Combining Generality and Practicality in a Conit-Based Continuous Consistency Model for Wide-Area Replication

Haifeng Yu
Computer Science Department
Duke University
Durham, NC 27708-0129
yhf@cs.duke.edu

Amin Vahdat
Computer Science Department
Duke University
Durham, NC 27708-0129
vahdat@cs.duke.edu

Abstract

Replication is a key approach to scaling wide-area applications. However, the overhead associated with large-scale replication quickly becomes prohibitive across wide-area networks. One effective approach to addressing this limitation is to allow applications to dynamically trade reduced consistency for increased performance and availability. Although extensive study has been performed on relaxed consistency models in traditional replicated databases, none of the models can simultaneously achieve the following two typically conflicting requirements imposed by wide-area applications: generality *(capturing application-specific consistency semantics) and* practicality *(enabling efficient application-independent consistency protocols to be designed and providing natural ways to express application semantics).*

In this paper, we propose a conit-based continuous consistency model designed to simultaneously achieve generality and practicality. Our conit theory provides generality, where application-specific consistency requirements are exported as conits*. Practicality is achieved by using a simple, spanning set of metrics for conit consistency and by using a per-write weight specification. We demonstrate the generality of our model through representative wide-area applications and by showing that a number of existing models can be expressed as instances of our model. Our efficient, application-independent consistency protocols and prototype implementation verify its practicality.*

1. Introduction

Replication is a key approach to scaling wide-area applications, such as e-commerce systems, dynamic content distribution, wide-area collaborative applications, sensor networks, and electronic bulletin boards. At the same time, the overhead associated with strong consistency in large-scale replication quickly becomes prohibitive across wide-area networks. One effective approach to addressing this limitations to allow these applications to dynamically trade reduced consistency for increased performance and availability[5, 10, 14, 24, 35, 34] based on a continuous (relaxed) consistency model. This diverse class of wide-area applications imposes the following two typically conflicting requirements, *generality* and *practicality*, on the continuous consistency model:

Generality These applications 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 maintaining such consistency semantics 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 to achieve 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 ways for application programmers to use the model in many cases.

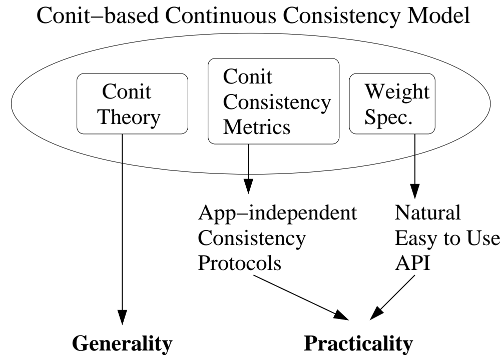

Figure 1. How the conit-based consistency model achieves two typically conflicting goals.

In the context of traditional replicated databases, much research [2, 3, 4, 8, 9, 10, 18, 19, 20, 24, 25, 26, 30, 31, 32] has been performed on relaxed consistency models. However, such traditional models typically achieve only one of generality and practicality. Some of the consistency models [2, 19, 20, 31] 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 API for application programmers, thus failing to meet the practicality requirement. Other relaxed consistency models [3, 4, 8, 9, 10, 18, 24, 25, 26, 30, 32] have easy to use interfaces and can be efficiently implemented, but they typically address the consistency requirements of a specific class of applications.

In this paper, we propose a conit-based continuous consistency model for wide-area data replication to simultaneously achieve generality and practicality (Figure 1). Generality is achieved by our conit theory. Each *conit* logically represents one particular application-specific consistency requirement. For example, in a replicated bulletin board, one possible consistency requirement is to bound the number of messages posted by other users but not seen locally. Another requirement can be the ordering among displayed news messages. These requirements serve as the definitions of conits. Consistency is defined on conits rather than data items and each conit has an application-independent consistency level. Each access (i.e. read or write) specifies the required consistency level for each conit it depends on.

Practicality is achieved in our model by i) using a simple, spanning set of metrics for conit consistency and ii) expressing semantics by simply specifying per-write weights. The flexibility of conits allows application-specific consistency semantics to be "absorbed" by the conit definition layer. Thus we can use a simple, application-independent, spanning set of metrics to define conit consistency, which enables the design of highly-efficient application-independent consistency maintenance protocols. To provide natural API to application programmers, we avoid the complexity of exporting abstract conit definitions. Instead, each access specifies the conit set it depends on and each write carries information about how it affects the consistency of each conit. The application programmer thus uses the model by attaching the necessary information to each access and never needs to explicitly define conits. Our prototype implementation and three sample application (replicated bulletin board, airline reservation and QoS load balancing for web servers)[34] have verified the practicality and scalability of the model.

The rest of this paper is organized as follows. Section 2 describes the replication model we assume. In Section 3, we present the conit-based continuous consistency model and discuss its generality and practicality. Next, Section 4 further explores the generality of our model through some representative wide-area applications and studies how some previous relaxed consistency models can be expressed as special instances of our model. An overview of the protocols implementing our consistency model is given in Section 5. Finally, Section 6 places our work in the context of related work and Section 7 presents our conclusions.

2. System model

Application data, referred to as the database for simplicity, 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. Writes in our model are procedures and replicas maintain consistency by propagating write procedures (rather than the data written) as in Bayou[23] and N-ignorant systems[18]. A write procedure checks for conflicts with the underlying database before updating the database state, allowing for application-specific conflict checking in a relaxed consistency environment. In case of a conflict, a write procedure may take an alternative action.

Each replica maintains a write log, containing all writes applied to its database image. Furthermore, each replica uses standard concurrency control mechanisms to ensure local serializability. The replica that first accepts an *access* (i.e., read or write) 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. The write can then be propagated to other replicas. Writes in a replica's write log may be reordered, e.g. rolled-back and then re-applied in a different order, with potentially different results. Write reordering is assumed to be isolated from reads and writes. At some point, a write becomes *commit-*

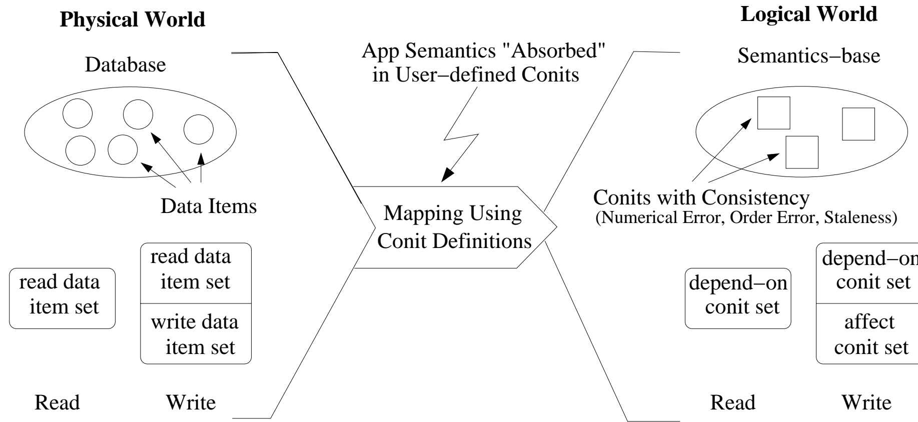

Figure 2. Role of the conit theory.

ted, which means it will never be reordered again. The *actual 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 *actual 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)[6]. 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, those reads can be considered to execute "in the past" by 1SR. As in timed consistency[28, 29] and external consistency[1], 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. Hereafter, we equate "strong consistency" with 1SR+EXT.

3. Conit-based continuous consistency model

In this section, we first present the conit theory and explain how it supports the consistency semantics of a broad range of replicated wide-area applications. Next, we formally define conits and their application-independent consistency level. We finish this section by demonstrating how applications can specify their consistency semantics by simply assigning weights to individual write operations.

3.1. 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 actual result. As we relax consistency, the observed result and the actual result begin to diverge. The meaning of the difference to the end users depends on application semantics. Thus, in order to quantify consistency and capture the semantic discrepancy between observed and actual results, we believe that a pre-defined uniform consistency 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*. For example, in a replicated bulletin board, sample consistency requirements include: i) the difference between observed/actual 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 2). 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 actual 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 actual 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, spanning set of metrics for conit consistency. We choose three metrics, *Numerical Error*, *Order Error* and *Staleness*, for conit consistency. 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 actual 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 actual 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 age of the oldest write (globally across the system) affecting the conit that has not been seen by the local replica. Depending on conit definitions, these three metrics for conit consistency will translate to different application semantics. Section 4 will further discuss the generality provided by user-defined conits and the meaning of these metrics in various situations.

3.2. 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 for reference purpose. 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 *actual prefix history* (PH_{actual}) of an access is the longest prefix of the ECG history that does not contain that access. $D_{init} + PH_{actual}$ is called the access's *actual database state* (D_{actual}), which determines the actual result of the access. The difference between the observed and actual result of an access is then determined by the "difference" between $D_{observed}$ and D_{actual}.

A *conit* is a function F that maps a database state D to a real number V. An application defines a conit set $\mathbf{F} = \{F_1, F_2, ...\}$, 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 non-negative 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 Error, Order Error, Staleness)** as in Figure 3. In the figure, function $stime(A)$ ($rtime(A)$) is the wall-clock time that access A is submitted by (returns to) the user.

Figure 4 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 one, we have $F_1(D_{actual}) = F_1(D_{init}) + 3$ in the ECG history. On the other hand, in

$$\textbf{Numerical Error}(absolute) = F_i(D_{actual}) - F_i(D_{observed}) \qquad \textbf{Numerical Error}(relative) = 1 - F_i(D_{observed})/F_i(D_{actual})$$

$$\textbf{Order Error} = \sum\{oweight(W, F_i) \mid W \in (PH_{observed}|\{F_1, .., F_n\} - prefix(PH_{observed}|\{F_1, .., F_n\}, PH_{actual}|\{F_1, .., F_n\}))\}$$

$$\textbf{Staleness} = stime(A) - min\{rtime(W) \mid W \in (PH_{actual} - PH_{observed}) \wedge nweight(W, F_i) \neq 0 \wedge W \ externally \ precedes \ A)\}$$

Figure 3. Conit consistency metrics.

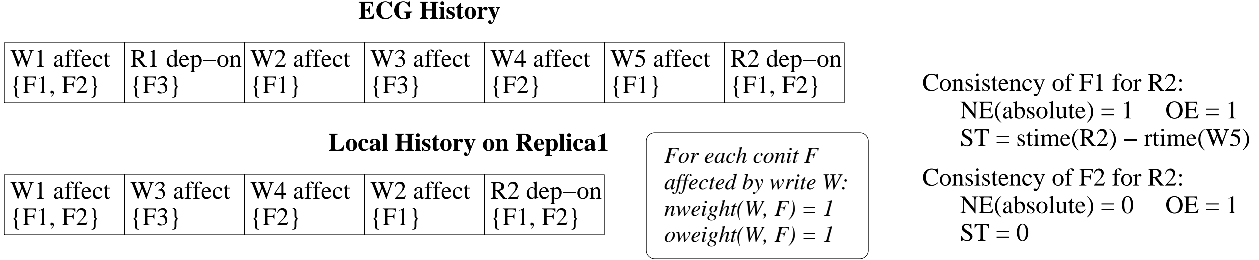

Figure 4. Conit consistency example.

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 $PH_{actual}|\{F_1, F_2\} = W_1W_2W_4W_5$. 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. The following theorem ensures that the result of each access is independent of the consistency level of other accesses. This self-determination property allows the application to provide differentiated consistency quality of service on a per access basis. Due to space limitations, the proof of this theorem and of all other theorems and corollaries are made available separately[33].

Theorem 1 (Self-Determination) *The semantic difference between the observed result and actual result of an access is guaranteed by the consistency level of the access, independent of the consistency of other accesses.*

Proof: Directly from the definition of the consistency of an access. □

3.3. 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 $(0, 0, 0)$. 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[6]. An execution on replicated data is 1SR if it is view equivalent[6] to a serial execution on non-replicated 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 2 *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

```
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)

Figure 5. Using weight specification in replicated bulletin board.

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_{actual}, where $PH_{observed}/PH_{actual}$ is the observed/actual 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_{actual}. So we know that the same set of writes from WS precedes A in $PH_{observed}$ as in PH_{actual}. 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_{actual}. So access A reads from a write W in $PH_{observed}$ iff it reads from W in PH_{actual}. Thus, the local histories are 1SR+EXT. □ In the last section, we discussed the self-determination of each access. Now we highlight the implications of this result for strongly consistent accesses.

Corollary 1 (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 actual result.*

Proof: Directly from Theorem 1 and Theorem 2. □ If we only require 1SR for the strong consistency extreme, then reads are allowed to observe non-zero numerical error:

Theorem 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 non-zero 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 $\{F_1, F_2, ..., F_n\}$, suppose W is the last write in $PH_{observed}|\{F_1, F_2, ..., F_n\}$, 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 $PH_{observed}|\{F_1, F_2, ..., F_n\}$. 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 $PH_{actual'}$ 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 $PH_{observed}|\{F_1, F_2, ..., F_n\} = PH_{actual'}|\{F_1, F_2, ..., F_n\}$. So R reads from a write W in the local history iff it reads from W in H'. Thus, the local histories are 1SR. □

3.4. Exporting conit definitions through weight specification

To achieve practicality, we use weight specification to provide natural API for application programmers and avoid the complexity of exporting abstract conit definition functions. Recall from Section 3.2 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. Rather, the application programmers can follow the following conceptual steps to use our model:

1. Crystallize high-level application consistency semantics.

2. Study how each write affects such semantics and determining the corresponding numerical/order weight.
3. Use `AffectConit()` statements to attach numerical/order weights to writes.
4. Determine the depend-on conit set and consistency level of each access according to application requirements.
5. Add `DependonConit()` statements to accesses to express such requirements.

In the weight specification step, 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 numerical weights accumulated. The application programmers may not even be aware of the conit functions they define in such a process.

Following is a concrete example of how this can be done in 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 one for each affected conit. For simplicity, we also use unit order weight. Figure 5(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 5(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 definition functions. For example, subjective numerical weight can be attached to each news message to export its relative importance.

4. Generality of the conit-based consistency model

4.1. Exporting application semantics through conits

In this section, we argue for the utility of our approach by discussing how a number of wide-area applications can specify their consistency semantics using conits. We will notice that not all applications below can fully utilize fine-grained continuous consistency in our model. 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, one hour, one day and one week. Such "non-continuity" on the consistency spectrum is inherent in the application's semantics and a continuous consistency model can only quantify consistency to the extent allowed by the applications' semantics. Also 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.

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 considered more important. Conits may further 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[7] 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 define the order weight of a modification also 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.

Airline Reservation System One important aspect of system consistency for this application is the percentage of reservations aborted as a result of conflicts. This aspect can be captured using numerical error in the following manner. A conit F is used for each flight and the value of the conit is 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 $1 - F(D_{actual})/F(D_{observed})$. Since relative numerical error NE of the conit equals $1-F(D_{observed})/F(D_{actual})$, 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[34]. Non-random reservation behavior will result in a higher conflict rate, but the application may still limit conflict rates by defining multiple conits over, for example, first class and coach seats.

Distributed Games/Virtual Reality/Teleimmersion [11, 14] Most of the consistency issue for these applications concerns the positions and orientations of objects in the virtual world. Since both position and orientation are pure numerical data, the semantics can be easily captured by numerical error. Furthermore, using different consistency levels for each conit/access can allow differentiated *focus* and *nimbus*[5] 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 help to collect/distributed 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 over-crowded. 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.

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.

4.2. Relationship to other consistency models

To further demonstrate the generality of our conit-based consistency model, in the following, we will discuss how some previous relaxed consistency models can be expressed as special instances of our model.

Conflict Matrix [4, 8, 30] The use of a conflict matrix 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 non-conflicting method invocations can be processed in parallel, while conflicting invocations have to be processed in a manner equivalent to 1SR. A correctness proof is omitted for brevity. 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 [21] Ladin et.al. propose 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 easily captured by our model. Sample conit specifications are included in the following table.

Transaction type	Causal (affect F_3)	Forced (affect F_2 and F_3)	Immediate (affect F_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

Cluster Consistency [24] Cluster consistency 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 inter-cluster 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"[24] for some entries in the matrix, we can allow non-zero numerical/order error for the conit corresponding to that row.

N-ignorant System [18] In an N-ignorant system, 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 applied to the database. A system bounding numerical error within N will behave the same as an N-ignorant system.

Timed Consistency/Delta Consistency [28, 29] These models 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. These timed models can be readily expressed using the staleness metric on conits.

Quasi-copy Caching and its Generalization [3, 10] Quasi-copy caching 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. Quasi-copy caching is later generalized and a few more coherency conditions proposed[10]. Due to space limitations, we will only discuss the more distinct one, object condition. This condition requires the copies of an object x to be synchronized when: i) at least i sub-objects of x have been modified, ii) at least q percent of the sub-objects 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 sub-objects. 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 sub-object y. The numerical error of F_3 is bounded to zero, which means updates on y will incur synchronization right away.

Memory Consistency Models in Multi-Processors [16, 22, 27, 36] Numerous number of memory consistency models have been proposed in the context of multi-processor/distributed shared memory. Due to space limitations, here we cannot discuss those 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 re-order 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, the conit concept can still be applied. Consider a consistency model for multi-processor and a program running under this model. The ordering requirements imposed by the model on the program can always be viewed as a DAG, whose nodes and edges are instructions and ordering among instructions, respectively. 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 in the DAG. 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. A correctness proof is omitted for brevity.

5. Implementation of the continuous consistency model and scalability issues

We have designed application-independent protocols to enforce conit consistency. Because of the simplicity of conit consistency metrics, the protocols can be highly optimized. We only give an overview here, detailed discussion of the protocols and their implementation can be found in [34, 35]. The absolute/relative numerical error bounding algorithms[35] for pure numerical data items are adopted for bounding numerical error of conits. Order error can be bounded with a write commitment algorithm, that is, an algorithm that allows replicas to agree on a write order[12, 13, 15, 23]. Staleness bounds can be enforced through a straightforward write pulling algorithm.

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 very 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, such information is 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.

Finally, a system prototype and wide-area evaluation of three sample applications (replicated bulletin board, airline reservation and QoS load distribution for web servers)[34] demonstrates the practicality of our approach.

6. Related work

In [35], we propose algorithms to enforce numerical error for pure numerical data records, even though the algorithms are also applicable to enforcing numerical error bounds for conit consistency. The prototype implementation and performance data of our consistency model are presented in [34]. However, in [34] we focus on how consistency can be traded for performance and no formal definition of conit, conit consistency or conit theory is provided. This paper concentrates on formal aspects of our consistency model and discusses how generality and practicality can be simultaneously achieved.

Most of the previous relaxed consistency models were not designed for the dual goals of generality and practicality. Agrawal et.al.[2] propose semantics-based consistency criteria using *guarded actions*, which are primitive reads/writes associated with arbitrary consistency assertions. Wong et.al.[31] apply 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[19, 20], 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[3, 10], N-ignorant systems[18], delta consistency[28], timed consistency[29], cluster consistency[24] and models based on a conflict matrix for abstract data types[4, 8, 30] 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 4.2, we showed that all these models can be expressed using our conit-based consistency model.

Pu et.al.[26] propose the concept of epsilon-serializability (ESR) to relax serializability and algorithms [9, 25, 32] 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[17] 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. Because replicas directly propagate writes in our system model, mutual consistency among data items in our model is always ensured.

7. Conclusions

In this paper, we propose a conit-based continuous consistency model to address the inherent overheads associated with large-scale replication in the Internet. Our model simultaneously achieve generality and practicality. These two goals usually conflict because generality requires application semantics to be exported, which typically precludes natural API and efficient, application-independent consistency protocols. Generality in our model is achieved by using user-defined conits to map the physical world to a logical world. We study the generality of our model by discussing how representative wide-area applications can export application-specific consistency semantics and how a number of existing relaxed consistency models can be expressed using our model. Practicality in our model is provided by i) using simple conit consistency metrics to allow application-independent consistency protocols and ii) using weight specification to simplify semantics expression. A number of efficient, application-independent protocols enforcing the consistency model and the prototype implementation verify its practicality.

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Syn-

chronized Clocks. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, May 1995.

[2] D. Agrawal, A. E. Abbadi, and A. K. Singh. Consistency and Orderability: Semantics-Based Correctness Criteria for Databases. *ACM Transactions on Database Systems*, September 1993.

[3] R. Alonso, D. Barbara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM Transactions on Database Systems*, September 1990.

[4] B. R. Badrinath and K. Ramamritham. Semantics-based Concurrency Control: Beyond Commutativity. *ACM Transactions on Database Systems*, March 1992.

[5] S. Benford, L. Fahlen, C. Greenhalge, and J. Bowers. Managing Mutual Awareness in Collaborative Virtual Environments. In *Proceedings of the ACM Conference on Virtual Reality and Technology*, 1994.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[7] P. Dewan, R. Choudhary, and H. Shen. An Editing-based Characterization of the Design Space of Collaborative Applications. *Journal of Organizational Computing*, pages 219–240, 1994.

[8] L. B. C. DiPippo and V. F. Wolfe. Object-based Semantic Real-time Concurrency Control. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1993.

[9] P. Drew and C. Pu. Asychronous Consistency Restoration under Epsilon Serializability. In *Proceedings of the 28th Hawaiian International Conference on Systems Sciences*, January 1995.

[10] R. Gallersdorfer and M. Nicola. Improving Performance in Replicated Databases through Relaxed Coherency. In *Proceedings of the 21st International Conference on Very Large Databases*, 1995.

[11] L. Gautier and C. Diot. Design and Evaluation of MiMaze, a Multi-Player Game on the Internet. In *Proceedings of IEEE Multimedia Systems Conference*, June 1998.

[12] R. A. Golding. A Weak-Consistency Architecture for Distributed Information Services. *Computing Systems*, 5(4):379–405, Fall 1992.

[13] J. Holliday, R. Steinke, D. Agrawal, and A. E. Abbadi. Epidemic Quorums for Managing Replicated Data. In *Proceedings of the 19th IEEE International Performance, Computing, and Communications Conference*, February 2000.

[14] Y. Honda, K. Matsuda, J. Rekimoto, and R. Lea. 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*, 1995.

[15] P. Keleher. Decentralized Replicated-Object Protocols. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, April 1999.

[16] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[17] R. Kordale and M. Ahamad. A Scalable Technique for Implementing Multiple Consistency Levels for Distributed Objects. In *Proceedings of the 16th IEEE International Conference on Distributed Computing Systems*, 1996.

[18] N. Krishnakumar and A. Bernstein. Bounded Ignorance: A Technique for Increasing Concurrency in a Replicated System. *ACM Transactions on Database Systems*, 19(4), December 1994.

[19] T.-W. Kuo and A. K. Mok. Application Semantics and Concurrency Control of Real-Time Data-Intensive Applications. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1992.

[20] T.-W. Kuo and A. K. Mok. SSP: A Semantics-Based Protocol for Real-Time Data Access. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1993.

[21] R. Ladin, B. Liskov, L. Shirira, and S. Ghemawat. Providing Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, 10(4):360–391, 1992.

[22] D. Patterson and J. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, Inc., second edition, 1996.

[23] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16)*, pages 288–301, October 1997.

[24] E. Pitoura and B. K. Bhargava. Maintaining Consistency of Data in Mobile Distributed Environments. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, May 1995.

[25] C. Pu, W. Hseush, G. E. Kaiser, K.-L. Wu, and P. S. Yu. Distributed Divergence Control for Epsilon Serializability. In *Proceedings of the International Conference on Distributed Computing Systems*, 1993.

[26] C. Pu and A. Leff. Replication Control in Distributed System: An Asynchronous Approach. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, May 1991.

[27] X. Shen, Arvind, and L. Rudolph. Commit-reconcile Fences (CRF): A New Memory Model for Architects and Compiler Writers. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[28] A. Singla, U. Ramachandran, and J. Hodgins. Temporal Notions of Synchronization and Consistency in Beehive. In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures*, June 1997.

[29] F. Torres-Rojas, M. Ahamad, and M. Raynal. Timed Consistency for Shared Distributed Objects. In *Proceedings of the 18th ACM Symposium on Principle of Distributed Computing*, May 1999.

[30] W. E. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Transactions on Computers*, December 1988.

[31] M. H. Wong and D. Agrawal. Tolerating Bounded Inconsistency for Increasing Concurrency in Database Systems. In *Proceedings of the 11th Symposium on Principles of Database Systems*, June 1992.

[32] K.-L. Wu, P. S. Yu, and C. Pu. Divergence Control for Epsilon-Serializability. In *Proceedings of 8th International Conference on Data Engineering*, February 1992.

[33] H. Yu and A. Vahdat. Combining Generality and Practicality in a Conit-Based Continuous Consistency Model for Wide-Area Replication. Technical Report CS-2000-09,

Duke University, Computer Science Department, Durham, NC, 2000. See http://www.cs.duke.edu/~yhf/icdcstr.pdf.

[34] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.

[35] H. Yu and A. Vahdat. Efficient Numerical Error Bounding for Replicated Network Services. In *Proceedings of the 26th International Conference on Very Large Databases*, September 2000.

[36] M. J. Zekauskas94, W. A. Sawdon, and B. N. Bershad. Software Write Protection for Distributed Share Memory. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, November 1994.