Paper ID: 314

Enabling Wide-Area Replication of Database Services with Continuous Consistency[*]

Kevin Walsh, Amin Vahdat, Jun Yang
Department of Computer Science, Duke University
{walsh,vahdat,junyang}@cs.duke.edu

Abstract

Recently, a number of industrial and academic efforts have demonstrated the utility of wide-area replication for improving both the performance and availability of static or slowly changing web content. However, the overhead of strong consistency limits the utility of wide-area replication for important classes of commercial database applications, such as e-commerce, where multiple users may simultaneously be operating on the same underlying data items. Optimistic consistency protocols are typically inappropriate for these workloads because of the potential for unbounded divergence. Thus, wide-area replication for such workloads is often relegated to asynchronous replication between a primary and backup, for example, as insurance for the case of catastrophic failure of one site.

This paper investigates the appropriateness of continuous consistency as the basic abstraction to support wide-spread replication of transactional network services. Our key insight is that applications benefit from dynamically bounding their maximum distance from strong consistency in response to changing database, network, and client characteristics. In this paper, we: (i) modify an existing implementation of TPC-W, a web e-commerce benchmark, to leverage continuous consistency, (ii) discuss how continuous consistency can be integrated into SQL-based legacy database applications (with the goal of enabling wide-area replication) in a straightforward and semi-automatic manner, and (iii) evaluate both the performance benefits and consistency costs of continuous consistency for our TPC-W implementation across a variety of replication scenarios and consistency bounds. At a high level, we find it possible to achieve much of the performance of local-area replication, with acceptable degradation of client-perceived data consistency for our target application, workload, and network scenarios.

1 Introduction

Wide-area replication is playing an increasingly important role in the deployment of robust network services. The realities of wide-area networks, including high latency, low bandwidth and frequent outages, dictate that service content and functionality be distributed as close as possible to end clients to achieve peak levels of performance and availability. Wide-area distribution of static content—through content distribution networks and web caching—has been largely successful at meeting these goals. Unfortunately, the overhead of strong consistency [5] limits the utility of

[*]This research is supported in part by the National Science Foundation (EIA-99772879, ITR-0082912), Hewlett Packard, IBM, Intel, and Microsoft. Vahdat is also supported by an NSF CAREER award (CCR-9984328).

replication for important classes of commercial database applications, such as e-commerce. Optimistic consistency models [7, 12, 13] are typically inappropriate for such workloads because of the potential for individual replicas to quickly become "delusional" [10].

In this paper, we investigate the costs and benefits of using a *continuous consistency* model called *TACT* [28] to enable wide-area replication for commercial database services. The key idea behind continuous consistency is that applications can dynamically bound their maximum distance from strong consistency. Through relaxing consistency in this model, applications receive commensurate benefits in performance and availability, at a high level by allowing an increasing portion of commit operations to occur asynchronously. At the same time, the maximum distance from strong consistency is tightly bounded, limiting the effects of inconsistency on user-perceived views of the data. Consistency bounds can be set at arbitrary granularity and dynamically in response to changing application and network characteristics. Importantly, continuous consistency allows application to specify strong consistency for specific data items or transactions where required and weaker consistency bounds where appropriate. We describe additional details of the TACT model in Section 2, and the adaptation of TACT as database middleware in Section 3.

While continuous consistency provides the potential for many improvements [28], its applicability will be limited if traditional SQL-based database applications are incompatible with the model. Thus, a key goal of this work is to explore the process of integrating continuous consistency into TPC-W [25], an industry-standard commercial database benchmark simulating an e-commerce web service. As a complete and realistic example of a web e-commerce service, TPC-W is well suited for evaluating the effectiveness of wide-area replication in network services. In Section 4, we present an overview of the TPC-W specification and discuss our experience in porting TPC-W from a conventional database approach to TACT. Based on this experience, we discuss how continuous consistency can be integrated into SQL-based legacy database applications (with the goal of enabling wide-area replication) in a straightforward and semi-automatic manner.

We evaluate performance benefits and develop a set of application-specific consistency metrics to measure both the costs and benefits of continuous consistency under a variety of conditions. The results in Section 5 show how service administrators can appropriately control consistency parameters to suit specific network conditions, business requirements, and performance goals. In general, we find that most of the performance benefits of optimistic consistency come with a relatively small relaxation in consistency. Thus, it possible to achieve much of the performance of local-area replication with acceptable degradation of client-perceived data consistency for our target application, workload, and network scenarios. For example, one result in Section 5 shows that a TPC-W service with six wide-area replicas can offer a three-fold improvement in throughput over a centralized service, while the application-specific inconsistency metric, measured by the percentage of orders for oversold items, is kept within 0.5%. If the service runs under strong consistency, the throughput would be an order of magnitude less because of the high overhead of strong consistency for wide-area replication.

2 TACT System Model

TACT is a middleware system providing weakly consistent replicated storage while guaranteeing application-specified bounds on the amount of inconsistency across the set of replicas. In this section, we present an overview of the generic TACT system model from [28]. In the next section, we discuss how we adapt TACT for use as database middleware appropriate for standard SQL-based database applications.

TACT mediates access to the underlying data store by accepting application-defined *read* and *write procedures*. With strong consistency, these procedures always return accurate results (as defined by the global commit order of these procedures). The key question in our continuous consistency model is whether the final and global commit order must be determined before successfully returning from a write or can be established through subsequent asynchronous reconciliation with remote replicas. With relaxed consistency, these procedures may return tentative results that differ from the accurate ones. The meaning and significance of this difference to the end users depend on application semantics.

To ensure that all replicas converge, all writes must be executed in a globally consistent order. This means that writes are sometimes reordered during the reconciliation process, potentially changing or invalidating the results of reads and writes. A write is considered *outstanding* if its final ordering is not known, meaning that it may be reordered in the process of reconciliation. A write is considered *stabilized* when its final ordering is known. Under continuous consistency, the system is guaranteed to achieve *eventual consistency* in the absence of additional writes [24].

TACT allows the application to export its own consistency requirements by defining *conits* and specifying bounds for them. A *conit* is the granularity over which the application specifies consistency requirements. The application may define multiple overlapping conits, each with its own set of tunable consistency bounds. Each write procedure may affect zero or more conits with positive or negative weight. The application controls consistency by bounding three metrics for each conit: the global weight of outstanding writes (*numerical error*), the number of outstanding writes on the local replica (*order error*), and the maximum age of outstanding writes (*staleness*). Strong consistency corresponds to numerical error of 0, order error of 0, and order error of 0.

As an example, consider an online bookstore with *basic* and *preferred* customers. Service replicas accept orders and process them locally against the current known state of the inventory. A transaction remains outstanding at a replica until all replicas have agreed on its global ordering. Hence, there is a delay before an order can be absolutely confirmed, during which time other replicas may not have learned of the order. To control this potential inconsistency, we can define a conit over the set of all customers' orders. Order error limits the number of orders a replica may have outstanding, and numerical error bounds the number of orders outstanding in the entire system. A second conit, defined over the set of preferred customer orders, can provide a higher level of guarantee through tighter consistency bounds. Administrators may define a third conit tracking the number of books ordered. The bounds on this conit ensure that inventory is bounded-consistent on each replica (ensuring, for example, that books corresponding to a given conit can only be oversold by a maximum amount). Finally, we can ensure that the inventory for each best-selling book is kept accurate by defining, for each best-seller, a conit tracking its current inventory.

We can view TACT's enforcement of consistency requirements as a pipeline, distributed across many replicas, and designed to hide the high latency of distributed transaction ordering. As shown in Figure 1, each replica maintains a history of write procedures. New procedures execute on the local data store, and return tentative results to the application as they enter the pipeline of outstanding writes. TACT performs background reconciliation with other replicas to stabilize older

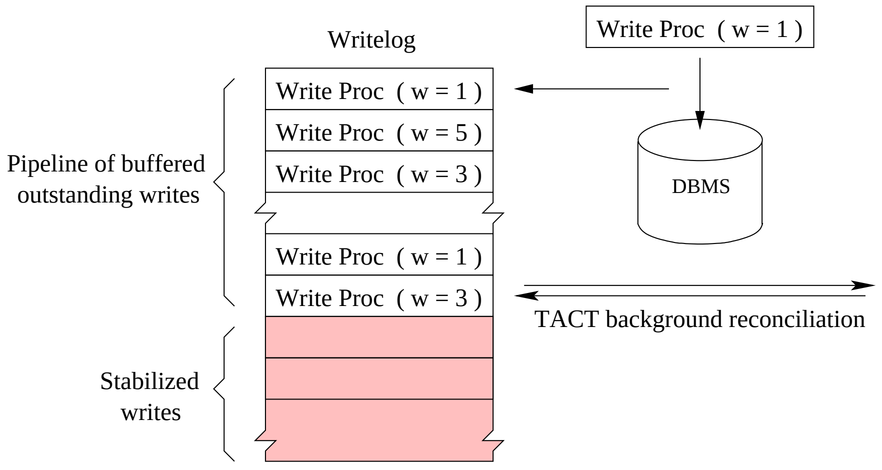

Figure 1: Pipelined reconciliation of outstanding writes.

writes. The history queue depth is controlled by consistency bounds. This logical pipeline stalls if the number of outstanding writes (either locally or globally) exceeds specified conit bounds. The pipeline starts again once enough writes have been committed to bring consistency back within bounds. For maximum performance, consistency bounds should be set at a level such that the pipeline never stalls, thus hiding the full latency of wide-area replication while achieving the benefits of increased throughput and availability. The relationship between consistency bounds and the history queue depth is essentially the same as that of buffer size and the bandwidth-delay product in a network pipeline situation. Of course, changing network conditions and client access rates can affect the optimal pipeline depth.

The power and flexibility of TACT come at a price of increased application complexity. The application needs to choose appropriate conits and specify the weight of each write procedure on each conit. Furthermore, the application must be able to tolerate or avoid fluctuations in the results of its operations. In the case of relaxed consistency, the results initially returned to the application may be tentative, since they are based on incomplete and possibly inaccurate data. Consider again the example of an online bookstore. A local replica observes, say, 5 copies of a book in stock, but the replica has not yet seen an earlier remote transaction a requesting 2 copies of the same book. A customer places an order b for 5 copies of the book. The order is accepted, executes locally (depleting the entire stock), and returns a success message to the customer. A second customer placing an order c for 3 copies will be rejected due to insufficient inventory. After the application-defined consistency bounds have ensured a unique global ordering, the local replica might reorder transaction a before transaction b. Now, the second order b must be rejected due to insufficient stock. The application may notify the customer of the problem, or choose to wait until stock is replenished and retry the order. Note however that the rejected order c could have in fact been accepted by a strong consistency service. By dynamically controlling consistency levels (potentially on a per-access basis), the application can control the maximum rate of such inconsistent access. In the remainder of this paper, we demonstrate that the increased application complexity is not only manageable (modifications to our target application are straightforward and can be done semi-automatically) but also worthwhile because of significant performance benefits from continuous consistency and because of a typically acceptable rate of inconsistent access. Further, we find that most of the performance benefits of optimistic consistency are available from relatively small relaxations in consistency, and hence acceptably low levels of inconsistent access for our target workload.

The TACT system model also introduces several restrictions on application programming. First, TACT requires all write procedures to be *reproducible*, i.e., they must produce the same effects when executed at a later time on remote nodes. This property ensures that each replica will converge to the same data values after executing all writes in the correct order. Second, TACT requires all write procedures to be undoable, because TACT may need to reorder writes during a reconciliation. In the general case, it may be necessary for the application to provide undo procedures, *compensations* [9], or conflict resolution logic explicitly. In the following section, however, we argue that neither limitation is likely to be significant when TACT is used as database middleware for standard SQL applications. For instance, we were able to recover all necessary undo state from the database log rather than requiring explicit undo support.

3 TACT as Database Middleware

A primary goal of our work is to demonstrate that the assumptions and limitations imposed by the TACT system model are compatible with the use of a commercial database system under the TACT layer. That is, we aim to show it is possible to gain the performance, availability, and semantic benefits of continuous consistency without sacrificing the ease of use and ubiquity of SQL and commercial database products.

To simplify porting JDBC-based database applications to TACT-based applications, we design the TACT database API based on JDBC. Other standard database API's can be similarly used as a basis for TACT. Our API provides functions to create and execute `ReadProc` and `WriteProc` objects (read and write procedures, respectively), defining conits, setting conit bounds, and specifying the weight of a `WriteProc` on a conit. Section 4 provides examples of code that uses this API.

In general, a `ReadProc` object executes on an underlying database as a read-only SQL transaction; a `WriteProc` object executes as an SQL transaction that modifies the database. In their simplest form, `ReadProc` and `WriteProc` correspond to single-statement SQL transactions, just like JDBC `Statement` under the auto-commit mode. Interfaces for creating and executing `ReadProc` and `WriteProc` objects mimic the JDBC interfaces for creating and executing `Statement` objects. To execute `ReadProc` and `WriteProc` objects, TACT simply passes the SQL strings to the underlying database system via JDBC. The results returned from the database system are passed back to the application directly, with no additional translation. In their most general form, `ReadProc` and `WriteProc` can encapsulate application code that dynamically generates multi-statement SQL transactions. To create these objects, applications need to define subclasses of `ReadProc` and `WriteProc` and override the default `execute()` method. TACT executes all SQL statements in the `execute()` method within a single SQL transaction on the underlying database system.

Handling Non-Reproducible Write Procedures

At each replica, TACT keeps a log of all `WriteProc` objects received from the application, so that they can be executed later at other replicas. In order to ensure the correctness of redo, we must require all `WriteProc` objects to be reproducible. An example of a non-reproducible `WriteProc` object would be an SQL statement that uses built-in SQL functions such as `RAND` and `CURRENT_TIME`, which may return different values when executed at a later time. Application code may also contain random number generators or system calls that return non-reproducible results.

It is possible for TACT to enforce the reproducible property automatically for simple `WriteProc` objects with only SQL statements and no application code. When a simple `WriteProc` object is created, TACT can parse and analyze the SQL string to identify non-reproducible calls. TACT evaluates these calls once, and then replaces them with their return values. The `WriteProc` object with the modified SQL statement is logged by TACT so its effects can be safely reproduced later at another replica.

For complex `WriteProc` objects containing application code, the burden is on the application programmer to identify and remove non-reproducible calls in code, because this kind of program analysis is intractable in general. Once these calls are identified, however, removing them is a simple and mechanical procedure: These calls are evaluated once and their results are stored in the `WriteProc` object for future executions. We show an example of this procedure in Section 4.2.

An alternative approach to handling non-reproducible write procedures would be to extract its redo information directly from the transaction log of the underlying database system. This approach has been used by many commercial products (e.g., DB2 DataPropagator and Oracle SharePlex) for replication. Unfortunately, it does not work in our case. Since write procedures may be reordered during reconciliation, they may see database states different from those they first executed in, and consequently, their behaviors may be different from those recorded in the database transaction log.

Undoing Write Procedures

As discussed at the end of Section 2, TACT must be able to revert the database to an earlier state during reconciliation. Normally, the application programmer would need to supply an undo or compensation procedure for each `WriteProc`. Fortunately, this need is eliminated when we use TACT as a database middleware. Since `WriteProc` objects execute as SQL transactions in the underlying database system, the database transaction log provides enough information for TACT to revert the database to any earlier state.

Specifying Conits and Weights of Write Procedures

By design, TACT is unaware of the meaning or the semantic definition of any conit. TACT only cares about what bounds are desired for each conit, and how each `WriteProc` affects each conit. Therefore, it is up to the application to compute the proper weights of a `WriteProc` on different conits, and specify these weights through the API. An additional requirement imposed by the current formulation of TACT is that the weight of a `WriteProc` must be independent of the database state. This requirement effectively restricts our choices of conits to those whose bounds can be maintained efficiently by looking at each `WriteProc` alone without examining the database state. For applications studied under TACT, this requirement has not presented any limitation [30]. In Sections 4.1 and 5.4, we will see this requirement is met by all conits used in our TPC-W implementation, including a non-trivial conit that controls the accuracy of best-seller lists.

One interesting possibility we intend to pursue is allowing the application to define the numerical value of a conit as a database view (most likely an aggregate view) whose content contains a single numerical value. Thus, the effect of each SQL transaction on each conit can be calculated automatically using techniques for view maintenance [11]. In the general case (e.g., when the view is not self-maintainable), the database state must be accessed in order to determine the effect of an update, which incurs a high overhead. We will need to study this trade-off between generality and efficiency carefully.

Generality of the Numerical Error Metric

Numerical error provides a general and flexible way for applications to define their own notions of distance from strong consistency. Besides measuring the difference between two numerical quantities, numerical error also can capture the difference between two sets. Here, we focus on several examples that are relevant in a database setting. For more discussion on the generality of TACT and examples beyond databases, please refer to [30].

- **Bounding errors in the results of `SUM`, `COUNT`, and `AVG` aggregates.** In the case of `SUM`, the weight of a `WriteProc` is the net effect (sum) of its changes on the column values being aggregated. We can bound the error in `SUM` by bounding the absolute value of the total weight of all outstanding `WriteProcs` globally. The case of `COUNT` can be seen as a special case of `SUM` where every row has value one. In the case of `AVG`, we can bound the error by bounding the errors in both `SUM` and `COUNT`.

- **Bounding errors in the results of `MIN`, `MAX` aggregates and top-k queries.** Like `SUM`, top-k queries are based on numerical quantities. However, unlike `SUM`, top-k queries return set-valued results, so the error must be defined differently. We show how to handle these queries in Section 5.4. `MIN` and `MAX` can be seen as special cases of top-k queries where $k = 1$.

- **Bounding errors in tables.** A reasonable numerical measure of error between the actual state of a table T and an observed state T' is $1 - \frac{|T \cap T'|}{|T \cup T'|}$. Suppose we want to bound this error. In the worst case, a deletion may decrease $|T \cap T'|$ by one while keeping $|T \cup T'|$ constant. In this case, the increase in error is at most $(1 - \frac{|T \cap T'|-1}{|T \cup T'|}) - (1 - \frac{|T \cap T'|}{|T \cup T'|}) = \frac{1}{|T \cup T'|} \leq \frac{1}{|T'|}$. Therefore, we can assign weight $\frac{1}{|T'|}$ to each deletion. In the worst case, an insertion may increase $|T \cup T'|$ by one while keeping $|T \cap T'|$ constant. In this case, the increase in error is at most $(1 - \frac{|T \cap T'|}{|T \cup T'|+1}) - (1 - \frac{|T \cap T'|}{|T \cup T'|}) = \frac{|T \cap T'|}{|T \cup T'|} \cdot \frac{1}{|T \cup T'|+1} \leq \frac{1}{|T'|+1}$, and we may assign weight $\frac{1}{|T'|+1}$ to each insertion. Since we can treat an update as a deletion followed by an insertion, the weight of an update should be $\frac{1}{|T'|} + \frac{1}{|T'|+1}$.

  Two points are worth noting here. First, we assign worst-case weights to `WriteProcs` since it is too expensive and hence impractical to compute the precise effects of `WriteProcs` on the numerical error. Using worst-case weights is appropriate because we only care about bounding the error instead of knowing its precise value. Second, weight computation requires the value of $|T'|$, the size of the table in the currently observed database state. Strictly speaking, the current formulation of TACT does not allow weights to be computed from the database state, even though in this particular case the value of $|T'|$ is not expensive to obtain. In practice, however, this limitation is not severe because we usually know a lower bound for the table size, which can be used in place of $|T'|$.

- **Bounding errors in the results of selection queries.** This case is a straightforward extension of the previous one. The error in this case is $1 - \frac{|\sigma_c T \cap \sigma_c T'|}{|\sigma_c T \cup \sigma_c T'|}$. For each deletion and insertion, we evaluate the selection condition c on the modified row. If c is not satisfied, the weight is zero; otherwise, the weight is $\frac{1}{|\sigma_c T'|}$ for a deletion and $\frac{1}{|\sigma_c T'|+1}$ for an insertion. Again, we can treat an update as a deletion followed by an insertion. To obtain the value of $|\sigma_c T'|$ at runtime for weight computation, we do not need to evaluate the selection query; instead, we can maintain the value of $|\sigma_c T'|$ (a single number) incrementally for each database modification. We plan to adopt this approach in the future when we lift the requirement of TACT that weight computation cannot reference the local database state.

- **Bounding errors in the results of joins.** Bounding errors for joins is inherently difficult because even a slight modification to one of the joining tables may cause large changes in the join results. Consider a join query $R \bowtie_c S$. The error is $1 - \frac{|(R \bowtie_c S) \cap (R' \bowtie_c S')|}{|(R \bowtie_c S) \cup (R' \bowtie_c S')|}$. For example, consider the case of an insertion into R'. Suppose this insertion increases $|R' \bowtie_c S'|$ by δ, where δ is between 0 (the inserted row joins with no row in S') and $|S'|$ (the inserted row joins with every row in S'). In the worst case, the error increases by $\frac{1}{1+|R' \bowtie_c S'|/\delta}$. Intuitively, when δ is small compared with the original size of the join, we have a small increase in error; otherwise, the error will likely exceed the bound, forcing immediate remote reconciliation. To facilitate efficient evaluation of $|R' \bowtie_c S'|$ and δ at runtime, we can either maintain $R' \bowtie_c S'$ as a materialized view (possibly with additional auxiliary views on R' and S'), or use histogram techniques to obtain rough bounds on $|R' \bowtie_c S'|$ and δ.

Implementation

The focus of our work is to demonstrate the feasibility and benefits of using TACT as a database middleware layer. However, integrating TACT into a production-quality database system would require significantly more effort and time, especially interfacing with the database transaction log. Therefore, for the purpose of evaluating the TPC-W implementation described in Section 4, we outfit TACT with a simple in-memory database system implemented in Java. This database supports a subset of SQL with hand-coded query execution plans. The database also implements a simple transaction log, which is used by TACT for undo. Due to memory limitations and the relatively poor performance of our simple database implementation, we restrict our attention to small datasets. However, we expect that continuous consistency in general, and the TACT model in particular, will scale well with the size of the dataset as shown by our earlier work [29].

4 TPC-W Implementation

The TPC-W benchmark is designed to evaluate the performance of a complete web e-commerce solution. The benchmark specification details database layout and sizing parameters, the set of database transactions used by the server, the set of possible web interactions, and the design of a load generator. The server implements a small but functional online book store. The load generator emulates the behavior of customers interacting with the web site through HTTP requests.

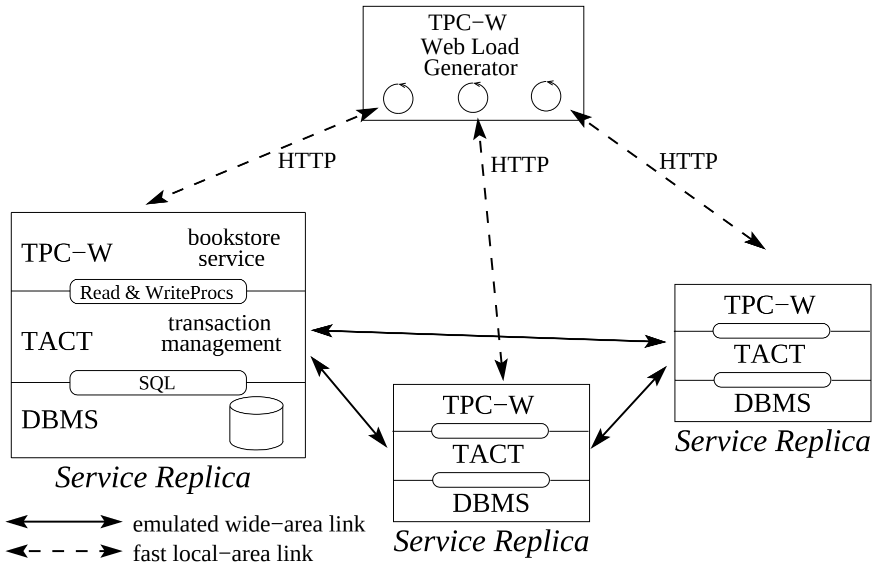

Figure 2: Design of TPC-W using TACT middleware layer for replication.

Figure 2 illustrates the overall design of our TPC-W implementation using TACT. Each service replica uses the TACT database middleware to manage all transactions. TACT maintains consistency bounds by reconciling with remote TACT instances. Our load generator uses multiple threads to send standard HTTP requests to each replica in a simple round-robin balancing scheme. Each emulated user session is assigned to a single replica for the duration of the session. In practice, this behavior is implemented in load-balancing switches or proxies, and is a natural assumption in commercial settings when replicas are distributed either across the wide area or a LAN. The problem of replica selection and load balancing [16, 19] is largely orthogonal to the issues examined in this paper.

We deviate from the TPC-W specification in several ways. First, we do not provide an external payment verification gateway or secure transactions, as required by the specification. We do not consider this omission to be a significant limitation of this work. Second, we disable all image retrieval code in the load generator. In practical settings, it is likely that images will be cached and served from dedicated image proxies or content distribution networks. Thus, we want our experimental results to focus on the performance of the database transactions themselves rather than auxiliary and static data. In general, we expect the cost of accessing dynamic content (and the associated database operations) to dominate the cost of image retrieval.

In this section, we describe our experience in porting TPC-W from a conventional database approach to TACT. A Java implementation of the TPC-W benchmark from the University of Wisconsin [6] serves as a base for our prototype. In addition to the changes detailed below, we have extended and modified the base implementation to allow for easier configuration and extra auditing of the server and load generator. We have equipped our load generator with a simple mechanism to limit the maximum rate of interactions in order to evaluate our implementation under both low and high constant loads. We have also made minor changes to allow the service to run under Apache with the Tomcat servlet extensions, the Java Web Server from Sun Microsystems, or as a stand-alone web server. For this paper, we evaluate TPC-W running under the Java Web Server.

In the base implementation, the server consists primarily of a set of Java servlets implementing 14 web interactions, and the `TPCW_datastore` class. Servlets access all service state through the `TPCW_datastore` class, which in turn uses JDBC to interface with an external SQL database system. In porting TPC-W to TACT, we only modify the `TPCW_datastore` class. At a high level, the tasks include defining conits and their consistency bounds, and converting JDBC transactions to TACT read and write procedures. Since TACT exposes an interface similar to JDBC, and similarly mediates access to an external SQL database, such modifications are fairly straightforward.

4.1 Defining Conits

In this section, we describe our choice of conits for TPC-W and the rationale behind the design. The six write transactions in TPC-W can be cleanly divided by functionality into four categories. We define four conits, each corresponding to a category of write transactions: those relating to *orders*, *carts*, *admin*, and *user* data. For more complex scenarios, it is possible to define, for example, separate conits over different classes of inventory (by author, genre, or popularity for example).

Each of the six write transactions affects exactly one conit, typically with unit weight. We use unit weight for most writes because there is no clear reason, administrative or otherwise, to give more weight to any particular write. For example, all *createNewCustomer* transactions affect the *user* conit with unit weight. In practice, weights might be chosen according to some subjective importance, with more weight placed on, say, *preferred* customer creation transactions.

By specifying a bound on the numerical error of the *user* conit we can limit the total weight (i.e. number of writes) of uncommitted *createNewCustomer* transactions system-wide. A bound on the order error for this conit will limit the number of these transactions allowed at any one replica. The staleness bound controls the maximum amount of time a new customer must wait before the registration is definitively committed. The conit bounds for *carts*, and *admin* have similar interpretations.

The *placeOrder* transaction places an order of one or more books. In this case, it is natural to assign weight to a transaction proportional to the size of the order, in number of books. Administrators may alternatively use the total dollar amount of the order, but this approach is less meaningful in TPC-W, where prices are chosen from a uniform random distribution and change randomly. Here, a numerical error bound on *orders* limits the total number of books ordered in uncommitted transactions system-wide. The interpretation of order error and staleness bounds is similar, bounding the number of outstanding book orders allowed on a single replica and the maximum lifetime of outstanding order transactions before being committed.

In TACT, conit bounds can be specified for each replica and for each transaction, and they can be changed in response to dynamically changing system characteristics. Thus, consistency may be tightened for a fast-selling book (or group of books with similar characteristics) with low inventory on hand. Conversely, consistency can be relaxed for a less popular book with significant inventory on hand. In this paper, we evaluate only static consistency bounds, and do not take advantage of relative or dynamic consistency bounds. In practice however, we believe that the availability of dynamic consistency bounds to be a significant advantage of our approach.

To simplify our evaluation of the effect of consistency bounds, we restrict our attention to consistency policies using a single metric, either numerical error or order error. The staleness bounds are left at the default value of *infinity*. Since the load is roughly constant throughout each experiment, bounding staleness will show results very similar to those of numerical error (experiments omitted for brevity verify this). In real-world situations, load will not be constant and the mix of transactions will vary over time. In such a situation, all three consistency bounds can be used to control different aspects of the consistency policy.

For each of our experiments we choose a single scale factor C and use it to control the numerical error or order error bound for each of the four conits. In our experiments (omitted for brevity), we find that the particular ratios between the bounds for each conit are not critical. A relatively higher or lower bound on a conit increases or decreases overall performance in proportion to the frequency of writes affecting the conit. For this paper, we chose the values C, $3C$, $C/2$, and C for the bound on the *orders*, *carts*, *admin*, and *user* conits, respectively, based on the the subjective importance and expected frequency of transactions. Here, we have placed higher subjective importance on the relatively infrequent administrative updates by specifying a low bound ($C/2$) on the *admin* conit.

4.2 Writing Read and Write Procedures

The bulk of the `TPCW_datastore` class is organized as a collection of 18 Java methods, each implementing a single logical SQL transaction. Access to service state is provided exclusively through these methods. In porting `TPCW_datastore` from JDBC to TACT, we change each of the 18 methods in a largely systematic manner. In total, 12 read-only JDBC statements and 6 read-write JDBC statements are converted to TACT `ReadProc` and `WriteProc` objects. A single programmer familiar with both TACT and TPC-W performed all modifications in three days.

The top half of Figure 3 presents an example of converting a JDBC query to a TACT read procedure. The JDBC query accesses a book record from the database based on a book id number, and returns an object containing the book information. The TACT read procedure is essentially identical in all respects. Most of the porting work requires only the simple conversion illustrated in this example.

The bottom half of Figure 3 provides an example of converting a JDBC update to a TACT write procedure. Line `38` on the right specifies the weight of this write procedure on the *user* conit. This complicated example is chosen purposefully to illustrate the major porting problems we have encountered. Some of these problems are the result of peculiar design decisions in the base implementation, such as the dependence on language-level locks rather than database transactions. Other problems represent artifacts of the TPC-W specification, such as exposing primary key values as user-visible handles, and the use of primary keys for generating user names and passwords. These peculiarities and artifacts present obstacles to the use of TACT. Rather than eliminating the offending code in favor of a more mainstream design, we choose to examine how TACT might be able to address each issue. The idea here is to demonstrate that even legacy code that was not written with continuous consistency in mind can be retrofitted in a straightforward (or in certain cases, even automatic) manner. This makes it more likely that the large existing code base associated with dynamic web services could benefit from continuous consistency with fairly little porting effort. In the rest of this section, we discuss these issues in detail.

First, all uses of random numbers or the system clock in write transactions must be removed, since TACT requires that each write procedure be a deterministic function of the current database state. In the Madison implementation of TPC-W, five transactions use the current system clock value as a timestamp; two of these also use random numbers. In Figure 3, lines `31`–`33` show how the

Madison JDBC Query

```
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;

Book getBook(int i_id) {
  Connection conn = ...
  Statement s = conn.createStatement(
     "SELECT * FROM item, author "
    +"WHERE item.i_a_id = author.a_id "
    +"AND item.i_id = "+i_id);
  ResultSet rs = conn.executeQuery(s);
  if (rs.next()) return new Book(rs);
  else return null;
}
```

TACT ReadProc

```
import tact.TactServer;
import tact.ReadProc;
import tact.ResultSet;

Book getBook(int i_id) {
  TactServer tact = ...
  ReadProc r = tact.createReadProc(
    "SELECT * FROM item, author "
   +"WHERE item.i_a_id = author.a_id "
   +"AND item.i_id = "+i_id);
  ResultSet rs = tact.submitReadProc(s);
  if (rs.next()) return new Book(rs);
  else return null;
}
```

Madison JDBC Update

```
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;

int createNewCustomer(Customer cust) {
  Connection conn = ...
  synchronized {
    Statement q = conn.createStatement(
      "SELECT MAX(c_id) FROM users");
    ResultSet rs = conn.executeQuery(q);
    next_id = rs.getInt(1) + 1;

    cust.c_id = next_id;
    cust.c_uname = chooseName(cust.c_id);
    cust.c_passwd = tolower(cust.c_uname);

    cust.c_discount = "RAND * 51"
    cust.c_login = "CURRENT_TIME";
    cust.c_last_visit = "CURRENT_TIME";

    Statement u = conn.createStatement(
      "INSERT INTO users VALUES("
      +cust.c_id+","+cust.c_uname+...);
    conn.executeUpdate(u);
  }
  return next_id;
}
```

TACT WriteProc

```
import tact.TactServer;
import tact.WriteProc;
import tact.ResultSet;

int createNewCustomer(Customer cust) {
  TactServer tact = ...
  try {
    next_id = centralServer.newUserId();
  } catch (java.rmi.RemoteException e) {
    return -1;
  }

  cust.c_id = next_id;
  cust.c_uname = chooseName(cust.c_id);
  cust.c_passwd = tolower(cust.c_uname);

  cust.c_discount = random() * 51;
  cust.c_login = currentTime();
  cust.c_last_visit = currentTime();

  WriteProc w = tact.createWriteProc(
    "INSERT INTO users VALUES("
    +cust.c_id+","+cust.c_uname+...);
  w.affectsConit("user conit", 1);
  tact.submitWriteProc(w);
  return next_id;
}
```

Figure 3: Pseudo-code examples of converting JDBC transactions to TACT read/write procedures.

TACT version replaces these non-reproducible function calls with a priori calculations. In TPC-W, all uses of random numbers are artificial, as can be clearly seen in this example. Furthermore, as discussed in Section 3, a more robust implementation of TACT would be able to handle non-reproducible SQL statements automatically.

A second issue is the use of Java synchronization blocks, shown on lines `21`–`39` of the Madison implementation. This practice is for the most part incompatible with any replication, and should therefore be abandoned in favor of database-level transactions. One solution would be to code the body of the synchronized block as a custom `WriteProc`, to be executed by TACT as a single database transaction. This cut-and-paste approach is sufficient for several transactions. In the case shown in Figure 3, however, a third problem requires more attention and thus a different solution.

The third, and most serious difficulty deals with external caching of tentative results. In the TACT model, the tentative results of a procedure are returned to the user after being executed only on the local replica. Later, TACT replicas reconcile outstanding writes and some writes may be re-ordered, potentially invalidating the tentative results. This is a problem only if the tentative results are cached outside of the server, or displayed to users. Besides being an annoyance to users, the invalid results may be re-introduced to the system later as part of a query. In the case of *createNewCustomer*, a unique user id is returned immediately to the user for use in later requests. In Madison TPC-W, this unique key is simply the primary key, or row number, of the customer data. Since TACT may re-order insertions at a later time, we must choose the user id a priori, rather than relying on insertion order.

The problem for our implementation thus becomes one of distributed primary key selection. Replicated databases typically use key-space partitioning or globally unique key generation to ensure unique primary keys. However, the TPC-W specification requires, for simplicity of the load generator, that all user id's be consecutive numbers, ruling out both of these approaches. We choose, instead, to implement a simple centralized key manager, which is used on lines `21`–`25` of Figure 3. This approach is used for the three transactions in which primary keys are passed to the client: *createNewCustomer*, *newShoppingCart*, and *placeOrder*. The approach works well since these transactions are relatively infrequent. More importantly, a real-world application is unlikely to have any of the constraints of TPC-W, and will then be free to use more traditional and efficient key management strategies. Alternatively, administrators may choose to implement certain transactions, such as those discussed here, at a central server for administrative or security reasons. Centralized operation does, of course, reduce performance and the benefits of replication.

A final issue, not illustrated by Figure 3, arises in the case of a complex read or write procedure involving multiple SQL statements dynamically generated by application code according to some business logic. In general, the application code can generate subsequent SQL statements based on the results of earlier SQL statements in the same transaction. Since TACT may reorder writes during reconciliation, it may be necessary to redo a write procedure in a different database state. Doing so requires re-executing the application code in order to generate the correct SQL statements. In this case, we must encapsulate the application code inside a custom-defined `ReadProc` or `WriteProc` class and pass an instance of it to TACT. The pseudo-code in Figure 3 in fact shows the simplest form of `ReadProc` and `WriteProc`, which just execute single SQL statements with no additional application code.

Other than the distributed primary key selection problem discussed above, the TACT version

Parameter	Value
Number of books in database	1000
Number of load generators	1
Transaction mix	*Shopping*
Number of loader threads	80
Startup and shutdown allowance	10 sec
Measurement interval	10 min
Inter-replica bandwidth	500 kbit/s
Inter-replica latency	100 ms

Table 1: TPC-W and network parameters in experiments.

of TPC-W does not require special conflict resolution or detection code. Moreover, it is not necessary to provide undo procedures in the application code, since TACT implements undo using the transaction log of the database system. We believe that this level of modification is acceptable and could be semi-automated, especially in the case of well-structured applications such as the TPC-W Madison implementation.

5 Results

We configure the TACT version of TPC-W on a cluster of machines (1 GHZ CPU, 256 KB cache, 256 MB RAM) running the Linux 2.4 kernel with a 100 Mbit/s FastEthernet network connection. We emulate wide-area network characteristics by routing all inter-replica IP traffic through a single FreeBSD router (1 GHZ CPU, 256 KB cache, 832 MB RAM) running the DummyNet traffic-shaping module [21]. A single load generator capable of generating roughly 1000 web interactions per second queries the replicas over switched FastEthernet with no intermediate routers. Relevant parameters for our TPC-W and network configuration are shown in Table 1.

We choose as our emulated wide-area topology a fully connected network between all replicas and the load generator. We restrict traffic between each pair of replicas to 500 kbit/s with 100 ms latency by default. No artificial restrictions are placed between the load generator and the replicas because we are interested in ascertaining the effects of consistency on replica reconciliation rather than on client/replica interaction.

Unless otherwise specified, for each network, conit, workload, and replica configuration, we execute four 10-minute runs of the load-generator. In the graphs presented below, we plot the average of the four runs for each configuration, along with error-bars for standard deviation where appropriate.

5.1 Costs and Benefits of Tunable Consistency

In the first set of experiments, we evaluate the effect of varying conit bounds on web interaction response time under a low constant load (10 interactions/s). The results are shown in Figure 4. Each point in the graph represents the average latency of web interactions over four runs of the load generator. In the first graph, we show the effect of varying numerical error bounds, while in the second we vary order error bounds.

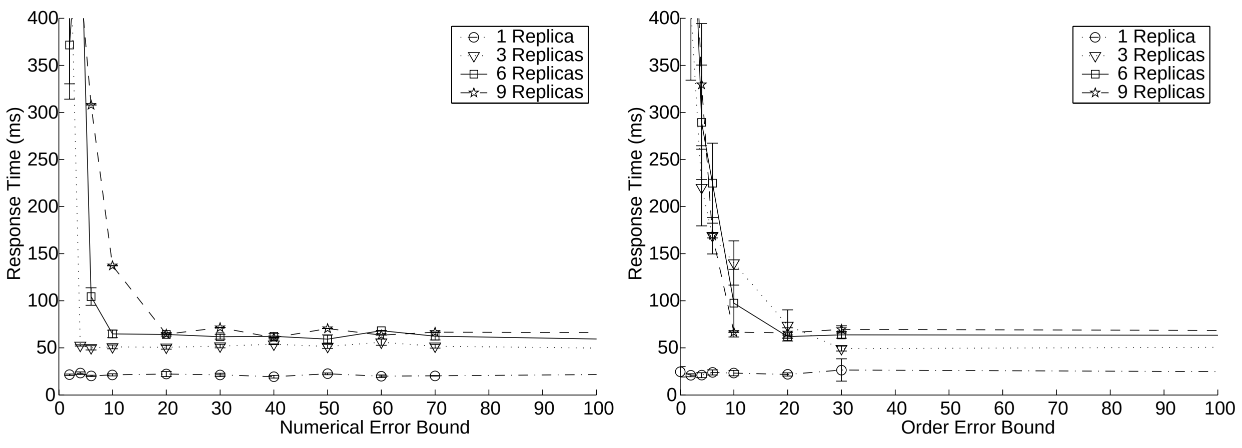

Figure 4: Average response time under varying consistency bounds and low constant load.

The graphs show that under relatively strong consistency, response time suffers due to the increased overhead of consistency management. With small relaxation of the consistency bounds (numerical error of 20 or order error of 20), response time improves dramatically (by an order of magnitude), indicating that by choosing appropriate conit bounds we can effectively limit the service latency. A breakdown of response time for each interaction type shows degraded response time for transactions that require an extra round-trip for key management, as discussed earlier in Section 4.2. This artifact accounts for the difference in response time between single- and multiple-replica trials under very weak consistency bounds. Our preliminary trials indicate that more appropriate key-management solutions would bring the low-consistency response time for the multiple-replica case down to essentially the same as that of the single-replica case. We also note that the current TACT implementation does not preemptively initiate state reconciliation with remote nodes, although it has been shown that this feature improves observed performance and availability [28]. The observed latency in this experiment thus includes the full latency of reconciliations. Thus, with additional tuning, our implementation can actually deliver lower end-to-end latency through pipelined background reconciliation, as discussed in Section 2.

Under high load, Figure 5 shows how aggregate service throughput scales with conit bounds. Again, order error and numerical error are varied and graphed separately. For numerical error, throughput degrades under tight consistency bounds proportionally to the number of replicas. As we relax consistency, the performance of replicated servers quickly surpasses the single-replica case. Under very weak consistency, throughput is approximately proportional to the number of replicas. Using order error bounds instead of numerical error, performance gain is similar but less pronounced. As consistency is relaxed, throughput levels off rather than scaling with the number of replicas. A trace of the server process reveals that the current implementation of TACT does not optimize the case of order error as well as that of numerical error. In particular, each anti-entropy pull due to order error is performed sequentially, each followed by an attempt to commit outstanding writes. For numerical error, all anti-entropy pushes are done in parallel, and only a single attempt is needed to commit writes. For the remainder of of this paper, then, we consider only the optimized case of numerical error.

Together, Figures 4 and 5 validate the concept and implementation of bounded consistency in TACT. Both response time and aggregate throughput can be tuned through appropriate choice of

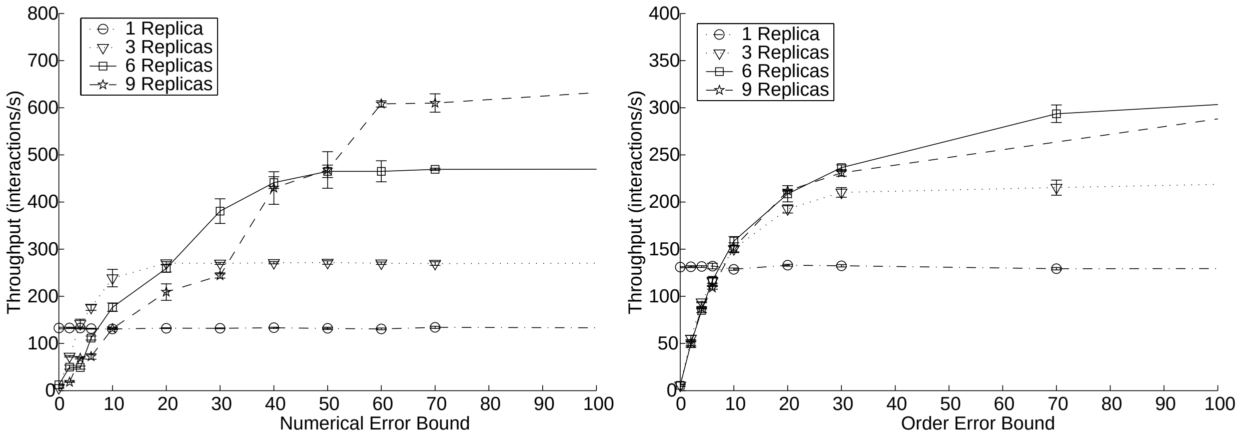

Figure 5: Maximum aggregate throughput under varying consistency bounds.

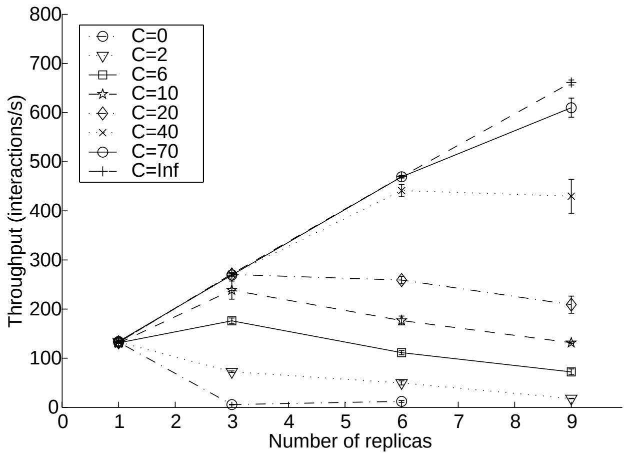

Figure 6: Maximum aggregate throughput under varying number of replicas.

consistency bounds.

5.2 Costs and Benefits of Replication

The effect of increasing the replication factor on aggregate service performance is illustrated in Figure 6. The underlying data for this figure is the same as for the tunable consistency experiments. Under very weak consistency ($C > 60$), aggregate throughput scales linearly with the number of replicas. With bounded consistency, throughput increases with the number of replicas only until the cost of replication becomes dominant. For example, with $C = 40$ the service scales to 6 replicas before the overhead of consistency management begins to limit throughput. This corresponds to upper limits on the weight of outstanding writes: at most 40 books ordered; 120 shopping cart updates; 20 administrative updates; or 40 new customer registrations. As we show in the remaining experiments, this level of relaxed consistency has a low user-visible impact. More importantly, we find that throughput does not degrade significantly when increasing replication beyond the optimal number of replicas. This result indicates that we can increase replication to improve availability without severely impacting performance.

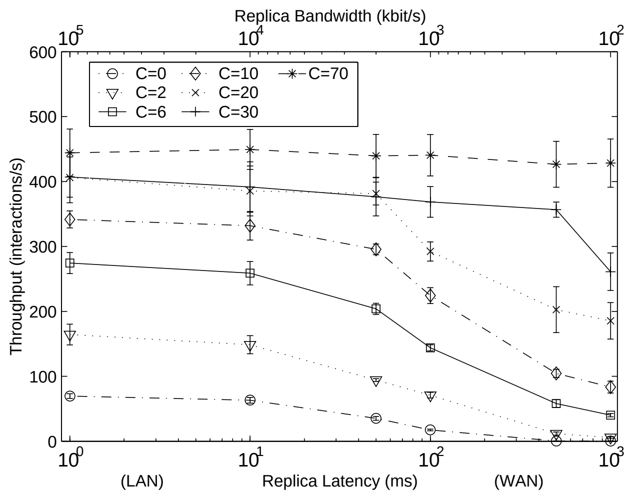

Figure 7: Maximum aggregate throughput sensitivity to inter-replica network characteristics.

5.3 Network Effects

We run identical configurations of TPC-W under varying inter-replica network characteristics to evaluate the sensitivity of the TACT relaxed-consistency protocol to network performance. Our emulated network remains a fully connected graph as before, but we vary latency and bandwidth together. Figure 7 plots the throughput of a six-replica service under different numerical error bounds. Note the two log-scale horizontal axes—the top one shows varying latency from 1 to 1000 ms and the bottom one shows varying bandwidth from 100 kbit/s to 100 Mbit/s. For a typical wide-area setting (100 ms latency, 1 Mbit/s bandwidth), the service compares favorably with the strong-consistency local-area service (1 ms latency, 100 Mbit/s bandwidth). A very modest relaxation of consistency ($C = 6$) results in a two-fold performance improvement over strong consistency in the a local area. A more aggressive relaxation ($C = 20$) results in a four-fold improvement, and shows little degradation due to network characteristics. Thus, we find that, even accounting for the sub-optimal implementation of the strong-consistency protocol in TACT, services using only slightly relaxed consistency can achieve performance in the wide-area comparable to traditional local area solutions.

5.4 Costs and Benefits of Relaxed Consistency

Relaxing consistency allows the service to achieve higher aggregate throughput and lower response times simultaneously, but at the cost of potentially returning inconsistent, inaccurate, or out of date information to users. In this section, we set out to quantify the semantic costs associated with the performance and availability gains accrued from continuous consistency. We quantify the user-visible impact of consistency by implementing auditing in the load-generator, and through off-line analysis of server logs. We track four metrics: *best-seller discrepancies*, *propagation delay*, *price discrepancies*, and *oversold books.* For simplicity, we consider only the case of six replicas.

First, we would like to quantify the difference between observed best-seller lists returned by each node, and the *ideal* best-seller list, assuming a centralized service. We modify the load generator to periodically query each node for one of the per-topic best-seller lists. Using a trace of the load-generator queries and the initial database state, we can construct the ideal list (that which

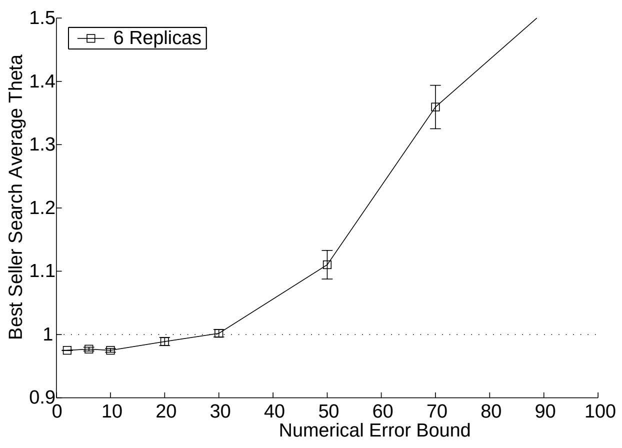

Figure 8: Inconsistency in best-seller lists.

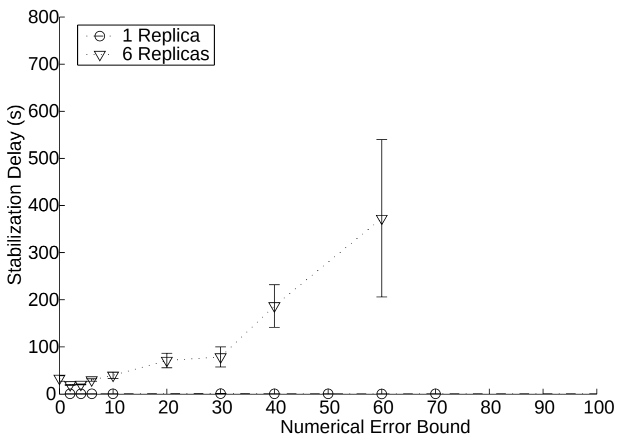

Figure 9: Average write stabilization delay.

would be returned under a strongly consistent system). We compare the observed and ideal lists using the idea of *θ-approximation* from [8]. Let *b.orders* denote the number of copies sold for book b. We define θ, the degree of inconsistency between an observed list L and the ideal list, as $\max_{b \in L, b' \notin L}(b'.orders/b.orders)$. In the ideal case, $\theta \leq 1$, meaning that any book reported as best-selling has sold no less than than any book not reported. Under relaxed consistency, a book b reported as best-selling at a replica may in fact have sold less than some other book b' not reported as best-selling. In this case, $\theta > 1$ bounds the ratio $b'.orders/b.orders$. For example, $\theta = 1.25$ means that any book reported as best-selling must have sold no less than 80% of any book not reported as best-selling.

A plot of θ observed under varying conit bounds is shown in Figure 8. We trace the server for 60 minutes in order to allow steady-state measurement of long-term ordering patterns. For a single replica, we find $\theta \approx 0.98$, meaning that an actual best-selling book sells at least 2% more than a non-best-seller in the TPC-W benchmark. For the replicated service, we find that θ remains less than 1 for moderate consistency bounds, and $\theta < 1.5$ for all but very weak consistency. This result is not surprising since, even in complete isolation, replicas starting from the same initial state should receive similar order patterns, and hence have somewhat similar best-seller lists.

A second metric likely to be important for administrators of the replicated service is the propagation delay of administrative updates. We measure the delay between the introduction of each *price-change* update and the stabilization of the update on all replicas. The results of this test are shown in Figure 9. For moderately relaxed consistency levels ($C \leq 30$), stabilization occurs within two minutes. Beyond moderately relaxed consistency levels, propagation time increases quickly, because reconciliations occur less frequently, while the overhead of each reconciliation (including exchanging, sorting, and committing many outstanding writes) increases. Although not discussed in this paper, the *staleness* conit bound is designed to explicitly bound the propagation delay measured here.

Our third metric measures observed discrepancies in reported prices versus the ideal case of a centralized service. All occurrences of book prices on the web pages returned to the load generator are logged. We can then calculate the percentage of price observations that are incorrectly reported due to relaxed consistency. These results are presented in Figure 10. In the TPC-W benchmark, new book prices for each administrative update are chosen randomly from a uniform distribution.

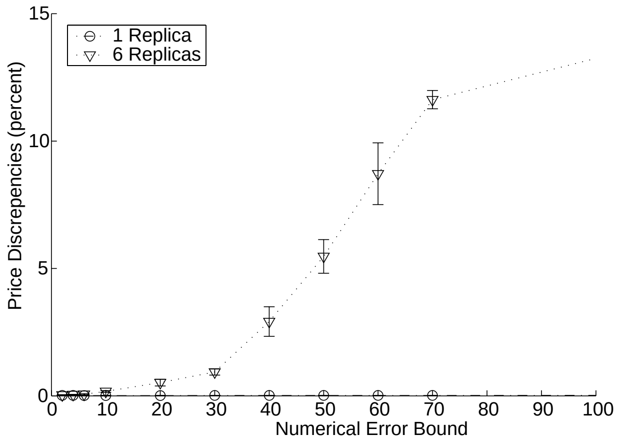

Figure 10: Percentage of out-of-date prices.

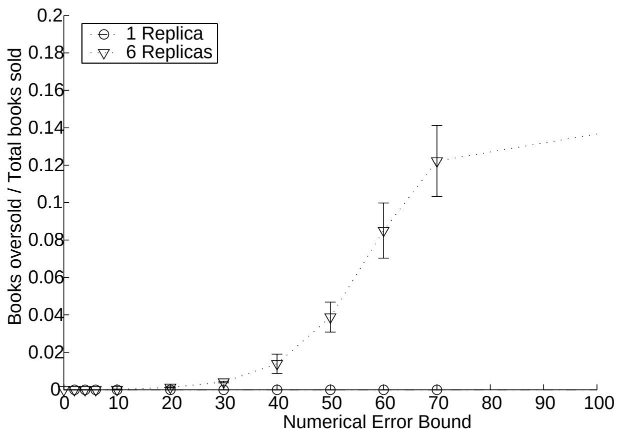

Figure 11: Fraction of oversold books to total book sales under varying consistency bounds.

Therefore, it is not meaningful to measure the difference in observed and ideal prices, but only the number of such differences. In more realistic settings, price updates are likely to follow a more meaningful pattern, making the measurement of price discrepancies simpler. According to Figure 10, incorrect prices are observed less than 3% of time for consistency bounds $C \leq 40$.

The last metric measures the impact that bounded inconsistency has on book sales. By analyzing trace files from the load generator and replicas, we can determine the number of books orders confirmed by replicas, but which would have been rejected by a non-replicated service due to low inventory. Figure 11 shows the number of books oversold as a fraction of total sales. As for the price discrepancy metric, it would not be meaningful to measure the total cost of these sales, since all prices are chosen at random. The cost of oversold book orders, from a business perspective, is not high so long as the number of inconvenienced customers remains low (at worst, some orders would have to be delayed until re-stocking, or some users notified of the problem). We find, however, that the fraction of orders oversold is low: In the case of six replicas, less than 3% of orders are oversold for consistency bounds $C \leq 40$. A more advanced feature of TACT is *relative* numerical error, which can be used to progressively tighten consistency as inventory drops for a particular book. Another possible approach is for applications to dynamically change the consistency requirements when inventory is low. However, our current implementation of TPC-W does not yet exploit these two features.

Combining the results shown in Figures 8–11 and those in Figure 5, we see that, for modestly relaxed consistency of $C = 30$, a 6-replica service gains a three-fold improvement in throughput over the single-replica configuration, while (i) the best-seller lists remain practically accurate (with θ very close to 1); (ii) the propagation delay of administrative updates is under two minutes; (iii) about 1% of all price observations are incorrect; and (iv) less than 0.5% of all orders are oversold.

Finally, we note that it is possible for an application to enforce upper bounds on these discrepancy metrics with appropriate conits. For example, best-seller discrepancies can be bounded by defining a special conit c_θ to track θ. For simplicity, assume that each transaction can order at most m copies of the same book, and that a best-selling book must have sold at least M copies. Typically, $M \gg m$. It is not difficult to see that each transaction can increase the value of θ by at most m/M. Thus, we can define the weight of a book-order transaction on c_θ to be m/M, and the numerical error bound on c_θ to be $\theta_{\max}-1$, where $\theta_{\max}$ specifies the maximum amount of best-seller discrepancies allowed. Propagation delay of administrative updates can be bounded by setting the staleness bound on the *admin* conit. Price discrepancies and oversold books can be bounded by setting the order error bounds on appropriate conits. In our implementation of TPC-W, we choose not to bound these discrepancy metrics directly, because the goal of the experiments in this section is to measure the observed discrepancies produced by a straightforward implementation that uses only the simple conit definitions in Section 4.1.

6 Related Work

Commercial databases offer SQL isolation levels as a means to improve performance at the cost of reduced consistency. Isolation levels have been problematic both in definition and in use by programmers [4]. There have been some recent efforts at clarifying these definitions and making them more rigorous [2, 1]. At each reduced isolation level, certain consistency requirements are eliminated, introducing a corresponding set of undesirable phenomena (e.g. *dirty reads*). Few tools exist to help applications cope with such effects, however. Further, isolation levels are not helpful in designing replicated systems, but only for applications that can cope without strong transactional consistency. In effect, the use of SQL isolation levels is somewhat orthogonal to the use of continuous consistency [28]. TACT provides isolation for local transactions within the framework of relaxed-consistency replication. This approach might be classified as *Lazy Update Everywhere* [26] but with multiple-operation transactions.

Most commercial databases also support some form of replication, following either an *eager* or a *lazy* approach [10]. The eager approach guarantees strong consistency by updating all replicas as part of a single distributed transaction. However, this approach is expensive in the wide area and unlikely to scale beyond a small number of replicas [10]. In practice, replicated database systems often take the lazy approach, where writes are propagated to other replicas asynchronously. To handle conflicting updates, commercial replication products use conflict resolution procedures, which are typically static, inflexible, and rule-based. Tools for managing data consistency in a flexible, application-defined manner are still lacking. In attempting to port TPC-W to two commercial database systems with replication support, we have found it nearly impossible to configure adequate conflict resolution procedures. Thus, we believe that these commercial replication products are only appropriate for applications that have been written from the ground up with the particular replication strategies in mind.

Some authors argue for complete transparency and automatic replication [20]. Under ideal circumstances, administrators and programmers can move from a single node server to a replicated service with no need to re-program the application. In the wide area, we contend that relaxed consistency is invaluable. We agree with the argument made previously [23] that, in order to fully benefit from relaxed consistency, applications need to be aware and in control of replication. Further, we have found that certain programming practices, such as those discussed in Section 4.2, would preclude any but the most thorough attempts to replicate. One example is the occasional use of application-level locks. No amount of intelligence in the database layer could have automatically detected these locks, though they are essential for maintaining data consistency. We are forced to re-implement all such locks using database-level transactions.

Our work is related to the research on relaxed notions of transaction commit, including *sagas* [9], *relaxed atomicity* [15], *optimistic commit* [14], etc. In the area of managing consistency of distributed numerical data, related work includes *data-value partitioning* [22] and *TRAPP* [17]. There is also a large body of work on general models of continuous and relaxed consistency. A thorough discussion can be found in earlier papers on TACT [29, 28, 30]. Here, we highlight the differences between our work and several others. *Quasi-copies* [3] use four conditions to control coherency, including an arithmetic condition which is similar to our numerical error. Under a quasi-copy scheme, only the master database may accept updates; on the other hand, TACT also supports updates at replicas. *Epsilon-serializability* [27] is another proposal for relaxing consistency. Compared with epsilon-serializability, TACT allows applications to express a broader range of semantics using flexible conit definitions. Another difference is that while epsilon-serializability focuses on increasing concurrency at a single site, TACT focuses on trading consistency for reduced communication among wide-area replicas.

7 Conclusions and Future Work

We have shown that continuous consistency enables services to take advantage of wide-area replication for either availability or performance. The tunable consistency bounds provided by the TACT middleware allow flexible control of both the user-observed latency and global aggregate throughput by trading consistency for performance. Furthermore, we find that significant performance benefits come with relatively small relaxations in consistency, and hence acceptably low levels of client-perceived inconsistencies. Further study is ongoing to characterize and maximize the availability of services using the TACT replication model.

We use a wide-area network emulator to study the sensitivity of the system to network characteristics. We find that the system is able to tolerate network characteristics representative of those found in wide-area networks, and that the system is not overly sensitive to inter-replica bandwidth or latency. However, it may be helpful to study the system under more realistic and varied topologies, taking into account also client-replica network paths. We plan to extend our approach of using the DummyNet traffic-shaping tool to evaluate TPC-W with TACT on more realistic topologies.

Through our experience of porting TPC-W to TACT, we have also demonstrated the feasibility of using TACT as a database middleware for traditional, SQL-based database applications. We find that the increased application complexity is manageable, and that the porting process is relatively straightforward and can be automated in many cases.

In addition to the future work mentioned above, we are investigating the possibility of applying the idea of continuous consistency to the problem of maintaining materialized views [11] in the context of view replication, semantic query caching, and data warehousing. In many cases, it is infeasible to update materialized views after every transaction. A solution to this problem is to batch updates and refresh the materialized views periodically. Traditionally, the application controls only the refresh interval. With TACT, the application can define more flexible refresh policies that can bound not only the staleness of the views, but also high-level, semantic measures of discrepancies between the views and the base data (such as the θ bound discussed in Section 5.4).

Another possible optimization orthogonal to replication is to partition the database into disjoint data sets, a technique that is well studied in distributed databases [18]. Currently, TACT does not efficiently handle the mix of partitioned and replicated data. A TACT node performing anti-entropy updates sends all transactions in the order they were introduced at the node, regardless of whether they only affect partitioned data in the local node. It may be possible for TACT to derive dependency relationships between transactions and data automatically, although it is likely to introduce complexity into the TACT transaction model. In addition, it may be desirable for TACT to prioritize updates. For example, high-priority administrative updates can be propagated to all replicas without sending lower-priority user updates.

References

[1] A. Adya, B. Liskov, and P. O'Neil. A new basis for an SQL isolation level standard. In *Proc. of the 1999 High Performance Transaction Systems Workshop*, 1999.

[2] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proc. of the 2000 Intl. Conf. on Data Engineering*, pages 67–78, Mar. 2000.

[3] R. Alonso, D. Barbará, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Trans. on Database Systems*, 15(3):359–384, 1990.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. of the 1995 ACM SIGMOD Intl. Conf. on Management of Data*, pages 1–10, June 1995.

[5] P. Bernstein and N. Goodman. The failure and recovery problem for replicated distributed databases. *ACM Trans. on Database Systems*, Dec. 1984.

[6] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti. An architectural evaluation of Java TPC-W. In *Proc. of the 2001 Intl. Symp. on High-Performance Computer Architecture*, Jan. 2001.

[7] W. K. Edwards, E. Mynatt, K. Petersen, M. Spreitzer, D. Terry, and M. Theimer. Designing and implementing asynchronous collaborative applications with bayou. In *Proc. of the 1997 ACM Symp. on User Interface Software and Technology*, Oct. 1997.

[8] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of the 2001 ACM Symp. on Principles of Database Systems*, pages 102–113, May 2001.

[9] H. Garcia-Molina and K. Salem. Sagas. In *Proc. of the 1987 ACM SIGMOD Intl. Conf. on Management of Data*, pages 249–259, May 1987.

[10] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the 1996 ACM SIGMOD Intl. Conf. on Management of Data*, pages 173–182, June 1996.

[11] A. Gupta and I. S. Mumick, editors. *Materialized Views: Techniques, Implementations and Applications.* MIT Press, June 1999.

[12] R. Guy, J. Heidemann, W. Mak, T. P. Jr., G. Popek, , and D. Rothmeier. Implementation of the Ficus replicated file system. In *Proc. of the 1990 Summer USENIX Conf.*, June 1990.

[13] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Trans. on Computer Systems*, 10(1):3–25, Feb. 1992.

[14] E. Levy, H. F. Korth, and A. Silberschatz. An optimistic commit protocol for distributed transaction management. In *Proc. of the 1991 ACM SIGMOD Intl. Conf. on Management of Data*, pages 88–97, 1991.

[15] E. Levy, H. F. Korth, and A. Silberschatz. A theory of relaxed atomicity. In *Proc. of the 1991 ACM Symp. on Principles of Distributed Computing*, pages 95–109, 1991.

[16] A. Myers, P. A. Dinda, and H. Zhang. Performance characteristics of mirror servers on the internet. In *Proc. of the 1999 IEEE Intl. Conf. on Computer Communications*, pages 304–312, 1999.

[17] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, Madison, Wisconsin, June 2002.

[18] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, second edition, 1999.

[19] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proc. of the 1998 Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[20] E. Riedel, S. Spence, and A. Veitch. When local becomes global: An application study of data consistency in a networked world. In *Proc. of the 2001 IEEE Intl. Performance, Computing, and Communications Conf.*, Apr. 2001.

[21] L. Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1), Jan. 1997.

[22] N. Soparkar and A. Silberschatz. Data-value partitioning and virtual messages. In *Proc. of the 1990 ACM Symp. on Principles of Database Systems*, pages 357–367, 1990.

[23] D. B. Terry, K. Petersen, M. J. Spreitzer, , and M. M. Theimer. The case for non-transparent replication: Examples from bayou. *IEEE Data Engineering Bulletin*, 21(4):12–20, Dec. 1998.

[24] 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 *Proc. of the 1995 ACM Symp. on Operating Systems Principles*, pages 172–183, Dec. 1995.

[25] Transaction Processing Performance Council. TPC-W benchmark specification. `http://www.tpc.org/wspec.html`.

[26] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proc. of the 2000 Intl. Conf. on Distributed Computing Systems*, pages 264–274, Apr. 2000.

[27] K.-L. Wu, P. S. Yu, and C. Pu. Divergence control for epsilon-serializability. In *Proc. of the 1992 Intl. Conf. on Data Engineering*, pages 506–515, Feb. 1992.

[28] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proc. of the 2000 USENIX Symp. on Operating Systems Design and Implementation*, Oct. 2000.

[29] H. Yu and A. Vahdat. Efficient numerical error bounding for replicated network services. In *Proc. of the 2000 Intl. Conf. on Very Large Data Bases*, Sept. 2000.

[30] H. Yu and A. Vahdat. Combining generality and practicality in a conit-based continuous consistency model for wide-area replication. In *Proc. of the 2001 Intl. Conf. on Distributed Computing Systems*, Apr. 2001.