Recovery Guarantees: Essentials and Incidentals

Cris Pedregal Martin and Krithi Ramamritham
cris@cs.umass.edu, krithi@cs.umass.edu

Technical Report 98-12

Computer Science Department
University of Massachusetts, Amherst, Mass. 01003–4610, USA

Abstract

In spite of the central role recovery plays in a transaction systems supporting the properties of Failure Atomicity and Durability, its theoretical underpinnings are not well understood. Specifically, there is a lack of abstractions to decompose the machinery of recovery, which leaves a wide semantic gap between high-level requirements and implementation. Realizing recovery, especially for novel applications and settings, is thus difficult and error-prone.

Recovery guarantees reduce the semantic gap by characterizing the assurances relevant to recovery that a subsystem must give to another; as such they describe the *what* but not the *how* of recovery at a given abstraction level. Guarantees are complemented by recovery protocols, which prescribe behaviors subsystems should follow in order to take advantage of the guarantees.

In this paper we use the recovery guarantees and protocols to shed light on the nature of recovery in different systems. We concentrate on two simple examples which show the breadth of application of our approach: a mobile system and an electronic commerce system. By comparing the characterizations of the two systems we separate the aspects of recovery that are germane to specific architectures and services from those that constitute the core of guaranteeing functionality in spite of failure. Our main contribution, through significant furthering of understanding of the essential and incidental recovery ingredients and their relationships, is a methodology to describe and reason about the recovery requirements and properties of reliable transactional and workflow. Other contributions include clarifying the mechanics and role of recovery in mobile and e-commerce systems.

1 Introduction

Transactional semantics provides a clean model for computation in the presence of concurrency and failures. For example, on transaction management systems that support some variant of the ACID[1] semantics [GR93] one can design applications that operate on a database without concern for concurrency or failure issues. The transaction management system maintains those properties by controlling how data are accessed and modified by different concurrent transactions (Concurrency Control) and by keeping enough information in persistent, safe storage to enable it to recover from failures (Recovery).

It is increasingly important to preserve the integrity of the data in the face of failures in systems and applications other than databases. We are interested in how the good properties of recovery can be described and supported in both in traditional database and in novel settings, such as workflows. In this paper we examine recovery in two scenarios: a mobile database and an electronic commerce example. We propose the notion of *recovery guarantee* and we use it as an abstraction tool, separating how the recovery properties are supported and what recovery properties other subsystems rely on.

Goals and contributions This paper is an initial effort aimed at bringing the benefits of abstraction to the characterization and crafting of (database transaction) recovery functionality in a broad range of applications and infrastructures.

We believe that exposing the components of recovery and their relationships will improve modeling and crafting of recovery. This will improve the understanding of recovery, leading to an increase of confidence in existing schemes (less obscurity) and making tradeoffs more apparent, thus allowing more informed choices of infrastructure and protocols. Also, it will make it easier to build recovery with both traditional and novel database applications.

The main contribution of this paper is a methodology to decompose and describe recovery in terms of the relationships between the subsystems that provide and the ones that use recovery properties, expressed as guarantees and protocols. Additionally, the paper sheds light on the nature of recovery by exposing which features of recovery are incidental to the infrastructure, which are incidental to the specific application domain, and which are essential reflection of the intuitive notions of using persistence, preparing for failure, and repairing after failure.

This paper is organized as follows. In the remainder of this section we introduce the notions of guarantee and protocol, along with some minimal notation. Section 2 discusses the recovery aspects of two examples, a mobile system and an electronic commerce system, in terms of guarantees and protocols. Section 3 offers some conclusions, discusses related work, and indicates future directions of research.

[1] ACID: Atomicity, Consistency, Isolation, Durability.

1.1 Recovery Guarantees and Protocols

It is well understood that in general recovery is about carefully and economically preserving information sufficient to bring a system to a consistent state in the face of (system or transaction) failures. This is a reflection of two facts. One, transformations of a system from one consistent state to another are not atomic, hence failures midway leave the system in an inconsistent state if not repaired. Two, system state (partially) resides in, and thus its transformations happen on, (volatile) storage whose contents are lost when failures occur, necessitating the transcription of crucial state information to reliable storage. The fundamentals of recovery in these abstract terms are covered in the literature [BHG87, GR93]. Most literature describes how to implement recovery [MHL+92, CMSW93, GR93] in very low-level terms, making it difficult to re-apply the basic ideas to novel scenarios. In other words, recovery is either discussed in general terms or in very detailed ones, with little in-between.

We propose a more detailed but still generally applicable way of talking about recovery is by introducing the notion of guarantees and protocols. Recovery is possible because different subsystems fail in different manners: an understanding of those failures (or, in other terms, of the reliability a subsystem has to offer), coupled with disciplined communications between subsystems, yields an aggregate system which is reliable as a whole in spite of the unreliability of its parts. Specifically, we describe the expectation a subsystem has on the reliability of another using the concept of *guarantee*.

In order to characterize the recovery properties of a system, we need to describe (i) what each subsystem expects from others (the guarantees) and (ii) how subsystems behave in order to create the conditions for the guarantees to be useful (the protocols).

We introduce the notion of *recovery guarantees*, which describe the assurances relevant to recovery that a subsystem (e.g., a fixed host in a mobile system, a bank in an e-commerce scenario) must give to other subsystems (e.g., a mobile host, a merchant); in general, recovery guarantees formalize expectations of reliability between parts of a system and have a broader applicability. Notice that, at a given level of abstraction, a guarantee describes *what* a subsystem offers as a recovery assurance, but not *how* it goes about doing it; in turn, how the guarantee is supported may be described in terms of guarantees offered by components of the subsystem.

We also introduce the notion of *recovery protocols*, which are prescriptions of how subsystems must behave in order to avail themselves of the benefits given by the guarantees; for example, stable undo logging before updating is a protocol that enables a data manager subsystem to use the guarantee of the disk subsystem in case the update needs to be undone later.

Recovery guarantees take the form:
if (p) **then** *guarantee(q)*

This states that:

> **if** p has been performed,
> **then if** q is invoked in the future, it will be performed.

The notation used henceforth to express such a guarantee is:

$$p \Rightarrow_G q.$$

Thus, a *recovery guarantee* is a promise that once p is executed, the guaranteed operation q, *if*

invoked, will succeed, in spite of any intervening failures.

Often *where* an operation is performed will be part of the guarantee specification. In general, the protocol is as follows: a (requester) subsystem S invokes an operation p of (recovery, i.e., guarantor) subsystem R; the success of p signifies that R guarantees to S that a future invocation of q will succeed. Basically, with the information made available by p (and perhaps other information maintained by R), R will be in a position to apply q, if requested. The above scenario is specified as

$$p_R \Rightarrow_G q_R.$$

It should be noted, however, that the subsystem on which p is performed and the subsystem guaranteeing q need not be the same!

Also, note that we do not talk about the "commitment" or "durability" of p. All that is stated above is that p is executed at R for R to guarantee that, when invoked, q will be executed. How R ensures that this guarantee will hold (say, by making p's results durable, etc.) is of interest only when one wants to "verify" the capabilities of subsystem R.

An example of a recovery protocol is the requirement that stable undo logging happen before updating. This protocol enables a data manager subsystem to use the guarantee of the disk subsystem in case the update needs to be undone later. For the purposes of this paper, we use protocols that specify precedence constraints: **if** q *happens* **then** p *must have preceded it*, i.e.:

if q has been performed,
then p must have happened before.

The notation used to express precedence protocols is simply: $p \rightarrow q$.

Some notation We represent the behavior of the system with events in a partial order –the history of the system. The subscript denotes in which subsystem the operation that the event represents takes place.

- p_S denotes a generic data operation which modifies the state of the subsystem S on which it is applied. Note that p may be invoked in one subsystem but be applied in another.
- $R(p_A)$ denotes the information necessary to produce the effects of operation p_A (on subsystem A). $R(p_A)$ is a data item and may be stored and transmitted; when subsystem B has such data we simply write $R(p_A) \in B$.
  $WRL_B(p_A)$ denotes the operation of creating a redo log record on B for operation p_A. (After $WRL_B(p_A)$ occurs, $R(p_A) \in B$ holds.)
- $U(p_A)$ denotes the information necessary to undo the effects of operation p_A (on subsystem A). $U(p_A)$ is a data item and may be stored and transmitted; when subsystem B has such data we simply write $U(p_A) \in B$.
  $WUL_B(p_A)$ denotes the operation of creating an undo log record on B for operation p_A. (After $WUL_B(p_A)$ occurs, $U(p_A) \in B$ holds.)

Example: Undo Log Writing the undo log on subsystem B for operation p guarantees that it will be possible (later) to apply the inverse of p on subsystem A (A and B may be the same subsystem):

G1 $WUL_B(p) \Rightarrow_G p_A^-$

We abuse notation and use G1 to signify the more precise guarantee G2, which requires that the inverse work correctly:

G2 $WUL_B(p_A[x]) \Rightarrow_G p_A^-[x] \wedge state(s, p_A[x]) = s' \wedge state(s', p_A^-[x]) = s$

The fuller notation of a guarantee generalizes the predicates that comprise its antecedent and consequent. In the shorter version, the appearance of an operation means the predicate "the operation succeeded," in the longer version, we incorporate an arbitrary predicate.

These guarantees[2] are supported by the recovery system, as follows: during normal (prevention) processing, the recovery system ensures that logs are made persistent and thus safe from failures; after a failure, during repair processing, the recovery systems accesses the logs and uses it to apply the undo as necessary.

The WAL (Write-ahead logging) protocol P1 ensures that the guarantee G1 will be applicable should a transaction need to be undone. WAL requires that an undo log record for an operation be stored before the operation is applied to preserve the ability to undo it (A and B may be the same subsystem):

P1 $WUL_B(p) \rightarrow p_A$

2 Two Recovery Examples

In this section we consider two examples: a mobile system and electronic commerce, and then discuss their similarities and differences.

2.1 Mobile System

The mobile system described by Pradhan et al. [PKV95] consists of a set of (fixed) base hosts and mobile hosts. Each *base* covers a *cell*, which is an area in which there may be zero or more mobile hosts. A mobile host may move from one cell to another, but at any given time it communicates with only (at most) one base host.

The state of a mobile host M changes by the application of operations. An operation has one of two origins: it may be caused by *user input at* M or it may be caused by *a message from another host* A.

Stable storage is needed for saving the process checkpoint and the logs. We consider that stable storage is at the base station (B). The disk storage at the mobile host (M) is deemed vulnerable to catastrophic failures and frequently disconnected.

Distributed applications require exchange of messages between (local and mobile) hosts and user inputs at the mobile hosts. A message sent to a mobile host M is first sent to the base host B which covers the cell that M is in. B forwards the message to M.

[2]The Redo Log guarantee is similar to the Undo Log guarantee G1, we omit it for brevity.

Two recovery approaches are proposed in [PKV95], based on whether any logging takes place at the base station B. In the logging approach, before a message is transmitted to a mobile host M, its base station host B logs it. Even inputs at M are sent to B and effected at M only after B's acknowledgment.

We introduce protocols and guarantees specific to the mobile system. We illustrate the behavior of the system via a sample history, H1. Consider an operation p_M which host A invokes to carry out at mobile host M:

H1 $invk_A(t, p_M)\ send_A(p_M, base(M))\ recv_{base(M)}(p_M)\ \ldots$
$\ldots WRL_{base(M)}(p_M)\ send_{base(M)}(p, M)\ recv_M(p)\ p_M$

where $base(M)$ is the fixed host for the cell where M is.

The operation is sent to M's base host (denoted $base(M)$) , which first logs it and then sends it on to M.

Mobile System Notation

- A, B, C denote any hosts; F, G denote hosts that are fixed in the network; M denotes a mobile host. In general, we have a predicate:
  $Mov(A)$ which is true if and only if host A is mobile.
- $Cnx(A, B)$ denotes that A and B are connected, i.e., that they can exchange messages with each other.
- We write $F = base(M)$ when host M is within the cell controlled by host F (a base station host). Base stations are always fixed.
- $send_A(p_C, B)$ denotes that subsystem (host) A sends a message to subsystem B containing operation p, to be applied on subsystem C. We omit C when $C = B$.
- $recv_A(p_B)$ denotes that subsystem A receives a message containing operation p, which must be applied on subsystem B.
- $invk_A(t, p_B)$ denotes that a transaction t requests, on subsystem A, the invocation of operation p on B. We sometimes omit t for brevity. We also omit B when $B = A$.
- $inpt_A(p_B)$ denotes that a user at A invokes operation p to be executed on B.
- $hndf_F(M, G)$ denotes that host F hands off mobile host M to host G, or in other words that M leaves F's cell and enters G's cell.[3]

The following properties characterize the architecture of the mobile system under study. Direct communication is always possible for any pair of *fixed* hosts. A mobile host can only communicate with the host that controls the cell in which the mobile host is. Specifically:

[3] In general, it is possible for M to leave A and be outside any other cell.

- A fixed host is connected with all other fixed hosts:
  $\neg Mov(A) \Rightarrow (\forall B, \neg Mov(B) \Rightarrow Cnx(A, B))$
- A mobile host is directly connected to its base host (if it is connected at all):
  $Mov(M) \Rightarrow ((Cnx(M, B) \Rightarrow B = base(M))$ and B is unique)
  $base(M) = B \Rightarrow Cnx(M, B)$

  We do not introduce notation for the transitive closure of Cnx but it is obvious that when a mobile host M is in a cell it can communicate with any fixed host via its base station.

The following protocols apply. Each *receive* always has its corresponding *send*:

P2 $send_A(B, p) \rightarrow recv_B(p)$

(The application of) an operation always stems from either an input or an invocation:

P3 $(invk_A(t, p_B) \rightarrow p_B) \vee (inpt_A(p_B) \rightarrow p_B)$

(Notice that it is possible that $A = B$; for example, at mobile host M, $A = B = M$ when a user inputs an operation to be carried out locally.)

When a base host receives an operation addressed to a mobile host within its cell, it logs it before forwarding it:

P4 $WRL_{base(M)}(p_M) \rightarrow send_{base(M)}(p_M, M)$

Before an operation is applied it must be logged at B, which guarantees that the operation will be reproducible later if necessary. The guarantee is:

G3 $WRL_B(p) \ \wedge \ Cnx(B, M) \ \Rightarrow_G \ p_M$

Where B is (the durable subsystem of) a fixed host, and M is the mobile host. The operation p may have been caused by an invocation (from a transaction) or user input; either at M or elsewhere.

The corresponding protocol prescribes redo-logging:

P5 $WRL_{base(p)}(p) \rightarrow p_M$

The first two protocols P2 and P3 are basic properties of correct executions; the third protocol P4 reflects a specific policy of logging before handing over messages to a mobile host. The guarantee G3, paired with protocol P4, captures the notion that the base station is expected to keep recovery information for the mobile host.

Remark The preceding protocols and guarantee are special cases of more general ones. For example, protocol P5 follows from protocol P3 and protocol P4:

$$\begin{aligned} &\{(invk_A(t, p_B) \rightarrow p_B) \vee (inpt_A(p_B) \rightarrow p_B)\} \ \wedge \\ &\{WRL_{base(M)}(p_M) \rightarrow send_{base(M)}(p_M, M)\} \\ &\qquad \Rightarrow \{WRL_{base(p)}(p) \rightarrow p_M\} \end{aligned}$$

Guarantee G3 is an instance of the more general redo guarantee G4, which states that if the information to effect operation p on host B is available at host A, and A and B can communicate, then p_B is guaranteed:

G4 $R(p_B) \in A \ \wedge \ Cnx(A, B) \Rightarrow_G p_B$

The antecedent of guarantee G4 may be satisfied in various ways:

- If a redo log for the operation was written at A, then A has the redo information for p: $WRL_A(p_B) \Rightarrow R(p_B) \in A$
- If a redo log was written at a different host C, which is accessible to A, it was/can be obtained by A and so A has/can get the redo information for p. These cases are considered in section 2.1.1.

2.1.1 Handoffs

The preceding does not take into account the effect on recovery of a mobile host's migration through the network. Migration is represented by a *handoff*, which happens when a mobile host M leaves a cell served by base station host F and enters a new cell base station host is G. We require that recovery-related information for M initially available on F must still be available should recovery be necessary while M is in G's cell.

We first discuss the handoff at an abstract level. M's migration from F to G causes F's guarantees to M to be honored by G because of (i) the guarantees F gave M, and (ii) a guarantee between fixed hosts on the network (given by their connectivity), in particular F to G. Thus the guarantees necessary to M are inferred from the guarantees between M and F and the connectivity guarantee between any two fixed hosts (F and G).

Guarantees between fixed hosts The system guarantees between fixed hosts in the networks say that if one of them has (redo or undo) recovery information for an operation, the other can obtain it too:

G5 $(\neg Mov(F) \ \wedge \ \neg Mov(G) \ \wedge \ R(p) \in A) \Rightarrow_G R(p) \in G$
$(\neg Mov(F) \ \wedge \ \neg Mov(G) \ \wedge \ U(p) \in A) \Rightarrow_G U(p) \in G$

These guarantees are supported by the architecture of the fixed host network; the connectivity, and some network management software, ensure that station G can obtain the recovery information from station F. The specification of *when* (or if) that information is transferred is done via protocols that prescribe what must be done at a handoff, and for which we specialize the guarantees G5 by replacing the antecedent with an implementation of how to record and convey recovery information. In the sequel we deal with redo guarantees; undo guarantees are similar.

Eager and Lazy handoffs To make the recovery information available to a mobile host, the approaches are *eager* (pessimistic in [PKV95]), in which the information follows M on the fixed network (i.e., when M moves from F to G, its recovery information is forwarded from F to G); and *lazy*, in which information is only fetched from the original host if and when necessary (i.e., the information remains at F, and G just keeps track of that fact).

For the eager approach, protocol P6 simply ensures that when a mobile host arrives in a new cell, as part of the handoff the previous base station transmits the recovery information to the current cell:

P6 $R(p_M) \in F \Rightarrow (send_F(R(p_M), G) \rightarrow hndf_F(M, G))$

Here again we extend the notation of protocol, adding a precondition for the protocol to apply, which appears as the antecedent of the implication above.

We specialize guarantee G5, by requiring a condition that yields the antecedent of G5, as follows. If recovery information resides on F (because p was invoked while M was in F's cell), then transmitting the recovery information to G will satisfy the antecedent of the guarantee:

$$(\exists F, R(p) \in F) \Rightarrow (send_G(R(p), C) \Rightarrow R(p_M) \in C)$$

(Here C denotes the current cell.) This is supported by protocol P6, which ensures the information is sent on handoff. Thus the eager protocol is correct because P6 ensures a the antecedent for G5; G5 with G4 yields G3.

Implementing the lazy approach only requires that there exist a linked list of the base stations visited by the mobile host in which there is extant recovery information relevant to the host. We do not need to specify this as an explicit protocol, i.e., we do not need to require that such forwarding information be updated on handoff, because we this information is in the history, in handoff ($hndf$) events. We show here only how to satisfy the antecedent of guarantee G5 in the lazy case:

$$WRL_F(p) \Rightarrow R(p) \in F$$

For completeness, notice that when the recovery information was generated in the same cell where the mobile host is, satisfying the guarantee G5 is trivial, because we have the consequent already.

2.1.2 Crash and Repair

The *repair phase* of recovery uses log information (possibly resident on various places) to reconstruct the state of the mobile host. The high-level property at work here is that the information necessary for undo or redo (as necessary) is durably stored in one of the nodes. In this subsection we briefly outline the protocols of the repair phase. One set of protocols delimits the repair actions, by prescribing which events must happen between the occurrence of a crash event and the completion of repair work. The other set of protocols describes the repair work itself, for example indicating that the effects of a 'loser' operation (e.g. one belonging to a transaction uncommitted before the crash) must be undone.[4]

The repair phase begins after a crash event. Thus we need to establish a protocol requirement that if a crash happened recovery must happen too. A committed operation (on M) must be *installed*[5] in M *and somewhere in the fixed network*, because we do not trust M's storage to be durable. Thus repairing the database will mean making the operation available on M (so M's state reflects a committed state) as well as somewhere else durable –a node on the fixed network, but not necessarily $base(M)$, as that's a policy choice. Thus we specify that for each operation p issued before the crash, repair must verify that p is committed but uninstalled, find the redo information for p; move the redo info for p to M; and using the guarantee that redo-info can be used to redo p, install p.

[4]Besides indicating what are correct repair histories, the protocols yield important properties that are otherwise stated ad-hoc: for example, the property that the system will not unilaterally undo an operation is a consequence that only while in the repair phase is this particular action permitted.

[5]Installing an operation usually means making its effects permanent on the (stable) database.

The repair phase ends when the state of the whole database has been restored to the committed projection of the history up to the crash. This may not be a single event; instead it is characterized on a per-object and per-operation basis.

Certain type of operations –repair work– can only happen while the database is being repaired; i.e., the events cannot appear at other times in the history of the system.

2.2 Electronic Commerce

We now give an informal description of a simplified scenario from an electronic commerce application.

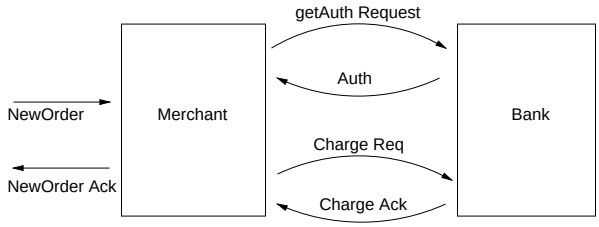

Figure 1: Merchant and Bank

The system consists of two entities connected over the Internet: a *Merchant* M and a *Bank* B (see Figure 1). A *customer* C contacts M through the Internet and places an order, which the customer pays by supplying a credit card number. After verifying its available stock, M obtains authorization from the bank for the amount of the order, by sending the credit card number and amount. Once authorization is obtained, M ships the item to the customer and asks B to process the credit card charge that had been authorized earlier.

The following is a history of operations that occur in this scenario:

H2 $newOrder_M$ $verifyStock_M$ $getAuth_B$ $shipOrder_M$ $charge_B$

Notice that once B authorizes the *charge*, M can commit to the sale, informing C that his or her order has been accepted. Subsequently, M calls upon B to execute *charge* to obtain payment for the sale from the bank.

From the customer's perspective, the activity begins with *newOrder* and ends when M commits to the sale at the end of *getAuth*. However, from M's perspective, the activity goes on until *charge* completes.

If there is a Merchant failure before the bank issues the authorization, the activity is aborted and the customer notified that the order cannot be processed. On the other hand, the successful completion of *getAuth* is a guarantee that if and when M invokes the *charge* operation, it will succeed. Based on this, the merchant can ship the order knowing fully well that the payment *will* come through. If the Bank crashes after issuing the guarantee, it is bound by the guarantee to be able to apply the charge once it is back up. That is to say, the authorization given by B is a guarantee for the successful completion of a subsequent *charge* operation.

The successful commitment of *getAuth* guarantees that if and when *charge* is subsequently invoked, it will succeed. It is thus possible, for example, to design the steps to take and fill an order at the Merchant without being aware of the nature of (interactions with) the Bank.

In terms of our notation, we can say that

G6 $getAuth_B(amount, account\#, ..., authID) \Rightarrow_G charge_B(authID)$

where *getAuth* returns a unique authorization ID which the Merchant must later present to validate its charge request. This is a typical Redo guarantee similar to the mobile system's guarantee G3.

The corresponding protocol dictates that the merchant not ship the order until the authorization is received, as indicated in the example history H2:

P7 $getAuth_B \rightarrow shipOrder_M$

At the Bank, the recovery guarantee G6 can be viewed as being supported by two requirements, one recovery-related and the other business-related:

1. The authorization ID and its associated information (e.g., how much was authorized to pay which merchant from which cardholder's account) must be preserved in spite of failures; it is implemented by the Bank's recovery machinery. To this end, information pertaining to the guarantee from the Bank must be durably stored by the Merchant. Alternatively, the Merchant must be able to later query the Bank if the Merchant failed after obtaining the guarantee, for example.
2. The bank must not allow a new charge if previous charges that have been authorized will not go through when they are applied in the future. More generally, the sum of all outstanding charges and authorizations must not exceed a function of an account's credit limit; it is implemented by the bank's business model, probably reflected by the operations (transactions) that issue authorizations and charges.

If the first requirement is not met, the promise made in the guarantee is obviously broken; if the second requirement is not met, the bank may be forced to take an extra risk or break its promise. This discussion is irrelevant to the merchant, but it does suggest that for a recovery guarantee to hold, the guarantor subsystem must provide future guarantees only if the previously provided guarantees will not be violated.

2.2.1 Discussion

While lack of space precluded developing them fully, the preceding examples show that one can decompose and explain recovery in terms of guarantees and protocols. The examples hint at the potential of our approach to make precise these properties:

- (De-)Composability/Abstraction of Recovery. The high-level guarantee offered by the bank in the e-commerce scenario may in turn be supported by the guarantees offered by some database system at the bank, and some application-business rules.

	requirement	guarantee	protocol
examples	FA + D	$WRL(p) \Rightarrow_G p$	$WUL(p) \rightarrow p$
what they are	system property, seen by user/apps	subsystem property, supported by recovery manager	execution property, supported by data manager
how implemented	guarantees + protocols	stable logging, recovery	logging operations

Figure 2: Relationships between recovery elements

- Equivalence of Recovery Implementations. The different handoff policies in the mobile example support the same recovery properties.
- Novel Applications. Recovery requirements can be specified for an application without reference to its underlying infrastructure.

3 Conclusions

The main contribution of this paper is a methodology to decompose and describe recovery in terms of the relationships between the subsystems that provide and the ones that use recovery properties, expressed as guarantees and protocols.

In some sense, protocols and guarantees are very closely tied to each other. Whereas protocols talk about what must be done to achieve correct behavior, a guarantee states what can be expected if certain events occur or operations are performed correctly.

By describing the mobile system and the electronic commerce scenario and their behavior in terms of guarantees and protocols, we obtained the following benefits. First, we used abstraction to separate what a component can expect from another; e.g., the mobile host's (recovery) expectations of the fixed hosts. The flip side of this abstract view serves to show what the system and each component must provide to others. This also documented precisely the challenges of providing recovery.

In this paper we have shown that *recovery guarantees* can be very helpful in exposing the connections and expectations that exist between different components of systems involved in a transactional activity. Figure 2 illustrates some of the normal processing and recovering processing relationships.

Related Work Few researchers have dealt with the formalization of aspects of recovery. For example, Kuo [Kuo97] used I/O automata to formally describe a recovery system based on ARIES [MHL$^+$92]; however, his description is at a low level of abstraction, close to the implementation, requiring a substantial reformulation to apply it to other scenarios.

Lomet and Tuttle [LT95] developed a redo recovery theory and algorithms that maintain the ability to recover a database provided the cache is flushed according to their installation graph (which imposes an ordering significantly weaker than the usual concurrency control conflict graph). More recently [LT99] they refined these ideas and applied them to supporting recovery of files and applications. Their work is a strong contribution to explaining recovery, especially the subtle issues of

cache management, and we expect our work can complement theirs by helping compose recovery in higher levels of abstraction.

Further Work We believe that the guarantee abstraction is at a level that can help us talk about recovery without getting into the details usually found in describing how recovery works. At the same time it allows us to delve into details that are not exposed if one were to stop simply at the properties of Failure Atomicity and Durability (FA + D). In this sense, we can use the idea of guarantees and the recovery protocols such as write-ahead-logging (WAL) as a mode of discourse to analyze and understand recovery in different domains; we will also attempt to integrate our ideas with the theory developed in [LT95, LT99].

We will use the approach of refining guarantees into lower level guarantees and protocols to precisely specify the recovery machinery, which in this paper we described informally.

Because guarantees help in understanding what a component is expected to do, they will enable using a divide and conquer approach to crafting recovery protocols and subsystems. To this end, we will continue to examine the crafting of recovery for other transaction processing platforms and for workflow systems.

References

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Mass., 1987.

[CMSW93] Luis-Felipe Cabrera, John A. McPherson, Peter M. Schwarz, and James C. Wyllie. Implementing Atomicity in Two Systems: Techniques, Tradeoffs and Experience. *IEEE Trans. on Software Engineering*, 19(10):950–961, October 1993.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, Calif., 1993.

[Kuo97] Dean Kuo. Model and Verification of a Data Manager Based on ARIES. *ACM Trans. on Database Systems*, 21(4):427–479, December 1997.

[LT95] David Lomet and Mark R. Tuttle. Redo recovery after system crashes. In *Proc. of the 21st Int'l Conf. on Very Large Data Bases*, pages 457–468, Zürich, September 1995.

[LT99] David Lomet and Mark R. Tuttle. Logical logging to extend recovery to new domains. In *Proc. of the 1999 ACM SIGMOD*, pages 73–84, Philadelphia, Penn., June 1999.

[MHL+92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks using Write-Ahead Logging. *ACM Trans. on Database Systems*, 17(1):94–162, March 1992.

[PKV95] Dhiraj K. Pradhan, P. Krishna, and Nitin H. Vaidya. Recoverable mobile environments: Design and trade-off analysis. Technical Report 95-053, Department of Computer Science, Texas A & M University, College Station, Texas, 1995.