 42*, 1 (May 2013), 5–14.

[17] Durrett, R. *Probability: Theory and Examples*, 4th ed. Cambridge University Press, 2010.

[18] Ebbinghaus, H., Flum, J., and Thomas, W. *Mathematical Logic.* Undergraduate Texts in Mathematics. Springer New York, 1996.

[19] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. The notions of consistency and predicate locks in a database system. *Comm. of the ACM 19*, 11 (Nov. 1976), 624–633. Also published as IBM RJ1487, December 1974.

[20] Garofalakis, M., Keren, D., and Samoladas, V. Sketch-based geometric monitoring of distributed stream queries. *Proceedings of the VLDB Endowment 6*, 10 (Aug. 2013), 937–948.

[21] Geng, Y., Liu, S., Yin, Z., Naik, A., Prabhakar, B., Rosenblum, M., and Vahdat, A. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symp. on Networked Systems Design and Implementation (NSDI)* (Apr. 2018), pp. 81–94.

[22] Giatrakos, N., Deligiannakis, A., Garofalakis, M., Sharfman, I., and Schuster, A. Prediction-based geometric monitoring over distributed data streams. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, ACM, pp. 265–276.

[23] Gray, J., Lorie, R., Putzolu, G., and Traiger, I. Granularity of locks and degrees of consistency in a shared database. In *Modeling in Data Base Management Systems.* Amsterdam: Elsevier North-Holland, 1976. Also available in Chapter 3 of *Readings in Database Systems, Second Edition*, M. Stonebraker Editor, Morgan Kaufmann, 1994.

[24] Gupta, A., Mumick, I. S., and Subrahmanian, V. S. Maintaining views incrementally. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (1993), pp. 157–166.

[25] Herlihy, M., and Wing, J. Linearizability: A correctness condition for concurrent objects. Technical Report CMU-CS-88-120, Carnegie Mellon University, Pittsburgh, Pa., 1988.

[26] Holt, C. C. Forecasting trends and seasonals by exponentially weighted averages. carnegie institute of technology. Tech. rep., Pittsburgh ONR memorandum, 1957.

[27] Keren, D., Sharfman, I., Schuster, A., and Livne, A. Shape sensitive geometric monitoring. *IEEE Transactions on Knowledge and Data Engineering 24*, 8 (Aug. 2012), 1520–1535.

[28] Kraska, T., Pang, G., Franklin, M. J., Madden, S., and Fekete, A. MDCC: Multi-data center consistency. In *ACM SIGOPS/EuroSys European Conference on Computer Systems* (Apr. 2013).

[29] Lee, K. S., Wang, H., Shrivastav, V., and Weatherspoon, H. Globally synchronized time via datacenter networks. In *SIGCOMM* (2016), pp. 454–467.

[30] Liskov, B. Practical uses of synchronized clocks in distributed systems. In *10th ACM Symp. on Principles of Distributed Computing* (Aug. 1991), PODC '91, pp. 1–9.

[31] Liskov, B., Shrira, L., and Wroclawski, J. Efficient at-most-once messages based on synchronized clocks. *ACM Trans. on Computer Systems 9*, 2 (May 1991), 125–142.

[32] Liu, J., George, M. D., Vikram, K., Qi, X., Waye, L., and Myers, A. C. Fabric: A platform for secure distributed computation and storage. In *22nd ACM Symp. on Operating System Principles (SOSP)* (Oct. 2009), pp. 321–334.

[33] Liu, J., Magrino, T., Arden, O., George, M. D., and Myers, A. C. Warranties for faster strong consistency. In *11th USENIX Symp. on Networked Systems Design and Implementation (NSDI)* (Apr. 2014), pp. 513–517.

[34] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *23rd ACM Symp. on Operating System Principles (SOSP)* (2011).

[35] Marzullo, K. *Loosely-Coupled Distributed Services: A Distributed Time Service.* PhD thesis, Stanford University, Stanford, Ca., 1983.

[36] Menth, M., and Hauser, F. On moving averages, histograms and time-dependentrates for online measurement. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (New York, NY, USA, 2017), ICPE '17, ACM, pp. 103–114.

[37] Mills, D. L. Network time protocol (version 3) specification, implementation and analysis. Network Working Report RFC 1305, Mar. 1992.

[38] O'Neil, P. The escrow transactional method. *ACM Trans. on Database Systems 11*, 4 (Dec. 1986), 405–430.

[39] Pantel, L., and Wolf, L. C. On the suitability of dead reckoning schemes for games. In *Proceedings of the 1st Workshop on Network and System Support for Games* (New York, NY, USA, 2002), NetGames '02, ACM, pp. 79–84.

[40] Papadimitriou, C. H. The serializability of concurrent database updates. *Journal of the ACM 26*, 4 (Oct. 1979), 631–653.

[41] Piantoni, R., and Stancescu, C. Implementing the Swiss exchange trading system. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on* (1997), IEEE, pp. 309–313.

[42] Ports, D. R. K., Clements, A. T., Zhang, I., Madden, S., and Liskov, B. Transactional consistency and automatic management in an application data cache. In *9th USENIX Symp. on Operating Systems Design and Implementation (OSDI)* (2010).

[43] Preguiça, N., Martins, J. L., Cunha, M., and Domingos, H. Reservations for conflict avoidance in a mobile database system. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services* (New York, NY, USA, 2003), MobiSys '03, ACM, pp. 43–56.

[44] Ramakrishnan, R., and Gehrke, J. *Database Management Systems*, 3 ed. McGraw-Hill, Inc., New York, NY, USA, 2003.

[45] Roy, S., Kot, L., Bender, G., Ding, B., Hojjat, H., Koch, C., Foster, N., and Gehrke, J. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2015).

[46] Shamis, A., Renzelmann, M., Novakovic, S., Chatzopoulos, G., Dragojevic, A., Narayanan, D., and Castro, M. Fast general distributed transactions with opacity. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (June 2019).

[47] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems* (Berlin, Heidelberg, 2011), SSS'11, Springer-Verlag, pp. 386–400.

[48] Shapiro, M., Preguiça, N. M., Baquero, C., and Zawirski, M. Convergent and commutative replicated data types. *Bulletin of the EATCS 104* (2011), 67–88.

[49] Sharfman, I., Schuster, A., and Keren, D. A geometric approach to monitoring threshold functions over distributed data streams. *ACM Transactions on Database Systems 32*, 4 (Nov. 2007).

[50] Singhal, S. K., and Cheriton, D. R. Using a position history-based protocol for distributed object visualization. Technical Report CS-TR-94-1505, Stanford University, Department of Computer Science, Stanford, CA, USA, Feb. 1994.

[51] Taylor, S. J., Saville, J., and Sudra, R. Developing interest management techniques in distributed interactive simulation using java. In *1999 Winter Simulation Conference Proceedings* (Dec. 1999), vol. 1, IEEE, pp. 518–523.

[52] Terry, D. B., Prabhakaran, V., Kotla, R., Balakrishnan, M., Aguilera, M. K., and Abu-Libdeh, H. Consistency-based service level agreements for cloud storage. In *24th ACM Symp. on Operating System Principles (SOSP)* (2013).

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

[54] Yu, H., and Vahdat, A. Design and evaluation of a continuous consistency model for replicated services. In *4th USENIX Symp. on Operating Systems Design and Implementation (OSDI)* (2000).

[55] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symp. on Networked Systems Design and Implementation (NSDI)* (2012).