The Homeostasis Protocol: Avoiding Transaction Coordination Through Program Analysis

Sudip Roy[1,2*], Lucja Kot[2], Gabriel Bender[2], Bailu Ding[2], Hossein Hojjat[2], Christoph Koch[3], Nate Foster[2], Johannes Gehrke[2,4*]

[1]Google Research [2]Cornell University [3]EPFL [4]Microsoft Corp.

{sudip, lucja, gbender, blding, hojjat}@cs.cornell.edu, christoph.koch@epfl.ch, {jnfoster, johannes}@cs.cornell.edu

ABSTRACT

Datastores today rely on distribution and replication to achieve improved performance and fault-tolerance. But correctness of many applications depends on strong consistency properties—something that can impose substantial overheads, since it requires coordinating the behavior of multiple nodes. This paper describes a new approach to achieving strong consistency in distributed systems while minimizing communication between nodes. The key insight is to allow the state of the system to be inconsistent during execution, as long as this inconsistency is bounded and does not affect transaction correctness. In contrast to previous work, our approach uses program analysis to extract semantic information about permissible levels of inconsistency and is fully automated. We then employ a novel *homeostasis protocol* to allow sites to operate independently, without communicating, as long as any inconsistency is governed by appropriate *treaties* between the nodes. We discuss mechanisms for optimizing treaties based on workload characteristics to minimize communication, as well as a prototype implementation and experiments that demonstrate the benefits of our approach on common transactional benchmarks.

1. INTRODUCTION

Modern datastores are huge systems that exploit distribution and replication to achieve high availability and enhanced fault tolerance at scale. In many applications, it is important for the datastore to maintain global consistency to guarantee correctness. For example, in systems with replicated data, the various replicas must be kept in sync to provide correct answers to queries. More generally, distributed systems may have global consistency requirements that span multiple nodes. However, maintaining consistency requires coordinating the nodes, and the resulting communication costs can increase transaction latency by an order of magnitude or more [41].

Today, most systems deal with this tradeoff in one of two ways. A popular option is to bias toward high availability and low latency, and propagate updates asynchronously. Unfortunately this approach only provides eventual consistency guarantees, so applications must use additional mechanisms such as compensations or custom conflict resolution strategies [36, 13], or they must restrict the programming model to eliminate the possibility of conflicts [2]. Another option is to insist on strong consistency, as in Spanner [11], Mesa [17] or PNUTS [10], and accept slower response times due to the use of heavyweight concurrency control protocols.

This paper argues for a different approach. Instead of accepting a trade-off between responsiveness and consistency, we demonstrate that by carefully analyzing applications, it is possible to achieve the best of both worlds: strong consistency and low latency in the common case. The key idea is to exploit the semantics of the transactions involved in the execution of an application in a way that is safe and completely transparent to programmers.

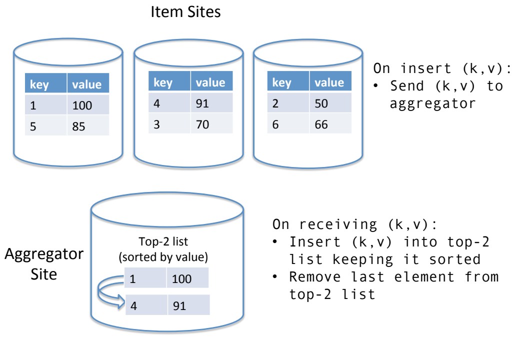
It is well known that strong consistency is not always required to execute transactions correctly [20, 41], and this insight has been exploited in protocols that allow transactions to operate on slightly stale replicas as long as the staleness is "not enough to affect correctness" [5, 41]. This paper takes this basic idea much further, and develops mechanisms for automatically extracting safety predicates from application source code. Our *homeostasis protocol* uses these predicates to allow sites to operate without communicating, as long as any inconsistency is appropriately governed. Unlike prior work, our approach is fully automated and does not require programmers to provide any information about the semantics of transactions.

Example: top-k query. To illustrate the key ideas behind our approach in further detail, consider a top-k query over a distributed datastore, as illustrated in Figure 1. For simplicity we will consider the case where $k = 2$. This system consists of a number of *item sites* that each maintain a collection of $(key, value)$ pairs that could represent data such as airline reservations or customer purchases. An *aggregator site* maintains a list of top-k items sorted in descending order by *value*. Each item site periodically receives new insertions, and the aggregator site updates the top-k list as needed.

A simple algorithm that implements the top-k query is to have each item site communicate new insertions to the aggregator site, which inserts them into the current top-k list in order, and removes the smallest element of the list. However, every insertion requires a communication round with the aggregator site, even if most of the inserts are for objects not in the top-k. A better idea is to only communicate with the aggregator node if the new *value* is greater than the minimal value of the current top-k list. Each site can maintain a cached value of the smallest value in the top-k and only notify the

*Work done while at Cornell University.

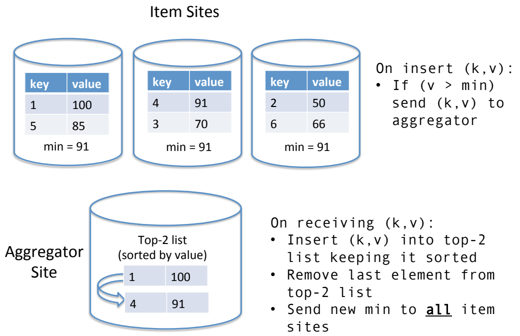
SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright © 2015 ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2723720.



```
On insert (k,v):
• Send (k,v) to
  aggregator

On receiving (k,v):
• Insert (k,v) into top-2
  list keeping it sorted
• Remove last element from
  top-2 list
```

Figure 1: Distributed top-2 computation, basic algorithm.



```
On insert (k,v):
• If (v > min)
  send (k,v) to
  aggregator

On receiving (k,v):
• Insert (k,v) into top-2
  list keeping it sorted
• Remove last element from
  top-2 list
• Send new min to all item
  sites
```

Figure 2: Distributed top-2 computation, improved algorithm.

aggregator site if an item with a larger value is inserted into its local state. This algorithm is illustrated in Figure 2, where each item site has a variable *min* with the current lowest top-k value. In expectation, most item inserts do not affect the aggregator's behavior, and consequently, it is safe for them to remain unobserved by the aggregator site.

This improved top-k algorithm is essentially a simplified distributed version of the well-known *threshold algorithm* for top-k computation [14]. However, note that this algorithm can be extracted *automatically* by analyzing the code for the aggregator site.

Our solution and contributions. This paper presents a new semantics-based method for guaranteeing correct execution of transactions over an inconsistent datastore. Our method is geared towards improving performance in OLTP settings, where the semantics of transactions typically involve a shared global quantity (such as product stock level), most transactions make small incremental changes to this quantity, and transaction behavior is not sensitive to small variations in this quantity except in boundary cases (e.g. the stock level dips below zero). However, our method will still work correctly with workloads that do not satisfy these properties.

We make contributions at two levels. First, we introduce our solution as a high-level framework and protocol. Second, we give concrete instantiations and implementations for each component in the framework, thus providing one particular end-to-end, proof-of-concept system that implements our solution. We stress that the design choices we made in the implementation are not the only possible ones, and that the internals of each component within the overall framework can be refined and/or modified if desired.

The first step in our approach is to analyze the transaction code used in the application (we assume all such code is known up front) to compute a global predicate that tracks its "sensitivity" to different input databases. We then "factorize" the global predicate into a set of local predicates that can be enforced on each node. In the example above, the analysis would determine that the code for the aggregator site leaves the current top-k list unchanged if each newly inserted item is smaller than every current top-k element.

Analyzing transaction code is challenging. It is not obvious what the analysis should compute, although it should be some representation of the transaction semantics that explains how inputs affects outputs. Also, we need to design general analysis algorithms that are not restricted in the same ways as the demarcation protocol [5] or conit-based consistency [41], which are either limited to specific datatypes or require human input.

Our first contribution (Section 2) is an analysis technique that describes transaction semantics using *symbolic tables*. Given a transaction, a symbolic table is a set of pairs in which the first component is a logical predicate on database states and the second component concisely represents the transaction's execution on databases that satisfy the predicate. We show how to compute symbolic tables from transaction code in a simple yet expressive language.

Our second contribution (Section 3) is to use symbolic tables to execute transactions while avoiding inter-node communication. Intuitively, we exploit the fact that if the database state does not diverge from the current entry in the symbolic table, then it is safe to make local updates without synchronizing, but if the state diverges then synchronization is needed. In our top-k example in Figures 1 and 2, as long as no site receives an item whose value exceeds the minimal top-k value, no communication is required. On the other hand, if a value greater than the minimal value is inserted at a site then we would need to notify the aggregator, compute the new top-k list, and communicate the new minimal value to all the sites in the system (including sites that have not received any inserts).

Our *homeostasis protocol* generalizes the example just described and allows disconnected execution while guaranteeing correctness. To reason about transaction behavior in a setting where transactions are deliberately executed on inconsistent data, we introduce a notion of execution correctness based on *observational equivalence* to a serial execution on consistent data. Our protocol is provably correct under the above definition, and subsumes the demarcation protocol and related approaches. It relies on the enforcement of *global treaties*; these are logical predicates that must hold over the global state of the system. In our example, the global treaty is the statement "the current minimal value in the top-k is 91". If an update to the system state violates the treaty, communication is needed to establish a new treaty.

In general, the global treaty can be a complex formula, and checking whether it has been violated may itself require inter-site communication. To avoid this, it is desirable to split the global treaty into a set of *local treaties* that can be checked at each site and that together imply the global treaty. This splitting can be done in a number of ways, which presents an opportunity for optimization to tune the system to specific workloads. Our third contribution (Section 4) is an algorithm for generating local treaties that are chosen based on known workload statistics.

Implementing the homeostasis protocol is nontrivial and raises many systems challenges. Our fourth contribution is a prototype implementation and experimental evaluation (Sections 5 and 6).

2. ANALYZING TRANSACTIONS

This section introduces our program analysis techniques that capture the abstract semantics of a transaction in *symbolic tables*. We first introduce our formal model of transactions (Sections 2.1) and symbolic tables (Section 2.2). We then present a simple yet expressive language $\mathcal{L}$ and explain how to compute symbolic tables for that language (Sections 2.3). Finally, we discuss a higher-level language $\mathcal{L}$++ (Section 2.4) that builds on $\mathcal{L}$ without adding expressiveness and is more suitable for use in practice.

2.1 Databases and transactions

We begin by assuming the existence of a countably infinite set *Obj* of *objects* that can appear in the database, denoted $x, y, z, \ldots$. A database can be thought of as a finite set of objects, each of which has an associated integer value. Objects not in the database are associated with a null default value. More formally, a database D is a map from objects to integers that has finite support.

A *transaction T* is an ordered sequence of atomic *operations*, which can be `read`s, `write`s, computations, or `print` statements. A `read` retrieves a value from the database, while a `write` evaluates an expression and stores the resulting value in a specific object in the database. A computation neither reads from nor writes to the database, but may update temporary variables. Finally, `print` statements allow transactions to generate user-visible output. We define transaction evaluation as follows:

Definition 2.1 (Transaction evaluation). *The result of evaluating a transaction T on a database D, denoted* $\text{Eval}(T, D)$*, is a pair* $\langle D', G' \rangle$ *where* D' *is an updated database and* G' *is a log containing the list of values that the transaction printed during its execution. The order of values in the log is determined by the order in which the print statements were originally executed.*

We assume that transactions are *deterministic*, meaning that D' and G' are uniquely determined by T and D.

2.2 Symbolic tables

We next introduce *symbolic tables*, which allow us to capture the relationship between a transaction's input and its output. The symbolic table for a transaction T can be thought of a map from databases to *partially evaluated transactions*. For any database D, evaluating T on D is equivalent to first finding the transaction associated with D in the symbolic table (which will generally be much simpler than T itself; hence the term "partially evaluated") and then evaluating this transaction on D.

More formally, a symbolic table for a transaction T is a binary relation Q_T containing pairs of the form $\langle \varphi_{\mathfrak{D}}, \phi \rangle$ where $\varphi_{\mathfrak{D}}$ is a formula in a suitably expressive first-order logic and ϕ is a transaction that produces the same log and final database state as T on databases satisfying $\varphi_{\mathfrak{D}}$. The symbolic tables for transactions T_1 and T_2 from Figure 3a and 3b are shown in Figure 4a and Figure 4b. For example, the transaction T_1 behaves identically to the simpler transaction $\texttt{write}(x = \texttt{read}(x) + 1)$ on all databases which satisfy $x + y < 10$, so the symbolic table contains the tuple

$$\langle x + y < 10, \texttt{write}(x = \texttt{read}(x) + 1) \rangle$$

Typically, each execution path through the transaction code corresponds to a unique partially evaluated transaction, although this is not always true if, for example, a transaction executes identical code on both branches of a conditional statement.

We can extend the notion of symbolic tables to work with a set of transactions $\{T_1, T_2, \cdots T_k\}$. A symbolic table for a set of K transactions is a $K + 1$-ary relation. Each tuple in this relation is now of the form $\langle \varphi_{\mathfrak{D}}, \phi_1, ..., \phi_N \rangle$ where ϕ_i produces the same final database and log as T_i when evaluated on any database satisfying $\varphi_{\mathfrak{D}}$. Such a relation can be constructed from the symbolic tables of individual transactions as follows. For each tuple $\langle \varphi_1, \phi_1, ..., \varphi_N, \phi_N \rangle$ in the cross-product of the symbolic tables of all transactions we add a tuple $\langle \varphi_1 \wedge \varphi_2 \wedge ... \wedge \varphi_K, \phi_1, ..., \phi_N \rangle$ to the symbolic table for $\{T_1, T_2, \ldots, T_k\}$. Figure 4c shows a symbolic table for $\{T_1, T_2\}$.

```
T_1 ::= {x̂ := read(x);
        ŷ := read(y);
        if (x̂ + ŷ < 10) then
          write(x = x̂ + 1)
        else
          write(x = x̂ − 1) }
```

(a) **Transaction T_1**

```
T_2 ::= {x̂ := read(x);
        ŷ := read(y);
        if (x̂ + ŷ < 20) then
          write(y = ŷ + 1)
        else
          write(y = ŷ − 1) }
```

(b) **Transaction T_2**

Figure 3: Example transactions for symbolic tables. x **and** y **are database objects,** $\hat{x}$ **and** $\hat{y}$ **are temporary program variables.**

$\varphi_{\mathfrak{D}}$	ϕ
$x + y < 10$	$\texttt{w}(x = \texttt{r}(x) + 1)$
$x + y \geq 10$	$\texttt{w}(x = \texttt{r}(x) - 1)$

(a) **Symbolic table for T_1.**

$\varphi_{\mathfrak{D}}$	ϕ
$x + y < 20$	$\texttt{w}(y = \texttt{r}(y) + 1)$
$x + y \geq 20$	$\texttt{w}(y = \texttt{r}(y) - 1)$

(b) **Symbolic table for T_2.**

$\varphi_{\mathfrak{D}}$	ϕ_1	ϕ_2
$x + y < 10$	$\texttt{w}(x = \texttt{r}(x) + 1)$	$\texttt{w}(y = \texttt{r}(y) + 1)$
$10 \leq x + y < 20$	$\texttt{w}(x = \texttt{r}(x) - 1]$	$\texttt{w}(y = \texttt{r}(y) + 1)$
$x + y \geq 20$	$\texttt{w}(x = \texttt{r}(x) - 1)$	$\texttt{w}(y = \texttt{r}(y) - 1)$

(c) **Symbolic table for the transaction set $\{T_1, T_2\}$**

Figure 4: Symbolic tables for T_1 and T_2. The `read` and `write` commands are abbreviated as `r` and `w` respectively.

2.3 Computing symbolic tables

We now explain how to use program analysis to compute symbolic tables automatically from transaction code. Our analysis is specific to a custom language we call $\mathcal{L}$. This language is not Turing-complete – in particular, it has no loops – but it is powerful enough to encode a wide range of realistic transactions. For instance, it allows us to encode all five TPC-C transactions [1], which are representative of many realistic OLTP workloads. We describe how to express a large class of relational database queries in $\mathcal{L}$ in the Appendix, Section A.

Let $\widehat{Var}$ be a countably infinite set of temporary variables used in the transactions (and not stored in the database). Metavariables $\hat{x}, \hat{y}, \hat{\imath}, \hat{\jmath}$... range over temporary variables. There are four types of statements in $\mathcal{L}$:

- arithmetic expressions *AExp* (elements are denoted $e, e_0, e_1, ...$)
- boolean expressions *BExp* (elements are denoted $b, b_0, b_1, ...$)
- commands *Com* (elements are denoted $c, c_0, c_1, ...$)
- transactions *Trans* (elements are denoted $T, T_0, T_1, ..$). Each transaction takes a list *Params* of zero or more integer parameters $p, p_0, p_1, \ldots$.

The syntax for the language is shown in Figure 5. The expression $\texttt{read}(x)$, when evaluated, will return the current value of the database object x, while the command $\texttt{write}(x = e)$ will evaluate the expression e and store the resulting value in database object x. There is also a `print` command that evaluates an expression e and appends the resulting value to the end of a log. Unlike other commands, `print` produces output that is visible to external observers.

Given a transaction T in $\mathcal{L}$, we can construct a symbolic table for T inductively using the rules shown in Figure 6. $\varphi\{\frac{e}{x}\}$ denotes the formula obtained from φ by substituting expression e for all occurrences of x.

Algorithmically, we can compute the symbolic table by working backwards from the final statement in each possible execution path. Each such path generates one tuple in the symbolic table. We give an example of how to compute the symbolic table for transaction T_1 from Figure 3a. The left side of Figure 7 shows the code and the right side the computation.

$$
\begin{array}{rrcl}
(AExp) & e & ::= & n \mid p \mid \hat{x} \mid e_0 \oplus e_1 \mid -e \mid \texttt{read}(x) \\
(BExp) & b & ::= & \texttt{true} \mid \texttt{false} \mid e_0 \odot e_1 \mid b_0 \wedge b_1 \mid \neg b \\
(Com) & c & ::= & \texttt{skip} \mid \hat{x} := e \mid c_0; c_1 \mid \texttt{if}\, b\, \texttt{then}\, c_1\, \texttt{else}\, c_2 \mid \\
 & & & \texttt{write}(x = e) \mid \texttt{print}(e) \\
(Trans) & T & ::= & \{c\}\,(P) \\
(Params) & P & := & \texttt{nil} \mid p, P \\
 & & & \oplus ::= + \mid * \qquad \odot ::= \; < \mid = \mid \leq
\end{array}
$$

Figure 5: Syntax for the language $\mathcal{L}$

$$[\![T, \{\}]\!] \to [\![c, \{\langle \texttt{true}, \texttt{skip}\rangle\}]\!] \quad \text{where } T = \{c\} \qquad (1)$$

$$[\![c_1; c_2, Q]\!] \to [\![c_1, [\![c_2, Q]\!]]\!] \qquad (2)$$

$$\left[\!\!\left[\begin{array}{l} \texttt{if}\, b\, \texttt{then}\, c_1 \\ \;\texttt{else}\, c_2 \end{array}, Q \right]\!\!\right] \to \left\{ \begin{array}{l} \{\langle b \wedge \varphi, \phi\rangle \mid \langle \varphi, \phi\rangle \in [\![c_1, Q]\!]\} \cup \\ \{\langle \neg b \wedge \varphi, \phi\rangle \mid \langle \varphi, \phi\rangle \in [\![c_2, Q]\!]\} \end{array} \right\} \qquad (3)$$

$$[\![(\hat{x} := e), Q]\!] \to \left\{ \langle \varphi\{\frac{e}{\hat{x}}\}, (\hat{x} := e; \phi)\rangle \mid \langle \varphi, \phi\rangle \in Q \right\} \qquad (4)$$

$$[\![\texttt{skip}, Q]\!] \to Q \qquad (5)$$

$$[\![\texttt{write}(x = e), Q]\!] \to$$
$$\left\{ \langle \varphi\{\frac{e}{x}\}, (\texttt{write}(x = e); \phi)\rangle \mid \langle \varphi, \phi\rangle \in Q \right\} \qquad (6)$$

$$[\![\texttt{print}(e), Q]\!] \to \left\{ \langle \varphi, (\texttt{print}(e); \phi)\rangle \mid \langle \varphi, \phi\rangle \in Q \right\} \qquad (7)$$

Figure 6: Rules for constructing a symbolic table for transaction T; Q is the running symbolic table.

First, Rule (1) initializes the symbolic table with a *true* formula representing all database states and a trivial transaction consisting of a single `skip` command. Next, the symbolic table is constructed by working backwards through the code as suggested by Rule (2).

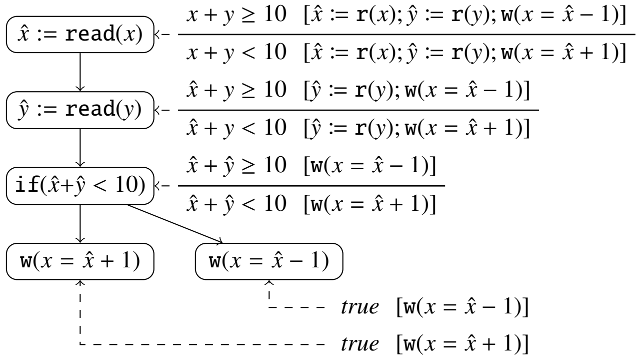
Because the last command is an `if` statement, we apply Rule (3) to create a copy of our running symbolic table; we use one copy for processing the "true" branch and the other for processing the "false" branch. On each branch, we encounter a write to the variable x and apply Rule (6). The statement immediately preceding the `if` is an assignment of a temporary variable. We apply Rule (4), performing a variable substitution on each formula and prepending a variable assignment to each partially evaluated transaction. We apply the same rule again to process the command $\hat{x} := \texttt{read}(x)$.

Note that each tuple in the symbolic table corresponds to a unique execution path in the transaction, and thus a given database instance D may only satisfy a single formula φ in Q_T.

2.4 From $\mathcal{L}$ to $\mathcal{L}$++

$\mathcal{L}$ is a low-level language that serves to set up the foundation for our analysis; we do not expect that end users will specify transactional code in $\mathcal{L}$ directly. In particular, although it is possible to encode many SQL `SELECT` and `UPDATE` statements in $\mathcal{L}$ as described in the Appendix Section A, the direct translation would lead to a large $\mathcal{L}$ program and an associated large symbolic table. Fortunately, we do not need to work with this $\mathcal{L}$ program directly. Because it was generated from a higher-level language, it will have a very regular structure which will translate into regularity in the symbolic table and enable compression.

Therefore, we have created a higher-level language we call $\mathcal{L}$++ which does not add any mathematical expressiveness to $\mathcal{L}$, but directly supports bounded arrays/relations with associated read, update, insert and delete operations. Our algorithm for computing symbolic tables for $\mathcal{L}$ programs can be extended in a straightforward way to compute symbolic tables for programs in $\mathcal{L}$++.



x̂ := read(x)
ŷ := read(y)
if(x̂+ŷ < 10)
w(x = x̂ + 1)
w(x = x̂ − 1)
x + y ≥ 10 [x̂ := r(x); ŷ := r(y); w(x = x̂ − 1)]
x + y < 10 [x̂ := r(x); ŷ := r(y); w(x = x̂ + 1)]
x̂ + y ≥ 10 [ŷ := r(y); w(x = x̂ − 1)]
x̂ + y < 10 [ŷ := r(y); w(x = x̂ + 1)]
x̂ + ŷ ≥ 10 [w(x = x̂ − 1)]
x̂ + ŷ < 10 [w(x = x̂ + 1)]
true [w(x = x̂ − 1)]
true [w(x = x̂ + 1)]

Figure 7: Example of symbolic table construction for T_1 from Figure 3a. Table construction occurs from bottom to top, and `read` and `write` are abbreviated as `r` and `w` respectively.

3. THE HOMEOSTASIS PROTOCOL

This section gives the formal foundation for our homeostasis protocol that uses semantic information to guarantee correct disconnected execution in a distributed and/or replicated system. We introduce a simplified system model (3.1) and several preliminary concepts (3.2) and present the protocol (3.3). The material in this section applies to transactions in any language, not just $\mathcal{L}$ or $\mathcal{L}$++ from Section 2.

3.1 Distributed system model

Assume the system has K sites, each database object is stored on a single site, and transactions run across several sites. Formally, a distributed database is a pair $\langle D, Loc\rangle$, where D is as before and $Loc : Obj \to \{1, \ldots, K\}$ is a mapping from variables to the site identifiers where the variable values are located. Each transaction T_j runs on a particular site i. Formally, there is a *transaction location function* ℓ that maps transactions T_j to site identifiers $\ell(T_j)$.

For ease of exposition, the discussion in Sections 3.2 and 3.3 will focus on systems where the following assumption holds.

Assumption 3.1 (All Writes Are Local). *If T_j runs on site i, it only performs writes on variables x such that* Loc$(x) = i$. *In other words, all writes are local to the site on which the transaction runs.*

This assumption can be lifted and we can apply the homeostasis protocol in systems and workloads with remote writes. We explain how and under what circumstances this can be done in the Appendix, Section B.

3.2 LR-slices and treaties

As explained in the introduction, the idea behind the homeostasis protocol is to allow transactions to operate *disconnected*, i.e. without inter-site communication, and over a database state that may be inconsistent, as long as appropriate *treaties* are in place to bound the inconsistency and guarantee correct execution. Our protocol relies on the crucial assumption that all transaction code is known in advance. This assumption is standard for OLTP workloads.

In our model of disconnected execution, a transaction is guaranteed to read up-to-date values for variables that are local to the site on which it runs. For values resident at other sites, the transaction conceptually reads a locally available older – and possibly stale – *snapshot* of these values. The actual implementation of how that snapshot is maintained can vary. If we are concerned about space overhead and do not wish to store an actual snapshot, we can simply transform the transaction code by inlining the snapshot value where it is needed. However, we will have to repeat this inlining

```
$T_3$ ::= {$\hat{x}$ := read($x$);
        if ($\hat{x} > 0$) then
          write($y = 1$)
        else
          write($y = -1$) }
```

(a) **Transaction T_3**

```
$T_4$ ::= {$\hat{x}$ := read($x$);
        $\hat{y}$ := read($y$);
        if ($\hat{y} = 1$) then
          write($z = (\hat{x} > 10)$)
        else
          write($z = (\hat{x} > 100)$) }
```

(b) **Transaction T_4**

Figure 8: Example transactions for protocol. x is remote for both transactions, while y and z are local.

at the start of each protocol round when the snapshots are synchronized with the actual remote values.

As a concrete example, consider transaction T_3 from Figure 8a. Suppose that T_3 runs on the site where y resides, but that x resides on another site. We can avoid the remote read of x by setting $\hat{x}$ to a locally cached old value of x, say 10. The transaction would still produce the correct writes on y even if the value of x should change on the remote site, as long as that site never allows x to go negative. Thus, if we can set up a *treaty* whereby the system commits to keeping x positive, we can safely run instances of the transaction on stale data that is locally available.

To formalize the intuition provided by the above example, we explore how the local and remote reads of a transaction affect its behavior. Consider transaction T_4 from Figure 8b. Assume that y and z are local and x is remote. The behavior of T_4 depends on both local and global state. Specifically, *if the local value y is* 1 then it matters whether x is greater than 10, and *if the local value y is not* 1, then it matters whether x is greater than 100.

The next few definitions capture this behavior precisely.

Definition 3.2 (Local-remote partition). *Given a database D, a* local-remote partition *is a boolean function p on database objects in D. If $p(x)$ = true we say x is* local *under p; otherwise, it is* remote.

Given a local-remote partition p, we can express any database D as a pair (l, r) where l is a vector containing the values of local database objects and r is a vector containing the values of remote objects. Any transaction T has an associated local-remote partition that marks exactly the objects on site $\ell(T)$ as local.

We next formalize what it means for transactions to behave *equivalently* on different databases. Our notion of equivalence is based on the *observable execution* of transactions: we assume that an external observer can see the final database produced by a transaction as well as any logs it has produced (using `print` statements), but not the intermediate database state(s). Informally, the executions of two (sequences of) transactions are indistinguishable to an external observer if (i) both executions produce the same final database state and (ii) each transaction in the first sequence has a counterpart in the second sequence that produces exactly the same logs.

Alternative notions of observational equivalence are possible: for example, we could assume that an external observer can also see the intermediate database states as the transactions run. Although it is possible to define a variant of the homeostasis protocol that uses this stronger assumption, the protocol will allow fewer interleavings and less concurrency. We therefore use the weaker assumption that only the final database state and logs are externally observable; we believe this is well-justified in practice.

To formally define observational equivalence under Assumption 3.1, we do not need to worry about transactions' effect on the remote part of the database, because remote variables will not be written. This leads to the following definition.

Definition 3.3 (Observational equivalence ($\equiv$)). *Fix some local-remote partition. Let l, l' be vectors of values to be assigned to local objects, let r, r' be vectors of values to be assigned to remote objects, and let G and G′ be logs. Then $\langle (l, r), G\rangle \equiv \langle (l', r'), G'\rangle$ if $l = l'$ and $G = G'$.*

The notion of equivalence defined above is reflexive, transitive, and symmetric. We can use it to formalize the assurances that our protocol provides when reading stale data.

Definition 3.4 (Local-remote slice). *Let T be a transaction with an associated local-remote partition. Consider a pair L, R, where each $l \in L$ is a vector of values to be assigned to local objects and each $r \in R$ is a vector of values to be assigned to remote objects. Then $\{(l, r) \mid l \in L, r \in R\}$ defines a set of databases. Suppose* $\mathrm{Eval}(T, (l, r)) \equiv \mathrm{Eval}(T, (l, r'))$ *for every $l \in L$ and $r, r' \in R$. Then (L, R) is a* local-remote slice *(or LR-slice for short) for T.*

For any database in the LR-slice we know that as long as the remote values stay in R, the writes performed and the log produced by the transaction are determined *only* by the local values.

Example 3.5. *Consider our transaction T_4 from above, with y being local and x remote. For notational clarity, we omit z and list the permitted values for y first, then those for x. One LR-slice for T_4 is* $(\{1\}, \{11, 12, 13\})$*, another is* $(\{1\}, \{11, 12, 13, 14\})$*, and yet another is* $(\{2, 3, 4\}, \{0, 1, 2, 3\})$.

There is no maximality requirement in our definition of *LR*-slices, as the above example shows. Next, we define global treaties.

Definition 3.6 (Global Treaty). *A* global treaty Γ *is a subset of the possible database states $\mathcal{D}$ for the entire system. The system* maintains *a treaty by requiring synchronization before allowing a transaction to commit with a database state that is not in* Γ.

For our purposes, it will be useful to have treaties that require the system to remain in the same *LR*-slice as the original database. Under such a treaty, by Definition 3.4, it will be correct to run transactions that read old (and potentially out-of-date) values for remote variables. We formalize the treaty property we need as follows.

Definition 3.7 (Valid Global Treaty). *Let $\{T_1, T_2, \ldots, T_n\}$ be a finite set of transactions. We say that Γ is a* valid global treaty *if the following is true. For each transaction T_j with associated site $\ell(T_j)$ and associated local-remote partition, let $L = \{l \mid (l, r) \in \Gamma\}$ and $R = \{r \mid (l, r) \in \Gamma\}$. Then (L, R) must be a LR-slice for T_j.*

3.3 Homeostasis protocol

We next present our protocol to guarantee correct execution of transactions executed on inconsistent data.

The protocol proceeds in rounds. Each round has three phases: treaty generation, normal execution and cleanup. Ideally, the system would spend most of its time in the normal execution phase of the protocol, where sites can execute transactions locally without the need for any synchronization. However, if a client issues a transaction T' that would leave the database in a state that violates the current treaty, the system must end the current round of the protocol, negotiate a new treaty, and move on to the next round.

Treaty generation: In this phase, the system uses the current database D to generate a valid global treaty Γ such that $D \in \Gamma$.

Normal execution: In this phase, each site runs incoming transactions in a disconnected fashion, using local snapshots in reads of remote values as previously explained. Transactions running on a single site may interleave, as long as two invariants are enforced.

First, for each site i there must exist a serial schedule $T_1, \ldots, T_m$ of the committed transactions at that site that produces the same log for each transaction and the same final database state as the interleaved schedule. This is a relaxation of view-serializability and can be enforced conservatively by any classical algorithm that guarantees view-serializability. Second, executing any prefix $T_1, T_2, \ldots, T_j$ of the transactions in this sequence (with $j \leq m$) must yield a final database in Γ. This can be enforced by directly checking whether Γ holds before committing each transaction. The details of how this check can be performed are discussed extensively in Section 4.

If a pre-commit check ever determines that a running transaction T' will terminate in a database state not in Γ, the transaction is aborted and the *cleanup* phase begins.

Cleanup: First, the sites synchronize: each site i broadcasts the value of every local object x that has been updated since the start of the round. Next, we determine the transaction T' that triggered the cleanup phase and was aborted, and we run T' to completion at every site. Finally, we start a new round of the protocol.

In a multi-site system, several transactions may try to make treaty-violating writes at the same time; we rely on a suitable voting algorithm to choose one "winner" T' among them. We abort any "loser" transactions that also tried to violate the treaty and wait for all transactions other than T' to terminate. (If, due to the local concurrency control algorithm used, some transactions are blocking/waiting for T', they are aborted as well.). At the start of the next round, we rerun any remaining "loser" transactions.

We now argue that the protocol above is correct. It is not trivial to capture a good notion of "correctness;" we are after all reasoning about transactions that execute on *inconsistent data*, which is not a common setting in classical concurrency theory.

Our notion of correctness is based on observational equivalence as introduced in Section 3.2. Even though transactions are permitted to operate on inconsistent data, it should be impossible for an external observer to distinguish between the execution of the transactions $T_1, T_2, \ldots, T_n$ using the homeostasis protocol and a serial execution of those same transactions that started on a consistent database. In particular, this means that (i) every transaction executed by the protocol must produce the "right" log, and (ii) we must end up with the "right" final database at the end of each round.

The following theorem captures the correctness of our protocol following the above intuition. In the theorem statement, we use the fact that the transactions executed at any given site can be totally ordered without altering the final database or logs that are produced – this is guaranteed by the invariants that the protocol enforces during the *normal execution* phase (discussed above).

Theorem 3.8. *Suppose that Γ is a valid global treaty, and let $T_1, T_2, \ldots, T_n$ be a total order on the transactions executed during a single round of the homeostasis protocol such that if T_j is executed before T_k on some site i then $j < k$. If we execute the transactions $T_1, T_2, \ldots, T_n$ in sequence then (i) each T_j produces the same logs as it did in the homeostasis protocol, and (ii) the final database state is the same as the one produced by the homeostasis protocol.*

The proof is in the technical report version of our paper [32].

4. GENERATING TREATIES

Having laid the theoretical groundwork for the homeostasis protocol, we move to the two most important practical considerations: how the global treaty is generated and how it is enforced. Our approach uses symbolic tables as described in Section 2; however, it does not depend on the way that symbolic tables are computed, or even on the language of the transactions. It can be applied to any workloads for which we can compute symbolic tables.

As a running example, we use transactions T_1 and T_2 from Figure 3. Assume that x and y are on different sites, T_1 runs on the site where x resides, and T_2 runs on the site where y resides.

For ease of exposition, we temporarily assume that we are working with transactions whose "output" (writes and logs) are functions of local database objects only; formally:

Assumption 4.1. *For each transaction T, each partially evaluated transaction ϕ_T in the symbolic table for T, and each database variable x that is read by ϕ_T we assume that $Loc(x) = \ell(T)$.*

This assumption holds for T_1 and T_2 from our previous example; we explain how to lift it in the Appendix, Section C.

4.1 Global and local treaties

In the homeostasis protocol, the first step in every round is to generate a valid global treaty (per Definition 3.7). A naïve but valid global treaty would restrict the database state to the initial D, i.e. require renegotiation on every write. This reduces the homeostasis protocol to a distributed locking protocol; of course, we hope to do better. Intuitively, a good global treaty is one that maximizes the expected length of a protocol round—i.e. one that has a low probability of being violated.

To find a good global treaty, we use the symbolic tables we computed for our transaction workload. At the start of a protocol round, we know the current database D, the set $\mathcal{T}$ of all transactions that may run in the system, and the symbolic table $Q_{\mathcal{T}}$ for $\mathcal{T}$, as shown in Figure 4c for our example set $\{T_1, T_2\}$.

We then pick the unique formula ψ in $Q_{\mathcal{T}}$ that is satisfied by D. For T_1 and T_2, assume that the initial database state is $x = 10$ and $y = 13$. Then ψ is $x + y \geq 20$ (third row of $Q_{\mathcal{T}}$ in Figure 4c). Note that the symbolic table can be precomputed once and reused at each treaty computation, so finding ψ can be done very quickly.

Unfortunately, ψ is not guaranteed to be a valid global treaty under Definition 3.7. In addition, even if it were, it would not be very useful to us as is, since it may be an arbitrarily complex formula and checking whether the database state satisfies ψ may itself require inter-site communication.

For these two reasons, our next step is to factorize ψ into a collection of *local treaties* $\langle \varphi_{\Gamma_1}, \varphi_{\Gamma_2}, \ldots, \varphi_{\Gamma_K} \rangle$. Each local treaty φ_{Γ_i} is a first-order logic formula associated with the site i and uses as free variables only the database objects on site i. The conjunction of all the local treaties should imply ψ:

$$\forall D'. \bigwedge_{1 \leq i \leq K} \varphi_{\Gamma_i}(D') \rightarrow \psi(D') \tag{H1}$$

We also require each local treaty to hold on the original database D; this excludes trivial factorizations such as setting all local treaties to *false*. Formally, for each i ($1 \leq i \leq K$) we require the following:

$$\varphi_{\Gamma_i}(D) \tag{H2}$$

Finally, we take the conjunction of all the local treaties to obtain a global treaty:

$$\varphi_{\Gamma} = \bigwedge_{1 \leq i \leq K} \varphi_{\Gamma_i}(D')$$

Due to the way the global treaty is constructed, all sites can enforce only their respective local treaties and still have the assurance that the global treaty is preserved.

Lemma 4.2. *Let Γ be the set of all databases satisfying φ_Γ. Under Assumption 4.1, Γ is a valid global treaty.*

The proof is in the technical report version of our paper [32].

From now on, we will blur the distinction between Γ and φ_Γ, and work with an intensional notion of a treaty as a first-order logic formula that represents a set of possible database states.

4.2 Computing good local treaties

The crux to computing φ_Γ is finding local treaties, which is difficult in the general case. The requirements in H1 and H2 create a set of Horn clause constraints $\mathcal{H}$. For an undecidable fragment of first-order logic (such as non-linear arithmetic over integers) finding the solution to $\mathcal{H}$ is undecidable, so the local treaties may not exist. For some first-order theories such as the theory of arrays, solving $\mathcal{H}$ is decidable but the local treaties are not necessarily quantifier-free. However, for certain theories such as linear integer arithmetic, the existence of solutions to $\mathcal{H}$ is decidable, solutions can be effectively computed and the solutions are guaranteed to be quantifier-free. These fragments of first-order logic are the theories that have the quantifier-free Craig's interpolation property. For a more elaborate discussion of solving Horn clause systems, see [33].

We present a method for computing local treaties that is not precise – i.e. it may not find the "optimal" local treaties that maximize protocol round length. However, it always works; in the worst case it degenerates to requiring synchronization on every write. It also allows us to optimize the local treaties based on a model of the expected future transaction workload. Such a model could be generated dynamically by gathering workload data as the system runs, or in other ways. The length of time spent in the optimization is a tunable knob so we can trade off treaty quality versus treaty computation time. This is important because local treaties need to be computed at the start of every protocol round; we do not want the cost of treaty computation to wipe out the savings from reduced communication during the protocol round itself.

There are three steps in our algorithm; preprocessing the global treaty, generating *templates* for the local treaties, and running an optimization to instantiate the templates into actual local treaties.

We begin by preprocessing the formula ψ to obtain a stronger but simpler formula that is a conjunction of *linear constraints*. A linear constraint is an expression of the form

$$\left(\sum_i d_i x_i\right) \odot n$$

where the d_i and n are integers, the x_i are variables and $\odot$ is either $=$, $<$ or $\leq$. The preprocessing causes a loss of precision in that we are now enforcing a stronger property than ψ. However, linear constraints and their variants arise frequently as global treaties in OLTP workloads and feature prominently in related work [5, 20, 41]; thus, we expect to handle most real-world global treaties with minimal precision loss. The preprocessing itself is straightforward and described in the Appendix, Section C.

The next step is to create *local treaty templates*; these contain fresh *configuration variables* that are instantiated later. We generate a template for each site k using the following method. Start with ψ and proceed clause by clause. Given a clause of the form

$$\left(\sum_i d_i x_i\right) \odot n,$$

we replace it by

$$\left(\sum_{Loc(x_i)=k} d_i x_i + c_k\right) \odot n$$

where c_k is a fresh configuration variable.

With our running example transactions T_1 and T_2, initial database state $x = 10$ and $y = 13$ and $\psi : x+y \geq 20$, this process would yield the local treaty templates $\varphi_{\Gamma_1} : (x + c_y \geq 20)$ and $\varphi_{\Gamma_2} : (c_x + y \geq 20)$, where c_x and c_y are configuration variables.

The following theorem states there is always at least one assignment of values to configuration variables that yields actual local treaties. We call any such assignment a *valid treaty configuration*.

Theorem 4.3. *Given a starting formula ψ and database D, and a set of local treaty templates generated using the above process, there always exists at least one way to assign values to the configuration variables that satisfies H1 and H2.*

The proof is in the technical report version of our paper [32].

In general, many valid treaty configurations may exist. Our final step is to run an optimization to find a configuration that yields local treaties with a low probability of being violated. As mentioned, we assume we have a model that describes expected future workloads; the details of this model and how it is generated are independent of our algorithm. All that we need is a way to use the model to sample (generate) a number of possible future system executions. Once these are available, we create a MaxSAT instance and use a solver to generate the optimal treaty configuration for these future executions. The number and length of future executions we consider are tunable parameters that allow us to trade off quality of treaties versus time spent in the solver. The details of how we generate the MaxSAT instance are in the Appendix, Section C.

4.3 Discussion and scope of our solution

We have presented our approach for supporting disconnected yet correct transaction execution. The homeostasis protocol allows for provably correct disconnected execution as long as a *global treaty* is maintained. The global treaty is enforced using *local treaties* at each site. We compute the treaties automatically, by analyzing transaction code semantics to generate symbolic tables; treaty computation also takes into account expected workload characteristics.

Our key contributions are, first, a fully general homeostasis protocol, and second, a concrete instantiation and implementation of the protocol. Our instantiation uses the language $\mathcal{L}$++ for specifying transactions and performing analysis, and it uses the algorithm described in Section 4.2 for computing local treaties. However, it is possible to use the homeostasis protocol with another language and/or local treaty computation algorithm.

We now discuss the scope of the homeostasis protocol versus that of the most similar existing approaches such as the demarcation protocol [5] and conit-based consistency [41]. Our current algorithm for local treaty computation only generates treaties which are conjunctions of linear constraints; the protocols in the above related work are able to enforce any such constraints if they are specified manually. However, generating linear constraints manually is tedious and error prone, and an automatic approach is preferable.

Consider the problem of maintaining a more complex aggregate than top-k, such as a top-k of minimums. This might arise in a weather monitoring scenario, where an application records temperature observations and displays the k record daily low temperatures for a date, the k *highest* daily low temperatures, and so on. The constraints for correct disconnected computation of the top-k daily low temperatures are linear and in principle derivable manually, but nontrivial. Suppose we now make the program more complex and also request the top-k days with the highest temperature difference (between the daily low and high temperatures). The treaties required remain linear but become quite difficult to infer manually. We discuss both these examples in the Appendix, Section D.

Moreover, there may be applications where the treaties cannot be expressed as linear constraints unless they are trivial (i.e., force synchronization on every write). In such cases, our algorithm from Section 4.2 will generate trivial treaties; however, our protocol is compatible with more powerful local treaty generation algorithms which are not so restricted, whereas the demarcation protocol is fundamentally limited to handle only linear constraints.

In summary, our automated approach removes the error-prone task of manually inferring constraints. Also, our protocol is fully independent of the language used for the transactions and for the treaties. Thus, our solution is open to handling more expressive treaties with suitable algorithmic extensions, which are future work.

5. HOMEOSTASIS IN PRACTICE

In this section, we highlight some of the practical challenges in implementing the homeostasis protocol and discuss specific design decisions we made in our prototype implementation.

5.1 System design

Our system has two main components – an offline component which analyzes application transaction code, and an online server which receives transaction execution requests from clients and coordinates with other servers to establish and maintain treaties. We discuss each of these components next.

Offline preprocessing As shown in Figure 9, there are two main offline components: the analyzer and the protocol initializer.

The analyzer accepts as input transactions in $\mathcal{L}$++ (Section 2.4) and computes (joint) symbolic tables. In doing so, it applies a number of compression techniques to exploit independence properties and keep the size of the symbolic tables small.

Often transaction code operates on multiple database objects independently; for example, the TPC-C New Order transaction orders several different items. The stock level of each item affects the transaction behavior, but each item affects a different portion of the code. Using a read-write dependency analysis like the one in SDD-1 [31], we identify such points of independence and use them to encode symbolic tables more concisely in a factorized manner.

Moreover, transactions may take integer parameters, and the behavior of the transaction obviously depends on the concrete parameter values. Rather than instantiate parameters now, we push the parameterization into the symbolic tables for further compression.

The protocol initializer sets up the *treaty table* – a data structure that at any given time contains the current global treaty and the current local treaty configuration. The treaty table is thus dependent on the current database state; it is initialized offline based on the database state before the system starts accepting transaction requests. Subsequently, it is updated at each treaty negotiation in the online component.

The protocol initializer also performs some further setup for the online component. For every partially evaluated transaction in the symbolic tables produced by the analyzer, it creates and registers a stored procedure which executes this partially evaluated transaction. The stored procedure also includes checks for the satisfaction of the corresponding treaty as maintained in the treaty table. The stored procedure returns a boolean flag indicating whether the local treaty is violated after execution. The protocol initializer also creates a catalog that maps transactions to corresponding stored procedures in the treaty table.

Online execution The online component accepts and executes transactions using the homeostasis protocol. When a transaction execution request arrives from the clients, the system identifies the appropriate stored procedure in the catalog created during offline preprocessing. The server executes the stored procedure within the scope of a transaction. If the local treaty associated with the stored procedure is satisfied, then the transaction commits locally. Otherwise, the server invokes the treaty negotiator to synchronize with other servers and renegotiate a set of treaties. The negotiator uses an optimizer such as a SAT solver to determine local treaties. It then updates the treaty table and propagates the new treaties to all the other nodes. Therefore, every treaty negotiation requires two rounds of global communication—one for synchronizing database state across nodes and one for communicating the new treaties. However, it is possible to eliminate the second round of communication if the solver is deterministic and therefore arrives at the same configuration at each of the replicas independently.

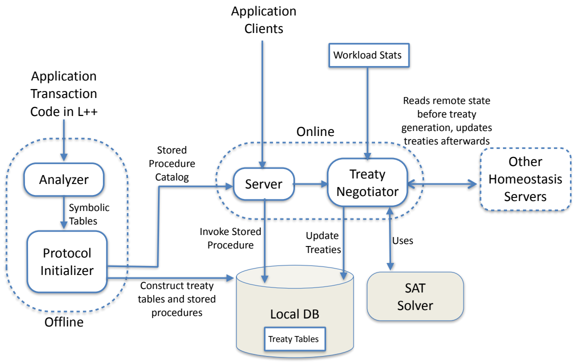


Figure 9: Homeostasis System Architecture.

Our implementation uses an underlying 2PC-like protocol for negotiation, and relies on the concurrency control mechanism of the transaction processing engine to ensure serializable execution locally. However, it would be easy to port it to any infrastructure which supports strongly consistent transaction execution.

5.2 Implementation details

Our system is implemented in Java as middleware built on top of the MySQL InnoDB Engine. Each system instance has a similar setup and communicates with the other instances through network channels. When handling failures, we currently rely on the recovery mechanisms of the underlying database. All in-memory state can be recomputed after failure recovery. In the offline component, we use ANTLR-4 to generate a parser for transactions in $\mathcal{L}$++. For finding optimal treaty configurations, we use the Fu-Malik Max SAT procedure [15] in the Microsoft Z3 SMT solver [12].

6. EVALUATION

We now show an experimental evaluation of a prototype implementation of the homeostasis protocol. We run a number of microbenchmarks (Section 6.1), as well as a set of experiments based on TPC-C [1](Section 6.2) to evaluate our implementation in a realistic setting. All experiments run in a replicated system, using the transformations described in the Appendix, Section B.

6.1 Microbenchmarks

With our microbenchmark, we wanted to understand how the homeostasis protocol behaves in our intended use case – an OLTP system where treaty violations and negotiations are rare. In particular, we were interested in the following questions:

- Since our protocol reduces communication, will it yield more performance benefits as the network round trip time (RTT) between replicas increases?

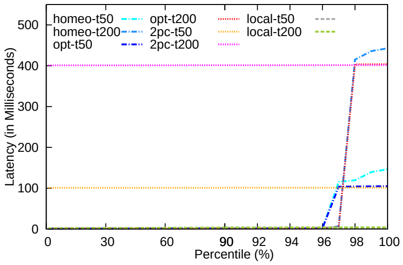


Figure 10: Latency with network RTT ($N_r = 2, N_c = 16$)

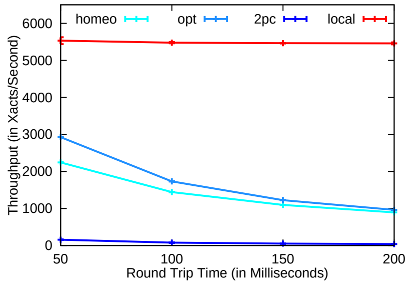


Figure 11: Throughput with network RTT ($N_r = 2, N_c = 16$)

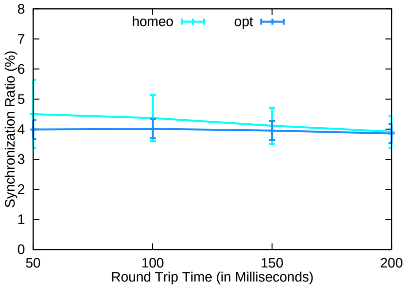


Figure 12: Synchronization Ratio with RTT ($N_r = 2, N_c = 16$)

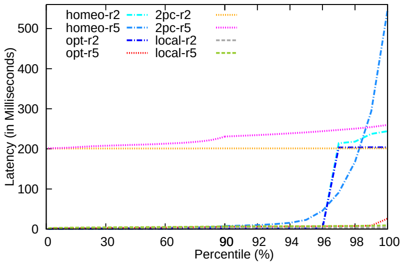


Figure 13: Latency with the number of replicas ($RTT = 100ms, N_c = 16$)

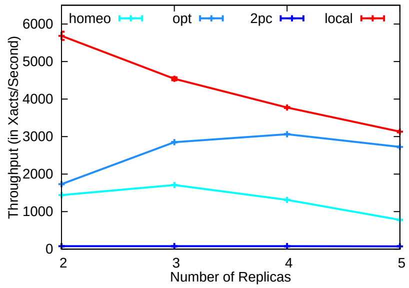


Figure 14: Throughput with the number of replicas ($RTT = 100ms, N_c = 16$)

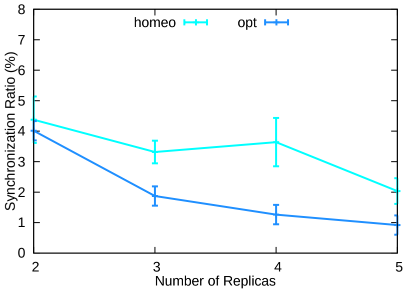


Figure 15: Synchronization Ratio with the number of replicas ($RTT = 100ms, N_c = 16$)

- As the number of replicas increases, treaty negotiations will become more frequent, because each replica must be assigned a "smaller" treaty that is more likely to be violated. How does the performance change with the degree of replication?
- Each replica/server runs multiple clients that issue transactions. How does the performance change when the number of clients per server (i.e., the degree of concurrency) increases?
- How much benefit can we gain from the protocol as compared to three baselines — two-phase commit (2PC), running all transactions locally without synchronizing, and a hand-crafted variant of the demarcation protocol [5]?

To answer these questions, we designed a configurable workload inspired by an e-commerce application. We use a database with a single table `Stock` with just two attributes: item ID (`itemid` INT) and quantity (`qty` INT). The item ID is the primary key. The workload consists of a single parameterized transaction which reads an item specified by `itemid` and updates the quantity as though placing an order. If the quantity is initially greater than one, it decreases the quantity; otherwise, it refills it. SQL pseudocode for the transaction template is shown below.

Listing 1: The microbenchmark transaction; @itemid is an input parameter, while REFILL is a constant.

```
SELECT qty FROM stock WHERE itemid=@itemid;
if (qty>1) then
    new_qty=qty-1
else
    new_qty=REFILL-1
UPDATE stock SET qty=new_qty WHERE itemid=@itemid;
```

We implemented two baseline transaction execution solutions: *local* and *two phase commit (2PC)*. In *local* mode, each replica executes the transactions locally without any communication; thus, database consistency across replicas is not guaranteed. Whereas the local mode provides a bare-bones performance baseline for how fast our transactions run locally, the 2PC mode provides a baseline of the performance of a geo-replicated system implemented in a classical way. In addition to these baselines, we also compare against a hand-crafted solution (OPT) which exploits the transaction semantics in the same way as the demarcation protocol [5]. At each synchronization point, this solution splits and allocates the remaining stock level of each item equally among the replicas. For uniform workloads, it is therefore the optimal solution.

Our workload has several configurable parameters: network RTT, number of replicas (N_r), number of clients per replica (N_c), and the REFILL value. By default, we set RTT to 100 ms, the number of replicas to two, the number of clients per replica to 16 and REFILL to 100. The database is populated with ten thousand items.

All the experiments are run on a single Amazon EC2 c3.8xlarge instance, with 32 vCPUs, 60GB memory, and 2x320GB SSDs, running Ubuntu 14.04 and MySQL Version 5.5.38 as the local database system. We run all replicas on the same instance, and we simulate different RTTs. For each run, we start the system for 5 seconds as a warm-up phase to allow the system to reach a steady state, and then measure the performance for the next 300 seconds. All data points are averages over three runs, and error bars are given in the figures to account for the differences between runs.

Varying RTT Our first experiment varies the network RTT from 50 ms to 200 ms, using the default values for all other parameters. Figure 10 shows the transaction latency by percentile. When using the homeostasis protocol, 97% of the transactions execute locally, with latency less than 4 ms. When a transaction requires treaty negotiation, the latency goes up to around 2RTT plus an additional overhead of less than 50 ms to find new treaties using the solver. This solver overhead manifests at the far right of Figure 10 where the latency for the homeostasis protocol is higher than for OPT for the same RTT setting. Under 2PC, each transaction requires two RTTs, and thus the transaction latency is consistently twice the RTT. In local mode, all the transactions complete in about 2 ms.

Figure 11 shows the throughput per second for each replica. In 2PC mode, it is less than 10 transactions per second due to the network communication cost. The homeostasis protocol allows 100x-1000x more throughput than 2PC, depending on the RTT setting. The difference between the throughput for the homeostasis protocol and local mode can be attributed to the small fraction of trans-

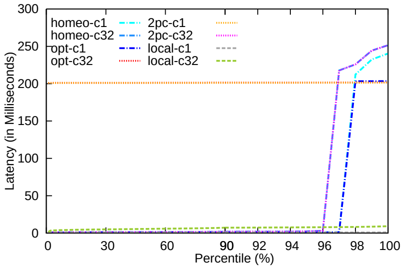


Figure 16: Latency with the number of clients ($N_r = 2, RTT = 100ms$)

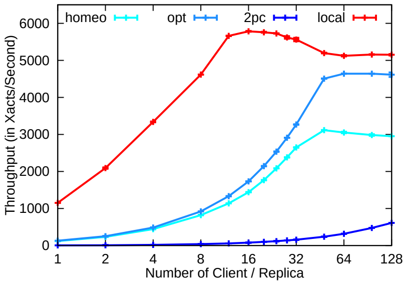


Figure 17: Throughput with the number of clients ($N_r = 2, RTT = 100ms$)

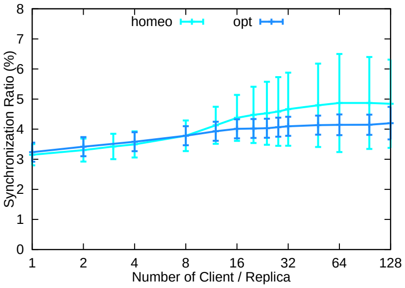


Figure 18: Synchronization Ratio with the number of clients ($N_r = 2, RTT = 100ms$)

actions which require synchronization; for example, if only 2% of transactions require treaty negotiation and the RTT is 100ms, this leads to an average latency of 4*0.98+200*0.02=7.92ms. Finally, Figure 12 shows the synchronization ratio, i.e., the percentage of transactions which require synchronization under the homeostasis protocol and under OPT. The ratio is almost identical, showing that we achieve near optimal performance for this workload.

Varying number of replicas Next, we vary the number of replicas from 2 to 5, while setting the other parameters to their default values. Figure 13 shows the transaction latency profile. With a higher number of replicas, the local treaties are expected to be more conservative and therefore lead to more frequent violations; this leads to an increase in latency. The transaction latency also increases for the local and 2PC modes. In the local case, this is due to the increased resource contention since our experimental setup requires us to run all the replicas on the same server. In 2PC, each transaction stays in the system longer since it has to wait for communication with more replicas; this causes an increase in conflict rates which further increases transaction latency. Figure 14 shows the throughput per second for each replica. As expected, the throughput decreases for all modes as the degree of replication increases. The synchronization ratio shown in Figure 15 also decreases with the decrease in the overall throughput of the system.

Varying number of clients Finally, we vary the number of clients per replica from 1 to 128, while setting the other parameters to their default values. Figure 16 shows the transaction latency profile. In all modes, the transaction latency increases with the number of clients due to higher data and resource contention but is mostly dominated by network latency. Figure 17 shows the throughput per replica. When using the homeostasis protocol with 16 clients, the throughput per client reaches 80% of the throughput per client we observe for 4 clients, indicating good scalability with the number of clients. The curve for the local mode shows a plateau or even exhibits a drop in throughput as the number of clients per replica approaches 16; with a 32-core instance, we reach a point where all cores are in use and the system is overloaded. When running the homeostasis protocol or OPT in the same case, transactions in the treaty negotiation phase free up the CPU and therefore exhibit a plateau at a higher number of clients per replica.

We discuss additional experiments that explore the behavior of the system when we vary other parameters in Appendix Section F.

6.2 TPC-C

To evaluate the performance of the homeostasis protocol over more realistic workloads with multiple transactions, larger databases, and non-uniform workload characteristics, we created a set of experiments based on the TPC-C benchmark.

Data The benchmark describes an order-entry and fulfillment environment. The database contains the following tables: `Warehouse`, `District`, `Orders`, `NewOrder`, `Customers`, `Items`, `Stock` and `Orderline` with attributes as specified in the TPC-C benchmark. The initial database is populated with 10,000 customers, 10 warehouses, 10 districts per warehouse and 1000 items per district for a total of 100,000 entries in the `Stock` table. Initial stock levels are set to a random value between 0 and 100.

Workload We use three transactions based on the three most frequent transactions in TPC-C. The `New Order` transaction places a new order for some quantity (chosen uniformly at random between 1 to 5) of a particular item from a particular district and at a particular warehouse. The `Payment` transaction updates the customer, district and warehouse balances as though processing a payment; we assume that the customer is specified based on customer number. The `Delivery` transaction fulfills the oldest order at a particular warehouse and district. We explain how we encode the transactions in $\mathcal{L}$++ and what treaties are produced in the Appendix, Section E.

For all experiments, we issue a mix of 45% `New Order`, 45% `Payment` and 10% `Delivery` transactions. To simulate a skew in the workload, we mark 1% of the items as "hot" and vary the percentage of `New Order` transactions that order hot items. We denote this percentage as H. For example, a value of $H = 10$ indicates that 10% of all `New Order` transactions order the 1% hot items.

Setup We run all our experiments on c3.4xlarge Amazon EC2 instances (16 cores, 30GB memory, 2x160GB SSDs) deployed at the Virginia (UE), Oregon (UW), Ireland (IE), Singapore (SG) and Sao Paolo (BR) datacenters. The average round trip latencies between these datacenters are shown in Table 1. For all the experiments, we use a single c3.4xlarge node per datacenter. All two-replica experiments use instances from the UE and UW datacenters. There are eight clients per replica issuing transactions. All measurements are performed over a period of 500s after a warmup period of 100s. We only report measurements for the `New Order` transactions, following the TPC-C specification. For comparison, we run the same workload against an implementation of the two-phase commit protocol and a version of the homeostasis protocol with hand crafted treaties (OPT) which minimize the expected number of treaty violations for uniform workloads. All reported values are averages of at least three runs with a standard deviation of less than 6% in all experiments.

	UE	UW	IE	SG	BR
UE	< 1	64	80	243	164
UW	-	< 1	170	210	227
IE	-	-	< 1	285	235
SG	-	-	-	< 1	372
BR	-	-	-	-	< 1

Table 1: Average RTTs between Amazon datacenters (in milliseconds)

Varying Workload Skew For this experiment we vary H, i.e., the percentage of transactions that involve hot items, from 1 to 50. The latency profile for different values of H is shown in Figure 19. As the value of H increases, the treaties for the hot items are vio-

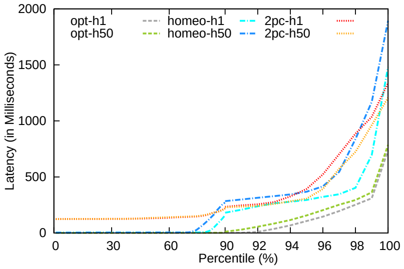


Figure 19: Latency with workload skew($N_r = 2, N_c = 8$)

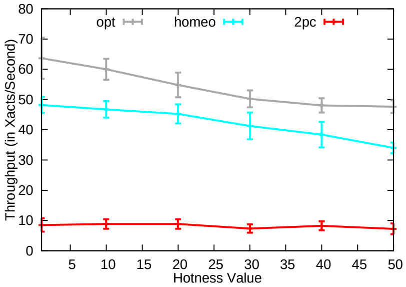


Figure 20: Throughput with workload skew($N_r = 2, N_c = 8$)

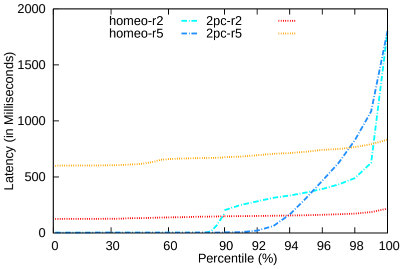


Figure 21: Latency with the number of replicas($N_c = 8, H = 10$)

lated more often, so a higher fraction of transactions takes a latency hit. In comparison, the latency profile for two-phase commit (2PC) is relatively unaffected as it always incurs a two RTT latency hit. As shown in Figure 20, the throughput for 2PC drops with increased H due to an increased rate of conflicts. The throughput for the homeostasis protocol drops as well, but the throughput per replica is still significantly higher than that for 2PC. Note that we only show throughput numbers for the `New Order` transaction, which constitutes 45% of the workload. The actual number of successful transactions committed by the system per second is more than twice this value. We can increase throughput by running more clients per replica; we omit these results due to space constraints.

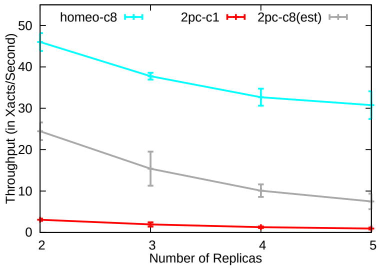


Figure 22: Throughput with the number of replicas($H = 10$)

Varying the number of replicas For this experiment, we set the value of H to 10 and measure the latency and throughput of the `New Order` transactions as we increase the number of replicas. The replicas are added in the order UE, UW, IE, SG and BR. The latency profile and the throughput per replica are shown in Figures 21 and 22. As we add replicas, the maximum RTT between any two replicas increases. This manifests itself towards the 98^{th} percentile as an upward shift in the latency profiles. With five replicas, the treaties become more expensive to compute, which also contributes to the upwards latency shift. On the other hand, with fewer replicas, the throughput is significantly higher, which means that with more replicas, a higher fraction of transactions cause treaty violations. This explains the leftward shift of the inflection point on the curves as the number of replicas decreases. In all cases, the `New Order` throughput values for the homeostasis protocol are substantially higher than the 2PC baseline. In our 2PC implementation, we only use a single client per replica: with a larger number of clients, conflicts caused frequent transaction aborts. Figure 22 also shows a very conservative upper bound on maximum throughput for 2PC that is obtained by multiplying the measured throughput for a single client by a factor of 8. Clearly, even this estimate still has a significantly lower throughput than the homeostasis protocol. Thus, the homeostasis protocol clearly outperforms 2PC in all situations.

The long tail for latencies is due to the fact that the minimum allowable value of MySQL's lock wait time-out is 1 second. In high-contention environments, more transactions time out, giving long tail latencies for high values of H or high numbers of replicas.

Distributed Deployment The goal of this experiment is to study the feasibility of deploying the homeostasis protocol in a realistic setting where each database replica is distributed across a number of machines. We distribute our database so that each machine handles all requests pertaining to one TPC-C warehouse. This distributed database is replicated across the UE and UW datacenters. We use 10 warehouses (and therefore 10 machines per datacenter), 100 districts per warehouse, 1M customers and a total of 1M entries in the stock relation. We use a transaction mix of 49% New Order, 49% Payment and 2% Delivery transactions. Due to space limitations we only highlight our results here and present them more fully in the Appendix, Section F.2.

Using the homeostasis protocol, we achieve an overall system throughput of nearly 9000 transactions per second, or around 80% of what is achievable under OPT. As expected, the fraction of transactions requiring synchronization under homeostasis is higher than under OPT and increases as we increase the skew in the workload. Unsurprisingly, given the average latency between the two datacenters, the maximum possible throughput achievable using 2PC is around an order of magnitude smaller than with OPT.

7. RELATED WORK

Exploiting application semantics to improve performance in concurrent settings has been explored extensively, from thirty years ago [16] up to today [3]. We refer the reader to [39] for a more complete survey, and only highlight the approaches most similar to ours.

The demarcation protocol [5, 21] and distributed divergence control protocols [40, 29] based on epsilon serializability [30] exploit semantics to allow non-serializable interleavings of database operations. The demarcation protocol allows asynchronous execution of transactions in a replicated system by maintaining an integrity constraint. The distributed divergence control protocol allows some inconsistency but bounds it to ensure that the schedules produced are within epsilon of serializability. More recently, Yu and Vahdat proposed a continuous consistency model for replicated services [41] which allows bounded inconsistency and uses anti-entropy algorithms [27, 28] to keep it at an acceptable level. As discussed in Section 4.3, the distinguishing feature of our work is the use of program analysis techniques to automatically infer the constraints to be maintained, and the fact that the homeostasis protocol is independent both of the transaction language and the constraint language.

The Calvin [37] and Lynx [42] systems also perform static analysis of transactions to improve performance; however, they ensure that all transactions operate on consistent data, whereas we allow them to see inconsistent data if the inconsistency is "irrelevant."

There is extensive research on identifying cases where transactions or operations commute and allowing interleavings which are non-serializable yet correct [22, 23, 19, 34, 2, 24]. Our solution extends this work, as the use of LR-slices allows transactions to commute in special cases where they may not commute in general. The approach in [9] allows for state-dependent commutativity, but at the interface level rather than the read/write level. $\mathcal{I}$-confluence invariants [3] and warranties [25] both have similarities to our treaties, but neither is inferred automatically from code.

Some systems allow inconsistency that does impact transaction semantics when deemed acceptable cost-wise for the application [20, 4]. Including the cost of inconsistency into our treaty optimization metric is future work.

There is more general distributed systems work that explains how to detect whether global properties hold [7, 26], and how to enforce them using local assertions [6]. However, this work does not address how the local assertions may be computed.

There is a large body of work on using synthesis to develop concurrent programs [8, 38, 35, 18]. Our applications are database transactions rather than general-purpose code, and we deal with extra challenges such as the need to optimize based on workload.

8. CONCLUSION

We have presented the homeostasis protocol, which allows transactions to execute correctly in a multi-node system while minimizing communication. We extract consistency requirements from transaction code *automatically via program analysis* rather than requiring human input, which enables a fully automated adaptive consistency management protocol. We have presented the method as a general framework and provided concrete implementations for each component in the framework.

9. ACKNOWLEDGMENTS

We would like to thank Peter Bailis and the anonymous SIGMOD reviewers for their insightful comments.

This research has been supported by the NSF under Grants CNS-1413972, CCF-1253165, CNS-1012593 and IIS-0911036, by the European Research Council under grant ERC Grant 279804, by the iAd Project funded by the Research Council of Norway and by a Google Research Award. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsors.

10. REFERENCES

[1] http://tpc.org/tpcc.

[2] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR*, pages 249–260, 2011.

[3] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. In *VLDB*, 2015.

[4] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow.*, 5(8):776–787, Apr. 2012.

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

[6] O. S. Carvalho and G. Roucairol. On the distribution of an assertion. In *PODC '82*, pages 121–131, 1982.

[7] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[8] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *PLDI*, 2008.

[9] A. T. Clements et al. The scalable commutativity rule: Designing scalable software for multicore processors. In *SOSP*, pages 1–17, 2013.

[10] B. F. Cooper et al. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[11] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *OSDI*, pages 251–264, 2012.

[12] L. De Moura and N. Bjørner. Z3: An efficient smt solver. TACAS'08/ETAPS'08, 2008.

[13] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[14] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[15] Z. Fu and S. Malik. On solving the partial max-sat problem. SAT'06.

[16] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM TODS*, June 1983.

[17] A. Gupta et al. Mesa: Geo-replicated, near real-time, scalable data warehousing. In *VLDB*, 2014.

[18] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Concurrent data representation synthesis. In *PLDI*, 2012.

[19] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPOPP*, 2008.

[20] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: pay only when it matters. *Proc. VLDB Endow.*, 2(1):253–264, Aug. 2009.

[21] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: multi-data center consistency. In *EuroSys*, 2013.

[22] A. Kumar and M. Stonebraker. Semantics based transaction management techniques for replicated data. In *SIGMOD*, 1988.

[23] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguica, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, pages 265–278, 2012.

[24] C. Li et al. Automating the choice of consistency levels in replicated systems. In *USENIX ATC 14*, pages 281–292, June 2014.

[25] J. Liu, T. Magrino, O. Arden, M. D. George, and A. C. Myers. Warranties for faster strong consistency. NSDI'14, pages 503–517.

[26] K. Marzullo and G. Neiger. Detection of global state predicates. In *In WDAG '91*, pages 254–272. Springer-Verlag, 1991.

[27] C. Olston, B. T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. In *SIGMOD*, 2001.

[28] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *VLDB*, 2000.

[29] C. Pu and A. Leff. Replica control in distributed systems: as asynchronous approach. In *SIGMOD*, pages 377–386, 1991.

[30] K. Ramamritham and C. Pu. A formal characterization of epsilon serializability. *IEEE TKDE.*, 7(6):997–1007, Dec. 1995.

[31] J. Rothnie,Jr. et al. Introduction to a system for distributed databases (sdd-1). *ACM TODS*, 1980.

[32] S. Roy, L. Kot, N. Foster, J. Gehrke, H. Hojjat, and C. Koch. Writes that fall in the forest and make no sound: Semantics-based adaptive data consistency. *CoRR*, abs/1403.2307, 2014.

[33] P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for horn-clause verification. In *CAV*, pages 347–363, 2013.

[34] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical report, INRIA, Rocquencourt, France, 2011.

[35] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching concurrent data structures. In *PLDI*, volume 43, pages 136–148, 2008.

[36] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[37] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, pages 1–12, 2012.

[38] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, volume 45, pages 327–338, 2010.

[39] G. Weikum and G. Vossen. *Transactional Information Systems.* Morgan Kaufmann, 2002.

[40] K.-L. Wu, P. S. Yu, and C. Pu. Divergence control for epsilon-serializability. In *ICDE*, pages 506–515, 1992.

[41] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *OSDI*, 2000.

[42] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *SOSP*, pages 276–291, 2013.

APPENDIX

A. EXPRESSING TRANSACTIONS IN $\mathcal{L}$

In this section, we explain how to translate programs in a higher-level language such as SQL or $\mathcal{L}$++ to $\mathcal{L}$. The translation lets us formalize the intuition that adding native support for bounded relations to $\mathcal{L}$ does not alter the expressive power of the language.

$\mathcal{L}$ does not have built-in support for relations or arrays. However, these data structures and read/write accesses to them can be simulated using nested if-then-else statements. An array a of length n is stored in the database as a set of objects $\{a_0, a_1, \ldots, a_{n-1}\}$. The following code reads the $\hat{\imath}$th element of the array a into a temporary variable $\hat{x}$:

```
     if $\hat{\imath} = 0$ then $\hat{x} :=$ read($a_0$)
else if $\hat{\imath} = 1$ then $\hat{x} :=$ read($a_1$)
else if ... else skip
```

We use the command $\hat{x}$:= read($a(\hat{\imath})$) as syntactic sugar for the code above. Analogously, we use the command write($a(\hat{\imath}) := \hat{x}$) as syntactic sugar for a sequence of if-then-else statements that writes the value of the temporary variable $\hat{x}$ into the $\hat{\imath}$th element of a.

Once we have one-dimensional arrays, we can extend them to two-dimensional arrays in the standard manner, by storing a two-dimensional array in row-major order inside a single-dimensional array and accessing elements by offset computation. Thus $a(\hat{\imath})(\hat{\jmath})$ (accessing the i, jth element in the two dimensional array a) is syntactic sugar for $a(\hat{\imath} * D + \hat{\jmath})$, where D is the size of the array along the first dimension.

Two-dimensional arrays allow us to represent relations (tables). If we assume a bound on the maximum number of tuples in any relation, we can encode bounded (foreach) iteration over relations and express SELECT-FROM-WHERE clauses as a sequential scan over the entire relation. In the sequential scan, for each tuple in the array we check whether it matches the selection condition using if-then-else statements.

For SQL UPDATE statements, database insertions and deletes are modeled by "preallocating" extra space in the array and keeping track of used vs. unused space with suitable placeholder values for unused portions of the array.

The above encoding is very straightforward and can easily be performed automatically by a compiler. Encoding more expressive language constructs than SELECT-FROM-WHERE will require more elaborate compilation techniques. Depending on the source language, the process may run into the theoretical expressiveness limits of $\mathcal{L}$; however, we have determined (through manual inspection) that $\mathcal{L}$ is expressive enough to encode all five TPC-C transactions, which are representative of many realistic OLTP workloads.

It is possible to add extensions to $\mathcal{L}$ and make it more expressive so it provides support for unbounded iteration or complex data structures. These extensions are compatible with our analysis, as long as we add appropriate rules for symbolic table computation and understand that the symbolic table formulas φ may need a more expressive logical language.

B. HANDLING REMOTE WRITES

In this section, we explain how to adapt the homeostasis protocol to handle non-local writes, removing the need for Assumption 3.1. The most common use case for remote writes is replicated systems, where – conceptually – a write to an object requires a remote write on each site which keeps a replica of the object.

The basic idea is to add extra logic to the existing protocol that transforms arbitrary transactions into transactions containing only local writes prior to execution. The transformations are conceptually simple and are easily applied automatically, without human intervention. We first explain in detail how to do this for programs in our language $\mathcal{L}$ from Section 2.3, and then discuss how the transformation would apply to programs in other and/or richer languages.

```
$\hat{x}$ := r($x$)
if $0 < \hat{x}$ then
  w($x = \hat{x} - 1$)
else
  w($x = 10$)
```

***(a)* Original transaction**

```
$\hat{x}$ := r($x$) + r($dx_1$)
if $0 < \hat{x}$ then
  w($dx_1 = \hat{x} - 1 -$ r($x$))
else
  w($dx_1 = 10 -$ r($x$))
```

***(b)* Transformed for Site 1**

```
if $0 <$ r($x$) + r($dx_1$) then
  w($dx_1 =$ r($dx_1$) $- 1$)
else
  w($dx_1 = 10 -$ r($x$))
```

***(c)* Transformed and simplified**

Figure 23: Transforming transactions to eliminate non-local writes. The read and write commands are abbreviated as r and w respectively.

Given a set of transactions in $\mathcal{L}$, we proceed as follows. For each database object x we introduce a set of fresh database objects $\{dx_1, dx_2, \ldots, dx_M\}$ – one for each site that runs transactions that perform remote writes on x. Each dx_i is local to site i and is initialized to 0. We modify any transaction executed on site i so it writes to dx_i instead of x.

An example of this transformation is shown in Figure 23. We assume that the original transaction (Figure 23a) runs on site 1, but x resides on some other site. To keep the example simple, we also assume that this transaction is the *only* transaction in the system that reads or writes to x.

The transformed transaction (Figure 23b) writes to the local dx_1 instead of the remote x; however, it maintains the invariant that $x + dx_1$ will always be equal to the "real" value of x. Thus every remote write is replaced with a local one. However, synchronization may still be required because of remote reads. For example, the transaction from Figure 23b must always perform a remote read to determine the value of x.

If we had other transactions in the system that read x, they would need to be transformed to read both x and dx_1 each time, and to work with the sum of these two values instead of just the value of x. If we had transactions running on sites other than i that *write* to x, we would need to introduce additional dx_i objects and apply appropriate transformations. In the common case where x is a replicated object that is read and written on each of K sites, we would introduce a dx_i for every site. Every statement of the form read(x) would be translated to

$$\texttt{read}(x) + \sum_{j\,=\,1 \text{ to } K} \texttt{read}(dx_j)$$

Every write($x = \hat{x}$) for a transaction running on site i would be translated to

$$\texttt{write}(dx_i) = \hat{x} - \texttt{read}(x) - \sum_{j\,=\,1 \text{ to } K,\ j \neq i} \texttt{read}(dx_j)$$

The next step is to eliminate as many remote reads as possible using semantics-preserving program transformations. In our running example, the resulting transaction for Site 1 is shown in Figure 23c. This transaction replaces the expression $\hat{x} - 1 -$ r(x), which references the remote variable x, with the equivalent expression

$$(\mathtt{r}(x) + \mathtt{r}(dx_1)) - 1 - \mathtt{r}(x) \quad = \quad \mathtt{r}(dx_1) - 1$$

which does not. This simplified transaction can avoid remote reads much of the time: if we can generate a treaty which ensures that $0 < x + dx_1$ then the transaction can be executed on Site 1 using only locally available information about the value of dx_1.

All the transformations described above are easy to perform automatically. Once we have applied the transformations to all transaction code, we can run the system using the homeostasis protocol as described previously, because Assumption 3.1 now holds. In practice, we might initialize the dx objects to 0 and reset them to 0 at the end of each protocol round, but this is not required for correctness. The protocol still yields provably correct execution; however, to complete the proof we now need to show that the above transformations preserve the observable behavior of transactions. A full treatment of the issue requires us to define formal evaluation semantics for $\mathcal{L}$, and is beyond the scope of this paper.

For transactions in a language that is richer than $\mathcal{L}$ and may support data types other than integers, it is still possible to perform the above transformation if each data type comes with a suitable *merge function* that is the equivalent of addition in the integer case. Under this merge function, elements of the data type must form an Abelian group. For example, we can apply the transformation to multisets (bags) of integers, using bag union as the merge function.

If the data type we wish to use does not come with a suitable merge function, the above transformations cannot be used and it is necessary to synchronize on every update to a replicated object of this type. However, existing research has shown that there are numerous commutative replicated data types which do come with the merge functions we need [34]. Thus we expect our transformation technique to be broadly applicable in practice.

C. TREATY COMPUTATION DETAILS

C.1 Preprocessing the formula ψ

The preprocessing of ψ involves identifying all subexpressions θ that prevent it from being a conjunction of linear constraints, for example θ may be a negated boolean expression $\neg b$. For each such expression, we replace it within ψ by the actual value of θ on D. In addition, if the variables in θ are $x_1, x_2, \cdots x_n$, we transform ψ to $\psi \wedge \bigwedge_i (x_i = D(x_i))$. Intuitively, any variables involved in the subexpression have their values fixed to the current ones. It is clear that the transformed ψ implies the original one so we can use it in what follows without sacrificing correctness.

C.2 Finding a good treaty configuration

Algorithm 1 computes valid treaty configurations. It has two tunable parameters: a lookahead interval L and a cost factor f. We examine f possible future system executions, where each execution is a sequence of L transactions and is constructed using our workload model. We search for a configuration that minimizes the number of treaty violations in our f executions. For larger f and L, the probability of a future treaty violation decreases, but the search for an optimal configuration takes more time to run.

The configuration search is performed by a suitable MaxSAT solver. We assume that we have an efficient solver that takes as input a first-order logic formula, finds the largest satisfiable subset of constraints that includes all the *hard constraints* and also produces a model (a concrete assignment of values to configuration variables) which satisfies this subset of constraints.

We now discuss the algorithm in more detail. The algorithm first generates θ_h—this is a formula that says the local treaties must enforce the global treaty. This must always be true for a valid treaty configuration, so we will have θ_h as a hard constraint (Line 4). Next (Lines 6-12) the algorithm generates f sequences of transactions of length L using the workload model. For each sequence, it simulates the execution of transactions in the sequence to produce a sequence of databases, one per transactional write (Line 8). The desired behavior is that none of the databases in the sequence violate the local treaties. We therefore add an appropriate constraint for each database in the sequence (Line 10). This process yields a set Θ_s of soft constraints, which are fed to the MaxSAT solver together with the hard constraints in θ_h (Line 13).

Algorithm 1 Finding a valid treaty configuration that reduces the probability of future violation.

```
 1: Let $L$ be the lookahead interval.
 2: Let $f$ be the cost factor.
 3: Let $\varphi = \bigwedge_{i \in [1,K]} \varphi_{\Gamma_i}$ {Conjunction of local treaty templates.}
 4: $\theta_h = \{\forall\ Vars(\varphi_\Gamma),\ \bigwedge_{\{i \in [1:K]\}}(\varphi_{\Gamma_i}) \Rightarrow \varphi_\Gamma\}$ {Local treaties enforce
    global treaty.}
 5: $\Theta_s = \emptyset$
 6: for $i = 1$ to $f$ do
 7:     Generate a sequence $S$ of $L$ transactions using model
 8:     Let $[D_0, D_1, ..., D_L]$ be the sequence of databases obtained
        by executing $S$ on $D$ (we have $D_0 = D$).
 9:     for $j = 1$ to $L$ do
10:         $\Theta_s = \Theta_s \cup \{\varphi(D_j)\}$ {Treaty holds on $D_j$.}
11:     end for
12: end for
13: $C = MaxSAT(\{\theta_h\} \cup \Theta_s)$ {Hard constraints $\theta_h$, soft $\Theta_s$}
```

We illustrate an execution of Algorithm 1 by continuing with our example and our local treaty templates $\varphi_{\Gamma_1} : (x + c_y \geq 20)$ and $\varphi_{\Gamma_2} : (c_x + y \geq 20)$. We have $\theta_h = (\forall[x, y], \varphi_{\Gamma_1} \wedge \varphi_{\Gamma_2} \Rightarrow \varphi_\Gamma)$.

Suppose that $L = 3$ and $f = 3$ and transaction T_1 is twice as likely as T_2. We construct 3 possible transaction sequences of length L drawn from this distribution. Suppose these are $S_1 = [T_1; T_1; T_2]$, $S_2 = [T_1; T_1; T_1]$ and $S_3 = [T_1; T_2; T_1]$. If we execute S_1 on the initial database we obtain the sequence of states $[(10, 13); (9, 13); (8, 13); (8, 12)]$ where each element represents an ordered pair of x and y values. This sequence can be computed by looking up in the symbolic table the partially evaluated transaction for each transaction in S_1 and applying it.

The desired behavior is that no database in the above sequence violates a local treaty. Thus $\varphi_{\Gamma_1} \wedge \varphi_{\Gamma_2}$ must hold on each of the four databases. Plugging in the x and y values from each database into $\varphi_{\Gamma_1} \wedge \varphi_{\Gamma_2}$, taking the conjunction and performing some simplification yields the soft constraints $\{(c_y \geq 12), (c_x \geq 8)\}$. Repeating the procedure for S_2 and S_3, we get soft constraints $\{(c_y \geq 13), (c_x \geq 7)\}$ and $\{(c_y \geq 12), (c_x \geq 8)\}$.

By passing all these constraints to a MaxSAT solver, we find that not all of the soft constraints can be satisfied in addition to θ_h. However, for $c_y = 12$ and $c_x = 8$, we can satisfy both the first and third set of soft constraints. The algorithm therefore chooses $C = \{(c_y = 12, c_x = 8)\}$. Note that this configuration allows more flexibility to the site at which T_1 runs, since it is more frequent. As desired, we have minimized the probability of a treaty violation subject to the computational constraints f and L.

C.3 Lifting Assumption 4.1

We now explain how to handle transactions that violate Assumption 4.1. An example of such a transaction would be one that reads the value of remote object y and copies it into the local object x. Intuitively, if we wish to avoid the remote read of y yet guarantee correct execution, we need a global treaty that guarantees the value of y will never change. We can achieve this by adding a constraint to the local treaty *on the site where y resides*. To do this, we need to perform a post-processing step on the local treaty templates. Specifically, for every transaction T executing at site i, we consider the partially evaluated transaction ϕ_T found in the relevant row of the symbolic table. For every variable x such that `read`(x) appears in ϕ_T but $Loc(x) \neq i$, we substitute it with a new local variable c_x and modify $\varphi_{\Gamma_{Loc(x)}}$ to $\varphi_{\Gamma_{Loc(x)}} \wedge (x = c_x)$. Once this postprocessing step is complete, we can run Algorithm 1 as before.

D. EXAMPLES BEYOND TOP-*K*

In this section, we elaborate on the examples from in Section 4.3.

The first example is the "maximum of minimums" case, which we encode as a program which maintains a list of lists in a two-dimensional array. In our weather scenario, each list represents a day and contains the temperature measurements for that day. The program adds a value to a specific list, finds the minimum value in each list, finds the top k values among these minimums and prints them to the screen.

The output of the program changes if the inserted value is a minimum in its own list and is one of the k highest minimum values among the other lists. Thus, there are $k + 2$ high-level cases to handle: one where the new value is not a minimum in its own list and the remaining $k + 1$ for the case where it is, and has a particular relative ordering with respect to the current top-k minimums. For each of these cases, the program will print a different result.

The actual local treaties required for correct disconnected execution at each site depend on the above information as well on what (portions of the) lists are stored on each site. For instance, if each list is stored on a different site, it is necessary to synchronize when a new minimum value is added to the local list, and this value is higher than the lowest value in the current top-k. If multiple lists are stored on one site or a list is distributed (or indeed replicated) across multiple sites, the treaties will be more complex. The treaties can be expressed as conjunctions of linear constraints and can in principle be derived manually, although the derivation process will require some care if the lists are stored on the various sites according to one of the more complex schemes we described.

Our second example looks for the top-k days with the highest temperature difference, where the temperature difference is the daily high temperature minus the daily low temperature. It can be implemented like the previous example, but now upon inserting a value we find and print the indexes of the top k lists which have the highest difference between max and min.

There are two high-level cases that affect how this program behaves – either the list that was updated enters the top-k after the update or it does not. If it does not, there are three possible reasons and subcases. Either the the new value that is inserted is neither a new min nor a new max in its list, or it is a new max but the resulting new min-max difference for that list is not greater than the current top-k, or it is a new min but the resulting new min-max difference is not greater than the current top-k. If the updated list does enter the top-k, we need $k + 1$ subcases for each relative ordering. As before, the actual treaties required for correct disconnected execution will depend on what is stored where.

Like the first example, this too can in principle be handled manually by a careful and motivated human treaty generator. However, it is unclear how much more complexity can be added without overwhelming the human and introducing errors. On the other hand, our analysis can compute correct symbolic tables and local treaties for both examples automatically.

E. ANALYZING TPC-C TRANSACTIONS

In this section, we explain how to express the TPC-C transactions `New Order`, `Delivery` and `Payment` in $\mathcal{L}$++, and we explain what treaties the homeostasis protocol produces for them.

E.1 Encoding the TPC-C transactions

Recall that the `New Order` transaction places a new order for some quantity (between 1 to 5) of a particular item from a particular district and at a particular warehouse. The `Payment` transaction updates the customer, district and warehouse balances as though processing a payment, and the `Delivery` transaction fulfills orders at a particular warehouse and district.

Expressing these three transactions in $\mathcal{L}$++ is largely straightforward, since they consist primarily of a sequence of SQL `SELECT`, `UPDATE`, `INSERT` and `DELETE` statements, and map directly to relation read and write statements in $\mathcal{L}$++ that are syntactic sugar for $\mathcal{L}$ code, as discussed in the Appendix Section A.

`New Order` and `Delivery` both write to the `NEW-ORDER` table; the TPC-C specification requires an ordering where `New Order` creates orders with monotonically increasing order ids for a given warehouse and district, whereas `Delivery` fulfills the oldest order (i.e. the one with the smallest id) for a given warehouse and district each time. However, the specification does not describe how this ordering should function in a replicated version of TPC-C and what constraints it needs to obey.

For our implementation, we assume that the ordering must exist and that `Delivery` must fulfill the lowest-numbered order each time, but that the assignment of new order ids does not need to be monotonic across all sites. That is, each site generates monotonically increasing order ids (for each district and warehouse) and no two sites can ever generate the same order id. Thus we ensure the existence of a global ordering, but no synchronization is required when `New Order` executes.

Finally, to make our code suitable for a replicated setting, we apply transformations as described in Appendix Section B.

E.2 Treaties for TPC-C

We now explain in more detail what treaties were generated from our $\mathcal{L}$++ implementation described above.

The `Payment` transaction with customer specified by number updates several balance amounts. These amounts are never read by any transaction, including other instances of `Payment`. As explained in Appendix Section B, incrementing a balance by a given amount can be done without reading the amount itself if we use auxiliary local objects to store the local "deltas" of the balance. Thus, no treaties are required on the balance values for the homeostasis protocol, and instances of `Payment` run without ever needing to synchronize.

The `New Order` transaction requires treaties on the stock values for each item and for each warehouse, i.e. on the `S_QUANTITY` for each row of the `STOCK` table in the TPC-C schema. The global treaty for each such quantity q ensures that q may never be negative, i.e. $q >= 0$. The local treaties generated by our protocol constrain q further and have the form $dq_i > c$, where dq_i is the auxiliary local object corresponding to q on site i (see Appendix Section B) and c is chosen through optimization as explained in Section 4.

The `Delivery` transaction, as explained above, needs to delete a tuple from the `NEW-ORDER` table which has the lowest order id for a given district and warehouse. This does require synchronization. In our $\mathcal{L}$++ code, we keep a database object for each warehouse and district that stores the current lowest order id in `NEW-ORDER` for that warehouse and district. Each such object has an associated global treaty that fixes it to its current value in the database. Be-

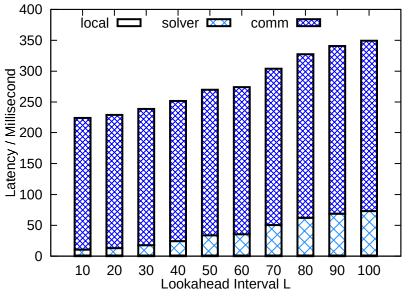


Figure 24: Transaction Latency Breakdown with Lookahead Interval L ($RTT = 100ms, N_c = 16, REFILL = 100$)

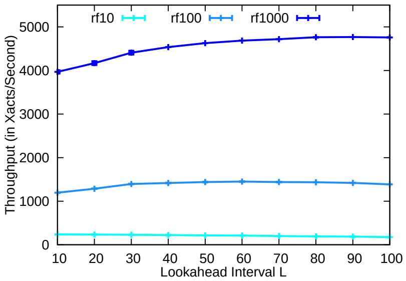


Figure 25: Throughput with Lookahead Interval L for different REFILL values ($RTT = 100ms, N_c = 16$)

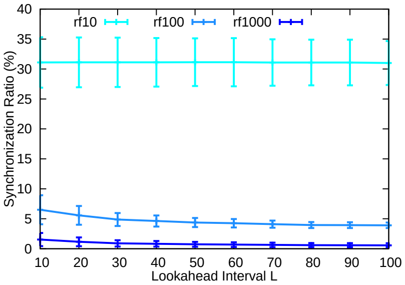


Figure 26: Synchronization Ratio with Lookahead Interval L for different REFILL values ($N_r = 2, RTT = 100ms, N_c = 16$)

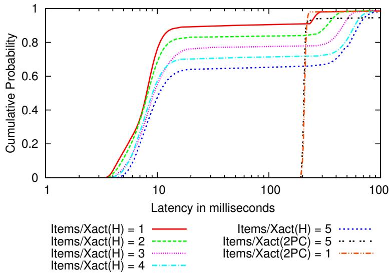


Figure 27: Latency with number of items accessed per transaction ($RTT = 100, REFILL = 100, N_c = 20, L = 20$)

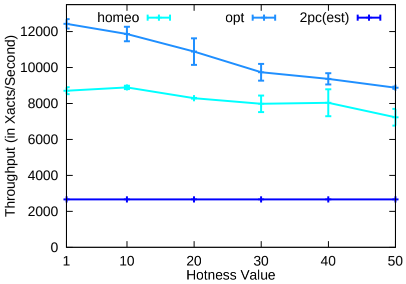


Figure 28: Overall system throughput with H ($REFILL = 100, N_c = 16$)

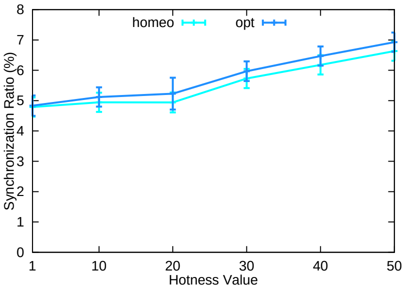


Figure 29: Synchronization Ratio with H ($REFILL = 100, N_c = 16$)

cause Delivery also updates (increments) that object, executing it causes a treaty violation and forces a synchronization. The only transaction reading or writing to that object is Delivery, and so synchronization occurs every time that transaction runs.

In addition, we need to ensure that the Delivery transaction never sees an empty NEW-ORDER table unless the table is truly empty (rather than containing inserts which have not been propagated to the site running Delivery). To achieve this, we maintain a treaty for each warehouse and district that requires the number of unfulfilled orders to remain above zero. If a Delivery transaction processes the order with the *highest* order id that it is "aware of," this will cause a violation of the treaty and synchronization will occur.

In summary, for the TPC-C transactions we use, there is a global treaty of the form $q > 0$ for every quantity q in the STOCK table. There is also a treaty for the current lowest unprocessed order id in the NEW-ORDER table for each warehouse and district, which fixes that order id to its current value. Finally, there is a treaty for each warehouse and district stating that the highest unprocessed order id is greater than the lowest unprocessed order id.

F. ADDITIONAL EXPERIMENTS

In this section, we present a number of additional experiments.

F.1 Microbenchmark setting

Varying the lookahead interval Within the microbenchmark setting from Section 6.1, we experiment with using the homeostasis protocol and varying the lookahead interval L used in Algorithm 1 from Appendix Section C. Varying this parameter trades off the time overhead of treaty generation versus the quality of the generated treaties, where a better treaty is one that requires less frequent synchronization. We also vary the value of REFILL, since a larger REFILL value allows for more flexible treaties. The REFILL values we used were 10, 100 and 1000.

Figure 24 gives the time breakdown for transaction latencies of transactions which caused a treaty violation as we vary L, showing time spent in actual transaction execution (*local*), in communication with other replicas(*comm*), treaty computation and in generating treaties using Algorithm 1 (*solver*). The values for *local* are negligibly small as compared to *comm* and *solver* and therefore do not appear in Figure 24. As L increases, the time spent in finding new treaties also increases. However, since the algorithm is able to find better treaties with larger values of L, the percentage of transactions which actually cause a treaty violation decreases as shown in Figure 26. As expected, using a larger value for REFILL leads to a significantly lower synchronization ratio and higher throughput as shown in Figures 25 and 26.

Varying the number of items accessed For this experiment, we modify our microbenchmark transaction to order more than one item at a time. We vary the number of items purchased in a single transaction from 1 to 5. The latency CDF for the experiment is shown in Figure 27. As expected, as the transactions purchase more items, the probability of synchronization increases since it is more likely that the treaty governing any one of the purchased items is violated. This manifests itself in the downward shift of the inflection point in the CDF. This inflection point differentiates between locally executed transactions and those which cause a treaty violation. Under 2PC, the system behavior is unaffected by increasing the number of items ordered per transaction.

F.2 TPC-C setting

Finally, we give detailed results for the distributed TPC-C experiment introduced in Section 6.2. Figure 28 shows how increasing the value of H (the percentage of transactions ordering "hot" items) affects transaction throughput. As the value of H increases, the treaties corresponding to hot items are violated more often, thus throughput falls. This increase in treaty violations is also apparent in Figure 29 where the synchronization ratio increases with increasing values of H. In summary, in a realistic workload and deployment scenario, the homeostasis protocol far outperforms 2PC and achieves near optimal throughput without sacrificing consistency.