Replica Control in Distributed Systems: An Asynchronous Approach

Calton Pu[*] and *Avraham Leff*[†]
Department of Computer Science
Columbia University
New York, NY 10027

Abstract

An asynchronous approach is proposed for replica control in distributed systems. This approach applies an extension of serializability called *epsilon-serializability* (ESR), a correctness criterion which allows temporary and bounded inconsistency in replicas to be seen by queries. Moreover, users can reduce the degree of inconsistency to the desired amount. In the limit, users see strict 1-copy serializability. Because the system maintains ESR correctness (1) replicas always converges to global serializability and (2) the system permits read access to object replicas before the system reaches a quiescent state.

Various replica control methods that maintain ESR are described and analyzed. Because these methods do not require users to refer explicitly to ESR criteria, they can be easily encapsulated in high-level applications that use replicated data.

1 Introduction

Data replication offers the benefits of autonomy, performance, and availability. Unfortunately, ensuring that the replicas remain mutually consistent (coherency control) is a difficult problem because of the tradeoffs involved. Typical coherency control methods are synchronous, in the sense that they require the atomic updating of some number of copies. From the point of view of performance, synchronous methods decrease system availability and throughput as the size of the system increases. From the point of view of autonomy, federated databases may not wish to support this kind of tight coupling. On the other hand, a basic problem with *asynchronous* coherency control methods is that the system enters an inconsistent state in which replicas of a given object may not share the same value. Standard correctness criterion for coherency control such as 1-copy serializability (1SR) [4] are hard to attain with asynchronous coherency control.

Epsilon-serializability (ESR) is a correctness criterion which offers the possibility of maintaining mutual consistency of replicated data asynchronously. First, ESR allows inconsistent data to be seen, but requires that data will eventually converge to a consistent (1SR) state. Moreover, ESR allows the degree of inconsistency to be controlled so that the amount of error (departure from consistency) can be reduced to a specified margin. A distributed system which supports ESR permits temporary and limited differences among object replicas: these replicas are required to converge to the standard 1SR coherency as soon as all the update messages arrive and are processed. These systems benefit from the increased asynchrony allowed under ESR, and which results from (1) controlled inconsistency in queries and from (2) use of operation semantics that go beyond the usual Read/Write operations.

Several practical *replica control* mechanisms can guarantee ESR, combining high autonomy, performance, and availability. A high-level interface called *epsilon-transaction (ET)* encapsulates the ESR abstraction so users need not explicitly deal with the theoretical conditions satisfying ESR. The advantages of ESR are not supplied for "free": each replica control method imposes some particular restrictions to achieve asynchronous consistency. For example, the ordered-update method works by requiring update ETs to be executed in an SR order.

[*]This work was partially supported by Digital Equipment Corp., New York State Center for Advanced Technology on Computer and Information Systems, and an NSF CISE Instrumentation Grant CDA-88-20759.

[†]This author was partially supported by an IBM Fellowship.

[0]To appear in SIGMOD 91.

In exchange, this method allows query ETs to be processed in *any* order.

Just as coherency control and atomic transactions ensure synchronous mutual consistency without requiring references to 1SR, replica control and ETs ensure asynchronous mutual consistency without references to ESR. In this paper we view the role of both coherency control and replica control as consistency maintenance among replicas of a given "logical" object. We therefore only discuss replica control methods: the replicated system is assumed to use standard (synchronous) concurrency control [4] or (asynchronous) divergence control methods [20] in order to maintain consistency among different objects in the system.

In Section 2 we introduce the ESR terminology, model, and its properties. In Section 3, asynchronous replica control methods are described, analyzed, and are shown to provide ETs with ESR properties. Finally, in Section 4, we discuss some related work.

2 Replica Control

Even if distributed systems are willing to pay the price of some inconsistency in exchange for the freedom to do asynchronous updates, they will insist that (1) the degree of inconsistency be bounded precisely, and that (2) the system guarantee convergence to standard notions of "correctness". Without such properties, the system in effect becomes partitioned as the replicas diverge more and more from one another [9]. These important properties are provided by ESR, which is a framework of constrained inconsistency/divergence [20]. The key idea is that asynchronous updates to replicas can be done *approximately atomically*, i.e., updates are atomic within certain time lags, but have the same value when completed.

2.1 ESR and ETs

An ET is a sequence of operations on data objects. These operations are divided into two class: reads and writes. An ET containing only reads is a *query ET* (denoted by Q^{ET}) and an ET containing at least one write is an *update ET* (denoted by U^{ET}). An U^{ET} preserves data consistency. In other words, if the objects modified by an U^{ET} are initially in a consistent state then, after the ET finishes (without interference from operations outside the ET), the objects will remain in a consistent state.

If update ETs are executed concurrently, we require them to be serializable (SR) [4]. However, ETs take advantage of operations which increase concurrency and allow more interleaving. For example, commutative operations can be interleaved more freely than reads and writes. Semantics-based consistency maintenance methods are in general termed *divergence control* methods. Divergence control methods play an analogous role in replicated systems to that of concurrency control methods: just as the latter maintain overall system consistency by ensuring *SR* correctness, the former maintain overall system consistency by ensuring ESR correctness.

Query ETs are allowed to interleave with other ETs (both queries and updates) freely. Therefore, query ETs may see an inconsistent object state produced by update ETs. This property does not disturb data consistency since query ETs do not change object state. This property *does* increase concurrency, because it increases the number of allowed interleavings.

In order to define the ET interface more precisely, we use an extension of the standard SR model, which we first briefly summarize. A history or *log*[1] is a sequence of operations. A *serial* log is a sequence of operations composed of consecutive transactions. A history (also called a schedule or a log) of transaction operations (typically reads and writes) is said to be serializable (SRlog) if it produces results equivalent to some serial schedule, in which the same transactions executed sequentially, one at a time. Concurrency control methods that preserve SR (e.g., two-phase locking) are algorithms that restrict the interleaving of operations in such a way that only SRlogs are allowed. In the standard model, a log is shown to be an SRlog by rearranging its operations according to certain rules. These rules are called read-write (R/W) and write-write (W/W) dependencies. Once a dependency is established between two operations, then we cannot move one past the other in the rearrangement, since the resulting log would have produced a different database.

A log containing only query ETs and update ETs is called an ϵ-serial log if, after deleting query ETs from the log, the remaining update ETs form an SRlog. An ESRlog is a history of operations of ETs that produce results equivalent to an ϵ-serial

[1] The term *log* is unrelated to the write-ahead log in database recovery.

log. An example of an ϵ-serial log is:

$$R_1(a)W_1(b)W_2(b)R_3(a)W_2(a)R_3(b) \qquad (1)$$

Even though $U_2^{ET} = W_2(b)W_2(a)$ and $Q_3^{ET} = R_3(a)R_3(b)$ are not SR, the deletion of Q_3^{ET} results in the log being an SRlog (actually a serial log) formed by U_1^{ET} and U_2^{ET}. As a result, log (1) still qualifies as an ϵ-serial log.

In an ϵ-serial log, a query ET may overlap with update ETs. If the query ET accesses objects that are affected by these update ETs (R/W dependencies), then the potential for inconsistency exists. We define the *overlap* of a query ET as the set of all update ETs that had not finished at the first operation of the query ET, plus all the update ETs that started during the query ET. (The term "update ETs" refers here to the set of update ETs that actually affect objects that the query ET seeks to access). In our example log (1), U_2^{ET} and Q_3^{ET} overlap. The overlap is an upper bound of error on the amount of inconsistency that a query ET may accumulate. If a query ET's overlap is empty, then it is SR.

2.2 Replica Control and ESR

Replica control maintaining ESR correctness involves asynchronous propagation of replica updates in a distributed system. Our model is that of a number of sites connected by a network, where both individual sites and network links may fail. We factor out the problem of internal system consistency due to site failures by encapsulating it in the local message processing, which assumes each site is capable of maintaining local consistency. Similarly, we factor out the problem of message losses by encapsulating it in stable queues which persistently retry message delivery until successful. The problem of replica control, therefore, is to keep replicas consistent with each other with a technique that is robust in face of very slow links, network partitions, and site failures.

We reiterate here the distinction between replica control, which ensures asynchronous mutual consistency under ESR, and traditional coherency control, which ensures synchronous mutual consistency under 1SR. Also, we distinguish replica control from the maintenance of system internal consistency, termed divergence control. This distinction is analogous to the distinction between coherence control (replicas of a single "logical" object) and concurrency control (system internal consistency).

We therefore restate the ESR model in terms of replica control in a distributed system. First of all, we are only concerned with ETs in the system that carry information about replicated data. At each site, an ET is represented by a *message set* or *MSet*. Query ETs use query MSets to read the values of an object's copy. An update MSet is a set of replica maintenance operations which propagates updates to object replicas. That is, when an update is originated in a client node the results of the update are propagated to the replicas in MSets. Each local system is responsible for applying its MSet and preserving internal consistency.

Note that the propagation of MSets to each site is asynchronous. We assume the system maintains the unprocessed MSets in some stable storage, such as stable queues [5] and persistent pipes [16]. Each MSet is stored as an element in a stable queue. Due to the asynchronous propagation of MSets, replicas of a "logical" object can differ at any given moment. This is the source of inconsistency seen by the query ETs. A key observation, however, is that under ESR all replicas converge to the same 1SR value when the update MSets queued at individual sites are processed, and the system reaches a quiescent state.

Since a query ET may see the intermediate results of update ETs, it is clear that a query ET may yield a results some distance away from that of an SR query. One of the important points of ESR replica control is the ability to control the length of overlap (amount of inconsistency) in practice. At the one end of spectrum, replica control may allow zero inconsistency and no overlap, producing SR queries. At the other end of spectrum, replica control may let a query ET's error grow, by allowing a very long overlap – but ultimately the length of the overlap still bounds the query ET's error.

2.3 Asynchronous Replica Control

All replica control methods have to solve the following problem: how can operations be permitted to execute with the greatest possible concurrency and not interfere with one another. The problem is complicated because replica control methods must permit updates to proceed asynchronously. Each of the methods that we discuss restricts the set of potential concurrent executions in one way or another. The key point is that these methods, at the same time, *implicitly* specify ESR correctness criteria. In section 3 these methods are shown to satisfy the ESR requirements discussed in section

2.1 and 2.2. As a result, these methods maintain ESR correctness for ETs.

The advantages of using a replica control method that maintains ESR can best be seen in the following context. Picture a distributed system comprised of a collection of cooperating components. Each component maintains a set of objects that support sophisticated operations, such as read, write, increment, append a timestamped version, etc.

If the overall system is correct in terms of ESR, this implies that when all ongoing ET updates have been applied, the system converges to SR. ESR allows more concurrency than strict SR in two ways. First, query ETs may interleave other ETs. Second, update ETs may be serialized by divergence control methods that apply operation semantics. Furthermore, the amount of inconsistency in the query ETs is bounded by the number of concurrent update ETs with which they interleave.

The known replica control methods can be classified into two families. The first, termed "forward methods", prevents inconsistency by restricting some system parameter. For example, if update ETs contain only *commutative* operations, then the system supports ESR since update operations can be reordered into a serial schedule. Another example of replica control is *ordered updates*, which maintains the processing order of update ET operations. Ordered updates maintain system consistency, and query ETs can be processed in any order to increase concurrency. *Read-independent timestamped updates* also preserve ESR, because update ETs can be scheduled in any order, and query ETs are scheduled more or less freely depending on consistency and concurrency trade-offs.

The second family of replica control, termed "backward methods", is based on *compensation* operations. The replicated system may optimistically allow operations to proceed in parallel. If inconsistencies are detected later, then the system rolls them back with compensation operations or compensation ETs. Since update ETs are serializable with respect to other update ETs, ESR compensation is safe and carries low overhead.

In summary, the forward methods assume the updates have been "committed" and are being propagated through a reliable communication mechanism, while the backward methods supply some recovery mechanism in case the updates are "aborted". Table 1 lists some important characteristics of each of the replica control methods discussed in the paper. We have already discussed the issue of "forwards" *versus* "backwards" methods. The significance of the other dimensions will be discussed as each method is analyzed.

2.4 Framework for Replica Control

In [26] formal arguments are given for the assertion that divergence control methods analogous to these replica control methods maintain ESR. We are concerned here with the issue of *implementing* these methods in a replicated system. Traditional coherency control methods, such as weighted voting [14], update a number of replicas (e.g., write quorum) in an atomic transaction. Similarly, replica control methods apply the updates in an update ET. We say that a coherency control method is synchronous because a distributed transaction requires a commit agreement protocol to synchronize the transaction outcome. This is a big handicap when network links have very low bandwidth or moderately high latency. To solve this problem, replica control propagates updates independently; these methods are therefore asynchronous.

The first step in replica control is the generation of update MSets and their delivery to the replica sites. Each MSet is delivered asynchronously to its destination, and local sites execute the MSet independently of the processing of other MSets that update the same replica. At this stage, potential inconsistency arises because of the asynchronous updates, so we need to analyze the degree of divergence and control it.

Replica control depends on some additional restriction (in addition to that of SR) on the execution of MSets to control the divergence. This restriction may happen during the first "MSet delivery" step, which is the case of ordered updates (ORDUP), described in Section 3.1. Alternatively, the restriction may be on the kind of operations allowed, during the second "MSet processing" step, as in the case of both commutative operations method (COMMU), described in Section 3.2, and read-independent timestamped update method (RITU), described in Section 3.3. Finally, the restriction may be on the MSet processing as a whole, which is the case of replica control methods based on compensations (Section 3.4).

We note that each restriction is independent of the other restrictions. Concretely, ORDUP does not restrict the *kind* of operations in any way. Similarly, COMMU does not restrict the *ordering* of MSet execution, nor does RITU. However, the analysis of replica control combinations is beyond the

	ORDUP	COMMU	RITU	COMPENSATION
Kind of Restriction	message delivery	operation semantics	operation semantics	"operation value"
Applicability	Forwards	Forwards	Forwards	Backwards
Asynchronous Propagation	Query only	Query & Update	Query & Update	Query & Update
Sorting Time	at update	doesn't matter	at read	N/A

Table 1: Replica-Control Methods

scope of this paper.

For each replica control method, we will describe its steps, from "MSet delivery", to "MSet processing", ending with an analysis of the divergence allowed and an algorithm to limit the divergence on replicas ("Divergence bounding"). There are several ways divergence may arise and users may want to control each of them (see a discussion in Section 4.1). Here, we exhibit an existential proof of such techniques, without any aspiration for a complete coverage of the subject.

3 Replica Control Methods

3.1 ORDUP Replica Control

The idea behind the ORDUP replica control method is to execute the MSets by updating different replicas of the same object asynchronously–but in the same order. In this way the update ETs are SR. We can process query ETs in any order because they are allowed to see inconsistent results. It is easy to see that the log composed of ordered updates is ϵ-serial since the update ETs are SR.

MSet delivery: The client generating the MSets does not have to deliver them in order. In other words, a "later" MSet can be delivered before an "earlier" MSet. However, since their *execution* must be in order, the MSet must include information about its execution order. Each site simply waits for the next MSet in the execution sequence to show up before running other MSets. Although such ordering can be generated easily by a centralized order server, sometimes true distributed control is desired. In those cases we may use a Lamport-style global timestamp [19] to mark the ordering. In that case the MSets should somehow be delivered in order, since it is not easy to see whether there is another MSet coming in with just a slightly earlier timestamp.

MSet processing: Once the local system determines the next MSet, that MSet may be processed immediately. Note that the execution of MSets at each site may be locally interleaved, as long as the end result is an ESRlog. For example, the basic-timestamp (or other timestamp-based) concurrency control method applied to update ETs will produce an SRlog. Query ETs may interleave with the update ETs (and other query ETs) in any order.

Divergence bounding: The amount of inconsistency which a query ET may see is bounded by the length of its overlap with update ETs. If we allow arbitrary interleaving, a query ET can conceivably start at the beginning of log and finish at the end. Such a query might contain as much inconsistency as there are conflicts.

To limit the amount of inconsistency seen by query ETs, we use the divergence control of update ETs. The detailed algorithm depends on the particular global ordering algorithm adopted. The idea is to give each query ET its own global order number. If these are ordered the same way as the update ETs, then the overlap will be empty, yielding an SRlog. To control the degree of inconsistency of a query ET, we maintain an "inconsistency counter" for each. Each time a Q^{ET} is found to overlap an U^{ET} their inconsistency counters are incremented. When the inconsistency counter reaches the limit, no further overlap is allowed.

The detection of out-of-order execution depends on the particular divergence control method used for local operation ordering. In case of basic times-

	R_U	W_U	R_Q
R_U	OK	—	OK
W_U	—	—	OK
R_Q	OK	OK	OK

Table 2: 2PL Compatibility for ORDUP ETs

tamps, for example, each object maintains the timestamp of the latest access. The divergence control checks the ordering of each access. In an SR execution, out-of-order reads are either rejected or cause an abort of a write. In an ESR execution, the divergence control increments the inconsistency counter and decides whether to allow the read depending on the specified divergence limit.

Although a complete presentation of divergence control methods is beyond the scope of this paper, we present here an outline of how the standard two-phase locking (2PL) concurrency control algorithm would be modified. The standard 2PL lock compatibility table shows R/R compatible and the other cases (R/W, W/R, W/W) incompatible. Table 2 shows the resulting lock compatibility for ETs. R_U denotes a read lock by an update ET, W_U a write lock by an update ET, and R_Q denotes a read lock by a query ET.

3.2 COMMU Replica Control

The idea behind the COMMU replica control method is the use of operation semantics. If the final result is equivalent to some serial execution, then the actual execution order does not matter. In essence, we order updates at their completion time. Since query ETs are allowed to interleave update ETs, Q^{ET} reads become commutative with respect to the commutative U^{ET} writes. Total concurrency increases since we have eliminated a major bottleneck– the lack of commutativity between reads and updates.

MSet delivery: Since the MSets are commutative, there is no restriction on the ordering of messages delivered. We still need stable queues, however, since lost MSets cannot be recovered.

MSet processing: We assume that update operations on each object are commutative. If this is not the case, then care must be taken to preserve the serialization of non-commutative operations. Commutative update MSets can be processed asynchronously in any order. Since query MSets can be interleaved arbitrarily, they become commutative with respect to the updates and can therefore be processed asynchronously in any order as well.

Divergence bounding: The inconsistency that query ETs may see is derived from intermediate states between U^{ET} operations through overlap. There are no problems with overlapping update ETs, since their MSets are commutative. If there is no hard limit on query ET divergence, then the system can run freely. However, if a limit is placed on the degree of divergence of query ETs, their serialization may be affected by the interleaving of update ETs. For example, if all the update ETs on a log are conflicting and interleaved, then the only way to make query ETs SR is to put them at the beginning or at the end.

One way to limit the amount of inconsistency seen by query ETs is to put a lock-counter on each object being accessed. When updating an object, the U^{ET} increments the object lock-counter by one. The replica control keeps track of these lock-counters in the same way as it handles the usual locks held by transactions. At the end of U^{ET} execution all the lock-counters are decremented. We can control object access consistency by regulating the lock-counter usage. For example, we can allow update ETs to run freely. In this case the query ETs are responsible for determining their own inconsistency. Each lock-counter different from zero means a certain degree of inconsistency added to the query ET.

Alternatively, we can limit the update ETs in addition to query ETs. For example, if the lock-counter of an object exceeds a specified limit, then the update ET trying to write must either wait or abort. Query ETs still have to take into consideration the inconsistency shown in lock-counters, but the overlap of update ETs will be limited and query ETs have a better chance of completion without waiting due to inconsistency limitations.

The modification needed to use two-phase locking for ETs under COMMU changes Table 2 slightly. The squares for incompatible locks in Table 2 may be changed to allow additional commutative operations. For example, increment and decrement would change the W_U locks to be compatible. R_U locks remain compatible with R_Q, and possibly with the commutative W_U.

3.3 RITU Replica Control

The Read-Independent Timestamped Updates (RITU) replica control method also uses update operation semantics, but postpones access ordering to subsequent read time. If updates do not have R/W dependencies, they can be executed asynchronously. Of course, arbitrary updates may still have W/W dependencies, but RITU updates are commutative with respect to reads.

MSet delivery: Since RITU MSets are commutative with respect to themselves and reads, we can apply the results and methods from Section 3.2.

MSet processing: An RITU update MSet may generate a new immutable version (in append-only systems) or overwrite a previous version that has an older timestamp. (An RITU update trying to overwrite a newer version is ignored.) The read-independent overwrite is sometimes called a "blind-write".

Divergence bounding: In case of single-version overwrites, the system assumes that the latest version is the desired data. In these cases, there is no divergence since by definition all the reads request the latest version, and RITU reduces to COMMU.

The case of multiple versions assumes that each query should be synchronized at some fixed time (for all the reads). For simplicity of presentation, we use the Modular Synchronization Method [1] to maintain versions, which makes versions of objects visible to queries in such a way that no smaller version can be created by any active or future transactions. This visibility control, called a visible transaction number counter (VTNC), produces SR queries. Query ETs may read versions newer than VTNC, knowing that the newer value may introduce inconsistency. Each time a query ET reads such a version its "inconsistency counter" is increased by one. (This is a conservative approach since it is possible that the value will be committed and become valid.) The replica control limits the amount of inconsistency in a query ET by not allowing reading versions that are newer than VTNC, when its inconsistency counter has reached a specified limit.

3.4 COMPE Replica Control

MSet processing: The processing of MSet can be unconstrained, in which case we need to rollback the entire log. This is the case with ORDUP operations. On the other hand, if all MSets are commutative, then the system can simply apply the compensation without any overhead. The COMPE replica control method must remember the executed MSets until there is no risk of rollback.

Compensation MSet delivery: If the entire log needs to be rolled back (e.g., in case of ORDUP), COMPE runs the compensation MSet for each MSet in the log (up to the desired MSet) in reverse order. The log is then replayed, the MSets re-executed, and the system continues. In order to rollback RITU with overwrite we must also record the value being overwritten on the log. If all MSets on the log are commutative, then COMPE simply runs the compensation MSet and continues. This is the case with COMMU and RITU with multiple versions. Multiple versions can support compensation by deleting the version, or by adding another version with the same timestamp but bearing the previous value.

Divergence bounding: In compensation-based replica control, inconsistency is introduced because other ETs may see results that are rolled back later. By limiting the time between an ET commit and its rollback we limit the amount of potential inconsistency.

4 Related Work

4.1 Specification of Inconsistency

ETs provide an interface to ESR. ESR is a correctness criterion that allows bounded divergence. Replica control methods can enforce different divergence bounds. ESR is a general framework of controlled inconsistency [20]; this paper has applied it within the more limited domain of replica control. Other examples of specification methods have been proposed for this domain.

Wiederhold and Qian [24] have introduced the notation called *identity connection* to specify the constraints binding the replicas of an object. They classify the update propagation into four classes: immediate updates, deferred updates, independent updates, and potentially inconsistent updates. Although Wiederhold and Qian propose a temporal constraint resolver [25] to implement transaction processing satisfying these specifications, it would require considerable overhead to test their satisfiability. Replica control methods offer more efficient execution at the price of less concurrency.

Sheth and Rusinkiewicz [23] have proposed a taxonomy for interdependent data management. They separate data consistency criteria into temporal and spatial dimensions. The temporal consistency has two kinds: eventual consistency (analogous to identity connections), and lagging consistency (analogous to quasi-copies [2]). The spatial consistency criteria are divided into three cases. Inconsistency is controlled by limiting either (1) the number of data items changed asynchronously, (2) the data value changed asynchronously, or (3) the number of allowed asynchronous operations. An implementation of interdependent data management is described in [22]: it essentially corresponds to ORDUP. Replica control and divergence control can implement such criteria [26].

Another specification approach is Controlled Inconsistency proposed by Barbara and Garcia-Molina [3], which generalizes their work on quasi-copies. Controlled Inconsistency specifies arithmetic consistency constraints, similar to the data value limit in interdependent data management. They constrain updates to be *safe* operations, defined by some semantic correctness criteria appropriate to the application. In contrast, the replica control methods discussed in this paper are meant to be more application independent.

4.2 Read-only Redundancy

Replica control methods must deal with the divergence introduced by updates. Read operations, however, can access existing replicas: if updates are 1SR, then the system will remain consistent. Such *read-only redundancy* can be maintained in several ways, and is useful in many applications. ETs offer an additional benefit in situations such as time-critical applications where a 1SR update has not yet been propagated. The available values may then be too old to be useful.

An early example of read-only redundancy is timestamped versions [4]. Queries that are serialized in the "past" do not block, and immutable versions can be replicated freely.

Another early proposal using the idea of read-only redundancy is *weak consistency* [13] defined over the class of read-only transactions. A read-only transaction satisfies weak consistency if it is locally consistent, but may cause non-SR results in the global log formed by the union of local logs. ESR differs in two major ways from weak-consistency: first, it allows more interleaving (because query ETs are permitted to see local inconsistency), and, second, weak consistency does not incorporate methods that can tune the *degree* of inconsistency.

Quasi-copies [2] offers a theoretical foundation for increased read-only availability, but require that all updates be 1SR. As a result, the primary copy is always consistent in the 1SR sense. Inconsistency is only introduced because quasi-copies may lag the primary copy. One similarity between quasi-copies and ESR is the notion of specifying the relationship between replicas. Quasi-copies uses a "closeness" specification in the trigger mechanism which propagates updates to quasi-copies. Replica control methods, in contrast, constrain the degree of inconsistency of ETs directly.

4.3 Network Partition Merging

Communication failures cause divergence between object copies in different partitions. Davidson et al [9] have surveyed the approaches to data replication under network partitions. They divide the approaches into two groups: pessimistic and optimistic. Pessimistic algorithms are synchronous, since they use commit protocols to maintain replica mutual-consistency. Optimistic algorithms allow updates to proceed asynchronously, but try to merge the operations at partition reconnection time. Optimistic techniques are also "off-line": repairs are based on merging logs from the different partitions. The replica control methods used by ETs differ principally in that they allow controlled divergence while queries and updates are in progress. That is, instead of processing logs at reconnection time, our methods control divergence dynamically.

An early example of such off-line algorithm work is Faissol's thesis [12]. He identifies 5 classes of methods: they can be roughly equated to the methods discussed in this paper. Class A is similar to RITU overwrite; classes B and C are similar to COMMU; and classes D and E are similar to COMPE.

Another example, log transformation [8] is a method proposed to speed up the merging of updates from different partitions when they reconnect. They use operation properties such as commutativity and overwrite to merge independent updates. If some updates cannot be merged then they try backward recovery by rolling back some updates and redoing them. This is an off-line algorithm that may be useful when logs grow long in prolonged partitions.

More recent work has also focussed on optimizing the work needed for log-merging. One example of such a system is OSCAR [11]. Their architecture is based on two cooperating agents, called replicators and mediators. The replicators propagate replica updates and the mediators recover from replicator failures. Three methods in OSCAR can be used by the appropriate application using the so-called weak-consistency updates: commutative and associative, overwrite, and site-sequential. ETs are a high-level interface to such methods; in addition, replica control methods give users the ability to control the amount of inconsistency in the system.

4.4 Services for Asynchronous Updates

The idea of applying updates asynchronously is not new, and specific applications and primitives for their implementation have been proposed in the literature. ESR differs principally by (1) providing a high-level and general (ET) interface and by (2) allowing fine-grained control over inconsistency through replica control methods.

Grapevine [7] is an early example of directory systems developed at Xerox. It propagated the changes between replicas periodically. Its successor, Clearinghouse, has adopted a randomized algorithm [10] for update propagation. They show that random propagation converges quickly, but needs a reliable mechanism to recover from rare failures.

Ficus distributed file system [15] uses a two-phase flooding algorithm to detect replica propagation stability and conflicts asynchronously. However, they neither prevent nor resolve conflicts.

Coda distributed file system [21] provides optimistic replication through a manual repair tool [17]. The tool supports semantics-dependent rules for automatic recovery of specific applications such as directories. No general rules are supplied.

In addition to directory/file-system propagation, asynchronous communication techniques have also been developed. An example of the latter is Lazy Replication [18]. It provides three classes of ordered messages, plus unordered delivery. This facility is at a much lower level of abstraction than transactions and ETs. Another example is Isis [6], which provides four kinds of multicast and broadcast facilities, including causal broadcasts. These communications facilities have been used to provide replication in a way similar to ORDUP.

5 Conclusion

Asynchronous updates have the potential of increasing performance and availability while allowing autonomy. The major problem has been the trade-off between asynchrony and consistency among the replicas. In this paper we describe an approach that guarantees replica convergence while at the same time taking advantage of asynchronous update propagation.

Updates to replicas are done via epsilon-transactions (ETs), which ensure the epsilon-serializability (ESR) correctness criterion. Update ETs remain serializable as in the standard (1-copy serializable) model and the system therefore remains consistent in a strong sense. Query ETs, however, can interleave with other ETs freely, thus seeing inconsistent values of replicas. The inconsistency visible to query ETs is bounded, and can be controlled by the user.

Just as coherency control methods ensure 1SR, there are replica control methods that ensure ESR. Practical replica control methods work by imposing restrictions on the set of concurrent executions. Several methods, such as one based on commutative operations and one based on compensations are described and analyzed in this paper. These methods enable applications to using ETs to omit explicit reference to ESR. The analysis of the replica divergence permitted under these methods shows that, in the limit, the divergence of ETs can be reduced to zero, making all ETs strictly serializable.

References

[1] D. Agrawal and S. Sengupta. Modular synchronization in multiversion databases: Version control and concurrency control. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, Portland, 1989. ACM/SIGMOD.

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

[3] D. Barbara and H. Garcia-Molina. The case for controlled inconsistency in replicated data. In *Proceedings of the Workshop on Management of Replicated Data*, pages 35–42, Houston, November 1990.

[4] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, first edition, 1987.

[5] P.A. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. In *Proceedings of 1990 SIGMOD International Conference on Management of Data*, pages 112–122, May 1990.

[6] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the*

Eleventh Symposium on Operating Systems Principles, pages 123–138. ACM/SIGOPS, November 1987.

[7] A.D. Birrell, R. Levin, R.M. Needham, and M.D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of ACM*, 25(4):260–274, April 1982.

[8] B.T. Blaustein and C.W. Kaufman. Updating replicated data during communication failures. In *Proceedings of the Eleventh International Conference on Very Large Data Bases*, pages 49–58, Stockholm, August 1985.

[9] S.B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[10] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, August 1987.

[11] A.R. Downing, I.B. Greenberg, and J.M. Peha. OSCAR: An architecture for weak-consistency replication. In *Proceedings of PARBASE-90 International Conference on Databases, Parallel Architectures, and Their Applications*, pages 350–358, 1990.

[12] S.Z. Faissol. *Operation of Distributed Database Systems Under Network Partitions*. PhD thesis, Department of Computer Science, University of California, Los Angeles, 1981.

[13] H. Garcia-Molina and G. Wiederhold. Read-only transactions in a distributed database. *ACM Transactions on Database Systems*, 7(2):209–234, June 1982.

[14] D.K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 150–162. ACM/SIGOPS, December 1979.

[15] R.G. Guy, J.S. Heidemann, W. Mak, T.W. Page, G.J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. In *Proceedings of 1990 Usenix Summer Conference*, pages 63–71, Anaheim, CA, June 1990. Usenix.

[16] M. Hsu and A. Silberschatz. Persistent transmission and unilateral commit. In *Proceedings of Workshop on Muldatabases and Semantic Interoperability*, pages 48–52, November 1990.

[17] P. Kumar. Coping with conflicts in an optimistically replicated file system. In *Proceedings of the Workshop on the Management of Replicated Data*, pages 60–64, Houston, November 1990. IEEE/TCOS.

[18] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: Exploiting the semantics of distributed services. In *Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing*, Quebec City, August 1990. ACM/SIGACT-SIGOPS.

[19] L. Lamport. Time, clocks and ordering of events in a distributed system. *Communications of ACM*, 21(7):558–565, July 1978.

[20] C. Pu and A. Leff. Epsilon-serializability. Technical Report CUCS-054-90, Department of Computer Science, Columbia University, December 1990. An extended abstract was submitted to PODS.

[21] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, C-39(4):447–459, April 1990.

[22] A. Sheth and P. Krishnamurthy. Redundant data management in Bellcore and BCC databases. Technical Report TM-STS-015011/1, Bell Communications Research, December 1989.

[23] A. Sheth and M. Rusinkiewicz. Management of interdependent data: Specifying dependency and consistency requirements. In *Proceedings of the Workshop on Management of Replicated Data*, pages 133–136, Houston, November 1990.

[24] G. Wiederhold and X. Qian. Modeling asynchrony in distributed databases. In *Proceedings of the Third International Conference on Data Engineering*, pages 246–250, February 1987.

[25] G. Wiederhold and X. Qian. Consistency control of replicated data in federated databases. In *Proceedings of the Workshop on Management of Replicated Data*, pages 130–132, Houston, November 1990.

[26] K.L. Wu, P. S. Yu, and C. Pu. Divergence control for epsilon-serializability. Technical Report CUCS-002-91, Department of Computer Science, Columbia University, February 1991. Also available as IBM Tech Report No. RC16598.