Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services

HAIFENG YU and AMIN VAHDAT
Duke University

The tradeoffs between consistency, performance, and availability are well understood. Traditionally, however, designers of replicated systems have been forced to choose from either strong consistency guarantees or none at all. This paper explores the semantic space between traditional strong and optimistic consistency models for replicated services. We argue that an important class of applications can tolerate relaxed consistency, but benefit from bounding the maximum rate of inconsistent access in an application-specific manner. Thus, we develop a conit-based continuous consistency model to capture the consistency spectrum using three application-independent metrics, *numerical error*, *order error*, and *staleness*. We then present the design and implementation of TACT, a middleware layer that enforces arbitrary consistency bounds among replicas using these metrics. We argue that the TACT consistency model can simultaneously achieve the often conflicting goals of generality and practicality by describing how a broad range of applications can express their consistency semantics using TACT and by demonstrating that application-independent algorithms can efficiently enforce target consistency levels. Finally, we show that three replicated applications running across the Internet demonstrate significant semantic and performance benefits from using our framework.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*; H.2.4 [**Database Management**]: Systems—*Distributed databases*

General Terms: Design, Experimentation

Additional Key Words and Phrases: Conit, consistency model, continuous consistency, network services, relaxed consistency, replication

1. INTRODUCTION

Replicating distributed services for increased availability and performance has been a topic of considerable interest for many years. Recently however, the exponential increase in access to popular Web services provides concrete examples of the types of services that benefit from replication, their requirements, and semantics. One of the primary challenges to replicating network services across

The University of Utah Network Testbed used in conjunction with this research was supported by NSF grant ANI-00-82493, DARPA/AFRL grant F30602-99-1-0503, and Cisco.
Author's addresses: Dept. of Computer Science, Levine Science Research Center (D308), Box 90129, Duke University, Durham, NC 27708-0129; email: {yhf,vahdat}@cs.duke.edu.

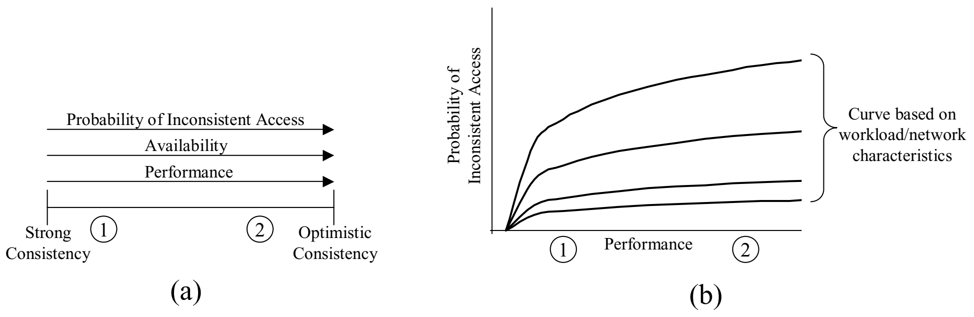
Probability of Inconsistent Access
Availability
Performance
Strong Consistency
1
2
Optimistic Consistency
(a)
Probability of Inconsistent Access
Curve based on workload/network characteristics
Performance
1
2
(b)

Fig. 1. (a) The spectrum between strong and optimistic consistency as measured by a bound on the probability of inconsistent access. (b) The tradeoff between consistency, availability, and performance depends upon application and network characteristics.

the Internet is maintaining consistency among the replicas. Providing strong consistency (e.g., one-copy serializability [Bernstein and Goodman 1984]) imposes performance overhead and limits system availability. Thus, a variety of optimistic consistency models [Guy et al. 1990; Guy et al. 1998; Kistler and Satyanarayanan 1992; Saito et al. 1999; Terry et al. 1995] have been proposed for applications that can tolerate relaxed consistency. Such models require less communication, resulting in improved performance and availability.

Unfortunately, optimistic models typically provide no bounds on the inconsistency of the data exported to client applications and to end users. A fundamental observation behind this work is that there is a continuum between strong and optimistic consistency and that this continuum is semantically meaningful for a broad range of network services. This continuum is parameterized by the maximum distance between a replica's local data image and some final image "consistent" across all replicas after all writes have been applied everywhere. For strong consistency, this maximum distance is zero, while for optimistic consistency it is infinite. We explore the semantic space between these two extremes. For a given workload, providing a per-replica consistency bound allows the system to determine an expected probability, for example, that a write operation will conflict with a concurrent write submitted to a remote replica, or that a read operation observes the results of writes that must later be rolled back. No such analysis can be performed for optimistic consistency systems because the maximum level of inconsistency is unbounded.

The relationship between consistency, availability, and performance is depicted in Figure 1(a). In moving from strong consistency to optimistic consistency, application performance and availability increases. This benefit comes at the expense of an increasing probability that individual accesses will return inconsistent results, e.g., stale/dirty reads, or conflicting writes. In our work, we allow applications to bound the maximum probability/degree of inconsistent access in exchange for increased performance and availability. Figure 1(b) graphs different potential improvements in application performance versus the probability of inconsistent access, depending on workload/network characteristics. Moving to the right in the figure corresponds to increased performance, while moving up in the figure corresponds to increased inconsistency. To achieve increased performance, applications must tolerate a corresponding increase in inconsistent accesses. The tradeoff between performance and consistency depends upon a number of factors, including application workload, such as read/write ratios, probability of simultaneous writes, etc., and network characteristics such as latency, bandwidth, and error rates. At the point labeled “1” in the consistency spectrum in Figure 1(b), a modest increase in performance corresponds to a relatively large increase in inconsistency for application classes corresponding to the top curve, perhaps making the tradeoff unattractive for these applications. Conversely, at point “2,” large performance increases are available in exchange for a relatively small increase in inconsistency for applications represented by the bottom curve.

Thus, the goals of this work are to: (i) explore the issues associated with filling the semantic, performance, and availability gaps between optimistic and strong consistency models; (ii) develop a continuous consistency model that allows a broad range of replicated services to conveniently and quantitatively express their consistency requirements; (iii) ensure that the model is simultaneously *general*—applicable to a broad range of applications—and *practical*—able to efficiently enforce target consistency levels; (iv) quantify the tradeoff between performance and consistency for a number of sample applications; and (v) show the benefits of dynamically adapting consistency bounds in response to current network, replica, and client-request characteristics. To this end, we present the design, implementation, and evaluation of the TACT toolkit for wide-area replication. TACT is a middleware layer that accepts specifications of application consistency requirements and mediates read/write access to an underlying data store. It also enforces the consistency requirements specified by the application using built-in *consistency protocols* to shield application programmers from the complexities of consistency maintenance. At a high level, if an operation does not violate prespecified consistency requirements, TACT allows it to proceed locally (without contacting remote replicas). Otherwise, the operation blocks until TACT is able to synchronize with one or more remote replicas (i.e., push or pull some subset of local/remote updates) as determined by system consistency requirements.

To allow applications to export application-dependent consistency semantics, we base our consistency model on the concept of a *conit* (for *con*sistency un*it*). Conceptually, a conit is a logical consistency unit. Consistency is then defined on conits rather than physical data items. We propose three simple, application-independent metrics, *numerical error*, *order error*, and *staleness* to quantify conit consistency. Numerical error limits the total weight of writes on a conit that can be applied globally across all replicas before being propagated to a given local replica. Order error limits the number of tentative writes on a conit (subject to reordering) that can be outstanding at any one replica, and staleness places a real-time bound on the delay of write propagation among replicas. Consider the concrete example of a replicated bulletin board service where users may post/retrieve news messages to/from any replica. In the simplest case where one conit is defined to cover all news messages, numerical error bounds the total number of messages posted system-wide but not seen locally.

Order error then limits the number of out of order messages on a given replica, while staleness limits the delay of messages.

We present a number of algorithms to efficiently bound each of the three metrics: a push approach based solely on local information bounds numerical error; a write commitment algorithm allows replicas to agree on a global total write order and enforces the order error bound; and staleness is maintained using a real-time vector. At the lowest level, these protocols systematically control when and where to initiate communication between replicas based on the application-defined conits and consistency bounds. To evaluate the effectiveness of our system, we implement and deploy across the wide area three applications with a broad range of dynamically changing consistency requirements using the TACT toolkit: an airline reservation system, a distributed bulletin board service, and replicated load distribution front ends to a Web server. Relative to strong consistency techniques, TACT improves the throughput of these applications by up to a factor of 10. Relative to weak consistency approaches, TACT provides strong semantic guarantees regarding the maximum inconsistency observed by individual read and write operations.

The rest of this paper is organized as follows. Section 2 describes three network services implemented in the TACT framework to motivate our consistency model and system architecture. Section 3 presents the system model and assumptions we make. Section 4 elaborates the conit-based continuous consistency model, and Section 5 further discusses its generality. In Section 6, we summarizes our insights on the consistency model. Next, Section 7 gives an overview of the application-independent consistency protocols we implement, and Section 8 details the TACT architecture. Section 9 evaluates the performance of our three applications in the TACT framework. Finally, Section 10 places our work in the context of related work, and Section 11 presents our conclusions.

2. APPLICATIONS

In this section, we describe three applications implemented in our framework. Our system architecture and performance evaluation of these applications are detailed in Sections 8 and 9, respectively. We present the requirements of these applications here to motivate both the need for a continuous consistency model and our particular solution.

2.1 Airline Reservations

Our first sample application is a simple replicated airline reservation system designed to be representative of replicated e-commerce services that accept inquiries (searches) and purchase orders on a catalog. In our implementation, each server maintains a full replica of the flight information database and accepts user reservations and inquiries about seat availability. Consistency in this application is measured by the percentage of requests that access inconsistent results. For example, in the face of divergent replica images, a user may observe an available seat, when in fact the seat has been booked at another replica (false positive). Or a user may see a particular seat is booked when in fact, it is available (false negative). Intuitively, the probability of such events is proportional to the distance between the local replica image and some consistent final image.

One interesting aspect of this application is that its consistency requirements change dynamically based on client, network, and application characteristics. For instance, the system may wish to minimize the rate of inquiries/updates that observe inconsistent intermediate states for certain preferred clients. Requests from such clients may require a replica to update its consistency level (by synchronizing with other replicas) before processing the request or may be directed to a replica that maintains the requisite consistency by default. As another example, if network performance among replicas is high, the absolute performance/availability savings available from relaxing consistency may not be sufficient to outweigh the costs associated with reduced consistency. Finally, the desired consistency level depends on individual application semantics. For airline reservations, the cost of a transaction that must be rolled back is fairly small when a flight is empty (one can likely find an alternative seat on the same flight), but grows as the flight fills.

2.2 Bulletin Board

The bulletin board application is a replicated message posting service modeled after more sophisticated services such as USENET. Messages are posted to individual replicas. Sets of updates are propagated among replicas, ensuring that all messages are eventually distributed to all replicas. This application is intended to be representative of interactive applications that often allow concurrent read/write access under the assumption that conflicts are rare or can be resolved automatically.

Desirable consistency requirements for the bulletin board example include maintaining causal and/or total order among messages posted at different replicas. With causal order, a reply to a message will never appear before the original message at any replica. Total order ensures that all messages appear in the same order at all replicas, allowing the service to assign globally unique identifiers to each message. Another interesting consistency requirement for interactive applications, including the bulletin board, is to guarantee that at any time t, no more than k messages posted before t are missing from the local replica.

2.3 QoS Admission Control

The final application implemented in our framework is an admission control mechanism that provides Quality of Service (QoS) guarantees to a set of preferred clients. In this scenario, front-ends (as in LARD [Pai et al. 1998]) accept requests on behalf of two classes of clients, standard and preferred. The front ends forward requests to back end servers with the goal of reserving some predetermined portion of server capacity for preferred clients. Thus, front ends allow a maximum number of outstanding requests (assuming homogeneous requests) at the back end servers. To determine the maximum number of "standard" requests that should be forwarded, each front end must communicate current access patterns to all other front ends.

One goal of designing such a system is to minimize the communication required to accurately distribute such load information among front ends. This QoS application is intended to be representative of services that independently track the same logical data value at multiple sites, such as a distributed sensor array, a load balancing system, or an aggregation query [Hellerstein et al. 1997]. Such services are often able to tolerate some bounded inaccuracy in the underlying values they track (e.g., average temperature or server load) in exchange for reduced communication overhead or power consumption (in the case of a mobile node).

3. SYSTEM MODEL

For simplicity, we refer to application data as a database, though the data can actually be stored in a database, file system, persistent object, etc. The database is replicated in full at multiple sites. Each replica accepts logical *reads* and *writes* from users that may consist of multiple primitive read/write operations. Reads and writes are both called *accesses*. Our reads and writes are query transactions and update transactions in database terminology except that they may or may not be atomic or persistent when replicas fail. Besides primitive reads/writes, a read or write also contains the associated application logic (e.g., conditional branch and computation) that controls the execution of the primitive reads/writes.[1] On a single replica, a read or write is isolated from other reads or writes during execution.

Replicas maintain consistency by propagating writes rather than the data written, as in Bayou [Petersen et al. 1997] and N-ignorant systems [Krishnakumar and Bernstein 1994]. Writes are reexecuted at each replica against the local database image. Because the system reexecutes writes rather than directly applies new data, it does not need to resolve write conflicts. The application logic in the write must have already handled conflicts, otherwise race conditions may occur even if the write is processed by a centralized database server. Compared to directly propagating new data images, this model allows the application to specify alternative actions in case of conflicts, rather than dictating that all conflicting writes be aborted. Reexecuting writes at each replica does increase system load, but previous research on write procedures [Petersen et al. 1997] has shown this approach can be implemented efficiently.

TACT mediates all accesses to the data store. Each replica maintains a write log, containing all writes applied to its database image. Standard concurrency control mechanisms (e.g., two-phase locking) are used on each replica to ensure local serializability. The replica that first accepts an access from a client is called the *originating replica* for that access. All other replicas are *remote replicas*. When first applied to a replica, a write is in a *tentative* state and returns an *observed result* to the user. However, this observed result may not be accurate because of an inaccurate local database image under weak consistency.

[1]Even though the original transaction concept in database does contain application logic, the transaction definitions in previous consistency models are often simplified to only containing primitive data accesses. Here, we include application logic in the definition so that we can apply the same write against different database images (at different replicas) under weak consistency.

The write can then be propagated to other replicas. Writes in a replica's write log may be reordered, for example, rolled-back and then re-applied in a different order, with potentially different results. Write reordering is assumed to be isolated from the execution of reads and writes. At some point, a write becomes *committed*, which means it will never be reordered again. The *ideal result* of a write is thus defined to be its return value when finally committed. Reads are processed once and are never reordered. The *observed result* of a read is the value returned to a client query, while its *ideal result* is the value that should be returned to the user if 1SR with external order (defined below) were maintained.

The traditional definition of strong consistency for replicated data is one-copy serializability (1SR) [Bernstein et al. 1987]. However, the lack of timing information in 1SR makes it inappropriate for Internet applications. For example, in replicated stock quotes systems, stale values are allowed to be read even if 1SR is maintained, as such stale reads can be considered to execute "in the past" by 1SR. As in timed consistency [Singla et al. 1997; Torres-Rojas et al. 1999] and external consistency [Adya et al. 1995], we augment 1SR with *external order*, which is a partial order over all accesses. An access A_1 *externally precedes* another access A_2 if A_1 returns its observed result to the user (in strict wall-clock time) before A_2 is submitted to its originating replica. We say an execution on replicated data is *1SR with external order* (1SR+EXT) if the execution is equivalent to a serial execution that is compatible to external order. The concept of 1SR+EXT is not completely new and is equivalent to the *linearizability* concept for concurrent objects [Herlihy and Wing 1990]. Hereafter, we equate "strong consistency" with 1SR+EXT, and it will be the strongest consistency level we consider.

4. CONIT-BASED CONTINUOUS CONSISTENCY MODEL

In earlier sections, we motivated the need for a continuous consistency and described our target system model. In this section, we describe our particular continuous consistency model. We begin with a discussion of two key and typically conflicting requirements of any consistency model, *generality* and *practicality*. We then formally describe the definition of conit consistency using three application-independent metrics. Finally, we show how applications use the consistency model and how the model captures application-specific consistency requirements with relatively little effort from the application developer.

4.1 The Dual Requirements of Generality and Practicality

Broad diversity in the characteristics of wide-area applications imposes the following two typically conflicting requirements, *generality* and *practicality*, on the continuous consistency model:

Generality. Wide-area services have rich and application-specific consistency semantics. For example, a shared editor may have well-defined, but totally different consistency semantics from an inventory maintenance system for e-commerce. Thus, the consistency model must be sufficiently general and abstract to capture a wide range of consistency semantics.

Practicality. The wide applicability of Internet data replication requires the model to be practical to use in regular application design. More specifically, by practicality, we mean (i) in spite of application-specific semantics, the protocols enforcing consistency requirements should be application-independent and highly-efficient, and (ii) the way that consistency semantics are expressed must be natural and easy to use.

The goals of generality and practicality typically conflict with one another. One effective approach for achieving generality is to avoid defining a uniform consistency model for all applications. Instead, applications are allowed to specify their own consistency semantics. However, the consistency protocols enforcing such a model typically cannot be optimized in an application-independent manner. Also, to capture arbitrary semantics, the model has to be abstract, providing no natural way for application programmers to use the model in many cases.

In the context of traditional replicated databases, a large body of research [Agrawal et al. 1993; Akibsi et al. 1990; Badrinath and Ramamritham 1992; Dipippo and Wolfe 1993; Drew and Pu 1995; Gallersdorfer and Nicola 1995; Krishnakumar and Bernstein 1994; Kuo and Mok 1992, 1993; Pitoura nad Bhargava 1995; Pu et al. 1993; Pu and Leff 1991; Weihl 1988; Wong and Agarwal 1992; Wu et al. 1992] has focused on relaxed consistency models. However, such traditional models typically achieve only one of generality and practicality. Some of the consistency models [Agrawal et al. 1993; Kuo and Mok 1992, 1993; Wong and Agarwal 1992] are general enough to allow a wide range of applications to express their consistency semantics. However, they provide no practical, efficient, application-independent protocols to enforce the model and no natural application programmer interface (API) for application programmers, thus failing to meet the practicality requirement. Other relaxed consistency models [Alonso et al. 1990; Badrinath and Ramamritham 1992; DiPippo and Wolfe 1993; Drew and Pu 1995; Gallersdorfer and Nicola 1995; Krishnakumar and Bernstein 1994; Pitoura and Bhargava 1995; Pu et al. 1993; Pu and Leff 1991; Weihl 1988; Wu et al. 1992] have easy-to-use interfaces and can be efficiently implemented, but they typically only address the consistency requirements of a specific class of applications.

To simultaneously achieve generality and practicality, we propose a conit-based continuous consistency model for wide-area data replication (Figure 2). Generality is achieved by our conit theory. Practicality is achieved in our model by (i) using a simple set of metrics for conit consistency and (ii) expressing application semantics through per-write weights.

4.2 Conit Theory, Application Semantics, and Conit Consistency

Applications observe consistency from the results of reads and writes. With strong consistency, the observed result always equals its ideal result. As we relax consistency, the observed result and the ideal result begin to diverge. The meaning of this difference to end users depends on application semantics. Thus, in order to quantify consistency and capture the semantic discrepancy between observed and ideal results, we believe that a predefined uniform consistency

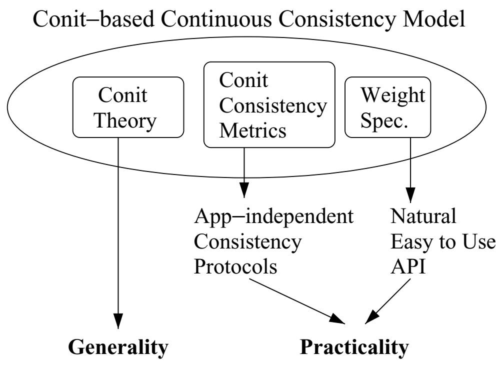
Conit–based Continuous Consistency Model
Conit Theory
Conit Consistency Metrics
Weight Spec.
App–independent Consistency Protocols
Natural Easy to Use API
Generality
Practicality

Fig. 2. How the conit-based consistency model achieves two typically conflicting goals.

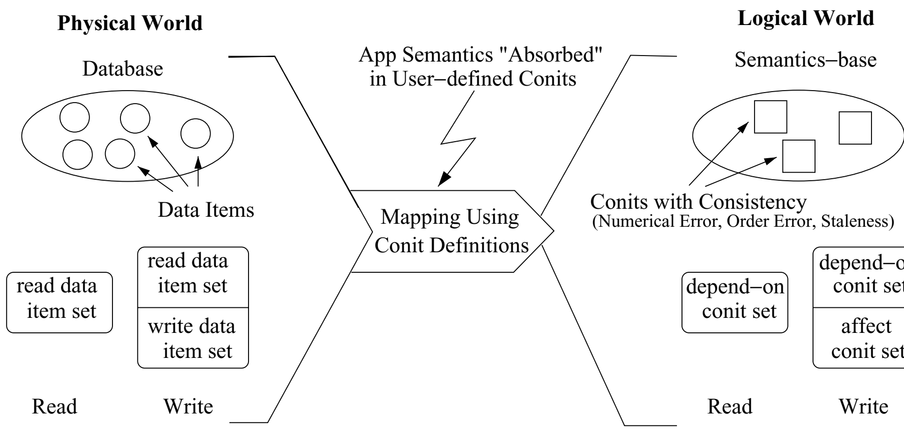

Fig. 3. Role of the conit theory.

model is inappropriate. Instead, the consistency model should allow the application to export its specific consistency requirements so that the model can address the consistency semantics that the application is sensitive to.

The approach we adopt in our model is to allow applications to define each consistency requirement as a *conit*. Conceptually, a conit is a logical consistency unit. For example, in a replicated bulletin board, sample consistency requirements include (i) the difference between observed/ideal number of messages, (ii) the number of out-of-order messages in the current view, and (iii) the consistency of messages posted by friends. These requirements can all serve as conit definitions. Using these conit definitions, our conit theory maps the physical world, composed of the physical database together with the reads and writes operating on physical data items, to a logical world (Figure 3). The logical world contains a *semantics-base*, consisting of application-specific consistency semantics (conits), and reads/writes conceptually operating on the semantics. Here a read/write *depends on* the conits with which it is concerned, and conits are *affected* by writes. The semantic difference between the observed and ideal return value of an access is then solely determined by the depend-on conit set. For example, suppose we define a conit to capture the consistency of messages posted by a user's friends. Then if the user only cares about messages posted by her friends, the semantic difference between the observed and ideal result of a read is solely determined by that conit. A write (message post) by a friend will affect the conit, while a write from other users has no effect on the conit.

In dealing with consistency, only the semantics-base is interesting to the application. Thus in our model, consistency is never specified on data items; rather, each conit has a consistency level. Each access then specifies the required consistency level for each conit it depends upon. Because the definition of each conit can be very flexible, we expect that the mapping between the physical world and the logical world can "absorb" most application-specific consistency semantics. This allows us to use a simple, application-independent set of metrics for conit consistency.

For each conit, we quantify consistency continuously along a three-dimensional vector:

$$\text{Consistency} = (\text{numerical error},\ \text{order error},\ \text{staleness})$$

Each conit has a logical numerical value. For example, in a bulletin board, the value of a conit could be the number of messages. Numerical error is the difference between the observed value of a conit and its ideal value if strong consistency were enforced. With the previous conit definition, numerical error will be reflected back to the physical world as the difference between the observed and ideal number of messages. Order error is the weighted out-of-order writes (subject to reordering and changing behavior) that affect a conit. In the bulletin board example, order error is the number of out-of-order messages. Staleness is the maximum age of the oldest write (globally across the system) affecting the conit that has not been seen by the local replica.

The intuition behind these three metrics comes from the execution of a replicated state machine [Schneider 1990]. The status of a state machine is uniquely determined by the instructions it executed. Replica coordination can be achieved by imposing two requirements: *agreement* (all state machines receive every instruction) and *order* (all state machines execute those instructions received in the same order). Numerical error is designed to capture how the agreement requirement is relaxed, while order error tries to quantify the order requirement. Finally, for applications with real-time consistency semantics, the staleness metric serves to explore the dimension of real-time, since such semantics cannot be expressed elegantly with the numerical error metric. Even though we do not claim that these three simple metrics form a spanning set of metrics for all possible consistency semantics, we believe they do capture some fundamental dimensions. Depending on conit definitions, these three metrics for conit consistency will translate to different application semantics. Section 5 will further discuss the generality provided by user-defined conits and the meaning of these metrics in various situations.

Figure 4 illustrates the definition of order error and numerical error in a simple example. Two replicas, A and B, accept updates on a conit containing

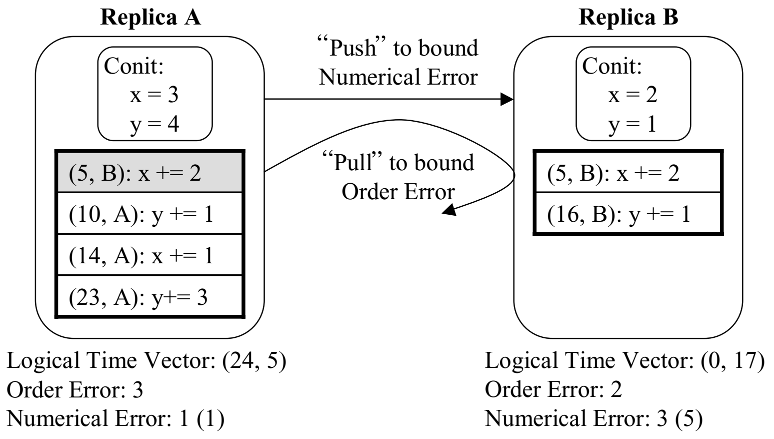

Fig. 4. Example scenario for bounding order error and numerical error with two replicas.

two data items, x and y. On A, one write is committed (indicated by the shaded box), leaving three tentative writes. Thus, order error on A is three. Similarly, with two tentative writes in its write log, B has an order error of two. Note that this is independent of whether these two writes have been seen by A: B's order error is solely determined by the number of tentative writes on B. Numerical error is the weight of all updates applied to a conit at *all* replicas not seen by the local replica. In the example, the weight of a write is set to be the update amount to either x or y, so that a "major" update is more important than a "minor" update. The replica A has not seen one update (with a weight of one) in this example, while B has not seen three updates (with a total weight of five).

One benefit of our model is that conit consistency can be bounded on a per-replica basis. Instead of enforcing a system-wide uniform consistency level, each replica can have its own independent consistency level for a conit. A simple analysis can show that as a replica relaxes its consistency while other replicas' consistency levels remain unchanged, the total communication of that replica is reduced. For relaxed numerical error, it means other replicas can push writes to that replica less frequently, resulting in fewer incoming messages. The amount of outgoing communication remains unchanged since that is determined by the consistency levels of other replicas. However, since numerical error is bounded using a push approach, if the replica is too busy to handle the outgoing communication, writes submitted to it will be delayed. Similarly, if the replica relaxes order error and staleness, the amount of incoming communication will be decreased. Thus, one site may have poor network connectivity and limited processing power, making more relaxed consistency bounds appropriate for that replica. Conversely, it may be cheap (from a performance and availability standpoint) to enforce stronger consistency at a replica with faster links and higher processing capacity. One interesting aspect of this model is that it potentially allows the system to route client requests to replicas with appropriate consistency bounds on a per-request basis. For instance, in the airline reservation system, requests from "preferred" clients may be directed to a replica that maintains higher consistency levels (reducing the probability of an inconsistent access).

4.3 Formal Conit/Consistency Definition

We now formalize the previous discussion on conit and consistency, starting from the concept of history. A *history* is a totally ordered (serial) set of reads and writes. Because standard concurrency control mechanisms on each replica ensure local serializability, we can define the *local history* of a replica to be the history corresponding to the equivalent serial execution of all accesses processed by that replica. The local histories are subject to reordering (due to write reordering). *Causal order* is a partial order defined over all accesses. An access A_1 *causally precedes* another access A_2 if A_1 is in the local history of A_2's originating replica when A_2 is accepted. To define a consistency spectrum, we need to use a global history that corresponds to a strongly consistent execution as a reference. Thus, we define *ECG history* (external-order-compatible, causal-order-compatible, global history) to be a history that is compatible with external and causal order and contains all accesses accepted by the system. Unless otherwise specified, the following discussion defines the consistency spectrum as the distance between local histories and a particular ECG history.

We use D to denote the database state at a particular time. Define D_{init} to be the initial state of the database. The notation $D+W$ denotes the database state after applying write procedure W to database state D, while $D+H$ means the database state after applying all writes in history H (in history order) to D. For each access, its *observed prefix history* ($PH_{observed}$) is its originating replica's local history when the access is submitted. $D_{init}+PH_{observed}$ is called the access's *observed database state* ($D_{observed}$), which determines the observed result of an access. The *ideal prefix history* (PH_{ideal}) of an access is the longest prefix of the ECG history that does not contain that access. $D_{init}+PH_{ideal}$ is called the access's *ideal database state* (D_{ideal}), which determines the ideal result of the access. The difference between the observed and ideal result of an access is then determined by the "difference" between $D_{observed}$ and D_{ideal}.

A *conit* is a function F that maps a database state D to a real number V. An application defines a conit set $\mathbf{F} = \{F_1, F_2, \ldots\}$, which can be infinite, to export its consistency semantics. Define the function *nweight* (numerical weight) of a write W, conit F, and database state D to be $nweight(W, F, D) = F(D+W) - F(D)$. Define the function *oweight* (order weight) to be a mapping from the tuple (W, F, D) to a nonnegative real value. To simplify discussion, we assume that *nweight* and *oweight* are independent of D (although our model is more general), so we can use the notations $nweight(W, F)$ and $oweight(W, F)$. A write *affects* a conit F if either $nweight(W, F) \neq 0$ or $oweight(W, F) \neq 0$. For a history H, define the *write order projection* of H on a conit set $\{F_1, F_2, .., F_n\}$ (denoted by $H|\{F_1, F_2, .., F_n\}$) to be the sequence of writes obtained by deleting all writes W in H, such that $oweight(W, F_i) = 0,\ \forall\, i, 1 \leq i \leq n$. Define $prefix(H_1, H_2)$ to be the longest common prefix of H_1 and H_2.

For an access A *depending on* a conit set $\{F_1, F_2, ..., F_n\}$ consistency $\mathcal{C}$ is defined for each F_i ($1 \leq i \leq n$) and is a three-dimensional vector (Numerical

$$
\begin{aligned}
\textbf{Numerical Error}(absolute) &= F_i(D_{ideal}) - F_i(D_{observed})\\
\textbf{Numerical Error}(relative) &= 1 - F_i(D_{observed})/F_i(D_{ideal})\\
\textbf{Order Error} &= \sum\{oweight(W, F_i) \mid W \in (PH_{observed}|\{F_1, .., F_n\} -\\
&\quad prefix(PH_{observed}|\{F_1, .., F_n\}, PH_{ideal}|\{F_1, .., F_n\}))\}\\
\textbf{Staleness} &= stime(A) - min\{rtime(W) \mid W \in (PH_{ideal} - PH_{observed}) \land\\
&\quad nweight(W, F_i) \neq 0 \land (W\ externally\ precedes\ A)\}, \text{where:}\\
&\quad stime(A) = \text{wall-clock time when access } A \text{ is submitted}\\
&\quad rtime(A) = \text{wall-clock time when access } A \text{ returns}
\end{aligned}
$$

Fig. 5. Conit consistency metrics.

ECG History

W1 affect {F1, F2}	R1 dep–on {F3}	W2 affect {F1}	W3 affect {F3}	W4 affect {F2}	W5 affect {F1}	R2 dep–on {F1, F2}

Local History on Replica1

W1 affect {F1, F2}	W3 affect {F3}	W4 affect {F2}	W2 affect {F1}	R2 dep–on {F1, F2}

For each conit F affected by write W:
nweight(W, F) = 1
oweight(W, F) = 1

Consistency of F1 for R2: NE(absolute) = 1; OE = 1; ST = stime(R2) – rtime(W5);
Consistency of F2 for R2: NE(absolute) = 0; OE = 1; ST = 0;

Fig. 6. Conit consistency example.

error, Order error, Staleness) as in Figure 5. Figure 6 illustrates the definition of our three consistency metrics. For simplicity, we assume that the writes do not depend upon any conit and carry unit numerical weight and unit order weight for each affected conit. In this example, the read R_2 depends on two conits, F_1 and F_2. Since W_1, W_2 and W_5 affect F_1 and each write has a numerical weight of 1, we have $F_1(D_{ideal}) = F_1(D_{init}) + 3$ in the ECG history. On the other hand, in the local history of $Replica_1$, we have $F_1(D_{observed}) = F_1(D_{init}) + 2$. Thus, the absolute numerical error of F_1 is 1 and staleness is $stime(R_2) - rtime(W_5)$. For order error, from the ECG history, we know that $PH_{ideal}|\{F_1, F_2\} = W_1W_2W_4W_5$. In the local history, for read R_2, $PH_{observed}|\{F_1, F_2\} = W_1W_4W_2$. Thus, the order error for F_1 is $oweight\,(W_4, F_1) + oweight\,(W_2, F_1) = 0 + 1 = 1$. Similarly, the consistency of F_2 for read R_2 is $(0, 1, 0)$.

To choose a consistency level, the application specifies bounds for the three metrics on a per-access and per-conit basis. Consistency is properly maintained if an ECG history, H, exists such that the numerical error, order error, and staleness of each (access, conit) tuple are within bounds with respect to H. Since consistency of an access is defined independently of the consistency of other accesses, the consistency model ensures a *self-determination property* for accesses. Such a property allows the system to provide differentiated consistency quality of service on a per-access basis.

4.4 Extremes of the Continuous Consistency Model

Tuning bounds on numerical error, order error, and staleness of each access/conit can provide different levels of consistency. To determine the range covered by our continuous consistency model, we study the two extremes of the spectrum: when the metrics are set to (∞, ∞, ∞) and $(0, 0, 0)$. In moving from strong to optimistic consistency, applications bound the maximum logical "distance" between the local replica image and the (unknown) consistent image that contains all writes in serial order. This distance corresponds directly to the percentage chance that a read will observe inconsistent results or that a write will introduce a conflict. The weak consistency extreme is achieved when none of the metrics are bounded and the system does not impose any restrictions on execution. We will explore the properties of the strong consistency extreme of our model by studying its relationship with 1SR [Bernstein et al. 1987]. An execution on replicated data is 1SR if it is view equivalent [Bernstein et al. 1987] to a serial execution on nonreplicated data. An access A *reads from* a write W if A reads some data item that was last written by W, and two executions are view equivalent if each access reads from the same write in the two executions.

THEOREM 4.1. *The conit-based continuous consistency model produces 1SR+EXT history if the application specifies the following consistency:*

(1) *A conit F is defined for each data item in the database, where $F(D)$ is the total number of writes applied to that data item.*
(2) *A write affects the conit set corresponding to the data items the write updates, with unit order weight.*
(3) *Each access depends upon the conit set corresponding to the data items it reads.*
(4) *Zero numerical error (implying zero staleness) and zero order error are enforced in all cases.*

PROOF. By definition of the continuous consistency model, an ECG history exists such that numerical error and order error are zero with respect to it. This ECG history is serial and compatible with external order and contains all accesses processed by the system. Thus, to prove the produced local histories are 1SR+EXT, we only need to show they are view equivalent to the ECG history. For each data item an access A reads, consider the set of writes (WS) that updates that data item. Since numerical error is zero and each write carries a unit numerical weight for each data item it writes, we know that the same number of writes from WS precedes A in $PH_{observed}$ as in PH_{ideal}, where $PH_{observed}/PH_{ideal}$ is the observed/ideal prefix history of A. Because ECG history is compatible with causal order, if W ($W \in WS$) precedes A in $PH_{observed}$, then W precedes A in PH_{ideal}. So we know that the same set of writes from WS precedes A in $PH_{observed}$ as in PH_{ideal}. Since each write also carries unit order weight for each data item it writes, zero order error ensures that these writes are in the same order in $PH_{observed}$ and PH_{ideal}. So access A reads from a write W in $PH_{observed}$ iff it reads from W in PH_{ideal}. Thus, the local histories are 1SR+EXT. □

In the Section 3, we discussed the self-determination of each access. Now we highlight the implications of this result for strongly consistent accesses.

COROLLARY 4.2 (SELF-DETERMINATION OF STRONGLY CONSISTENT ACCESSES). *If a conit is defined for each data item and each write carries a unit numerical/order weight for each affected conit, then for an access requiring zero numerical error and zero order error on all conits it depends upon, the observed result equals the ideal result.*

PROOF. Directly from Theorem 1. □

If we only require 1SR for the strong consistency extreme, then reads are allowed to observe nonzero numerical error:

THEOREM 4.3. *The conit-based continuous consistency model produces 1SR history if the application specifies the following consistency:*

(1) *A conit F is defined for each data item in the database, where $F(D)$ is the total number of writes applied to that data item.*
(2) *A write affects the conit set corresponding to the data items the write updates, with unit order weight.*
(3) *Each access depends upon the conit set corresponding to the data items it reads.*
(4) *Zero numerical error and zero order error are enforced for all conits a write depends upon.*
(5) *Zero order error is enforced for all conits a read depends upon.*

PROOF. Again, we consider the ECG history of the execution. However, because of nonzero numerical errors for reads, the produced local histories may have different read-from relations from those in the ECG history. We will construct another global history H' by reordering the reads in the ECG history in the following manner. For a read R depending upon conit set $\{F_1, F_2, \ldots, F_n\}$, suppose W is the last write in $PH_{observed}|\{F_1, F_2, \ldots, F_n\}$, where $PH_{observed}$ is the observed prefix history of R. Since the ECG history is compatible with causal order, we know W must precede R in the ECG history. To obtain H', every R in the ECG history is moved from its original place forward to the place immediately after the corresponding last write W in $PH_{observed}|\{F_1, F_2, \ldots, F_n\}$. Next, we will show that the produced local histories are view equivalent to the global history H'. From the proof of Theorem 1, we know that a write reads from the same write in the local history as in H. All writes in H and H' are in the same order; thus a write reads from the same write in the local history as in H'. For a read R, define $PH_{ideal'}$ to be the longest prefix of H' that does not contain R. Because the order error of each conit that R depends upon is zero and each write carries a unit order weight for at lease one conit, we know that $PH_{observed}|\{F_1, F_2, \ldots, F_n\} = PH_{ideal'}|\{F_1, F_2, \ldots, F_n\}$. So R reads from a write W in the local history iff it reads from W in H'. Thus, the local histories are 1SR. □

```
PostMessage(String msg) {
    // this write does not depend on any conit

    Append msg to the bulletin board;

    // unit nweight and unit oweight
    AffectConit("AllMsg", 1, 1);

    if (I am a friend of Alice)
        // unit nweight and unit oweight
        AffectConit("MsgFromFriends", 1, 1);
}
```

(a)

```
ReadMessages() {
    // require (10, 5, 99999) on "AllMsg"
    DependonConit("AllMsg", 10, 5, 9999);

    // require (3, 0, 60) on "MsgFronFriends"
    DependonConit("MsgFromFriends", 3, 0, 60);

    Retrieve news messages;
}
```

(b)

Fig. 7. Using weight specification in replicated bulletin board.

4.5 Exporting Conit Definitions through Weight Specification

To achieve practicality, we associate application-specific weights with each write to provide a natural API for application programmers and to avoid the complexity of exporting abstract conit definition functions. Recall from Section 4.3 that we define a conit as a function mapping database states to real numbers. However, to use our model, application programmers do not need to formally, or even conceptually, define such functions. Through the `AffectConit()` API call, the application directly tells the system how each write affects the return value of a conit F, and the system can then infer the return value of F by summing all accumulated numerical weights. The required consistency level on each access is declared using the `DependonConit()` API. With these two APIs, application programmers may even be unaware of the conit functions they implicitly define.

Figure 7 is a concrete example of a replicated bulletin board. We first define a conit with symbolic name "AllMsg," whose value is the number of news messages, to export the consistency requirements on all news messages. Besides these semantics, a user Alice also defines another conit with a symbolic name "MsgFromFriends," whose value is the number of news messages posted by Alice's friends. Thus, each write has a numerical weight of 1 for each affected conit. For simplicity, we also use unit order weight. Figure 7(a) is the message posting routine. In this example, a write does not depend on any conits and each message posted affects the conit "AllMsg" with unit numerical weight and unit order weight. If the author of the message is a friend of Alice, the message also affects the conit "MsgFromFriends." When Alice uses the routine in Figure 7(b) to retrieve news messages, she specifies the required consistency levels for the two conits the read depends on. For example, she requires the numerical error, order error and staleness on conit "MsgFromFriends" to be within 3, 0, and 60 (seconds), respectively. In this way, the actual definitions of the two conits "AllMsg" and "MsgFromFriends" are never directly exported to the system. Weight specification can even express subjective conit functions. For instance, a subjective nonunit numerical weight can be attached to each news message to export its relative importance so that postings from certain friends may carry higher weight than others.

4.6 Continuous Consistency Programming

One philosophy behind our model is that only the application can know its specific consistency requirements. At a very high level, this philosophy is consistent with that of extensible systems [Bershad et al. 1995; Kaashoek et al. 1997]. The major implication of such an approach is that application programmers are now responsible for determining the application-specific consistency semantics. We call the process of consistency-aware application design (*continuous*) *consistency programming*. Consistency programming can conceptually be done in the following top-down steps, with more concrete examples in Section 5:

(1) Crystallize high-level application consistency semantics and determine what "continuous consistency" means for this application from the user's perspective. For example, application programmers may start by asking "What does 90% consistency mean for this application?" It is possible that consistency has multiple aspects for an application, in which case the designer needs to answer this question for each aspect. This step should be completely independent of the the conit-based consistency model. Note that a continuous consistency model cannot and should not quantify high-level application semantics for programmers. Rather, it should only serve as a unified way to express and enforce such quantified consistency semantics. We believe one common pitfall here is to start with determining conits and write weights directly, and then to try to interpret these conits and weights into high-level semantics. Such a bottom-up approach can easily result in cases where the three application-independent metrics and various conits do not translate to any meaningful high-level semantics.
(2) Based on the quantified high-level consistency semantics, study how each write affects such semantics and determine the corresponding numerical/order weight. In the previous step, the high-level consistency semantics have been quantified, which allows numerical/order weight to be naturally derived. In the case where consistency has multiple aspects, multiple conits should be used and numerical/order weight should be determined for each conit. In most cases (in fact in all applications we studied), the weight can be solely determined by the write itself, without considering the state of the local database or remote replicas.

(3) Use the `AffectConit()` API calls to attach numerical/order weights to writes.
(4) Based on the quantified high-level consistency semantics, determine the depend-on conit set and consistency level of each access according to user requirements.
(5) Add the `DependonConit()` API calls to accesses to express these requirements.

Consistency programming can be directly incorporated into application logic, which entails modifying or instrumenting legacy code. An alternative is to export such semantics using a *semantics library* layer. The library layer sits between the application and TACT, intercepting reads and writes from applications. By examining the access from the application, the library layer inserts `AffectConit()` or `DependonConit()` into the access, then passes it on to TACT. This mechanism is similar to that of a locking-based scheduler [Bernstein et al. 1987] for traditional databases that inserts `lock()` and `unlock()` calls into transactions. In the extreme case, each application will have its own semantics library. However, similar to libOSes in the exokernel approach [Kaashoek et al. 1997], we believe many applications with similar consistency semantics may share the same semantics library, allowing some library reuse. We envision the development of a suite of common semantics libraries to aid application programmers with the use of our model.

In cases where the overhead of modifying legacy code or designing application-specific semantics libraries is prohibitive, some "general" semantics libraries can be used to run unmodified applications. For example, one "general" semantics library may define one conit per object, and attach unit weight to all writes. With "general" semantics libraries, our model can have the benefits of application-independent relaxed consistency models (e.g. epsilon-serializability [Pu and Leff 1991]), namely, supporting legacy code. In fact, an application-independent model can be viewed as a consistency model with a single semantics library already incorporated into that model. The relationship between application-independent models and our application-dependent model is then similar to the relationship between a monolithic operating system kernel and an extensible exokernel.

It may appear that continuous consistency programming imposes an extra burden on application programmers, since they now need to determine consistency semantics. However, we believe quantifying consistency in the most appropriate way entails such an effort from programmers (hopefully with the help of semantics libraries). A possible analog here is the relational data model [Codd 1970]. To exploit the power of the relational model, applications must be designed with such a model in mind. While it imposes an extra burden on programmers and potentially requires special training, the benefits of the relational model outweigh the costs. A long-term goal of this work is to demonstrate the requisite performance, availability, and semantic benefits of a continuous consistency model necessary to justify any additional cost associated with the model.

Table I. Expressing High-Level Application-Specific Consistency Semantics Using the TACT Continuous Consistency Model: NE = numerical error, OE = order error, ST = staleness

Application	Bulletin board	Airline reservation	Adm. control, resource Acc., sensor networks	Dynamic Web content
Consistency semantics	A. Message ordering B. Unseen messages C. Message delay	A. Resv. conflict rate R B. Inconsistent query result	A. Accuracy of resource info.	A. Subjective freshness of Web content
Conit Def.	A Newsgroup	Seats on a flight	Resource info.	Web object
Weight Def.	Subjective importance of the news message	Reservation: 1	Req. forward: 1 Req. return: −1	Update significance
Metrics capturing semantics	A. OE B. Absolute NE C. ST	A. Relative NE $R_{max} = 1 - 1/(1+\gamma)$ $R_{avg} = R_{max}/2$ B. OE and ST	A. Relative NE	A. Absolute NE and ST

Table II. Expressing High-Level Application-Specific Consistency Semantics Using the TACT Continuous Consistency Model: NE = numerical error, OE = order error, ST = staleness. (Continued)

Application	WAN shared editor	Distributed games, virtual reality	Traffic monitoring	Abstract data types
Consistency semantics	A. Unseen modifications B. Stability of local version	A. Accuracy of object position and orientation	A. Accuracy of traffic condition B. Unseen reservations C. Reservation stability	Varies
Conit def.	Characters in a paragraph	Object position, object orientation	Section of a road	Varies
Weight def.	Insert: 1 Delete: −1	Change in position and orientation	Size of the vehicle	Varies
Metrics capturing Semantics	A. Absolute NE B. OE	A. NE	A. NE B. NE C. OE	Varies

5. GENERALITY OF THE CONIT-BASED CONSISTENCY MODEL

5.1 Exporting Application Semantics through Conits

In Section 4, we elaborated our consistency model and explain how our model combines generality and practicality using conit theory, application-independent metrics and simple APIs. In this section, we further argue for the general utility of our approach by discussing how a number of wide-area applications (in addition to the previous bulletin board example) can specify their consistency semantics using conits. We also describe how a number of existing consistency models can be expressed using TACT. We will further address practicality issues in later sections. Note that because our consistency model is designed to capture a wide range of semantics, not all applications below will use all three consistency metrics. Table I and II summarize these application-specific consistency semantics and their expression using TACT, as detailed in the discussion below.

Airline Reservation System. One important aspect of consistency for this application is the percentage of reservations that must later be aborted as a result of simultaneous, conflicting updates at remote replicas. This consistency requirement can be captured using numerical error in the following manner. A conit F is used for each flight, with the conit value defined to be the number of available seats on that flight. Assuming single-seat reservations (though our model is more general) and that reservations are randomly distributed among all available seats, the probability P that a reservation conflicts with another remote (unseen) reservation is $1 - F(D_{ideal})/F(D_{observed})$. Since relative numerical error NE of the conit equals $1 - F(D_{observed})/F(D_{ideal})$, we can use NE to express the conflict rate: $P = 1 - 1/(1 - NE)$. Thus, the system can limit the rate of reservation conflicts by bounding relative numerical error. The above formula has been verified through experiments with our prototype implementation (see Figure 13). Nonrandom reservation behavior will result in a higher conflict rate, but the application may still limit conflict rates by defining multiple conits over the flight. For example, two separate conits can be defined for first class seats and coach seats. Other consistency semantics for the airline reservation example can be expressed using order error or staleness. For example, the system may wish to limit the percentage of queries that access an inconsistent image, for instance, that see a multiseat reservation that must later be rolled back because of a conflicting single-seat reservation at another replica. Such consistency semantics can be enforced by properly bounding order error.

QoS Admission Control. In this example application, we consider the case where replicated front-end load balancing switches wish to maintain some percentage of back-end server capacity for “preferred” clients. Here, front ends estimate the total resource consumption for standard clients as the total number of outstanding standard requests on the back ends. This value also serves as the definition of a conit for this application. Front ends increase this value by 1 upon forwarding a request from a standard client and decrease it by 1 when the request returns. If this value exceeds a predetermined resource consumption limit, front ends will not forward new standard client requests until resource consumption drops below this limit. The relative numerical error of each front end’s estimate of resource consumption captures this application’s consistency semantics—each front end is guaranteed that its estimate of resource consumption is accurate within a fixed bound. Note that this load balancing application is not concerned with order error (writes are interchangeable) or staleness (no need to synchronize if the mix of requests does not change).

Dynamic Content Distribution. Modern Web services produce much of their content dynamically based on database state. Consistency is a key hurdle to replicating dynamic services across the wide area. Conits address this problem by applying application-specific semantics to allow services to relax from strong consistency under certain circumstances. Consider a dynamic Web page tracking the score of a football game. The application can define a conit for this page and attach subjective numerical weights to changes in the score. For example, score changes near the end of a close game may be assigned more weight (signifying more importance). Conits may similarly be used to limit discrepancies in inventory for e-commerce services or the error in stock quotes provided by financial services.

Shared Editor. We use this application to represent wide-area collaborative applications [Dewan et al. 1994]. In a shared editor, multiple authors work on the same document simultaneously. Consistency requirements include the "amount" of modifications from remote authors not seen by a user and the "instability" of the current version due to uncommitted modifications. Several definitions of conits are possible. One approach is to define two conits per paragraph representing the number of characters in the paragraph. One conit tracks character additions, while the other tracks deletions. Numerical error then captures the "amount" of modifications not seen by a user. We can also define the order weight of a modification to be the number of characters it affects, and order error will capture the "instability" of the observed version. More functionality can be provided by, for example, defining a conit for each (paragraph, author) pair, so that modifications from different authors can have different consistency levels. Finally, staleness can be used to enforce a bound on modification propagation delay.

WAN Resource Accounting / Sensor Networks. These two very different applications represent a broader class of services that maintain pure numerical records that are read/updated from multiple locations. In resource accounting, the data records are the resource consumption of principles, while in sensor networks, the data records are the data measured by the sensors. A conit can be defined for each data record or group of records with numerical error capturing the accuracy of the record values.

Distributed Games / Virtual Reality / Teleimmersion. Most of the consistency issues for these applications [Gautier and Diot 1998; Honda et al. 1995] concern the positions and orientations of objects in the virtual world. Since both position and orientation are pure numerical data, their consistency semantics can be captured by numerical error. Furthermore, using different consistency levels for each conit/access can allow differentiated *focus* and *nimbus* [Benford et al. 1994] to represent the degree of interest objects have in each other.

Traffic Monitoring and Road Reservation. Advances in mobile technology have made "road reservation" possible. Here a mobile device is equipped to each vehicle and base stations collect and distribute traffic information to allow drivers to choose the "best" route. Road reservation helps to avoid the situation where many drivers choose the same "best" route and suddenly the route becomes overcrowded. Consistency here is the accuracy of the traffic/reservation information. We can define each section of the road to be a conit, its value being the number of vehicles in that section. To be more precise, different weights can be assigned to different vehicles to take into account the vehicle size, etc. Numerical error, again, describes the accuracy of the traffic/reservation information. Order error of the conit also plays a role here. A larger order error indicates that more "reservations" on the road have not been committed and may be canceled.

Abstract Data Types. Abstract data types naturally fit into our consistency model. For example, consider a set (or hashtable) with methods `add()`, `remove()`, `size()`, and `contains()`. We can define a conit whose value is the number of elements in the set. The accuracy of the return value of `size()` can then be reflected in the numerical error of the conit. Similarly, the probability of `contain()` returning a correct value is determined by the numerical error. Using the same reasoning as in airline reservations, we can derive a functional relationship between the conflict probability of `add()` (`remove()`) and numerical error. Other abstract data types, such as FIFO queues, can also express their consistency requirements naturally using our conit-based model.

5.2 Relationship to Other Consistency Models

To further demonstrate the generality of our conit-based consistency model, below we discuss how a number of existing relaxed consistency models can be expressed as special instances of our model.

Conflict Matrix. The use of a conflict matrix [Badrinath and Ramamritham 1992; DiPippo and Wolfe 1993; Weihl 1988] is a well-studied technique for relaxing the consistency of abstract data types. Each entry in the conflict matrix determines whether two methods on the same object can proceed in parallel. Our consistency model can achieve the same functionality using the following conit definition. Each method is considered a write. The ith row of the conflict matrix (associated with method M_i) is assigned a conit F_i, $1 \leq i \leq n$. For a method M_j corresponding to the jth column of the conflict matrix, M_j affects F_i iff the matrix entry (i, j) is a "conflict" entry. For each conit affected, M_j carries a unit numerical weight. Each method M_i depends on conit F_i and requires zero numerical error. In this way, all pairs of nonconflicting method invocations can be processed in parallel, while conflicting invocations have to be processed in a manner equivalent to 1SR. Note that if we enforce finite, instead of zero/infinity, numerical error for a matrix entry, we can provide the semantics of "bounded conflict" that cannot be obtained from a conflict matrix. For example, a `getBalance()` method on a bank account is allowed to miss no more than $50 deposited by `deposit()` operations.

Three-Level Consistency in Lazy Replication. Ladin et al. [1992] proposed three different consistency levels in lazy replication. A *causal transaction* is causally ordered to all other causal transactions, a *forced transaction* is totally ordered across all replicas with respect to all other forced transactions, and *immediate transactions* are totally ordered across all replicas with respect to all transactions. These consistency levels can be expressed using the following conflict matrix regarding the three types of transactions, and thus can be captured

Table III. Emulating Three-Level Consistency

Transaction type	Causal (affect F_3)	Forced (affect F_2 and F_3)	Immediate (affectF_1, F_2, and F_3)
Causal (dep-on F_1)	No conflict	No conflict	Conflict
Forced (dep-on F_2)	No conflict	Conflict	Conflict
Immediate (dep-on F_3)	Conflict	Conflict	Conflict

using TACT as described above. Sample conit specifications are included in the Table III.

Cluster Consistency. Cluster consistency [Pitoura and Bhargava 1995] is a two-level consistency model proposed for mobile environments. In this model, data copies are partitioned into *clusters*, where consistency constraints within a cluster must be preserved while intercluster consistency may be violated. Two kinds of operations are allowed: strict operations and weak operations. The consistency requirements of these operations can again be expressed as a conflict matrix, and thus can be captured by our model. To enforce "m-consistency" [Pitoura and Bhargava 1995] for some entries in the matrix, we can allow nonzero numerical/order error for the conit corresponding to that row.

N-ignorant System. In an N-ignorant system [Krishnakumar and Bernstein 1994], a transaction can run in parallel with at most N other transactions. To emulate the behavior of an N-ignorant system, we define a conit whose value is the number of transactions being applied to the database. A `beginTransaction` operation will increment the conit value, while an `endTransaction` operation will decrement the value. Bounding numerical error within N will make the system behave the same as an N-ignorant system.

Timed Consistency/Delta Consistency. These models [Singla et al. 1997; Torres-Rojas et al. 1999] address the lack of timing in traditional consistency models such as sequential consistency. They require the effect of a write to be observed everywhere within time Δ. TACT can readily express such timed models by using the staleness metric on conits.

Quasi-Copy Caching/Its Generalization. Quasi-copy caching [Alonso et al. 1990; Gallersdorfer and Nicola 1995] proposes four coherency conditions: delay condition, frequency condition, arithmetic condition, and version condition. Delay condition imposes an upper bound on propagation delay for a data item, which is a special case of staleness on conits. Frequency condition requires the copies of a data item to be synchronized every t seconds. We believe in most cases frequency condition can be more efficiently achieved by bounding staleness. Arithmetic condition bounds the difference between copies of numerical data items, which can be captured by the numerical error on conits. The last condition, version condition, bounds the version difference among copies. It can be achieved by using a conit whose value is the number of updates applied to a data item and by bounding the absolute numerical error of the conit. More recently, quasi-copy caching has been generalized and a few more coherency conditions have been proposed [Gallersdorfer and Nicola 1995]. Due to space limitations, we will only discuss the most distinct one, object condition. This condition requires the copies of an object x to be synchronized when: (i) at least i subobjects of x have been modified, (ii) at least q percent of the subobjects of x have been modified, or (iii) sub-object y of x has been modified. Emulating this condition requires three conits (F_1, F_2, and F_3), one for each of the three cases, to be defined for each object. The value of both F_1 and F_2 is the number of modified subobjects. To enforce case (i) and case (ii), we bound the absolute error of F_1 within i and bound the relative error of F_2 within q. The value of the third conit F_3 is the number of updates on subobject y. The numerical error of F_3 is bounded to zero, which means updates on y will force immediate synchronization.

Memory Consistency Models in Multiprocessors. Numerous memory consistency models [Keleher et al. 1992; Patterson and Hennessy 1996; Shen et al. 1999; Zekauskas et al. 1994] have been proposed in the context of multiprocessors/distributed shared memory. Due to space limitations, we cannot discuss these models individually and can only give a high-level abstract discussion. Most of the consistency models are defined by imposing ordering requirements on load, store, and other synchronization (e.g. fence, barrier, lock acquire) instructions. The system is allowed to reorder instructions, that is, violate program order, during execution, as long as the ordering imposed by the consistency model is preserved. Although our consistency model cannot be directly used in computer architecture (because reads/writes are heavy-weight compared to load/store instructions), the conit concept can still be applied. Consider a consistency model for a multiprocessor and a program running under this model. The ordering requirements imposed by the model on the program can always be viewed as a directed acyclic graph (DAG), whose nodes are instructions and edges are ordering requirements among instructions. To apply the conit theory, we need to redefine the reference history to be the history compatible to the DAG. Next, we assign a conit for each edge in the DAG. Each node in the DAG is modeled as a write and it depends on/affects the conit set corresponding to the set of incoming/outgoing edges of the node. Last, we enforce zero numerical error on all conits. It can then be shown that the resulting model is equivalent to the original memory consistency model. Different from the way we emulate other consistency models in this section, here conits must be dynamically defined and the number of conits can be quite large, making the implementation impractical. In Section 6, we further discuss the class of applications to which our consistency model may not be applicable.

6. DISCUSSION

In this section, we discuss our insights and experiences with the model.

Diminishing Returns from Additional Metrics. Section 5 demonstrated that our model is able to express a wide range of consistency semantics and that many previous models can be viewed as special instances of our model. Interestingly, the numerical error metric itself achieves much of the flexibility in our model, while the order error metric and staleness metric capture much of the semantics not already covered by numerical error. Adding additional metrics beyond our three metrics may make the model more general, but the marginal benefit in generality will become smaller and smaller and will come at the cost of higher complexity. Thus, we believe our metric set is a good tradeoff between complexity and the semantics space covered. Furthermore, our model mainly targets wide-area applications and our experiences suggest these three metrics are a good fit for these applications.

Amount of Continuity in Consistency. Following from our belief that the exact definition of consistency is application-specific, the question of whether consistency is "continuous" or how "continuous" it is also depends upon application semantics. Even though our consistency model allows consistency quantification to the finest granularity (all three metrics are continuous as real numbers), application programmers should only quantify consistency to the extent allowed by application semantics. Not all applications require the full spectrum of possible consistency values. For example, a distributed sensor system monitoring traffic conditions may be interested in all possible values of staleness bounds, while a banking system may be interested in only four different staleness bounds: zero, 1 hour, 1 day, and 1 week. Such "noncontinuity" on the consistency spectrum is inherent to the application's semantics. While consistency is not fully continuous for some applications, our goal of developing a general consistency model with broad applicability leads us to export the entire consistency spectrum to applications. In general, it is difficult to predict which discrete consistency levels are appropriate for which applications. It is straightforward to layer simpler or more restricted models on top of TACT.

Limitations of the Weight Specification API. Using per-write weights to avoid explicitly defining conits simplifies our model's API. However, such simplification does come with a cost. We have been assuming that the numerical weight and order weight of a write are independent of the database state, which enables the application to determine the weight solely based on the write itself. However, this precludes some possible conit definitions such as $F(D) = x \times y$, where x and y are two numerical values in D. Interestingly, from the generality discussion in previous sections, we can see that imposing such a restriction does not significantly impact our model's generality, since all of the considered examples can be addressed using weights that are independent of the database state.

Limitations of the Conit-Based Consistency Model. The ability to define conits arbitrarily makes our model quite flexible. For example, the model is able to precisely express all ordering relations in an arbitrary DAG for desired access ordering, as we explained earlier. However, for an arbitrary DAG, conits must be dynamically defined on the fly as new accesses are added into the DAG. The set of affected conits of a write may also expand after the write is accepted, since new outgoing edges need to be created for new accesses. Furthermore, because the number of conits grows with time and is unbounded, garbage collection for old conits may become necessary. A similar, but less severe problem exists in the replicated bulletin board example if we want to define one conit per message thread. Here no modification to the affected conit set of a write is necessary, but conits still need to be generated on the fly. In cases where the set of conits changes rapidly, further study on protocol design and implementation is necessary to make our model practical.

Combining Generality and Practicality. One major merit of the conit-based consistency model is that it simultaneously achieves generality and practicality. This allows us to reuse the consistency protocols and implementation for a broad range of application semantics. Fundamentally, however, we believe that generality and practicality conflict. By reusing consistency protocols, we necessarily sacrifice some opportunities for case-specific optimizations. Our conit-based consistency model can only combine generality and practicality to a particular point, beyond which application-specific optimizations based on semantics are necessary to make the implementation efficient. One simple example is causal consistency, which dictates causal order among writes. Causal order can be expressed using a DAG, and the discussion in Section 5 demonstrated that the conit-based model can theoretically emulate causal consistency. However, even if we solve the efficiency issues with dynamically defined conits, it is unlikely that the implementation will be as efficient as a straightforward protocol that always propagates writes causally. In this case, we believe optimizations exploiting the properties of the DAG become necessary to make the implementation efficient. Similar conclusions can be drawn for using our conit-based model for achieving release consistency [Keleher et al. 1992].

In these cases with special consistency semantics, specialized protocols can be used in place of our consistency protocol to achieve better performance. For example, we can add additional binary choices (besides the three continuous metrics) to our model so that the system can utilize better protocols when the semantics fit. One such binary choice may be causal consistency. When applications choose this option, a specialized protocol that propagates writes causally will be used to achieve causal consistency.

Replica Control and Concurrency Control. Informally, *replica control* is the coordination of replicas to properly maintain consistency across replicas in a distributed system, while *concurrency control* algorithms schedule the accesses to data within a single machine to ensure consistency. Continuous or relaxed consistency can be applied to both replica control and concurrency control. The TACT consistency model is for replica control only, since we dictate that local histories on each replica are serializable (Section 3). Most consistency models we mentioned in Section 5.2 are designed for replica control. In the traditional database context, there are also many relaxed consistency models designed for concurrency control, including the ANSI SQL isolation levels [ANSI 1992], its improved version [Adya et al. 2000], and relative serializability [Agrawal et al. 1994]. Whether these relaxed concurrency control models can be combined orthogonally with our model requires further study. However, our current design of requiring local serializability on each replica does offer some advantages. It potentially allows us to use off-the-shelf database servers on each replica with standard concurrency control and recovery mechanisms. Our continuous consistency will not interfere with the various (potentially tightly coupled) modules in a database management system. Furthermore, since our model targets wide-area data replication where communication overhead among replicas can easily dominate, replica control tends to be the bottleneck.

Having discussed the difference between replica control and concurrency control, we also believe there is a high-level one-to-one mapping between consistency models for replica control and consistency models for concurrency control. A natural question, then, is whether new consistency models for concurrency control or replica control can be derived using this mapping. We believe this is not always practical because, in replica control, the information needed to make a decision may or may not be available locally, while for concurrency control, the algorithm has all knowledge regarding the execution. Thus, from this perspective, replica control is harder than concurrency control, and the goal of replica control is to increase concurrency while reducing the communication overhead incurred by information collection. Mapping a consistency model for concurrency control to replica control may result in excessive communication overhead. A further difference between consistency models for replica control and those for concurrency control is that, in concurrency control, external consistency can be ensured trivially, since there is only one copy of the data. Thus, external consistency is not explicitly discussed in most consistency models for concurrency control, because it is ensured by standard implementation techniques. On the other hand, external consistency is a major issue in replica control; two of our metrics, numerical error and staleness, were developed particularly to address this requirement.

7. ENFORCING CONIT CONSISTENCY

In previous sections, we described the TACT continuous consistency model, its application to a broad range of applications, and its relationship to existing relaxed consistency models. However, a consistency model will not be of general interest unless efficient algorithms can be developed to enforce desired levels of consistency. In this section, we provide an overview of our algorithms for enforcing conit consistency; one benefit of our model is that the simplicity of the consistency metrics enables the protocols to be highly optimized.

The absolute/relative numerical error bounding algorithms [Yu and Vahdat 2000] for pure numerical data items are adopted for bounding the numerical error of conits. Note that the details and correctness proofs for our numerical error algorithms are available separately [Yu and Vahdat 2000]; we present an overview here for completeness. The first algorithm, *split-weight AE* (absolute error), employs a "push" approach to bound absolute numerical error. It "allocates" the allowed positive and negative error for a server evenly to other servers. Each $server_i$ maintains two local variables x and y for $server_j$, $j \neq i$. Intuitively, the variable x is the total weight of negatively weighted writes that $server_i$ accepts but have not been seen by $server_j$. $server_i$ has only conservative knowledge (called its *view*) of the writes $server_j$ has seen. The variable x is updated when $server_i$ accepts a new write with a negative weight or when *server*$_i$'s view is advanced. Similarly, the variable y records the total weight of positively weighted writes. Suppose the absolute error bound on *server*$_j$ is α_j. In other words, we want to ensure that $|F(D_{ideal} - F(D_{observed})| \leq \alpha_j$ on *server*$_j$. To achieve this, *server*$_i$ makes sure that at all times, $x \geq -\alpha_j/(n-1)$ and $y \leq \alpha_j/(n-1)$, where n is the total number of servers in the system. This may require *server*$_i$ to push writes to *server*$_j$ before accepting a new write. Split-weight AE is pessimistic in the sense that *server*$_i$ may propagate writes to *server*$_j$ when not actually necessary. For example, the algorithm does not consider the case where negative weights and positive weights may offset each other. We developed another algorithm, *compound-weight AE* [Yu and Vahdat 2000], to address this limitation at the cost of increased space overhead. However, simulations have indicated that potential performance improvements do not justify the additional computational complexity and space overhead [Yu and Vahdat 2000].

A third algorithm, *inductive RE* (relative error), provides an efficient mechanism for bounding the relative error in numerical records. The algorithm transforms relative error into absolute error. Suppose the relative error bound for *server*$_j$ is γ_j, that is, we want to ensure $|1 - F(D_{observed})/F(D_{ideal})| \leq \gamma_j$, equivalent to $|F(D_{ideal}) - F(D_{observed})| \leq \gamma_j \times F(D_{ideal})$. A naive transformation would use $\gamma_j \times F(D_{ideal})$ as the corresponding absolute error bound, requiring a consensus algorithm to be run to determine a new absolute error bound each time $F(D_{ideal})$ changes. Our approach avoids this cost by conservatively relying upon local information as follows. We observe that the current value $F(D_{observed})$ on any *server*$_i$ was properly bounded before the invocation of the algorithm and is an approximation of $F(D_{ideal})$. So *server*$_i$ may use $F(D_{observed})$ as an approximate norm to bound relative error for other servers. More specifically, for *server*$_i$, we know that $F(D_{ideal}) - F(D_{observed}) \geq -\gamma_i \times F(D_{ideal})$, where γ_i is the relative error bound for *server*$_i$, which reduces to $F(D_{ideal}) \geq F(D_{observed})/(1 + \gamma_i)$. Using this information to substitute for $F(D_{ideal})$ on the right-hand side in the previous inequality produces

$$|F(D_{ideal}) - \textit{server}_j\text{'s } F(D_{observed})| \leq \gamma_j \times \frac{\textit{server}_i\text{'s } F(D_{observed})}{1 + \gamma_i}$$

Thus, to bound relative error, *server*$_i$ only needs to recursively apply split-weight AE, using $\gamma_j \times (\textit{server}_i\text{'s } F(D_{observed}))/(1 + \gamma_i)$ as α_j. Note that while this approach greatly increases performance by eliminating the need to run a consensus algorithm among replicas, it uses local information $(F(D_{observed})/(1+\gamma_i))$ to approximate potentially unknown global information $(F(D_{ideal}))$ in bounding relative error. Thus it behaves conservatively (bounding values more than strictly necessary) when relative error is high, as will be shown in our evaluation of these algorithms in Section 9.

To bound order error on a per-conit basis, a replica first checks the number of tentative writes on a conit in its write log. If this number exceeds the order error limit, the replica invokes a write commitment algorithm to reduce the number of tentative writes in its write log. A write commitment algorithm is an algorithm that allows replicas to agree on a write order. A number of write commitment algorithms [Golding 1992a; Holliday et al. 2000; Keleher 1999;

Petersen et al. 1997] have been proposed and they can all be used to bound order error. We have implemented three popular write commitment algorithms (Golding's algorithm [Golding 1992a], primary copy [Petersen et al. 1997], and voting [Holliday et al. 2000; Keleher 1999]) in our prototype. However, our evaluation focuses on Golding's algorithm.

To bound the staleness of a replica, each server maintains a *real time vector*. The vector has an entry for each replica in the system. Similar to logical time vector [Golding 1992a], here a "coverage property" is preserved between the writes a server has seen and the real time vector. If A's real time vector entry corresponding to B is t, then A has seen all writes accepted by B before real time t. Even though the size of the real time vector grows linearly with the number of replicas, various techniques [Ratner et al. 1997; Torres-Rojas and Ahamad 1996] for reducing the vector size can be used for better scalability. To bound staleness within l, a server checks whether $\textit{current time} - t < l$ holds for each entry in the real time vector.[2] If the inequality does not hold for some entries, the server must pull writes from corresponding replicas to advance its real time vector. This pull approach may appear to be less efficient than a push approach because of unnecessary polling when no updates are available. However, a push approach cannot bound staleness if there is no upper limit on network delay or processing time.

All our protocols for bounding the three metrics are scalable relative to the number of conits. Such scalability is crucial for our model because the number of conits can be large (on the order of the number of data items in the database), depending on application semantics. In numerical error bounding protocols, we avoid maintaining constant-size bookkeeping information for each conit. Instead, necessary states are dynamically created when necessary and deleted when no longer in use. In the write commitment algorithms, scalability can be achieved by ignoring order relaxations enabled by multiple conits. In the extreme, if we simply use a conventional write commitment algorithm to generate a total order on all writes, the overhead incurred will be independent of the total number of conits. Our staleness bounding algorithm, by nature, is insensitive to the number of conits, since the algorithm needs to pull writes based on the smallest staleness bound specified across all conits.

Finally, we would like to emphasize that application consistency semantics, conit definitions, etc., are completely independent of the consistency protocols used to enforce the model. In some sense, the model, together with exported application consistency semantics, serves as a "specification," while our consistency protocols are just one possible "implementation" to achieve this specification. Decoupling implementation from specification allows application programmers to concentrate on high-level semantics without worrying about underlying protocols, while simultaneously enabling protocol

[2]We assume that server clocks are loosely synchronized. This assumption can be removed if we only update A's real time vector entry for B when A directly receives B's local writes from B. This will allow us to use A's clock to advance the real time vector entry rather than B's. Of course, this design precludes the possibility that A can advance the vector entry for B when it receives B's writes via other replicas.

designers to optimize the implementation without knowledge of application semantics.

8. SYSTEM ARCHITECTURE AND IMPLEMENTATION

To evaluate the performance benefits of dynamically setting service consistency levels, we have built and deployed across the Internet a prototype system implementing the TACT consistency model. In this section, we describe our architecture and design decisions before presenting our performance results in Section 9. Our first decision involves the method for distributing writes among replicas. Write propagation can take the form of gossip messages [Ladin et al. 1992], antientropy sessions [Golding 1992b; Petersen et al. 1997], group communication [Birman 1993], broadcast, etc. We choose antientropy exchange as our write propagation method because of its flexibility in operating under a variety of network scenarios. Each write bears an accept stamp composed of a logical clock time [Lamport 1978] and the identifier of the accepting replica. Replicas deterministically order all writes based on this accept stamp. As in Bayou [Petersen et al. 1997; Terry et al. 1995], updates are procedures that check for conflicts with the underlying data store before being applied in a *tentative* state.

Each TACT replica maintains a write log, and allows redo and undo on the write log. It is also responsible for all antientropy sessions with remote replicas. The system supports parallel antientropy sessions with multiple replicas, which can improve performance significantly for antientropy across the wide area. For increased efficiency, we also implement a one-round antientropy push. With standard antientropy, before a replica pushes writes to another replica, it first obtains the target replica's logical time vector to determine which writes to propagate. However, we found that this two-round protocol can add considerable overhead across the wide area, especially at stronger consistency levels (where the pushing replica has a fairly good notion of the writes seen by the target replica). Thus, we allow replicas to push writes using their local view as a hint, reducing two rounds of communication to one round at the cost of possibly propagating unnecessary writes. While the current implementation uses this one-round protocol by default, dynamically switching between the variants based on the consistency level would be ideal.

While TACT's implementation of antientropy is not particularly novel, a primary aspect of our work is determining when and with whom to perform antientropy in order to guarantee a minimum level of consistency. Replicas may propagate writes to other replicas at any time through *voluntary antientropy*. However, we are more concerned with write propagation required for maintaining a desired level of consistency, called *compulsory antientropy*. Compulsory antientropy is necessary for the correctness of the system, while voluntary antientropy only affects performance.

TACT replicas also implement a consistency manager responsible for bounding numerical error, order error and staleness. In bounding numerical error, a replica may need to push writes in its log to other replicas before the write can return, for example, if a write has a weight that is larger than another replica's absolute error bound. There are two possible approaches for addressing this requirement. One approach is a one-round protocol where the local site applies the write, propagates it to the necessary remote replicas, awaits acknowledgments, and finally returns. This one-round protocol is appropriate for applications where writes are interchangeable, such as resource accounting/load balancing. For other applications, such as the airline reservation example, a reservation itself observes a consistency level (the probability it conflicts with another reservation submitted elsewhere). In such a case, a stronger two-round protocol is required where the replica first acquires remote data locks, pushes the write to remote replicas, and then returns after receiving all acknowledgments. Such a two-round protocol ensures the numerical error observed by a write is within bounds at the time it is submitted. Our prototype implements both protocols and allows the application to choose the proper approach based on its requirements.

The current prototype of TACT is implemented in Java 1.2 using remote method invocation (RMI) for communication (e.g., for accepting read/write requests and for write propagation). TACT replicas are multithreaded. Thus, if one write incurs compulsory write propagation, it will not block writes on other conits. We implement a simple custom database for storing and retrieving data values, though our design and implementation is compatible with a variety of storage mechanisms.

9. PERFORMANCE EVALUATION

Given the description of our system architecture, we now discuss our experience in building the three applications described in Section 2 using the TACT infrastructure. We define conits and weights in these applications according to the analysis in Section 5.1. The experiments below focus on TACT's ability to bound numerical error and order error. While implemented in our prototype, we do not present experiments addressing staleness for brevity and because bounding staleness is well studied, for example, in the context of Web proxy caching [Fielding et al. 1997].

9.1 Bulletin Board

For our evaluation of the bulletin board application, we deployed replicas at three sites across the wide area: Duke University (733-Mhz Pentium III/Solaris 2.8), University of Utah (350-Mhz Pentium II/FreeBSD 3.4) and University of California, Berkeley (167-Mhz Ultra I/Solaris 2.7). All data was collected on otherwise unloaded systems. Each submitted message was assigned a numerical weight of 1 (all messages were considered equally important).

We conducted a number of experiments to explore the behavior of the system at different points in the consistency spectrum. Figure 8 plots the average latency for a client at Duke to post 200 messages as a function of the numerical error bound on the x-axis. For comparison, we also plot the average latency for a conventional implementation using a read-one write-all protocol. For each write, this protocol first acquires necessary remote data locks, then propagates the update to all remote replicas. The figure shows how applications are able to

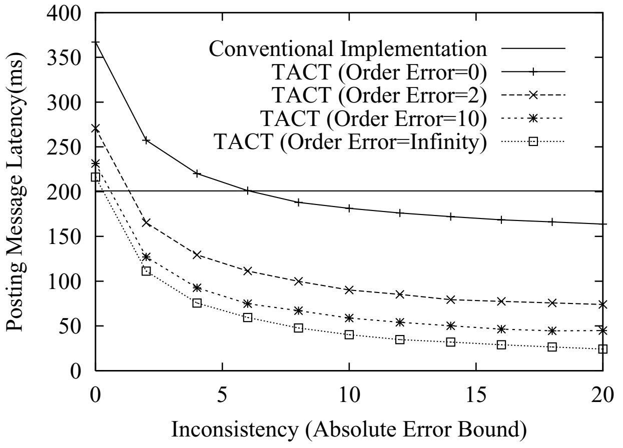

Fig. 8. Average latency for posting messages to a replicated bulletin board as a function of consistency guarantees.

continuously trade performance for consistency using TACT. As the numerical error bound increases, average latency decreases. Increasing allowable order error similarly produces a corresponding decrease in average latency. Relative to the conventional implementation, allowing each replica to have up to 20 unseen messages and leaving order error unbounded reduces average latency by a factor of 10.

A simple analysis on the number of synchronous RMIs per message can help us to better understand the results. Suppose the numerical error bound is $bound_{NE}$. With three replicas, $bound_{NE}$ will be split evenly to two other replicas, resulting in a local limit of $bound_{NE}/2$. Once the local limit is exceeded, two rounds of RMIs will be incurred by the protocol to propagate writes to remote replicas. The first round of RMIs acquires remote data locks, which is done sequentially to avoid deadlock. This will result in two synchronous RMIs. In the second round, the replica pushes writes to all other replicas in parallel. Thus, the number of synchronous RMIs per message is $3/(bound_{NE}/2 + 1)$. For order error, once the bound $bound_{OE}$ is violated on a replica, Golding's algorithm [Golding 1992a] will commit by pulling writes from all other replicas in parallel to advance logical time vector and commit writes. Thus, the number of synchronous RMIs per message is $1/(bound_{OE} + 1)$. If we ignore the possible overlap of the RMIs, the total number of RMIs per messages in TACT is $3/(bound_{NE}/2 + 1) + 1/(bound_{OE} + 1)$. A similar analysis can show that conventional implementation incurs three synchronous RMIs per message. We plot these calculated values in Figure 9, which is qualitatively similar to Figure 8.

One interesting aspect of Figure 8 is that TACT performs worse than the standard read-one write-all protocol at the strong consistency end of the spectrum. To investigate this overhead, Figure 10 summarizes the performance overhead associated with message posts using TACT at four points in the consistency spectrum (varying order error with numerical error set

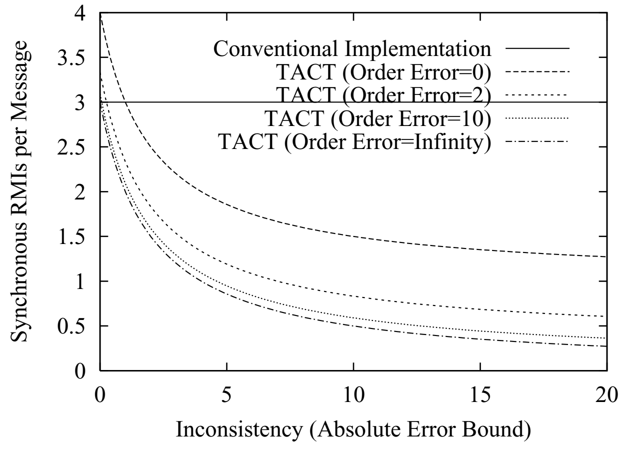

Fig. 9. Calculated number of synchronous RMIs per message as a function of consistency guarantees.

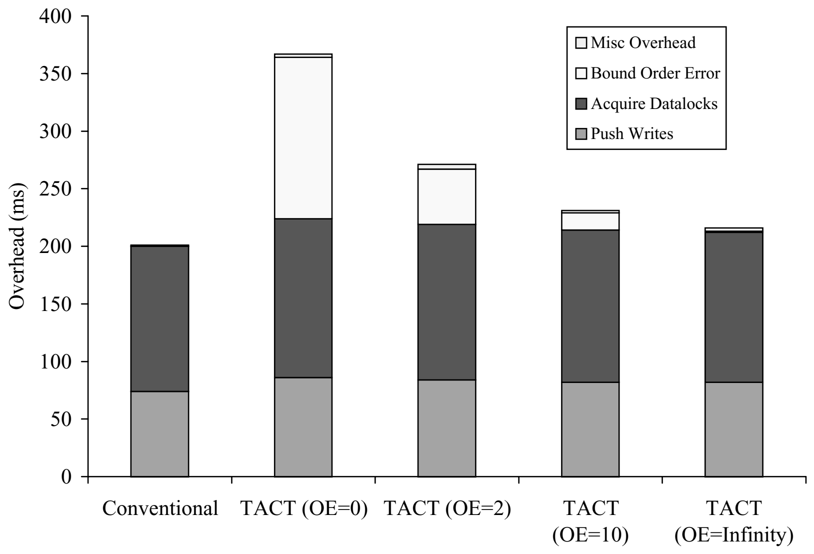

Fig. 10. Breakdown of the overhead of posting a message under a number of scenarios.

to zero) in comparison to the conventional read-one write-all protocol. All five configurations incur approximately 130 ms to sequentially (required to avoid deadlock) acquire data locks from two remote replicas and 80 ms to push writes to these replicas in parallel. Since the cost of remote processing is negligible, this overhead comes largely from wide-area latency. Compared to the conventional implementation, TACT with zero numerical error and zero order error (i.e., same consistency level) incurs about 83% more overhead. This additional overhead stems from the additional 140 ms to bound order error. This is an interesting side effect associated with our implementation. Our design decomposes consistency into two orthogonal components

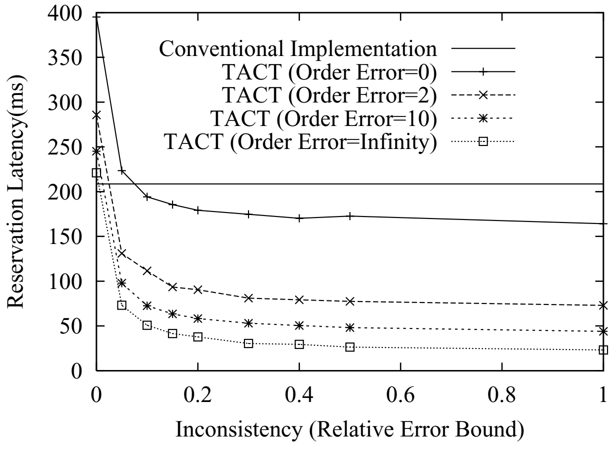

Fig. 11. Average latency for making a reservation as a function of consistency guarantees.

(numerical error and order error) that are bounded using two separate operations, doubling the number of wide-area round trip times. When order error and numerical error are both zero, TACT should combine the push and pull of write operations into a single step as a performance optimization, as is logically done by the conventional implementation. This idea is especially applicable if we use the recently proposed quorum approach [Holliday et al. 2000; Keleher 1999] to commit writes. A preliminary implementation of this optimization shows that TACT's overhead (at strong consistency) drops from 367 ms to 217 ms, within 8% of the conventional approach.

9.2 Airline Reservation System

We now evaluate our implementation of the simple airline reservation system using TACT. Once again, we deployed three reservation replicas at Duke, Utah, and Berkeley. We considered reservation requests for a single flight with 400 seats. Each client reservation request was for a randomly chosen seat on the flight. If a tentative reservation conflicted with a request at another replica, a merge procedure attempted to reserve a second seat on the same flight. If no seats were available, the reservation was discarded. A conit was defined over all seats on the flight, with an initial value of 400. Each reservation carried a numerical weight of -1.

The latency and throughput measurements summarized in Figures 11 and 12 for airline reservations are similar to the bulletin board application described above. The latency experiments were run on the same wide-area configuration as the bulletin board. The plotted latency was the average observed by a single Duke client making 400 reservations. As in the bulletin board example, the latency comes largely from the blocking communication required to maintain consistency. For throughput, we ran two client threads at each of the replica sites, with each thread requesting $400/(2 \times 3) = 67$ (random) seats in a tight loop. We also plot the application's performance using the read-one write-all protocol, showing the same trends as the results for the bulletin board application.

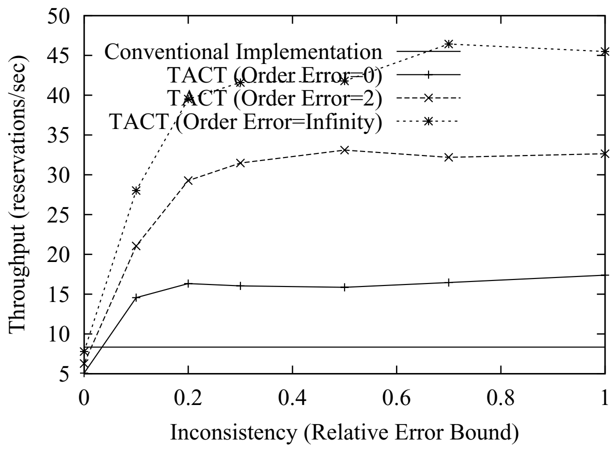

Fig. 12. Update throughput for airline reservations as a function of consistency guarantees.

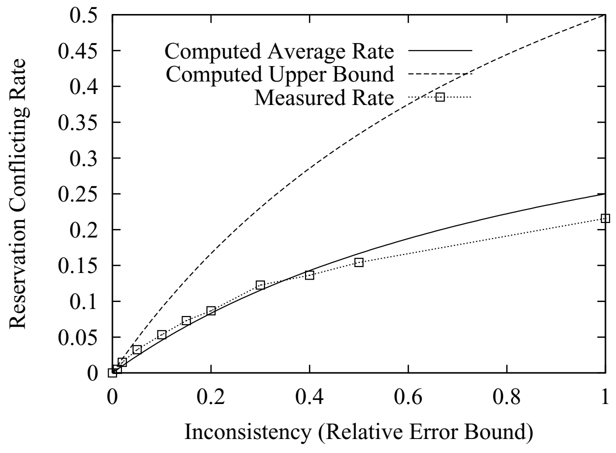

Fig. 13. Percentage of conflicting reservations as a function of the bound on numerical error.

As consistency was gradually relaxed, TACT achieved better performance by reducing the amount of required wide-area communication.

The improved performance does not, of course, come without cost. When consistency is relaxed, there is large possibility that two reservations may conflict. In Section 5.1, we derived a relationship between the reservation conflict rate R and the relative error bound γ: $R_{max} = 1-1/(1+\gamma)$ and $R_{avg} = (1-1/(1+\gamma))/2$. We conducted the following experiment to verify that an application can limit the reservation conflict rate by simply bounding the relative numerical error. Figure 13 plots the measured conflicting reservation rate R, the computed upper bound R_{max}, and the computed average rate R_{avg} as functions of relative numerical error. Order error and staleness are not bounded in these experiments. The experiments were performed with two replicas on a LAN at Duke, each attempting to make 250 (random) reservations with the results averaged across four runs.

The measured conflict rate roughly matches the computed average rate and is always below the computed upper bound, demonstrating that numerical error can be used to bound conflicting accesses as shown by our analysis. Note that, as the bound on relative error is relaxed, the discrepancy between the measured rate and the computed average rate gradually increases because of conservativeness inherent to the design of our inductive RE algorithm (i.e., at relaxed consistency, our algorithm performed more write propagation than necessary). As described in Section 3, this conservative behavior greatly improves performance by allowing each replica to bound relative error using only local information.

9.3 Quality of Service for Web Servers

For our final application, we demonstrate how TACT's numerical error bound can be used to enforce quality of service (QoS) guarantees among Web servers distributed across the wide area. Recall that in this application a number of front-end machines forward requests on behalf of both standard and preferred clients to back-end servers. In our implementation, we use TACT to dynamically trade communication overhead in exchange for accuracy in measuring total resources consumed by standard clients. The front ends estimate the standard client resource consumption as the total number of outstanding standard requests on the back ends. If this resource consumption exceeds a predetermined resource consumption limit, front ends will not forward new standard client requests until resource consumption drops below this limit. For simplicity, all our experiments were run on a local-area network at Duke on seven 733-Mhz Pentium III's running Solaris 2.8. Three front ends (each running on a separate machine) generated requests in a round-robin fashion to three back end servers running Apache 1.3.12.

For our experiments, the three front end machines generated an increasing number of requests from standard clients. As a whole, the system desired to bound the number of outstanding standard client requests to 150. A fourth machine, representing a preferred client, periodically polled a random back end to determine system latency. Each of the three front ends started a new standard client every 2 s, which then continuously requested the same dynamically generated Web page requiring 10 ms of computation time. If all front ends had global knowledge of system state, each front end would start a total of 50 standard clients. However, depending on the bound placed on numerical error, front ends may in fact start more than this number (up to 130 in the experiment described below). For simplicity, no standard clients are torn down even if the system learns that too many (i.e., more than 150) are present in aggregate. Ideally, this aggregate number would oscillate around 150 with the amplitude of the oscillation being determined by the relative numerical bound.

Figure 14 depicts latency observed by the preferred client as a function of elapsed time (corresponding to the total number of standard clients making requests). At time 260, each front end has tried to spawn up to 130 standard clients. The curves show the average latency observed by the preferred client for different bounds on numerical error. For comparison purposes, we also show the

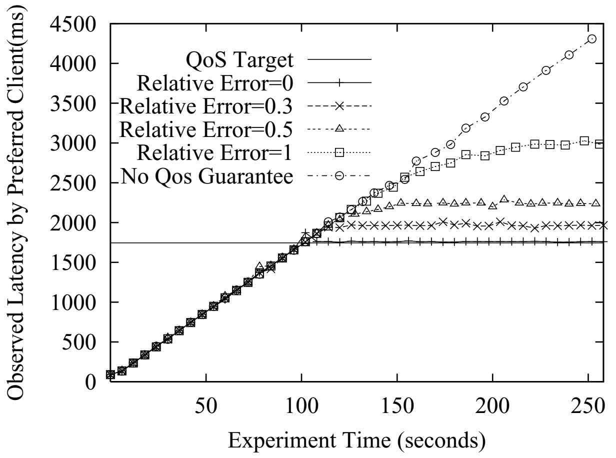

Fig. 14. The average latency seen by a preferred client as a function of time.

Table IV The Tradeoff Between TACT-Enforced Numerical Error and Communication Overhead

Configuration	Number of Messages
Relative Error = 0	300
Relative Error = 0.3	46
Relative Error = 0.5	30
Relative Error = 1	16
No QoS Guarantee	0

latency (1745 ms) of a preferred client when there are exactly 150 outstanding standard client requests. In the first curve, labeled "Relative Error=0," the system maintains strong consistency. Therefore, the front ends are able to enforce the resource limit strictly. The curve corresponding to a relative error of 0 flattens at 100 s (when three front ends have created a total of 150 standard clients) with latency very close to the ideal of 1745 ms. As the bound on relative error is relaxed to 0.3, 0.5, and 1, the resource consumption limit for standard clients is more loosely enforced. The curve "No_QoS" plots the latency where no resource policy is enforced. Similar to the airline reservation application, the discrepancy between the relative error upper bound of 1 and the "No_Qos" curve stems from the conservativeness of the inductive RE algorithm.

Table IV quantifies the tradeoff between numerical error and communication overhead. Clearly, front ends can maintain near-perfect information about the load generated from other replicas at the cost of sending one message to all peers for each event that takes place. This is the case when zero numerical error is enforced by TACT: each replica sends 50 messages to each of two remote replicas (for a total of 300) corresponding to the number of logical events that take place during the experiment. Once each front end starts 50 standard clients, strong consistency ensures that no further messages are necessary. Of course, such accuracy is typically not required by this application. Table IV shows that communication overhead drops rapidly in exchange for some loss of accuracy. Note that this dropoff will be more dramatic as the number of replicas is increased as a result of the all-to-all communication required to maintain strong consistency.

10. RELATED WORK

The tradeoff between consistency and performance/availability is well understood [Coan et al. 1986; Davidson et al. 1985]. Many systems have been built at the two extremes of the consistency spectrum. Traditional replicated transactional databases use strong consistency (one-copy serializability [Bernstein and Goodman 1984]) as a correctness criterion. At the other end of the spectrum are optimistic systems such as Bayou [Petersen et al. 1997; Terry et al. 1995], Ficus [Guy et al. 1990], Rumor [Guy et al. 1998], and Coda [Kistler and Satyanarayanan 1992]. In these systems, higher availability/performance is explicitly favored over strong consistency. Besides Bayou, none of the above systems provide support for different consistency levels. Bayou provides session guarantees [Edwards et al. 1997; Terry et al. 1994] to ensure that clients switching from one replica to another view a self-consistent version of the underlying database. However, session guarantees do not provide any guarantees regarding the consistency level of a particular replica. In some sense, session guarantees provide guarantees regarding the consistency *trend* across accesses, while our model provides guarantees for the consistency of a single access. Using the techniques in Bayou [Terry et al. 1994], session guarantees can be orthogonally incorporated into TACT.

Most of the previous relaxed consistency models under the traditional database context were not designed for the dual goals of generality and practicality. Agrawal et al. [1993] proposed semantics-based consistency criteria using *guarded actions*, which are primitive reads/writes associated with arbitrary consistency assertions. Wong and Agrawal [1992] applied similar ideas to abstract data types. In their model, a history is consistent if the assertions are satisfied when the system executes the associated read/write. In the similarity model [Kou and Mok 1992, 1993], applications define certain database states to be indistinguishable for concurrency control purposes. These three models can capture a broad range of application semantics. However, they place a significant burden on the application to match the model to their requirements. Further, they do not provide any practical, efficient protocols to enforce the requested consistency level in the general case. On the other hand, quasi-copy caching [Alonso et al. 1990; Gallersdorfer and Nicola 1995], N-ignorant systems [Krishnakumar and Bernstein 1994], delta consistency [Singla et al. 1997], timed consistency [Torres-Rojas et al. 1999], cluster consistency [Pitoura and Bhargava 1995], and models based on a conflict matrix for abstract data types [Badrinath and Ramamritham 1992; DiPippo and Wolfe 1993; Weihl 1988] have developed efficient application-independent protocols to enforce the relaxed consistency model. However, because they use a uniform consistency model for all applications, generality is sacrificed in favor of the consistency requirements of a specific class of applications. In Section 5.2, we showed that all these models can be expressed using our conit-based consistency model.

Pu and Leff [1991] proposed the concept of epsilon-serializability (ESR) to relax serializability and algorithms [Drew and Pu 1995; Pu et al. 1993; Wu et al. 1992] have been developed to enforce ESR. Relative to ESR, our conit-based model allows a broader range of application semantics to be expressed through flexible conit definitions. Another fundamental difference is that while we focus on trading consistency for reduced wide-area communication among replicas, ESR aims to increase the concurrency at a single site. The lifetime-based mutual consistency detection mechanism [Kordale and Ahamad 1996] can provide several discrete mutual consistency levels for different objects. Their mechanism is targeted to a different problem from ours, that is, to determine mutual consistency of objects in a system where client caches may retrieve individual objects from servers. Mutual consistency among data items is ensured because replicas directly propagate writes in TACT. Garcia-Molina and Wiederhold [1982] discussed *consistency* and *currency* requirements for query transactions. The consistency requirements roughly correspond to concurrency control (Section 6), while currency requirements are for replica control. The authors concentrated on partial replication with limited discussion on the currency requirement. On the other hand, we assume full replication and focus on quantifying application-specific consistency for both query and update transactions. One of our three metrics, staleness, can capture the currency requirement.

In fluid replication [Noble et al. 1999], clients dynamically create service replicas to improve performance. Their study on when and where to create a service replica is complementary to our study on tunable consistency issues among replicas. Similar to Ladin's system [Ladin et al. 1992], fluid replication supports three consistency levels: last-writer, optimistic, and pessimistic. Our work focuses on capturing the spectrum between optimistic and pessimistic consistency models. Varying the frequency of reconciliation in fluid replication allows applications to adjust the "strength" of the last-writer and optimistic models. Bounding staleness in TACT has similar effects. However, as motivated earlier, staleness alone does not fully capture application-specific consistency requirements.

Fox and Brewer [1999] argued that strong consistency and one-copy availability cannot be achieved simultaneously in the presence of network partitions. In the context of the Inktomi search engine, they showed how to trade harvest for yield. Harvest measures the fraction of the data reflected in the response, while yield is the probability of completing a request. In TACT, we concentrate on consistency among service replicas, but a similar "harvest" concept can also be defined using our consistency metrics. For example, bounding numerical error has similar effects to guaranteeing a particular harvest. Olston and Widom [2000] addressed tunable performance/precision tradeoffs in the context of aggregation queries over numerical database records. Finally, HOPE [Cowan et al. 1995] is a generic optimistic programming tool that uses speculative execution to exploit concurrency. However, HOPE provides no way to bound the probability that a speculative execution can be committed, while in TACT the amount of inconsistency can be clearly declared and enforced.

11. CONCLUSIONS AND FUTURE WORK

Traditionally, designers of replicated systems have been forced to choose between strong consistency, with its associated performance overhead, and optimistic consistency, with no guarantees regarding the probability of conflicting writes or stale reads. In this paper, we have explored the space in between these two extremes. We have presentd a continuous consistency model where application designers can bound the maximum distance between the local data image and some final consistent state. This space is parameterized by three metrics, *numerical error*, *order error*, and *staleness*. We showed how TACT, a middleware layer that enforces consistency bounds among replicas, allows applications to dynamically trade consistency for performance based on current service, network, and request characteristics. We argued for the generality of our approach by describing how a variety of services can express their consistency requirements using TACT and by showing how a number of existing consistency models can be expressed within the TACT framework. A performance evaluation of three replicated applications, an airline reservation system, a bulletin board, and a QoS Web service, implemented using TACT demonstrated significant semantic and performance benefits relative to traditional approaches.

Several topics remain unexplored in this paper. TACT quantifies consistency and makes it tunable, leaving the actual tuning policy unspecified. Can we build an adaptation layer on top of TACT that can tune application consistency levels in response to changing wide-area network characteristics to achieve application-specified targets for performance or availability? Are experiences from other adaptive systems applicable here? Further theoretical work on conits can also be interesting. Though simple, the conit concept provides much flexibility to the consistency model. In this paper, we have used numerous examples to explain the model's generality. However, we were not able to formally argue about it. Can we distill a set of computational/data access properties that are necessary to use the model? How stringent are these properties? Can we further use these properties to rigorously discuss the relative generality of various consistency models?

ACKNOWLEDGMENTS

We would like to thank David Becker for maintaining the experimental infrastructure at Duke. Jay Lepreau and Stephen Clawson were very helpful in providing access to the Utah computing resources available at the University of Utah Network Testbed. Jeff Chase and Doug Terry provided a number of insightful comments on the TACT design. Special thanks to Phil Bernstein for helpful discussion on the presentation of our work and future directions. Finally, the comments of the anonymous reviewers greatly improved this paper.

REFERENCES

ADYA, A., GRUBER, R., LISKOV, B., AND MAHESHWARI, U. 1995. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. ACM Press, New York, NY, 23–24.

ADYA, A., LISKOV, B., AND O'NEIL, P. 2000. Generalized isolation level definitions. In *Proceedings of the IEEE International Conference on Data Engineering*. IEEE Computer Society Press, Los alamitos, CA, 67–78.

AGRAWAL, D., ABBADI, A. E., AND SINGH, A. K. 1993. Consistency and orderability: Semantics-based correctness criteria for databases. *ACM Trans. Database Syst. 18*, 3 (Sept.), 460–486.

AGRAWAL, D., BRUNO, J. L., ABBADI, A. E., AND KRISHNASWAMY, V. 1994. Relative serializability: An approach for relaxing the atomicity of transactions. In *Proceedings of the 13th ACM Symposium on Principles of Database Systems*. ACM Press, New York, NY, 139–149.

ALONSO, R., BARBARA, D., AND GARCIA-MOLINA, H. 1990. Data caching issues in an information retrieval system. *ACM Trans. Database Syst. 15*, 3, 359–384.

ANSI. 1992. *American National Standard for Information Systems—Database Language—SQL*, ANSI X3. 135–1992. American National Standards Institute, New York, NY.

BADRINATH, B. R. AND RAMAMRITHAM, K. 1992. Semantics-based concurrency control: Beyond commutativity. *ACM Trans. Database Syst. 17*, 1 (March), 163–199.

BENFORD, S., FAHLEN, L., GREENHALGE, C., AND BOWERS, J. 1994. Managing mutual awareness in collaborative virtual environments. In *Proceedings of the ACM Conference on Virtual Reality and Technology*. ACM Press, New York, NY, 233–236.

BERNSTEIN, P. AND GOODMAN, N. 1984. The failure and recovery problem for replicated distributed databases. *ACM Trans. Database Syst. 14*, 2, 264–290.

BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA.

BERSHAD, B. N., SAVAGE, S., P. PARDYAK, E. G. S., FIUCZYNSKI, M., BECKER, D., CHAMBERS, C., AND EGGERS, S. 1995. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. ACM Press, New York, NY, 267–284.

BIRMAN, K. P. 1993. The proecss group appraoch to reliable distributed computing. *Commun. ACM 36,* 12, 36–53.

COAN, B., OKI, B., AND KOLODNER, E. 1986. Limitations on database availability when networks partition. In *Proceedings of the 5th ACM Symposium on Principle of Distributed Computing*. ACM Press, New York, NY, 187–194.

CODD, E. F. 1970. A relational model of data for large shared data banks. *Commun. ACM 13,* 6, 377–387.

COWAN, C., LUTFIYYA, H., AND BAUER, M. 1995. Performance benefits of optimistic programming: A measure of HOPE. In *Proceedings of the 4th IEEE International Symposium on High-Performance Distributed Computing*. ACM Press, New York, NY, 197–204.

DAVIDSON, S., GARCIA-MOLINA, H., AND SKEEN, D. 1985. Consistency in partitioned networks. *ACM Comput. Surv. 17,* 3, 314–370.

DEWAN, P., CHOUDHARY, R., AND SHEN, H. 1994. An editing-based characterization of the design space of collaborative applications. *J. Org. Comput. 4*, 3, 219–240.

DIPIPPO, L. B. C. AND WOLFE, V. F. 1993. Object-based semantic real-time concurrency control. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, Los alamitos, CA, 135–147.

DREW, P. AND PU, C. 1995. Asychronous consistency restoration under epsilon serializability. In *Proceedings of the 28th Hawaiian International Conference on Systems Sciences*. IEEE Computer Society Press, Los alamitos, CA, 16–25.

EDWARDS, W. K., MYNATT, E., PETERSEN, K., SPREITZER, M., TERRY, D., AND THEIMER, M. 1997. Designing and implementing asynchronous collaborative applications with Bayou. In *Proceedings of 10th ACM Symposium on User Interface Software and Technology*. ACM Press, New York, NY, 119–128.

FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., AND BERNERS-LEE, T. 1997. Hypertext Transfer Protocol—HTTP/1.1. RFC 2068.

FOX, A. AND BREWER, E. 1999. Harvest, yield, and scalable tolerant systems. In *Proceedings of HOTOS-VII*. IEEE Computer Society Press, Los alamitos, CA, 174–178.

GALLERSDORFER, R. AND NICOLA, M. 1995. Improving performance in replicated databases through relaxed coherency. In *Proceedings of the 21st International Conference on Very Large Databases*. VLDB Society, Mumbai, India, 445–456.

GARCIA-MOLINA, H. AND WIEDERHOLD, G. 1982. Read-only transactions in a distributed database. *ACM Trans. Database Syst. 7*, 2, 209–234.

GAUTIER, L. AND DIOT, C. 1998. Design and Evaluation of MiMaze, a multi-player game on the Internet. In *Proceedings of IEEE Multimedia Systems Conference*. IEEE Computer Society Press, Los alamitos, CA, 233–236.

GOLDING, R. 1992a. A weak-consistency architecture for distributed information services. *Comput. Syst. 5*, 4 (Fall), 379–405.

GOLDING, R. 1992b. *Weak-Consistency Group Communication and Membership*. Ph.D. thesis, University of California, Santa Cruz.

GUY, R., HEIDEMANN, J., MAK, W., JR., T. P., POPEK, G., AND ROTHMEIER, D. 1990. Implementation of the Ficus replicated file system. In *Proceedings Summer USENIX Conference*. USENIX, Berkeley, CA, 63–72.

GUY, R. G., REIHER, P., RATNER, D., GUNTER, M., MA, W., AND POPEK, G. J. 1998. Rumor: Mobile data access through optimistic peer-to-peer replication. In *Proceedings of the 17th International Conference on Conceptual Modeling* (ER'98).

HELLERSTEIN, J. M., HAAS, P. J., AND WANG, H. J. 1997. Online aggregation. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 171–182.

HERLIHY, M. AND WING, J. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12,* 3 (July), 463–492.

HOLLIDAY, J., STEINKE, R., AGRAWAL, D., AND ABBADI, A. E. 2000. Epidemic quorums for managing replicated data. In *Proceedings of the 19th IEEE International Performance, Computing, and Communications Conference*. IEEE Computer Society Press, Los alamitos, CA, 93–100.

HONDA, Y., MATSUDA, K., REKIMOTO, J., AND LEA, R. 1995. Virtual society: Extending the WWW to support a multi-user interactive shared 3D environment. In *Proceedings of the First Annual Symposium on the Virtual Reality Modeling Language*. 109–116.

KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., BRICEO, H. M., HUNT, R., MAZIRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. 1997. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. ACM Press, New York, NY, 52–65.

KELEHER, P. 1999. Decentralized replicated-object protocols. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, New York, NY, 143–151.

KELEHER, P., COX, A. L., AND ZWAENEPOEL, W. 1992. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*. ACM Press, New York, NY, 13–21.

KISTLER, J. J. AND SATYANARAYANAN, M. 1992. Disconnected operation in the Coda file system. *ACM Trans. Comput. Syst. 10,* 1 (Feb.), 3–25.

KORDALE, R. AND AHAMAD, M. 1996. A Scalable technique for implementing multiple consistency levels for distributed objects. In *Proceedings of the 16th IEEE International Conference on Distributed Computing Systems*. IEEE Computer Society Press, Los alamitos, CA, 23–30.

KRISHNAKUMAR, N. AND BERNSTEIN, A. 1994. Bounded ignorance: A technique for increasing concurrency in a replicated system. *ACM Trans. Database Syst. 19,* 4 (Dec.), 586–625.

KUO, T.-W. AND MOK, A. K. 1992. Application semantics and concurrency control of real-time data-intensive applications. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, Los alamitos, CA, 35–45.

KUO, T.-W. AND MOK, A. K. 1993. SSP: A semantics-based protocol for real-time data access. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, Los alamitos, CA, 76–86.

LADIN, R., LISKOV, B., SHIRIRA, L., AND GHEMAWAT, S. 1992. Providing availability using lazy replication. *ACM Trans. Comput. Syst. 10,* 4, 360–391.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21,* 7 (July), 558–565.

NOBLE, B., FLEIS, B., AND KIM, M. 1999. A case for fluid replication. In *Proceedings of the 1999 Network Storage Symposium* (Netstore). 1–5.

OLSTON, C. AND WIDOM, J. 2000. Bounded aggregation: Offering a precision-performance tradeoff in replicated systems. In *Proceedings of the 26th International Conference on Very Large Databases*. VLDB Society, Mumbai, India, 144–155.

PAI, V. S., ARON, M., BANGA, G., SVENDSEN, M., DRUSCHEL, P., ZWAENEPOEL, W., AND NAHUM, E. 1998. Locality-aware request distribution in cluster-based network servers. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, New York, NY, 205–216.

PATTERSON, D. AND HENNESSY, J. 1996. *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufman, San Francisco, CA.

PETERSEN, K., SPREITZER, M., TERRY, D., THEIMER, M., AND DEMERS, A. 1997. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (SOSP-16). ACM Press, New York, NY, 288–301.

PITOURA, E. AND BHARGAVA, B. K. 1995. Maintaining consistency of data in mobile distributed environments. In *Proceedings of the 15th International Conference on Distributed Computing Systems*. IEEE Computer Society Press, Los alamitos, CA, 404–413.

PU, C., HSEUSH, W., KAISER, G. E., WU, K.-L., AND YU, P. S. 1993. Distributed divergence control for epsilon serializability. In *Proceedings of the International Conference on Distributed Computing Systems*. IEEE Computer Society Press, Los alamitos, CA, 449–456.

PU, C. AND LEFF, A. 1991. Replication control in distributed system: An asynchronous approach. In *Proceedings of the ACM SIGMOD Conference on Management of Data*.

RATNER, D., REIHER, P., AND POPEK, G. J. 1997. Dynamic version vector maintenance. Technical report CSD-970022, University of California, Los Angeles, Los Angeles CA.

SAITO, Y., BERSHAD, B., AND LEVY, H. 1999. Manageability, availability and performance in Porcupine: A highly scalable Internet mail service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. ACM Press, New York, NY, 1–15.

SCHNEIDER, F. B. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4, 299–319.

SHEN, X., ARVIND, AND RUDOLPH, L. 1999. Commit-reconcile Fences (CRF): A new memory model for architects and compiler writers. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*. ACM Press, New York, NY, 150–161.

SINGLA, A., RAMACHANDRAN, U., AND HODGINS, J. 1997. Temporal notions of synchronization and consistency in Beehive. In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures*. ACM Press, New York, NY, 211–220.

TERRY, D., DEMERS, A., PETERSEN, K., SPREITZER, M., THEIMER, M., AND WELCH, B. 1994. Session guarantees for weakly consistent replicated data. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*. IEEE Press, Los alamitos, CA, 140–149.

TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. ACM Press, New York, NY, 172–183.

TORRES-ROJAS, F. AND AHAMAD, M. 1996. Plausible clocks: Constant size logical clocks for distributed systems. In *Proceedings of the 10th International Workshop on Distributed Algorithms*. IEEE Computer Society Press, Los alamitos, CA, 71–88.

TORRES-ROJAS, F., AHAMAD, M., AND RAYNAL, M. 1999. Timed consistency for shared distributed objects. In *Proceedings of the 18th ACM Symposium on Principle of Distributed Computing*. ACM Press, New York, NY, 163–172.

WEIHL, W. E. 1988. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput. 31*, 12, 1488–1505.

WONG, M. H. AND AGRAWAL, D. 1992. Tolerating bounded inconsistency for increasing concurrency in database systems. In *Proceedings of the 11th Symposium on Principles of Database Systems*. ACM Press, New York, NY, 236–245.

WU, K.-L., YU, P. S., AND PU, C. 1992. Divergence control for epsilon-serializability. In *Proceedings of 8th International Conference on Data Engineering*. IEEE Press, Los alamitos, CA, 506–515.

YU, H. AND VAHDAT, A. 2000. Efficient numerical error bounding for replicated network services. In *Proceedings of the 26th International Conference on Very Large Databases* (VLDB). USENIX, Berkeley, CA, 305–318.

ZEKAUSKAS, M. J., SAWDON, W. A., AND BERSHAD, B. N. 1994. Software write protection for distributed share memory. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*. USENIX, Berkeley, CA, 87–100.

Received May 2001; revised January 2002; accepted March 2002