Execution Autonomy in Distributed Transaction Processing

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

Technical Report No. CUCS-024-91

Abstract

We study the feasibility of execution autonomy in systems with asynchronous transaction processing based on epsilon-serializability (ESR). The abstract correctness criteria defined by ESR are implemented by techniques such as asynchronous divergence control and asynchronous consistency restoration. Concrete application examples in a distributed environment, such as banking, are described in order to illustrate the advantages of using ESR to support execution autonomy.

Index terms: epsilon-serializability, autonomy, asynchronous transaction processing, divergence control, consistency restoration.

Contents

1 Introduction

A key assumption made by techniques that provide global recovery and concurrency atomicity distributed databases such as R* [19], is that system components want to *cooperate* in providing global transaction properties. However, there are many areas (e.g., performance and availability) where participation in a classic distributed transaction will infringe on a local transaction system's capabilities. Because of this concern, much on-going effort (e.g., [5, 6, 14, 16, 17, 18, 26, 27]) is devoted to providing global transaction capabilities, while at the same time allowing local systems to maintain individual autonomy. In addition to motivations such as heterogeneity [21], autonomy *per se* is increasingly important because of the change in the cost/benefit ratio of intersite cooperation in large systems. For example, two-phase commit becomes less desirable as the diameter of the network increases, since both the response time and the probability of failure become higher [16]. Also, two-phase commit may impose unnecessary synchronization and prevent independent local commit [21, 14]. Supporting autonomous operations can potentially improve both performance and availability.

Before discussing the issues posed by "autonomy" we note that the definition itself is in a state of flux [21]. Some researchers distinguish between "design autonomy" and "execution autonomy". Design autonomy involves the question of who controls the design and maintenance of data schema. Although design autonomy poses many difficult problems, we will not pursue it here.

The focus of this paper is on execution autonomy because of its dramatic impact on transaction processing (TP) systems. In this context, autonomy allows local systems to control local transaction execution without being forced to accede to a global transaction coordinator in the scheduling and execution of transactions. Execution autonomy in effect calls the entire notion of distributed TP into question because participating sites are no longer willing to sacrifice independence simply in order to implement a globally serializable transaction schedule. Instead of the many political and administrative issues posed by autonomy, we discuss *mechanisms* that support autonomous TP. Furthermore, we are only interested in mechanisms that support a wide range of autonomy *policies*. Mechanisms that impose a particular policy are a premature optimization and too inflexible in a large distributed system. Thus, the issues of administrative and design autonomy will become relevant *after* suitable and general mechanisms supporting autonomy have been developed.

If sites are not willing to surrender control to a global transaction coordinator because of the performance penalties which arise in large and highly replicated systems, they will not participate in algorithms that maintain the traditionally *synchronous* notions of transactions. Sites lose much availability if they must coordinate updates with many other sites in the system; they may then be willing to trade reduced consistency for increased availability. We believe that one direction for the evolution of autonomous TP systems is towards *asynchronous* TP. Asynchrony clearly avoids the problems that were just discussed for autonomous systems. On the other hand, a basic problem in any kind of asynchronous processing is the existence of inconsistency while components have not synchronized. Since a major reason for using transactions in the first place is consistency maintenance, we need to be able to specify rigorous consistency criteria that can be maintained even with asynchronous TP. We discuss such a framework, termed *epsilon-serializability*, in Section 3.

In Section 2 we motivate this approach by showing example applications that are limited by both classic TP methods and some previously proposed asynchronous techniques. Section 3 then summarizes the properties of and interface to epsilon-serializability. Some practical algorithms for the implementation of ESR are outlined in Sections 4 and 5. Section 6 revisits the motivating applications, and uses ESR to solve them. Some related work is summarized in Section 7; some conclusions are offered in Section 8.

2 Motivating Applications

In this section, we discuss the limitations of classic TP as well as the limitations of some particular solutions previously proposed for some of these problems. Several shortcomings of classic TP systems hinder the programmer of distributed and autonomous TP applications due to the lack of asynchronous TP support to enable a site to execute autonomously. Some limitations of other proposed solutions are:

- the lack of smooth integration of asynchronous TP execution with classic TP execution,
- the lack of consistency guarantees that a transaction can rely on to maintain the desired trade-off between consistency and performance,
- the lack of a transparent specification, and treatment, of inconsistency-related uncertainty so that applications can be written independently of particular implementation techniques.

2.1 Autonomous TP

For historical reasons, and also for high availability of computer hardware, most airline reservation systems are centralized mainframe TP systems. Given that most of flight bookings are made at the end points of each flight, both performance (e.g., response time) and availability of network communications (e.g., temporary network partitions) may improve if a distributed TP system is used instead. For example, a network of smaller and cheaper machines could handle flights departing from a single city. In such a distributed airline reservation system, autonomous execution is important: we do not want to sacrifice local performance due to remote requests (e.g., locking a flight record at the busy check-in time), nor do we want to compromise local availability due to remote crashes (e.g., being unable to book Chicago because of a hurricane in Florida). (The issue of replication in asynchronous TP is discussed in [24]).

Normally, when the distributed TP system is running well, no-shows are sufficient protection for inconsistent data (application semantics exploited by the practice of overbooking). Since the system does not have to be absolutely consistent, high performance is achieved by free access to the database. Since most of flights are not full, this strategy optimizes for the normal case. However, as the flight becomes full, an airline would like to be more careful since the penalty for passengers with reservation left on the ground is great (monetary compensations and/or alternative travel arrangements). Avoiding overflows is very important to airlines, both to reduce liabilities and to improve the company's image as being responsible and competent. Therefore, as a particular flight is getting full, the application needs to become conservative in maintaining database consistency.

Another classic TP application is banking. Consider a large bank that has been acquiring a number of smaller banks in different states. Due to differences in legislation, it is advantageous to maintain the original computing systems of each acquired bank (the "sites"). Each site is assumed to run a classic TP system. However, the parent bank also wants to offer integrated TP service in order to to attract corporate customers that do business in several states. In this example, electronic funds transfer service must maintain serializable consistency so money would not be lost. However, the parent bank's investment services may want to obtain an approximate picture of the entire group's assets and investments each day. For example, the asset report may be rounded to the nearest thousand dollars for items less than a million, and rounded to the nearest million for items above a million. In this case, consistency requirements are relaxed since only the “big picture” is being sought. For example, for a report in which units are in millions of dollars, an inconsistency bounded by 100,000 will add only a maximum of 20% to the expected rounding error.

2.2 Service Guarantees

In time-critical applications such as the stock market, the ability to provide service-level guarantees may be important. Current requirements specify that response-time should be as fast as possible, with little concern for consistency. This works well during normal times, when the time lag is very short or zero so that results tend to be consistent. However, during a crisis, when the quality of data becomes important, the no-consistency requirement allows the system to return essentially arbitrary data.

Although the above example is also valid for a centralized system, consider a distributed system linking the New York Stock Exchange (NYSE) and the Chicago Stock Exchange. Under normal conditions, a local broker would have a minimal advantage over a remote competitor, since the transmission and execution delay between New York and Chicago is measured in seconds. However, if volume climbs and the system slows down, inconsistency differences can become quite large. For example, if NYSE slows down, then a Chicago broker may be getting increasingly less accurate information compared to a New York rival. It would be useful for her to know the exact “age” of each piece of information in order to make correct buy/sell decisions.

Consistency guarantees are similarly important to other kinds of service contracts between computer vendors and customers. A guarantee in performance, availability, and in this case, consistency, typically increases the value of the system because of the added dependability. A guaranteed service has a competitive advantage over one that does not. However, because such guarantees involve financial penalty in case of failure, an efficient way to offer such guarantees is important.

2.3 Integration with Classic TP

Although the airline reservation system may not need rigorous consistency in the sense of serializability (SR), a bank needs it for certain kinds of transactions such as electronic funds transfer. The same distributed database must then support a spectrum of “degrees of inconsistency” simultaneously. At one end of the spectrum is rigorous SR consistency. At the other end are varied trade-offs (for reasons of performance, availability, and autonomy), between consistency and asynchrony. Indeed, many autonomous TP systems will need to move along the spectrum as the circumstances change. For example, the airline reservation system and stock market would emphasize improved response time during normal times when inconsistency is minimal anyway, but emphasize consistency when few seats are left because the penalty for inconsistency is then high. Conversely, in banking or military operations, high consistency is maintained during normal operation, but in a crisis limited inconsistency is permitted so as to achieve operational availability and autonomy. In such cases, the system must repair inconsistency when the crisis is over.

Gray’s different degrees of consistency [11], offers an example of such a spectrum. Degree 3 consistency is equivalent to SR, but degree 2 consistency offers higher concurrency for queries— at the cost of reduced consistency— since updates are allowed to “dirty” data already read by queries. Degree 2 is reportedly used at many DB2 installations, underscoring the importance of integrating inconsistency specifications. However, there are two limitations in this approach. First, degree 2 is peculiar to a particular concurrency control algorithm, namely two-phase locking. This limits database interoperability, since other concurrency control algorithms cannot provide the same level of consistency. Second, because no bounds are set on the total amount of inconsistency, degree 2 queries will become less accurate as a system grows larger . This implies that service guarantees cannot be offered as a system scales up.

Quasi-serializability (QSR) has been proposed [6] as an abstract correctness criterion for a multidatabase environment. QSR specifies that local databases and global schedulers should maintain SR, but isolates a global scheduler from the local schedulers. This way, the local histories and the global histories considered in isolation are SR, even though their combination may not be. QSR is well-defined and easy to implement. However, its applicability is limited in the trade-off between consistency and performance. The number of non-SR histories allowed by QSR is limited by its global serializability requirement. At the same time, unbounded inconsistency may be found when we consider the global history and the local histories together.

2.4 Network Partitions

Another important problem in the synchronous execution of distributed TP is that when the communications network partitions, transactions tend to *block*; i.e., they have to wait for the other partition(s) to reconnect. The information required for distributed TP includes the transaction outcome decision (commit or abort) as well as the serialization order. Autonomous execution implies that blocking would be reduced.

Data-value Partitioning [27] has been proposed as a method to for increasing distributed TP system availability and autonomy by explicitly separating parts of the value of a data item into different sites. Since the different parts may operate asynchronously even during network partitions, Data-value Partitioning increases autonomy because of its non-blocking character. The basic idea is to allow more parallel processing by dividing the data item value. While this division increases concurrency and avoids blocking, it also introduces some uncertainty to the value of the data item as a whole. The problem is that if several updates on a data item are pending, some of them may commit and others abort, so that the definite value of the whole data item will be known only at the completion time of the updates. This makes reading a data value non-trivial, since, in principle, the uncertainty can be arbitrarily large. In practice, the uncertainty is limited by the number of partitions in Data-value Partitioning, assuming that each update has a ceiling. Leaving the handling of the uncertainty to application programmers may then be an adequate solution in a system of moderate size. However, as TP systems grow faster and larger, the uncertainty increases, making uncertainty control more difficult.

3 Epsilon-Serializability (ESR)

Epsilon-serializability is an abstract framework that offers a solution to the motivating applications described in Section 2. In the next three sections we first describe the properties of ESR in a distributed environment, and then discuss implementation techniques. In Section 6 we revisit the applications and use ESR to solve them.

The general idea of ESR is to preserve consistency, but, at the same time, allow some bounded slack. When this (in)consistency slack converges to zero, ESR reduces to serializability (SR). In addition, although ESR allows inconsistent data to be seen, it requires that data eventually converge to a consistent (SR) state. 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. This control is at a very fine granularity since each transaction may specify its own limit. An autonomous TP system should tolerate temporary inconsistency during autonomous operation and repair it later when the system returns to a cooperative mode. ESR-based TP systems permit limited inconsistency to arise, and repair it through two techniques— asynchronous divergence control (Section 4) and asynchronous consistency restoration (Section 5).

3.1 Standard SR Model

A database is a set of data items, which support Read and Write operations. Read operations do not change data and Write operations do. (Section 5.2 will introduce additional semantics to database operations.) A transaction is a sequence of operations that take a database from a consistent state to another. Transactions may be *updates* that contain at least one Write or *queries* that are read-only.

Our terminology follows the standard model of conflict-based serializability [2]. Two operations are said to conflict if at least one of them is a Write, so we have read-write (R/W) and write-write (W/W) conflicts. Each pair of conflicting operations establishes a *dependency*. A *history* is a sequence of operations such as Reads and Writes. A *serial* history is a sequence of operations composed of consecutive transactions. A history of transaction operations is said to be serializable (SR-history) if it produces results equivalent to some serial history, 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 SR-histories are allowed.

Intuitively, in the standard model, a history is shown to be an SR-history by rearranging its operations according to certain constraints imposed by R/W and W/W dependencies. The rules of rearrangement are given by concrete concurrency control methods. A more formal way to specify concurrency control uses a serialization graph (SG), in which each arc represents the *precede* relation [2]. Transaction T_1 precedes T_2 when one of T_1's operations precedes and conflicts (R/W or W/W) with T_2's operations. Since the Serializability Theorem [2] says that a history H is SR if and only if its serialization graph SG(H) is acyclic, an acyclic SG implies an SR-history.

3.2 Epsilon-Transaction (ET)

An epsilon-transaction, denoted by ET, is a sequence of operations that maintains database consistency when executed atomically. However, an ET differs from standard transactions (denoted by T), in that an ET also includes a specification of the amount of inconsistency permitted the ET. This per-ET limit of allowed inconsistency is called an ϵ-specification, (ϵ-spec for short); the specification must be met— despite the concurrent execution of other ETs.

Abstractly, an ϵ-spec is divided into two parts: a limit on *imported* inconsistency and a limit on *exported* inconsistency. Of course, many kinds of inconsistency can exist. An ϵ-spec can therefore take several forms, as long as it is defined using a monotonic distance metric [28]. In the airline example, the number of seats is a linear distance metric (therefore monotonic). Each ET includes two parameters: (1) *ImpLimit* denotes the maximum number of seats in non-SR conflicts that the ET can import from other ETs, and (2) *ExpLimit* denotes the maximum number seats in non-SR conflicts that the ET can export to other ETs. For simplicity of presentation, our ImpLimit/ExpLimit ϵ-spec lumps together all sources of inconsistency. In this paper, we use a concrete example for illustration. Our example is the number of seats in an airline reservation system. This distance metric, as well as other examples such as banking, is linear. The ADC methods can be generalized to other inconsistency specifications using techniques described in [28].

One important observation is that when ImpLimit = 0 and ExpLimit = 0, the ET is serializable, so ETs include the standard Ts as a boundary case. We continue to use T as a short

	ImpLimit = 0	ImpLimit > 0
ExpLimit = 0	Transaction	Q^{ET}
ExpLimit > 0	U^{ET}	G^{ET}

Table 1: Four Kinds of ETs

notation for ETs that are serializable. Usually, when we refer to an ET we mean one of the other kinds of ETs shown in Table 1: query ETs denoted by Q^{ET}, consistent update ETs denoted by U^{ET}, and general ETs denoted by G^{ET}. Intuitively, when ImpLimit > 0 and ExpLimit $= 0$, (the query) ETs may import some inconsistency, up to ImpLimit. When ImpLimit $= 0$ and ExpLimit > 0, (the consistent update) ETs may export some inconsistency, up to ExpLimit. When ImpLimit > 0 and ExpLimit > 0, the G^{ET} may both import and export inconsistency. In this case the inconsistency in the database may grow unboundedly. This is the problem addressed by consistency restoration methods.

A history containing query ET and update ET operations is called an ϵ-serial history if the difference between it and a serial history is less or equal than a specified ϵ-spec. For example, a 0-serial history is a serial history as defined in Section 3.1. But any arbitrary history is an ∞-serial history, since infinite amount of inconsistency is allowed. In our airline example, an ϵ-serial history for a finite ϵ-spec is a history in which some ETs have their operations interleaved. The interleaving is such that for every ET, the number of seats involved in the interleaved operations add up to less than its ϵ-spec.

An ESR-history is a history that produces results equivalent to an ϵ-serial history defined by the ϵ-spec of the set of ETs in the history. A database that executes ETs and produces an ESR-history is said to support epsilon-serializability (ESR). Note that the same way several SR-histories may exist for each serial history we have many ESR-histories for each ϵ-serial history. In this paper, we are not concerned about the existence of an "optimal" ϵ-serial history for a given set of ETs or a "minimal" ESR-history for a given ϵ-serial history. As long as we can rearrange the operations in a history to produce an ϵ-serial history, it is an ESR-history.

Consider two ETs in the banking example. An update U_1^{ET}(ImpLimit= 0 and ExpLimit= 400) is defined by the sequence of operations R_1(a) R_1(b) W_1(a) W_1(b), and a query Q_2^{ET}(ImpLimit= 2000 and ExpLimit= 0) defined by the sequence of operations Q_2(a) Q_2(b). In our notation, R_1(a) means a Read operation in U_1^{ET} on a, W_1(a) denotes a Write operation in U_1^{ET} on a, and Q_2(a) means a read operation in Q_2^{ET} on a. We further assume that W_1(a) changes a by 100 and W_1(b) changes b by 200 (this can be calculated by the query compiler, a programmer declaration, or at the lowest level, the raw amount of update).

A simple example of ESR-history is:

$$R_1(a)R_1(b)W_1(a)Q_2(a)Q_2(b)W_1(b). \tag{1}$$

History (1) is an ESR-history (in fact, an ϵ-serial history). It is easy to see that Q_2^{ET} precedes U_1^{ET} because of a R/W conflict on b and U_1^{ET} precedes Q_2^{ET} because of another R/W conflict on a. Thus the sub-history formed by U_1^{ET} and Q_2^{ET} is not serializable. However, the non-SR interleavings result in a difference of 200 from an SR-history, which is within the specified limits for both U_1^{ET} and Q_2^{ET}, so history (1) qualifies as an ESR-history.

3.3 Distributed ETs

In a traditional TP system such as R* [19], a distributed transaction is subdivided into subtransactions, each executing at one site. We adopt a similar model. When a distributed ET *(dET)* is submitted, it is divided into sub-ETs, called *asynchronous* sub-ETs (asET) for emphasis. Then the asETs are distributed from the originator site to the execution sites, one asET per site. At completion of an asET, the executing site returns the results to the originator site for assembly and delivery to the user.

Our model of a distributed TP system extends classic TP systems. The execution of dETs is handled by asynchronous divergence control *(ADC)* and asynchronous consistency restoration *(ACR)* methods. These are analogous to distributed concurrency control and crash recovery methods in classic distributed TP. The role of asynchronous divergence control is to *prevent* inconsistency from being introduced into the database; asynchronous consistency restoration *repairs* the global database once some inconsistency is detected. Our goal is to maximize database consistency, either through serializable transactions or ETs that limit the amount of inconsistency. This is in contrast to multidatabases [20] in which global inconsistency is tolerated.

Because our focus in this paper is on the *interaction* between sites, we assume that each site is capable of handling its own asETs. In fact, to avoid complications unrelated to the autonomy issue, we assume that each asET is a serializable transaction— i.e., there is no asynchronous execution within each site. The incorporation of other kinds of ETs as asETs (Q^{ET}, U^{ET}, and G^{ET}) is straightforward but non-trivial. Since our focus is on the useful properties of ESR for autonomous TP systems, we omit implementation and usage issues such as the distribution and negotiation of ϵ-spec among sites. This simplification brings ESR closer to QSR [6], since both assume local databases to maintain SR. However, even this simple case of ESR allows more concurrency than QSR, since QSR insists that global history to be SR when considered by itself.

4 Asynchronous Divergence Control (ADC)

Divergence control methods for ESR have similar goals to those of concurrency control methods for SR. We want efficient algorithms that guarantee some correctness criteria. One important observation is that a *global* property such as serializability will always require some global validation, since the union of locally serializable asETs do not make a globally serializable dET. In contrast, *local* properties such as linearizability [13] do not need global validation.

ESR spans the spectrum between global and local properties. On the one hand, ESR is compatible with SR, so that ESR is a global property when $\epsilon = 0$. On the other hand, ESR is a local property when $\epsilon = \infty$. (QSR is also a local property.) In other words, as the ϵ-spec increases, ESR becomes increasingly local. In this paper, we are primarily concerned with applications that use relatively tight bounds for their ϵ-spec. In the airline example, the bound is a small number of seats. As a result, the divergence control methods described here have a strong global flavor; i.e., they resemble classic concurrency control methods.

4.1 Designing ADC Methods

An important property of ESR is that it allows the amount of inconsistency (the ϵ-spec) that a query ET may see to be bounded rigorously. The type of inconsistency is expressed in a user specification. For example, in the banking context, an ϵ-spec is formulated in dollars. The job of an ADC method is to keep the amount of inconsistency in the dET calculating total checking deposits to within the ϵ-spec, despite any number of concurrent updates.

The design of ADC methods follows a two-stage methodology: extension and relaxation. In the first stage, existing concurrency control methods are extended, by identifying the places where the CC methods detect non-SR, conflicting operations. In the second stage, the extension is relaxed to allow for more concurrency. The underlying idea is that (conflict-based) concurrency control methods must be able to identify the non-SR conflicts and prevent a cycle from forming in the Serialization Graph. The extension stage isolates the identification part of concurrency control and the relaxation stage modifies the cycle prevention part so as to permit specific inconsistencies. The exact modification varies for each concurrency control method. For example, in a previous paper [28] we have extended two-phase locking (2PL), timestamp-based, and optimistic validation methods. For brevity we omit the details here.

In our banking example, a general implementation strategy of the relaxation stage for the ImpLimit and ExpLimit specifications (Section 3.2) could be based on two *inconsistency counters*, associated with each ET: ImpCount and ExpCount. ImpCount records the total amount in dollars of the non-SR conflicts that the ET has imported so far, and ExpCount maintains the total amount in dollars of the non-SR conflicts that the ET has exported so far. To bound the inconsistency according to the ϵ-spec, the ADC methods ensure for each ET that ImpCount $\leq$ ImpLimit and ExpCount $\leq$ ExpLimit. Since the ADC methods do not allow permanent inconsistency to be introduced into the database, an ET cannot have both ImpLimit> 0 and ExpLimit> 0. In other words, only Q^{ET} and U^{ET} can execute under ADC, not G^{ET}.

4.2 Distributed ADC Methods

The first case we consider is a distributed TP system where every site supports strict 2PL, i.e., they maintain the locks until the dET commits (or aborts). However, strict 2PL is a severe infringement of execution autonomy, since the release of local locks depends on the dET commit, which is a global event following a commit protocol. In other words, strict 2PL is a mechanism that inherently imposes a synchronous policy for concurrency control.

To avoid the problems of strict 2PL, one could adopt general 2PL (the problem of cascaded aborts is beyond the scope of this paper). In this case, each site may release locks independently, but the lock points of asETs at different sites may not be synchronized. Consequently, a global validation is necessary for checking the uniformity in ordering of asETs at each site. This checking would provide the amount of inconsistency introduced into the execution of a dET. Section 4.3 describes such a validation mechanism for optimistic validation.

A third alternative is all sites using basic timestamp ordering. We have two choices here. First, we could impose a global ordering (unique per dET) to the execution of an asET at each site. If we were to enforce SR (the same ordering at each site), this would be similar to strict 2PL and thus a violation of execution autonomy, since the sites must follow the ordering dictated by some central authority. Alternatively, if each site is allowed to establish the ordering of its own asETs, the situation is similar to general 2PL and we also need a validation mechanism (Section 4.3).

Finally, if all sites use optimistic concurrency control, a global validation phase is needed at the end of a dET. However, instead of aborting a dET at the detection of its first non-SR conflict, we relax the validation algorithm. In our banking example, when a non-SR conflict is detected, we sum up the update values into the ImpCount and ExpCount of the dETs involved. A dET is aborted only if one of its counters exceeds its ϵ-spec. Otherwise, it is allowed to proceed. The following section describes the validation with slightly more detail.

4.3 Heterogeneous ADC Methods

We note that the same motivations for execution autonomy of dETs also encourage the existence of heterogeneous local concurrency control mechanisms at each site. We consider here the three most popular concurrency control methods at the sites: 2PL, timestamps, and optimistic validation.

Each site is allowed to execute its asET independently. Each site is assumed to guarantee SR, and returns the ordering of each asET in its own context when the asET finishes its execution. This ordering may be sent back by the local concurrency control in an O-vector [25] or by the asET itself in a ticket [10]. In addition to the ordering, each update asET also provides the total value of the update.

When all the asETs of a dET have returned, the validation algorithm checks for non-SR orderings between different sites. If all the sites have serialized the asETs the same way, then the dET is SR. However, if the orderings are different, then the validation algorithm accumulates the update value in the ImpCount and ExpCount of the involved dETs. If the final counters are below the ϵ-spec value, the dET is allowed to commit. Otherwise it is aborted. Further details of the validation algorithm and its variants for other kinds of inconsistency specifications are topics of active research.

5 Asynchronous Consistency Restoration (ACR)

ADC methods preserve database consistency, since the database converges to SR when all the asETs arrive and are processed. However, the situation becomes more complicated with the introduction of G^{ET}, in which both ImpLimit> 0 and ExpLimit> 0. Inconsistency can now be introduced permanently, i.e., the database may be inconsistent in the SR sense even when quiescent. Database values may completely degenerate because of inconsistency if G^{ET} are executed without control. To restore consistency into the database we use ACR methods. Just as ADC methods reduce to classic concurrency control when $\epsilon \rightarrow 0$, ACR methods reduce to classic crash recovery under certain conditions.

5.1 Limitations of Classic Recovery

In classic TP, the main-memory view of the database always reflects the correct serialization order. Recovery algorithms are needed because we wish to increase system performance by decoupling the main memory processing from slow disk I/O operations. Once information on disk may differ from the main memory, we need to maintain enough information for resynchronization at recovery time. Remember that in classic TP, we have Read/Write operations without sophisticated semantics.

The model of inconsistency repair based on Read/Write compensations using REDO and UNDO [12] consists of three steps. First is the inconsistency detection where a specific operation is found to have introduced inconsistency. Second is the undoing of the effects of the offending operation. Third is the REDO of the other operations that have been undone as a side-effect during UNDO. In the worst case, the entire history needs to be undone up to the offending operation, and then redo everything else on the history. To illustrate the compensation overhead, consider an inventory item value decremented by \$200 (its compensation increments \$200). Suppose that it was an error and we wish to compensate for it after some processing. If another operation has multiplied the inventory value by a factor of 10 in the mean time, then we must first divide the value by 10, increment to compensate, and multiply by 10 again. This is necessary because the Read and Write operations do not commute in general (neither do multiplication and addition/subtraction).

In this paper, we are concerned only with asETs, which may contain just one operation or several, so our ACR methods do not distinguish between the two levels. If an ACR method does not rely on operation semantics such as commutativity and $\epsilon = 0$, then it reduces to applying all the compensation asETs in the reverse order up to the asET causing inconsistency. After the offender has been undone, the ACR method replays the history from that point to redo. When $\epsilon > 0$, the need for roll back is lessened by an amount roughly proportional to the ϵ-spec. This is explained in more detail in Section 5.3 using commutativity.

For operation-level compensation such as Time Warp [15] and classic TP crash recovery, the overhead of undoing the entire history is accepted as inevitable for the Read/Write model. The transaction-level equivalent of undoing the entire history is called *cascaded aborts*. However, if we take into account higher levels of abstraction for the operations (e.g., increment and decrement), compensation actions may be used to roll back sophisticated updates more cheaply. This is especially relevant at the transaction level. Sagas [9] and Compensating Transactions [16] are good examples. The most important property in operation semantics to reduce rollback overhead is commutativity.

5.2 Semantics-Based Compensation

As mentioned above, Sagas [9] and Compensating Transactions [16] rely on commutativity to reduce the rollback overhead in compensations. The idea is that if we are applying only commutative transactions then we can shuffle them all the way back to the original transaction introducing the inconsistency, thus avoiding the rollback. Sagas are composed of *steps*, each being an atomic transaction. If some steps need to be rolled back for any reason, a compensation step is run. They depend on general "application semantics", which include commutativity, as the underlying assumptions that allow compensations without rolling back the entire history.

To simplify the presentation we use the notation of Korth et al. [16] in the description of Compensating Transactions. When the updates of transaction T_1 are read by some other transaction T_2, T_1 is said to have been *externalized*. If we want to undo the effects of T_1, a Compensating Transaction CT_1 is run. T_1 is called a *compensated-for* transaction and T_2 a *dependent transaction* with respect to T_1. The goal of their recovery paradigm is to undo the compensated-for transaction but leave the effects of the dependent transactions intact. An important definition is that of soundness. (As usual [2], a *history* is a sequence of database operations.) If X is the history of transactions T, CT, and their set of dependent transactions $dep(T)$, and Y is some history of only the dependent transactions $dep(T)$, then X is said to be *sound* if for the same initial state S, $X(S) = Y(S)$. In other words, in a sound history, CT compensates for T cleanly, leaving the effects of $dep(T)$ intact. It can be shown that if CT commutes with every transaction in $dep(T)$ then the history is sound.

They further generalize the above definition to include weak forms of compensation soundness. The history X is sound with respect to a reflexive relation R (in short R-sound), if there exists a history Y of $dep(T)$ such that $Y(S)\ R\ X(S)$. For the case of R being equality, the general definition reduces to the "regular" soundness.

In ESR-based TP systems, the definition of inconsistency specification implies a monotonic distance metric underlying an ϵ-spec. So our focus is narrower than the predicate-based generality of Korth's reflexive relations. We are interested in a relation called "Within Bound", denoted by $W(B)$, such that $Y(S)\ W(B)\ X(S)$ if the database state $Y(S)$ is within the distance bound B of state $X(S)$ in the distance metric. If the distance metric is isotropic (as in airline and bank examples and all real-world applications) then the relation $W(B)$ is reflexive. The result is that $W(B)$-sound histories are ϵ-serial.

In our banking example, consider a transaction T that transfers funds and a query ET $Q(ImpLimit = \$10,000)$ in $dep(T)$ that produces the summary of total balance. If we need to compensate for T, the CT must reverse the fund transfer. According to the above definition, the history X is sound for a bound of \$10,000 if there is a $Y(S)$ in $dep(T)$ that produces a result within \$10,000 of $X(S)$. It is immediate that if the amount of T is less than \$5,000 then we can schedule the execution of Q freely, since it now *commutes* with T within the \$10,000 bound. It is the job of the ACR method, however, to accumulate the amount of inconsistency in the system for any given moment, to make sure the bound for each ET is not exceeded.

5.3 Designing ACR Methods

An informal description of a generic ACR method is that during the processing of a compensation ET it goes through the history checking for violations of the commutativity property. Whenever a violation is spotted, the method accumulates the update amount for all the involved ETs. If the ϵ-specs are not exceeded, then the compensation remains sound and the algorithm continues. When one of the bounds is exceeded, the ACR has three choices (to be made by the programmer or the system on behalf of the programmer), which we will consider in turn:

(1) to stop the compensation ET,

(2) to retroactively UNDO the committed ET, potentially starting a cascaded abort although at a lower intensity, and

(3) to mark this conflict and continue.

We think that alternative (1) is a viable choice, even though most researchers (including Korth et al [16]) define a Compensating Transaction as non-abortable. The main reason they do not consider the possibility of aborting a Compensating Transaction is because the abort would introduce some inconsistency into the database, which is not tractable in the classic TP theory. Although this alternative is not the focus of this paper, ESR allows some inconsistency to persist in the database. This is consistent with the real-world databases, where some amount of residual permanent inconsistency (e.g., due to data entry errors) is inevitable.

Alternative (2) continues the processing of the compensation ET by UNDOing the conflicting committed ET. This may be followed by a REDO, depending on the soundness of the resulting history. Although the phenomenon is the same as cascaded aborts, we believe that the intensity (number of aborts) is much smaller because only the ETs with exceeded bounds need to be undone.

In alternative (3), the ACR processes the entire history to the end, finds all the bound violations and reports the result to the compensation ET. This is useful if the compensation ET programmer wants to assess the magnitude of the total conflicts before making a decisions between options (1) and (2). For example, cascaded aborts may be preferable for a small number of aborts, but if $dep(T)$ is large then a large number of aborts may force the compensation to stop.

To illustrate this compensation-based ACR design methodology, let us consider an simple example. Each asET and compensation asET is a single operation and we have an ADC method derived from two-phase locking. Table 2 shows the lock compatibility matrix for dirty objects. ACR uses the usual two-phase locking compatibility table [28] to handle clean objects. The main difference is in squares marked LOK-1 (reading uncommitted data), LOK-2 (overwriting data read by uncommitted query –degree 2), and Commu-1 (if operations commute). In all

	No Lock	R^{ET}	W^{ET}
Read/Write ImpLimit= 0	—	—	—
R^{ET} ImpLimit> 0	AOK	AOK	LOK-1
W^{ET} ImpLimit> 0	AOK	LOK-2	Commu-1

Table 2: 2PL Compatibility for Dirty Objects

three cases (LOK-1, LOK-2, and Commu-1), the lock is granted if the ImpCount and ExpCount of involved ETs are within their respective ϵ-spec. In Table 2, R^{ET} includes the read requests from Q^{ET}. Note that we have identified the places where potential inconsistency may arise and made explicit the references to these situations. This type of analysis is facilitated by ESR because it is semantics-independent. This does not prevent ESR from incorporating the explicit specification of semantics-dependent inconsistency. In contrast, sagas [9] as proposed are implicitly dependent on application semantics for the maintenance of database consistency.

5.4 Independent Updates

Besides compensations, a second method of consistency restoration is *independent updates*. In these cases, an independent source of consistent data is available. From time to time the consistent data is used to overwrite potentially inconsistent data. The first important example of this method is the propagation of replica updates in primary copy methods, such as Grapevine [4]. Since all the updates are performed first in the primary copy, the secondary copies may be allowed to diverge (within bounds specified by each dET). A similar situation occurs with bank accounts. The bank database is processed in batch mode at night, at which time the updates are made. Although, each branch may log some local operations into the local replica (usually on paper), the official copy is the central database.

Another class of applications that use independent updates are the signal acquisition systems that receive fresh data every so often, such as radars or satellite photos. Even if the current data is inconsistent, a consistent version is expected to arrive at certain time intervals to solve the problem.

One way to use independent updates is to emulate the bank practice. An update is made locally and immediately, but the update is sent to the central site in a reliable message [3]. The update in the central site satisfies the necessary rigor in consistency constraints, for example, serializability. Periodically the central site propagates the official updates known to be consistent to the local sites.

The ϵ-spec of the local copy is limited by the amount of updates done locally in the worst case (when all the updates fail at the central site and are rolled back). To enforce this limit, the CR algorithm may need to store the accumulated ϵ-spec with each data item. This can be done in main memory, if the number of local updates between consecutive update propagations is relatively small. If it is too much for main memory, it can be stored on disk. Alternatively, the ϵ-spec can be calculated from the site update log when needed. This is a trade-off between storage and processing.

6 Applications Revisited

Having described ESR and outlined its implementation, we are ready to apply ESR to solve the problems detailed in Section 2. First, because ESR allows asynchronous execution of asETs, it decreases or eliminates the synchronicity requirement of classic distributed TP. Second, ESR-based systems offer dependable consistency levels that can be specified at the granularity of individual ETs at run-time. Thus, users are given the flexibility to precisely control the trade-off between consistency on the one hand and performance, availability, and autonomy on the other hand. We note that *ad hoc* solutions may have been previously proposed for each application, but ESR offers a uniform framework for a diverse suite of applications.

6.1 Autonomous TP

Recall that airline reservation systems run with few consistency checks during normal operation in order to optimize response-time. However, when a flight becomes full, it is important to maintain control consistency for queries and updates. ESR can model both situations. ETs with a relatively large ϵ-spec runs unimpeded normally; when a flight becomes full its ETs run with a tighter ϵ-spec.

To enable autonomous operation when the network is partitioned or a site is down, we can use either pre-allocation of seats such as the Data-value Partition method (Section 2.4), or a dynamic allocation using a standard ACR method. Data-value Partitioning offers higher autonomy, since negotiations between sites is necessary only when the seat allocation is exceeded at some site. A pure dynamic allocation method based on ACR simply processes reservation requests as usual during the normal execution. During a network partition, the system proceeds using estimates given by the load management system. Each site is given a quota of seats for the duration of the network partition, and they obtain reservations by specifying the remaining quota as the ET's ϵ-spec. When the network connectivity is restored, the difference is reconciled using an ACR method. For example, if the network link between New York and Chicago is temporarily broken, New York may be assigned k seats on each flight from Chicago to New York, even though the database is inaccessible. When the link is restored, the ACR method (e.g., based on compensations) will replay the reservations and make up to k reservations in the Chicago database.

In the banking example, each component bank (as well as the parent bank) may process its own asset analysis independently, perhaps seeking slightly different kinds of information to generate reports of different formats. If the inconsistency is below the tolerance level (which may be different for each bank), these reports may be run asynchronously. A more serious situation is when a network partition or a site crash makes part of the distributed database inaccessible to the rest of the system. It is desirable that clients be allowed to make withdrawals, for example from the automatic teller machines, while minimizing the risk to the bank. Current practice gives a flat limit to all customers ($400 daily for the New York Cash Exchange), which bounds the bank risk. Using a quota system as above, the bank may be able to offer a significant part of a customer's balance available with less risk. Assuming replication, the bank may allow a percentage of the last balance to be used, provided that limitations are placed on the other partition to protect the bank. This way, bank loses less in case the balance is less than the fixed limit and gains customer satisfaction when the balance is significantly higher than the limit.

6.2 Service Guarantees

One of the strong points of ESR is the ability to offer service guarantees in terms of consistency. In the stock market example, the inconsistency is represented by the time lag between the reported price and the last transaction update. Using ESR with fixed time bounds (explained in [28]), the stock exchange can offer the brokers some firm guarantees of the inconsistency level. Even if the system breaks down due to overload, it can still return the amount of inconsistency (in terms of time lag in this case) so brokers may judge the value of its information. For critical times when the volume threatens to overwhelm the system, as well as the remote connections to other cities and countries, the guarantee on consistency level is very important.

In general, ESR-based systems can offer dependable consistency levels, in contrast to degree 2 and QSR, for example. Further, the level of consistency can be specified at the granularity of each ET at run-time. This flexibility gives programmers previously unavailable facilities to control the trade-off between consistency on the one hand and performance, availability, and autonomy on the other hand.

6.3 Integration with Classic TP

One possible reason why in practice many DB2 installations can use degree 2 consistency is because current circumstances mitigate the actual amount of inconsistency. Assuming that each update value has an upper limit, that the query has a bounded duration, and that the number of update transactions per second is moderate, then the total amount of inconsistency of that query is bounded by the product of these factors. However, as the system grows, both the query duration and the update transaction rate increase. If update values also increase then the fortuitous bound on inconsistency may quickly become meaningless and degree 2 no longer useful.

Therefore, for scalability reasons we need a more abstract and precise integration with classic TP. Another example is using the Escrow Method [22] to improve system performance. The Escrow Method requires the programmers to accommodate the whole-data-item uncertainty explicitly, which may interact with degree 2 (in)consistency in subtle ways. As we move into more advanced applications [1] and heterogeneous databases [7], the integration of different kinds of concurrency control becomes increasingly important. In contrast to degree 2 consistency, ESR —by definition— provides a smooth integration between SR and any degree of inconsistency. Furthermore, because ESR guarantees a bound on inconsistency, applications will not produce increasingly inconsistent results when more processing power or nodes are added to the system.

6.4 Network Partitions

Another advantage of ESR-based distributed TP is the transparency of the ET interface. For example, the Escrow Method [22] increases the TP system throughput by implicitly reserving part of an aggregate field (e.g., a numerical value) for processing. The reason the Escrow Method obtains more concurrency is its ability to release the lock on the data item once the partial value has been put in "escrow". More relevant to this paper, the Data-value Partitioning method [27] increases the distributed TP system availability and autonomy by explicitly separating parts of the value of a data item into different sites. Since the different parts may operate asynchronously even during network partitions, Data-value Partitioning increases autonomy because of its non-blocking character. A serious problem with both methods is that they introduce uncertainty when trying to read the value of the entire data item. For both the Escrow Method and the Data-value Partitioning, this uncertainty is inevitable.

ESR can capture this uncertainty. For example, instead of having the application programmer look at the maximum and minimum possible values (as suggested in [22]), an ϵ-spec in a read ET is enforced by the system, so the ET programmer will not have to deal with the data value uncertainty explicitly. The same way the transaction interface hides details of concurrency control and crash recovery, the ET interface hides details of uncertainty. In an ESR-based TP system, the Escrow Method and Data-value Partitioning may be introduced as optimization techniques that are transparent to the application programs.

7 Related Work

Besides ESR, notions of correctness weaker than SR have been proposed. We have already discussed QSR [6] and degree 2 [11]. Garcia-Molina et al. [9] proposed sagas that use *semantic atomicity* [8] which rely on transaction semantics to define correctness. Sagas differ from ESR because an unlimited amount of inconsistency (revealed before a compensation) may propagate and persist in the database. Levy et al [18] defined *relaxed atomicity* to model non-atomic transactions similar to sagas. Non-atomic transactions are composed of steps, which may be a forward step or a recovery step. They also describe the Polarized Protocol to implement Relaxed Atomicity. The main difference between ESR and these notions of correctness is that ESR is independent of application semantics. ESR also allows a larger number of execution histories. The Polarized Protocol, for example, does not allow global state from an incomplete transaction to be seen by other transactions.

An implementation issue in asynchronous TP is to guarantee uniform outcome of distributed transactions running asynchronously. Unilateral Commit [14] is a protocol that uses reliable message transmission to guarantee that a uniform decision is correctly carried out. Optimistic Commit [17] is a protocol that uses Compensating Transactions [16] to undo the effects of partial results to reach a uniform decision. This is but one aspect of the autonomous TP problem.

Sheth et al [26] use the notion of *eventual consistency* to define *current copy serializability* (CPSR) for replicated data. Each update is done on a current copy and asynchronously propagated to the other replicas. Users have control over when the updates are propagated, and the scheme reduces to synchronous replication is the propagation delay is set to zero. In contrast, ESR is useful for general asynchronous TP, not just replication.

8 Conclusions

We study the feasibility of execution autonomy in systems with asynchronous transaction processing. By maintaining epsilon-serializability (ESR), autonomous sites can ensure consistency despite the concurrent and asynchronous execution of distributed transactions. We describe efficient implementation techniques for ESR. While allowing bounded inconsistency to be seen by other queries, asynchronous divergence control methods prevent inconsistency from entering the database. If limited amounts of inconsistency is found in the database, asynchronous consistency restoration methods restore database consistency.

To illustrate the benefits of ESR in asynchronous transaction processing, we describe several concrete applications: an airline reservation system, a bank account management system, and a stock exchange trading system. These examples make use of ESR to achieve autonomous transaction processing, guaranteed consistency levels, and integrated execution with classic transaction processing.

The work based on ESR is called Generalized Transaction Processing, several aspects of which are currently under development. One application of ESR is an asynchronous approach for replication [23]. An important part of ESR implementation is the divergence control methods [28]. We are now evaluating the performance benefits of divergence control methods resulting from added concurrency and decreased deadlock frequency.

References

[1] N. Barghouti and G.E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, September 1991.

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

[3] 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.

[4] 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.

[5] Y. Breitbart, A. Silberschatz, and G. Thompson. Reliable transaction management in a multidatabase system. In *Proceedings of 1990 SIGMOD International Conference on Management of Data*, pages 215–224, May 1990.

[6] W. Du and A. Elmagarmid. Quasi serializability: a correctness criterion for global concurrency control in InterBase. In *Proceedings of the International Conference on Very Large Data Bases*, pages 347–355, Amsterdam, The Netherlands, August 1989.

[7] A.K. Elmagarmid and C. Pu, editors. *Special Issue on Heterogeneous Databases*, volume 22:3 of *ACM Computing Surveys*. ACM, September 1990.

[8] H. Garcia-Molina. Using semantic knowledge for transactions processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

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

[10] D. Georgakopoulos and M. Rusinkiewicz. On serializability of multidatabase transactions through forced local conflitcs. In *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan, April 1991.

[11] J.N. Gray, R.A. Lorie, G.R. Putzolu, and I.L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *Proceedings of the IFIP Working Conference on Modeling of Data Base Management Systems*, pages 1–29, 1979.

[12] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[13] P.M. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[14] M. Hsu and A. Silberschatz. Unilateral commit: A new paradigm for reliable distributed transaction processing. In *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan, February 1990.

[15] D.R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[16] H. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[17] E. Levy, H. Korth, and A. Silberschatz. An optimistic commit protocol for distributed transaction management. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, Denver, Colorado, May 1991.

[18] E. Levy, H. Korth, and A. Silberschatz. A theory of relaxed atomicity. In *Proceedings of the 1991 ACM Symposium on Principles of Distributed Computing*, August 1991.

[19] B. Lindsay, L.M. Haas, C. Mohan, P.F. Wilms, and R.A. Yost. Computation and communication in R*: a distributed database manager. *ACM Transactions on Computer Systems*, 2(1):24–38, February 1984.

[20] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

[21] H. G. Molina and Kogan B. Node autonomy in distributed systems. In *International Symposium on Databases in Parallel and Distributed Systems*, pages 158–166, December 1988.

[22] P. E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11(4):405–430, December 1986.

[23] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, Denver, May 1991.

[24] C. Pu, A. Leff, and S.W.F. Chen. Heterogeneous and autonomous transaction processing. Technical Report CUCS-008-91, Department of Computer Science, Columbia University, April 1991.

[25] Calton Pu. Superdatabases for composition of heterogeneous databases. In Amar Gupta, editor, *Integration of Information Systems: Bridging Heterogeneous Databases*, pages 150–157. IEEE Press, 1989. Also appeared in Proceedings of Fourth International Conference on Data Engineering, 1988, Los Angeles.

[26] A. Sheth, Yungho Leu, and Ahmed Elmagarmid. Maintaining consistency of interdependent data in multidatabase systems. Technical Report CSD-TR-91-016, Computer Science Department, Purdue University, March 1991.

[27] N. Soparkar and A. Silberschatz. Data-value partitioning and virtual messages. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, Nashville, Tennessee, April 1990.

[28] 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.