Recovery Guarantees for General Multi-Tier Applications

Roger Barga
Microsoft Research
Redmond, WA USA

David Lomet
Microsoft Research
Redmond, WA USA

Gerhard Weikum
University of the Saarland
Saarbruecken, Germany

Abstract

Database recovery does not mask failures to applications and users. Recovery is needed that considers data, messages, and application components. Special cases have been studied, but clear principles for recovery guarantees in general multi-tier applications such as web-based e-services are missing. We develop a framework for recovery guarantees that masks almost all failures. The main concept is an interaction contract between two components, a pledge as to message and state persistence, and contract release. Contracts are composed into system-wide agreements so that a set of components is provably recoverable with exactly-once message delivery and execution, except perhaps for crash interrupted user input or output. Our implementation techniques reduce logging cost, allow effective log truncation, and provide independent recovery for critical server components. Interaction contracts form the basis for our Phoenix/COM project on persistent components. Our framework's utility is demonstrated with a case study of a web-based e-service.

1. Introduction

Database recovery does not mask failures to applications and users. Transaction atomicity guarantees all-or-nothing but not exactly-once execution. Applications need explicit code to retry failed transactions. Often such code is incomplete or missing, exposing failures to users. Or even worse, a failure occurs without being noticed, which can occur if the error handling part of an application crashes. For an e-commerce service, this can lead to user inconvenience and lost sales when this happens during shopping cart checkout. However, a user must not blindly re-initiate a transaction when no result is received, as the transaction may have succeeded and re-execution is not usually idempotent. Some e-services warn users not to hit the checkout/buy/commit button twice when a long delay occurs. Users who do not heed this warning may unintentionally purchase two seats on the same flight or two copies of the same book.

TP monitors, exploiting transactional message queues, have long been the preferred solution for coping with application failures. However, these prior solutions are limited to stateless applications and have not been fully carried over to middle-tier web application servers (e.g., Apache or IIS). And some e-services are even more complex, with layers of application servers (e.g., web server, workflow server, activity server) accessing several database servers but also directly maintaining persistent data in files and requesting services from external providers. For example, the Expedia travel service integrates services from travel industry providers such as Amadeus or Sabre. It is not clear how to adapt prior solutions.

For multi-tier applications with communicating components, a comprehensive form of data, process, and message recovery is needed, going beyond traditional database recovery. Designing such a protocol entails a number of issues:

- Which component logs which messages or state to provide recovery, mask failures, and provide exactly-once semantics to a user?
- How are logs managed, when is a log forced to disk, and how are logs coordinated for log truncation, crucial for fast restart and thus for high availability?
- How do critical components, e.g., database servers, avoid being "hostage" to other components (applications, servers, clients), which may hamper or block their independent recovery or normal operation?

1.1 Contributions

We develop a framework for recovery guarantees in general multi-tier applications that answers the above questions. It masks from users all failures (of clients, application servers, or data servers) such that a user's initial request, which may start a conversation or workflow, has exactly-once semantics, meaning 1) no output to the user is duplicated (to avoid confusion), 2) the user provides input only once (to prevent irritation), and 3) the user intent, e.g., buying airline tickets, is carried out exactly once.

We enable these strong guarantees for multi-tier applications, based on the common concept in fault-tolerant computing of piecewise determinism. We identify the logging required for specific non-deterministic events

Proceedings of the 18th International Conference on Data Engineering (ICDE'02)
1063-6382/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

so that after a failure, an application component can be replayed from an earlier installed state (in an extreme case its initial state) and arrive at the same (abstract) state as in its pre-failure incarnation.

- We introduce **interaction contracts** between two components, e.g., (web) application servers, workflow engines, mail servers, database servers, and client software systems (e.g., browsers). For example, a committed interaction contract between "persistent" components, whose state needs to survive failures, requires sender and receiver guarantees to ensure that the interaction persists at both components should either system fail. Contracts also exist for persistent component interactions with external components (including users) and transactional components, such as databases, which provide atomic state transitions but not exactly-once executions.
- We show how to compose bilateral contracts to make persistent components provably recoverable with exactly-once execution semantics. We strictly separate contract obligations from their implementation so that we can provide strong guarantees to users while many internal interactions can avoid forced logging. Such optimizations depend on the exact nature of the components and interactions.
- We present implementation techniques to a) minimize logging cost, especially log forces, b) enable log truncation to bound restart work and thus provide high availability, and c) permit independent recovery of certain mission-critical components.
- To demonstrate the applicability of our framework, we present a case study of a multi-tier e-service that integrates services from multiple providers.

1.2 Related Work

Recoverability for systems of communicating processes has been studied in the fault-tolerance community (e.g., [JoZw87, SBY88, Cr91, AlMa95, EJW96]), where the main focus has been long-running computations (e.g., scientific applications) with distributed checkpointing, to avoid a failure losing too much work. Most of this work has not masked failures from users. Methods masking failures exploit "pessimistic logging" (see, e.g., [HuWa95]), with forced log I/Os for sender and receiver upon every message exchange. Even more expensive techniques, such as process checkpointing (i.e., copying state to disk) upon every interaction, were used in early fault-tolerant systems of the eighties [Ba81, Bo89, Kim84]. So failure masking has been a luxury affordable only by mission-critical applications (e.g., stock exchanges). Only recently have vendors begun to serve this market with commodity features, but current solutions require explicit application code for failure handling and/or "stateless" components, or do not handle failures at all levels of a multi-tier application.

The most successful prior approach is queued transactions in OLTP [BHM90,GrRe93,BeNe96,WV01], supported by most TP monitors (e.g., MQ Series, Tuxedo, MTS). These involve three distributed transactions per user request to: enqueue the request on the queue (a separate resource manager); dequeue it, process it in the database server, and enqueue the reply; and dequeue the reply. This incurs the cost of three distributed commits (2PC) and requires "stateless" applications where the only state between transactions is in a database or queue. It requires significant programming effort to cast rich stateful applications into this quasi-stateless paradigm. And little work on failure masking for general, stateful, applications [FCK87, LoWe98] exists.

Fault tolerance is being discussed also for component middleware like EJB and CORBA [OMG00], but the focus is on service availability for stateless interactions (i.e., restarting re-initialized application server processes). Products (e.g., VisiBroker, Orbix, BEA WebLogic, Sun's J2EE suite) at best support simple failover techniques that do not relieve the application programmer from having to code failure handling logic and are not geared for masking process or message failures to users.

Others have raised the need for execution guarantees for e-services (e.g., [Ty98, MaRa99, FrGu00, SPS00, MaRa01, FBHS01]), but they have been concerned with specific applications such as payment protocols and do not specifically address system-wide failure masking in general multi-tier architectures, or are limited to stateless applications. Prior work on user-transparent database application recovery was restricted to applications embedded in the data server such as stored procedures [Lo98] and two-tier client-server systems [LoWe98, BLAB00, BL01]. Clients are trivially piecewise deterministic, but application servers typically receive asynchronous messages. Also, it is not obvious how the client-server protocol can be applied to more than two interacting components. Our interaction contract introduced here is the key for the generalization to multi-tier systems.

The recovery framework and protocols developed in this paper improve the state of the art in a number of ways. Compared to traditional techniques based on pessimistic logging or frequent process state saving, our protocols are less costly in terms of logging and state saving while providing very fast recovery. In contrast to solutions provided by TP monitors and CORBA- or EJB-oriented application servers, our approach can handle stateful applications without requiring a new application programming model. Finally, our method is unique in providing an end-to-end solution for masking all failures across an entire multi-tier federation of ap-

Proceedings of the 18th International Conference on Data Engineering (ICDE'02)
1063-6382/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

plication and data servers while, to a large extent, preserving the autonomy of these servers.

1.3 Computational Model

We consider a group of **components**: clients, application servers (e.g., web servers, Java servlet engines, workflow engines, etc.), data servers (e.g., database servers, mail servers, document servers), and users (viewed as components). We assume these components, which may be mapped to processes or threads, are **piecewise deterministic (PWD).** A PWD component is deterministic between two successive messages from other components, so that it can be replayed from an earlier state if it is fed the original messages. Such **deterministic replay** resends the original messages, and produces the same component end state. Replay starts from a previous component state on disk, perhaps the initial state. We call these saved states "installation points" (IP's), not checkpoints, to avoid confusion with the different notion of checkpoint in database recovery [GrRe93]. What constitutes component state varies greatly; sometimes a compact abstract state is sufficient rather than the full image of a component's address space. Usually, server state includes persistent data (e.g., a database), messages, and session information.

A client synchronously communicating with one or more servers, suspending its execution after a message send and awaiting a reply message from a uniquely identified server, is clearly PWD. For an application server that serves multiple clients and communicates in an asynchronous manner, the PWD assumption is not guaranteed without some effort. Such components have three types of non-determinism:

1. A component, e.g. database or application server, may execute on multiple **concurrent threads** accessing shared data (e.g., SAP-style systems or eBay-style web sites). Reproducing this access interleaving order is essential for successful replay. We assume that components do not directly share data. Multiple components accessing common data require that data be in a component, e.g., a database server. Non-determinism is removed by logging the interleaved accesses to that component, while that component logs the interleaved access to its data.
2. Component execution may depend on **asynchronous events**, e.g., received messages or interrupts that prompt component execution at arbitrary points. These events are not reproducible during replay. We log the order of asynchronous events to guarantee deterministic replay. Often short logical log entries are sufficient, e.g. message receive order, if message contents can be recreated by other means (e.g., from the sender). However sometimes, physical logging is inevitable, e.g., when reading the real-time clock.
3. Component re-creation after a crash may not exploit the same system elements as the original execution, a form of non-determinism. Ids for messages, processes, threads, or users may change. To remove **system resource mapping** non-determinism, we "virtualize" these resources, introducing logical ids for messages, component instances, etc. We log the logical ids. The logical ids are mapped to different physical entities after a crash, but at the abstract level the "logically identified" component becomes PWD.

We assume that failures are (i) soft, i.e., no damage to stable storage so that logged records are available after a failure, (ii) fail-stop so that only correct information is logged and erroneous output does not reach users or persistent databases, and (iii) failures are the result of race conditions (timing or "Heisenbugs"), and that replay does not reproduce the failure deterministically.

1.4 Outline of the Paper

Section 2 introduces the notion of interaction contracts between two components and their four flavours, and discusses its ramifications. Section 3 elaborates on the implementation techniques for realizing interaction contracts. Section 4 presents an application case study, while we end with a brief discussion in Section 5.

2. Interaction Contracts

2.1 Components and Interactions

2.1.1 Component Guarantees

Mostly we are concerned about persistent components (**Pcom**'s), for which we guarantee persistent state, i.e., state that survives failures (not simply re-initializing a failed and restarted component as in commercial failover solutions). But a multi-tier application is rarely composed solely of persistent components. We treat other components in their interactions with Pcom's. With transactional components (**Tcom**'s), state and messages are only guaranteed to persist at transaction boundaries. Transaction abort resets Tcom state to the beginning of the transaction, losing intra-transaction messages. Finally, external components (**Xcom**'s) cannot usually provide any guarantees. For example, when prompted for previous input, a user does not necessarily deliver identical input.

Implementing persistence guarantees requires a **log** and a **recovery manager** to capture the order of non-deterministic events. During **normal operation** log entries are created in a buffer for received messages, sent messages, and other non-deterministic events. The buffer is written to a stable log on disk at appropriate points or when full. Also, component state may be periodically "installed" (saved) to disk in an **installation**

Proceedings of the 18th International Conference on Data Engineering (ICDE'02)
1063-6382/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY

point to facilitate log truncation, frequently making log records preceding the installation point unnecessary. A data server can use its own log for this.

During **restart** after a failure, the recovery system scans the stable log. A component is re-incarnated from its last installation point and replayed from there. The recovery system intercepts all messages or other non-deterministic events; information is reconstructed from the corresponding log entry and fed to the component in place of the event. When log entries do not contain message contents, communication with the sender is required to obtain the contents. For this, a contract with the sender ensures that the message can indeed be provided again. Outgoing messages that the replaying component knows (either directly or via inference) have been successfully and stably received prior to a failure, may be suppressed. However, if the component cannot determine this, then the message needs to be re-sent, and the receiver must test for duplicates.

We can now establish an important property that relates the persistence guarantees of a component to the underlying implementation mechanisms.

Theorem 1 (for the proof see [BLSW01])**:** *A component can guarantee a) persistent state as of the time of the last sent message or more recent and b) persistent sent messages from the last installation point up to and including the last sent message if it:*

- *logs all non-deterministic events, such that these events can be replayed*
- *forces the log upon each message send (before actually sending it) if there are non-deterministic events that are not yet on the stable log, and*
- *can recreate, possibly with the help of other components, the contents of all messages received since its last installation point.*

2.1.2 Interaction Contracts

An interaction contract specifies the joint behavior of two interacting components in the presence of failures of one or even both of them. An interaction contract requires each of them to make certain guarantees, depending on the nature of the contract and components. Perhaps only one component can provide strong guarantees, whereas the other component cannot. Different contracts provide flexibility in the design space.

An interaction contract between two components specifies guarantees about a **state transition**. The guarantees are permanent, but log records needed to provide the guarantee can be garbage collected when both components agree they are no longer needed. Such agreements can be set up a priori, for example, by limiting the logging to the last state transition common to the two involved components, or dynamically negotiated.

We consider three types of components as contract partners: **persistent components** whose state should persist across failures, **transactional components**, usually data servers, that provide all-or-nothing guarantees for atomic transactions, and **external components**, which can be used to capture human users who usually do not provide any recovery guarantees.

2.2 Persistent-Persistent Interactions

Persistent components, when they interact with other persistent components, must ensure the persistence of both state and message at each interaction. Committed interaction contracts are used for this purpose.

2.2.1 Committed Interactions

A committed state transition involves a pair of persistent components, whose states persist across system failures. One Pcom sends a message and another receives it. A **committed interaction contract (CIC)** is the fundamental building block for making entire applications persistent and masking failures to users.

Definition 1: A **committed interaction contract** consists of the following obligations:

- **Sender Obligation S1: Persistent State**
  Sender promises that its state at the time of the message or later is persistent.
- **Sender Obligation S2: Persistent Message**
  - **S2a:** Sender promises to send the message periodically, driven by timeouts, until receiver releases it (perhaps implicitly) from this obligation.
  - **S2b:** Sender promises to resend the message upon explicit receiver request until the receiver releases it from this obligation. This is distinct from s2a, typically longer lasting and usually more explicit.
- **Sender Obligation S3: Unique Messages**
  Sender promises that its messages have unique contents (including all header information such as timestamps, http cookies, etc.).

Obligations S1 and S2 ensure that an interaction is **recoverable,** i.e. it is guaranteed to occur (at least once), though not with the receiver guaranteed to be in exactly the same state. Obligation S3, i.e. message uniqueness, is required so that the receiver can detect duplicates (i.e., resent messages) and does not confuse a message with another that happens to have exactly the same content as a previous one (e.g., same shopping cart contents with the same timestamp and the same cookies etc.).

- **Receiver Obligation R1: Duplicate Message Elimination**
  Receiver promises to eliminate duplicate messages (which sender may send to satisfy S2a).

Proceedings of the 18th International Conference on Data Engineering (ICDE'02)
1063-6382/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY

- **Receiver Obligation R2: Persistent State**.
  - **R2a:** Receiver promises that before releasing sender obligation S2a, its state at the time of message receive or later is persistent without the sender periodic re-sending. After S2a release, receiver must explicitly request the message from sender should it be needed. The interaction is **stable**, i.e. it persists (via recovery if needed) with the same state transition as originally.
  - **R2b:** Receiver promises that before releasing the sender from obligation S2b, its state at the time of the message receive or later is persistent without the need to request the message from the sender. After S2b release, the interaction is **installed,** i.e., replay of the interaction is no longer needed.

Note the contract asymmetry: The sender makes a strong immediate promise whereas the receiver merely promises to obey rules in releasing the contract. The sender exposes its current state and "commits" to that state and the resulting message. It doesn't know the implications on other components or, ultimately, external users, that could (transitively) result from subsequent receiver execution. Therefore, the sender must be prepared to re-send the identical message if needed by later recovery and also to recreate its exact same state during replay after a failure.

Each CIC pertains to one message. To fully discharge the CIC may require several messages. Only when the receiver later becomes a sender does it commit itself to the effects of the received message and to the newly sent one, but this involves a new CIC, perhaps with another component, perhaps with the original sender. Before this, receiver forced logging is not required.

Eventual CIC release is essential to free the sender from its obligations. The sender wants to garbage-collect data retained for persistence of previously sent messages, not only in-memory data, but also stable log, periodically truncating it to shorten restart time and reclaim disk space. Once a CIC is released, the sender can discard the interaction data; however, the sender still guarantees the persistence of its own state at least as recent as the interaction. This persistent state guarantee falls out naturally from our implementation techniques.

CIC behavior is depicted as a statechart [HaGe97, UML99] in Figure 1. Ovals show sender and receiver states; transitions are labeled with "event [condition] / action" rules where each element is optional and omitted when not needed. A transition fires if the specified event occurs and its condition is true, then the state transition executes the specified action. For example, the label "/ stability notification" of the receiver's transition from "interaction stable" state into "running" state specifies this transition fires unconditionally (i.e., its condition is "[true]") and its action is sending a stability notification. The sender transition labeled "stability notification" makes the corresponding state change when it receives the stability notification (i.e., when the event "stability notification" is raised). Sender and receiver return to "running" before proceeding further. Unlike two-phase commit, a CIC allows intermediate states for the two components to exist for an extended period, enabling logging optimizations. Note that, for simplicity, we have omitted all transitions for periodic re-sends (e.g., sender's periodic re-send of the message until it receives the stability notification).

While each message has its own contract, the nature of the interactions between two components can enable further optimizations. Request/reply interactions, as in the client-server setting, is an important situation because real protocols are frequently of that form, whether the reply contains application related information or is only an acknowledgement. Consider requestor Q and replier P. The pre-condition for the reply is that Q's state is persistent and that Q will resend the request until P announces the commit of its state that includes the reply. Hence the reply message need not be sent periodically as Q has already committed to receiving the reply (i.e., is synchronously waiting for the reply). P needs only resend the reply on request, which in this case is in response to re-sends of Q.

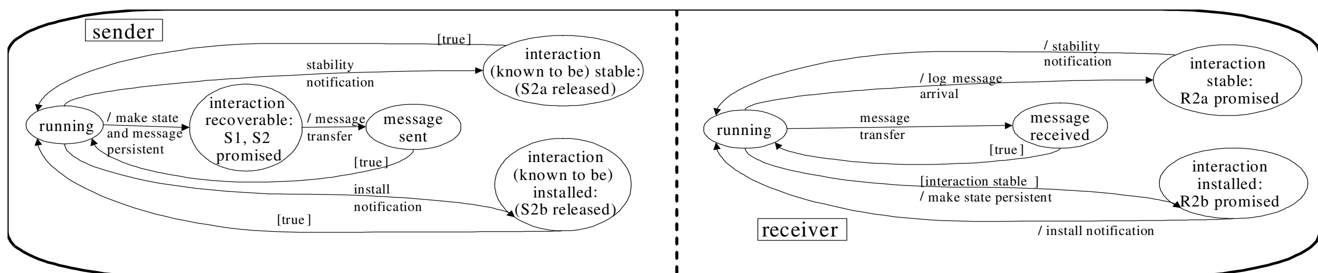

Fig. 1: Statechart for committed interaction contract

Proceedings of the 18th International Conference on Data Engineering (ICDE'02)
1063-6382/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY

2.2.2 Immediately Committed Interactions

Sometimes it is desirable to release a sender from its obligations all at once. This can be useful not only to the sender, as it enables the receiver to recover independently of the sender. This is achieved by strengthening the interaction contract into an **immediately committed interaction contract (ICIC)**.

Definition 2: An **immediately committed interaction** is a committed interaction where sender is released from both message persistence requirements, S2a and S2b, when receiver notifies sender (usually via another message) that the message-received state has been installed, without previously notifying sender that its state is stable. This makes the interaction both stable and installed simultaneously.

An ICIC can be viewed as two CIC's, the first for the original message and the second for a combined stability-and-install notification sent by the receiver. The first CIC requires the sender to make its state persistent, and the second CIC, for the notification sent by the original receiver, requires the receiver to make its state persistent, too. With an ICIC for the entire interaction, the sender waits synchronously for this notification, and the receiver's part in the committed interaction is not deferrable. This is like an optimized two-phase commit involving two participants, the sender as coordinator, making its commit right away, a form of "first agent optimization" (i.e. a "dual" of the well-known last agent optimization [GrRe93, BeNe96]). This a priori commitment is feasible because the sender guarantees that it will re-send the message until the receiver eventually commits the interaction.

With a committed interaction, whether either party requires logging depends on if there are non-deterministic events that need to be made repeatable. If not, then no logging is required, as the interaction is made persistent via replay, including the message contents. With an ICIC, the receiver must make the message contents stable so that its state, which includes the receipt of the message, is persistent without contacting the sender. Thus, an ICIC can be more expensive than a committed interaction both in log forces (when logging is used for message persistence) and in how much is logged (both message arrival and contents).

Because ICIC's always require forced logging by the receiver to immediately install the interaction, they are not always appropriate. It is the avoidance of this cost and its adverse impact on system throughput that makes the simple committed interaction useful. In traditional OLTP, expensive ICIC's have been the method of choice. CIC's substantially reduces the overhead of a recovery contract. ICIC's do, however, ensure independent recovery of the receiver; otherwise the receiver must rely on the sender for recreating the message contents. We discuss such recovery dependencies in more detail in Section 3.3.

2.3 Persistent-External Interactions

Sender and receiver must be Pcom's to engage in committed interactions. External components (Xcom's) may not be persistent, and hence cannot have committed interactions. Importantly, one form of Xcom is a human user. Our intent is to come as close as possible to providing an immediately committed interaction with Xcom's, including users. This leads us to introduce **external interaction contracts (XIC's)**.

Definition 3: An **external interaction contract** is between a Pcom that subscribes to the rules for an ICIC, and an external component, which does not. The impact on the Xcom (or users) is described below.

- **Output Message Send (X1)**. A Pcom (usually a client machine) sends (displays) a message to an Xcom (e.g., external user), after having logged the message send. The sender Pcom crashes before knowing whether the message was seen. Hence it must re-send the message. Because an Xcom might not eliminate duplicates, a user may see a duplicate message.
- **Input Message Receive (X2).** An external user (Xcom) sends a message, via keyboard, mouse, or other input device, to a (client) Pcom. The Pcom crashes before logging the message. On restart, the user must re-send the message. But the user (an Xcom) has not promised to re-send the message automatically, but rather makes only a "best effort" at this. Moreover, the failure is not masked.

The property of interest here is that in the absence of a failure during the XIC interaction, the result of an XIC is an immediately committed interaction that masks internal failures from the external components.

2.4 Persistent-Transactional Interactions

Another contract type covers interactions with a transactional component (Tcom), e.g. data server. These are request/reply interactions, where either a) a request message initiates the execution of a transaction (e.g., invoking a stored procedure) at the server and produces a reply reporting the transaction outcome or b) a sequence of request/reply interactions (e.g., SQL commands) occurs, the first initiating a transaction and the last being the server's reply to a commit or rollback request. The Tcom's state transition is all-or-nothing, but the interaction is not guaranteed to complete. In current practice, the Tcom final reply might not be delivered even though the transaction commits. We require a stronger guarantee. Furthermore, if a transaction

Proceedings of the 18th International Conference on Data Engineering (ICDE'02)
1063-6382/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY

aborts, the Tcom may forget the transaction, which can pose extra difficulties for failure handling at the requestor Pcom. This, frequent and widely accepted, behavior is captured by a **transactional interaction contract (TIC)** between a Pcom, the requestor, and a Tcom, the server that processes the transaction:

***Definition 4*: A transactional interaction contract** between a Pcom and a Tcom consists of the following.

The Tcom promises:

- **Atomic state transition (T1).** The Tcom eventually proceeds to one of two possible states, committing or aborting the transaction (or not executing it, equivalent to aborting). This state transition is persistent.
- **Faithful reply message (T2).** The Tcom's reply message to the Pcom's commit-transaction or rollback-transaction request faithfully reports the Tcom's commit or abort. If a transaction aborts following a sequence of interactions within the transaction, abort is signaled to the Pcom in reply to the next request (e.g., through a return error code).
- **Persistent commit reply message (T3).** Upon commit, the Tcom replies acknowledging the commit request, and guarantees persistence of this reply.

The Pcom promises:

- **Persistent state and commit request message (P1).** The Pcom's commit request and the Pcom's state as of the time in which the transaction reply is expected or later must persist. This guarantee thus includes all earlier Tcom replies within the same transaction (e.g., SQL results, return codes). Persistence of the expected reply state means that the Tcom, rather than repeatedly sending its reply (under T3), need send it only once, perhaps not at all when a transaction aborts. The Pcom asks for the reply message should it not receive it. This persistence guarantee must not depend on the Tcom being able to resend replies.

Guarantee P1 is conditional, applying only for commits, not for aborts. P1 also removes the need for a Tcom to persist earlier messages in the transaction. Guarantee T3, in conjunction with P1 means that the Tcom need only capture the transaction's effects on its database and final commit reply, since earlier messages in the transaction are not needed for Pcom state persistence. Thus the Tcom supports testable transaction status so that the Pcom can inquire whether a given transaction that has a persistent commit request was indeed committed. If the Tcom does not want to provide this testability over an extended time period, guarantee T3 can be implemented analogously to an ICIC with more eager measures by the receiving Pcom.

When a transaction aborts, the only guarantee is the transaction's effect on Tcom state is erased. If the Tcom aborts the transaction or the Pcom requests a transaction rollback, neither messages nor the Pcom's intra-transaction state need persist. There are two cases:

1. When the Tcom fails or autonomously aborts the transaction, the Pcom must re-initiate the transaction, but the Tcom treats this as a completely new transaction, a standard practice in transaction processing.
2. When the Pcom fails in the middle of the transaction, the Tcom will abort (e.g., driven by timeouts for the connection) and forget the transaction. When the Pcom later attempts to resume the transaction, the Tcom will respond with, e.g., a "transaction/connection unknown" return code and the Pcom proceeds as in the first case.

2.5 System-wide Composition of Contracts

We combine bilateral interaction contracts between component pairs into a system-wide agreement that provides the desired guarantees to external users. The key to such a **recovery constitution** is that multi-tier system behavior is based on these different kinds of interactions: internal ones that do not involve user or data server, external ones between a user and a (client) component, and transactional ones between components and data server. Then interaction contracts provide the following general solution:

1. Each internal interaction between a pair of Pcom's has a committed interaction contract (CIC or ICIC).
2. Each external interaction between Pcom and Xcom (user) has an external interaction contract (XIC).
3. Each request/reply interaction from Pcom to Tcom has a transactional interaction contract (TIC).

Note: Tcom's are not allowed to call Pcom's (else such persistent effects might not be undoable, breaching the transactional all-or-nothing paradigm) or Xcom's.

Our recovery constitution allows arbitrary interaction patterns, including, e.g., asynchronous message exchanges, callbacks from a server to a client or among servers, or conversational message exchanges with either one of two components being a possible initiator (e.g., in collaborative work applications). The only restriction is that Tcom's not call Pcom's or Xcom's but only reply to requests from Pcom's. Then, the following very general theorem holds:

Theorem 2 (for the proof see [BLSW01])**:** *Consider an arbitrary graph of message exchange relationships among a set of components with an arc from component A to B if A sends a message that B receives. The graph must have no edges between Tcom's and Xcom's, and the only edges from Tcom's to Pcom's are Tcom replies to Pcom transactional requests. Then the following holds: If there is a CIC (or ICIC) for each pair of*

Proceedings of the 18th International Conference on Data Engineering (ICDE'02)
1063-6382/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

Pcom's that interact directly, an XIC for each message sent or received by an Xcom, and a TIC for each message sent or received from a Tcom, then all failures can be masked with the exception of failures during the last external interaction.

Theorem 2 is the basis for building multi-tier systems with message, component state, and data recovery with failure masking. However, it does not capture important pragmatic issues. It says nothing about *when* CIC or ICIC contracts are released (we implicitly assumed they were never released) and the garbage collection and log truncation at the components. Before we describe this, we must discuss our underlying implementation techniques. We return to this issue in Section 3.

Inability to mask send or receive failures can occur only with a failure during an external interaction. This is possible with any conceivable recovery algorithm without special hardware support. For output messages, if a device has testable state, e.g., an ATM for dispensing cash with a mechanical counter that records when money is dispensed, then output messages (e.g., cash) are guaranteed to be delivered exactly once.

3 Implementing Recovery Contracts

To illustrate Theorem 2, consider a three-tier system, a client and two application server tiers, e.g., a workflow server with whom the client interacts directly and an activity server receiving requests from the workflow server. Assume all components are PWD. Interaction contracts ensure exactly-once semantics. However, these contracts may be implemented in different ways. By treating user input as an external interaction, the client can recreate all its requests (and its own state) to the workflow server (except for a failure during the user interaction). So the CIC's for client requests to the workflow server don't need forced logging. The workflow server needs to log client request order, and make sure it is stable before sending requests to the activity server. The workflow server can enforce its CIC's for both requests to activity server and replies to client without explicit measures by itself. Requests can be recreated by deterministic replay, with client requests re-obtained from the client. To re-create replies to the client the workflow server relies on the activity server for activity server replies. Finally, the activity server needs forced logging for its CIC's when sending replies to the workflow server because call order to the activity server is non-deterministic.

Note that interaction contracts and implementation measures are separate levels of abstraction in our framework. It is possible to set up strong contracts, in the sense of Theorem 2, for all bilateral interactions while implementing some of them with little or no overhead. Indeed, there are many potential ways for a collection of components to support CIC's. Here we describe one such way to do this.

3.1 Log Management

Each component needs its own log. The issues for normal operation are what to log, when to force the log, and how to minimize overall overhead of logging.

3.1.1 Data Servers

Data servers have the hardest logging requirements. They are usually heavily utilized, support many concurrent "users", maintain valuable data, and are carefully managed for high availability. When an application interacts with a data server, the data server constructs a session for it. When there is inter-transaction state (including perhaps control state), we regard this session as a Pcom maintained by the data server. It is subject to the usual events, deterministic and non-deterministic, related to the sending and receiving of messages. A session component indirectly interacts with other session components via a potentially non-deterministic sequence of data accesses mediated by a data component (a Tcom). If there is no session state, but only accesses to data, only the data component need exist.

We partition our persistence requirements into four elements: data component state, session component state, received messages, and sent messages.

Data Component State: Data servers log entries for updates of persistent (database) data in physiological, physical or logical form [Mo92, GrRe93, BeNe96, LoTu99], to provide persistent data. The data component for a database system is always a Tcom. In addition to logging for persistent data, the data component needs to log the final reply message for a caller's commit-transaction request, and the server log needs to be forced before sending this final reply. For aborted transactions no log forcing is necessary.

Session Component State: We also maintain persistent state for the session components when that state persists across transactions. SQL session state such as cursors or temporary tables can span transaction boundaries. This server maintained state is covered by interaction contracts. Phoenix/ODBC [BLAB00, BL01] persisted this state to provide persistent sessions.

A program executing in a session, e.g. stored procedure, need not persist if it lives entirely within a transaction. When it lives across transactions, e.g. a multi-transaction stored procedure, replay makes it persistent, via interaction contracts. During restart after a server failure, incomplete requests (interactions with the data component) must be replayed without altering previ-

Proceedings of the 18th International Conference on Data Engineering (ICDE'02)
1063-6382/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

ously committed data changes. This is done by message logging as described below. Optimizations exploiting the fact that all data server components share the same log manager are also possible.

Session Received Messages: Asynchronous message receives require logging (but no log forcing), logical logging being sufficient for CIC interactions. Logical log entries capture non-deterministic interleaving and uniquely identify sender and message, but do not contain message contents. Other "received" events need to be logged, too, log entries depending on the type of event (e.g., reading the system clock (an Xcom) requires logging its time).

Session Sent Messages: Data servers need to recreate sent messages. Logging for this can be either physical, including message contents, or logical. Messages can be treated like any other effect of request execution. CIC's require, however, that the server force its log to include the (chronologically ordered) log records that ensure the persistence of a sent message before actually sending the message.

3.1.2 Application Servers and Clients

The advantage of CIC's versus ICIC's in reducing recovery overhead shows up clearly with application servers and clients. For these components, often (but not necessarily) the only non-determinism is the result of user input or data server interactions. Further, these components usually have little reason for using ICIC's. What such components need to do for a CIC is to guarantee that replay will recreate their state and sent messages. In the absence of non-determinism, this is frequently possible without forcing the log at interactions between system components. Only user interactions need to be force-logged as external interactions.

For interactions with data servers (i.e., Tcom's), Pcom's (application servers or clients) must ensure state persistence as of the commit-transaction request. If the transaction consists of a sequence of request/reply interactions, the Pcom needs to create log entries for the replies and its commit-transaction request and force the log before sending the commit request. Otherwise (i.e., for transactions with a single invocation request, e.g., to execute a stored procedure, and single reply) no forced logging is needed, unless the commit request is preceded by non-deterministic events that have to be tracked. If the Pcom issues a rollback request no force logging is needed. The Pcom needs application logic for aborted transactions anyway.

Logging or installation points are needed because components must eventually release each other and data servers from the committed interaction requirement to resend messages upon request. But this is not forced logging, and a single application thread state installation or log write can serve to release contracts involving many committed interactions.

3.2 Component Restart after Failure

After a failure, each Pcom performs local recovery that re-incarnates the component at its last installation point and replays the component from there. The log is scanned in order to recreate persistent data and component state. To recreate component state, non-deterministic events are replayed from the log and the appropriate information, reconstructed via recovery, is fed to the component. This information can be from the local log, or requested of other components. The component is re-executed between message receives. All Pcom's use this procedure.

After recovery, a Pcom resumes normal operation. Part of this is periodic resend of committed-interaction messages that a receiver has not yet made stable. For a stable interaction, the message only needs to be resent when the receiver explicitly asks for it, so it needs to continue to be available. For an installed interaction (an ICIC is promptly installed), no action is needed, as the message contents are stable at the receiver.

A component may receive messages from other components that are resends of messages received before its failure (in its prior incarnation). There are two cases:

1. The restarted component finds a log entry for a message. It asks the sender to deliver the message again if waiting for a spontaneous resend takes too long.
2. The component doesn't find a log entry for the message. It restarts as if that message was never received. The message is treated as a new message. This works because the component cannot have committed its state (with the message receive) to other components, else a log force would have put the message on the log.

3.3 Recovery Independence

With complex multi-tier systems that span autonomous organizations, it is crucial that components can recover independently of other, potentially less reliable or untrusted components. These considerations lead to two notions of independent recovery as discussed below.

3.3.1 Isolated Recovery

To avoid one component's recovery forcing a second component to perform an expensive recovery when the second has not failed, we want **"isolated"** component recovery, i.e., no cascading restarts typical of many "optimistic" fault-tolerance algorithms [AlMa95, EJW96]. Interoperating components providing cross-organizational e-services are largely autonomous, and cascading restarts are absolutely unacceptable.

Proceedings of the 18th International Conference on Data Engineering (ICDE'02)
1063-6382/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

Nonetheless, an isolated component must resend messages as long as its contracts are not released. A solution is the volatile **message lookup table (MLT)** [LoWe98] that stores in main memory all uninstalled sent messages. These messages can be resent without component replay or reading the log. The MLT is reconstructed during recovery if the component fails; so it is always present during normal execution. Should the MLT become too large, we shrink it by replacing some (the oldest or longest) messages by their positions in the log. This is safe as the corresponding log entries can be obtained from the stable log, albeit at higher cost.

3.3.2 Autonomous Recovery

A (server) component wants to avoid communicating with and thus depending on other components during its recovery. This **autonomous recovery** [LoWe98] for the server in a client-server setting can be generalized to **component ensembles**. A component of an ensemble may rely on ensemble components, but wants to be autonomous of outside components. One example ensemble is a data server and application server at an e-commerce site, clients being outside the ensemble.

Autonomous recovery avoids having to ask that messages be resent from outside components for successful ensemble restart. This is achieved by using ***immediately committed interactions*** **(ICIC's)** for all messages received from such components. Messages within the ensemble need no force-logging when there is no non-determinism in the interactions. Should an ensemble component fail, it relies on other ensemble components to resend messages, but not on outside components. This requires a log force only upon the next message sent to an outside component. ICIC's for received messages require two forced log I/Os for every interaction.

3.4 Garbage Collection

Garbage collection is critical for server components, which need to discard information from the MLT to reclaim memory and reclaim log space for fast restart and thus high availability. Contracts with other components can hamper garbage collection. Therefore, another facet of component autonomy is to ensure that log and MLT entries kept on behalf of other components can be dropped within reasonable time. Each kind of log record has its own truncation point.

1. To recover component state, only log entries for messages and non-deterministic events that follow the most recent installation point are needed. To advance this truncation point, one performs another installation point for the component's state.
2. Log entries for data updates can be discarded which have LSN's (i.e., log sequence numbers [GrRe93, BeNe96]) less than the minimum of the LSN of the oldest update that has not yet been written back from the cache to disk and the LSN of the oldest update of all active transactions. A technique for advancing this minimum LSN is to flush the data pages with the oldest dirty updates from the cache.
3. Log entries for MLT entries kept to honor CIC's with other components (for possible recovery of these other components) can be discarded up to the oldest of log records for messages not yet (known to be) installed. To advance this truncation point requires asking other components to force their log or create an installation point. Once these actions are taken and our component receives an acknowledgement (i.e., install notification), it can garbage-collect the information. If autonomous garbage collection is desired, then the component should only use ICIC's.

The log can only be truncated up to the earliest of the truncation points. Often, this log entry can be copied forward in the log, though required interleaving with other log records must be preserved. However, "live" messages are only used to recover the MLT. We need only ensure that the original LSNs and message sequence numbers are kept in the log entries themselves.

Receivers usually release CIC's fairly continuously, periodically taking installation points and forcing the log. These events can be signaled lazily to senders. A low-cost technique is to piggyback on the next message to a sender a message sequence number of the oldest still uninstalled message from the sender. Other techniques can be based on predefined agreement, interaction patterns, or session boundaries. For example, end-of-session notification (perhaps via session time-out) might mean releasing the contracts for all session messages. Sometimes the next request from the same component could be an implicit form of such a release.

3.5 Phoenix/COM+

We are currently implementing the recovery guarantees framework as part of the Phoenix project on robust applications, in a system we call Phoenix/COM+. In Phoenix/COM+ we integrate interaction contracts into the Microsoft Component Object Model (COM) runtime environment, allowing programmers to build persistent component-based applications without requiring modifications to their applications.

With Phoenix/COM+, a programmer registers component classes as PCOM's or TCOM's. At runtime, applications are embedded into the COM+ runtime and an interceptor captures method calls and returns between components. Calls and returns between PCOM's are treated as CIC's, between PCOM and TCOM as TIC's, and between a PCOM and any other component (XCOM) as XIC's. Phoenix/COM+ flushes log records as necessary, and coordinates log truncation. In the

Proceedings of the 18th International Conference on Data Engineering (ICDE'02)
1063-6382/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

event of a failure, Phoenix/COM+ masks the error and automatically recovers failed components from log records using redo recovery. The result is a persistent component-based application, without any special application code or operator intervention. While our current implementation of the recovery guarantees framework is COM+ specific, the techniques are relevant to other runtimes, such as CORBA or Java Beans.

4 Application Scenario

The following application scenario demonstrates the benefits of our framework. The application is a multi-tier travel services e-service, e.g. Expedia or Travelocity. The system architecture, illustrated in Figure 2, can be characterized as a four-tier system with clients using Internet browsers, two tiers of application servers in the middle, and a suite of backend data servers.

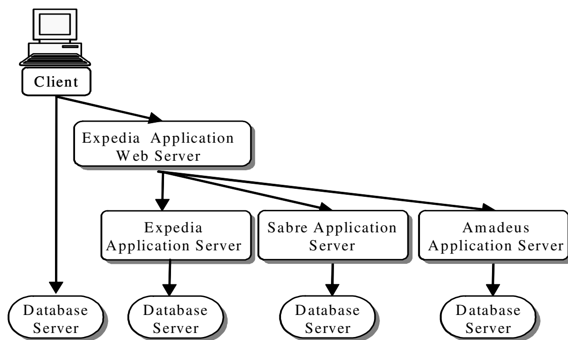

Fig. 2: Components in electronic travel service.

A client sends a travel request to the Expedia application server. The client, whose state is extended via cookies or applets for personalization, e.g., frequent flyer numbers, forwards such information to the Expedia web server, which may persist it at a data server. The web server runs workflow-style servlets on behalf of client requests. It hosts business logic and maintains itineraries for users. It keeps user state spanning conversational interactions with the client for the duration of a user session, typically using session objects to hold shared data on the web server. To query flight fares, hotel rates, etc., the web server interacts with application servers, including servers of autonomous travel companies with their own backend data servers, e.g., Amadeus and Sabre. One of the application servers is an integrated part of Expedia, again a server running servlets that communicate with a database for long-term information about customers. The client interacts with a data server to store user information such as credit card numbers, e.g. at a service such as Passport.

Client and Expedia components, web server and application server, and Amadeus and Sabre application servers are Pcom's, and data servers are Tcom's. Non-determinism resulting from Amadeus or Sabre interactions is captured via ICIC forced logging. But messages leading up to a purchase that are directed to the Expedia application server are treated as CIC's, and do not require forced logging. Queries to the Expedia data server are treated as TIC's. Interaction contracts for our e-service are set up as follows:

[**user⇔client**] The client handles user input and output with XIC's, promptly force logging. Current Internet browsers do not provide native support for logging, but could be enhanced through a plug-in or an applet.

[**client⇔data server**] Interactions between client and data server are TIC's. The data server commits modifications to the database when sending its final reply to the client, and force logs this final reply message.

[**client⇔Expedia web server**] Between client and upper tier web server, client request and server reply are handled with CIC's. No forced logging is required as client XIC logging captures all non-determinism.

[**Expedia web server⇔Expedia application server**] Expedia web server and application server requests and replies are CIC's. Forced logging isn't needed. Client XIC logging captures all non-determinism.

[**Expedia web server⇔external application server**] Between Expedia web server and external application servers, ICIC's, with forced logging by both servers, are used to capture the potential non-determinism as these external servers are autonomous.

[**application server→data server**] Requests from application server to data server are transactional, and require a TIC. A commit request exposes the effects of application server execution via changes to data server state, and hence this state must persist. However, since prior ICIC's with Expedia server or client have captured all non-determinism already, forced logging is not required. Being able to recreate the application server's request, so as to provide exactly-once semantics, is exactly what is missing in many of today's e-services.

[**data server→application server**] A data server commits modifications to a shared database when sending its final reply to the application server, exposing changes to other application servers. Thus the TIC requires a persistent reply message. This final reply (i.e., for the SQL "commit work") must be forced logged, which also captures the committed database changes.

The contracts above are necessary for system-wide recoverability. The data server may also require garbage collection and independent recovery. Specifically, the data server can treat its transaction ending reply to the application server as an ICIC so that it can discard messages once it knows that the application server has received them, and hence truncate its log at its discretion.

The number of forced log writes, which we characterize below, dominates the cost of our protocols. Let the user

Proceedings of the 18th International Conference on Data Engineering (ICDE'02)
1063-6382/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY

session consist of *u* input messages and *u* output messages, and let the client generate *one* request to its local data server and *x* requests to the Expedia server for each user's input message. In turn, Expedia will create *y* requests per incoming request to each of the three application servers, and let each of the external application servers creates *z* requests per incoming request to its local data server. Under these assumptions, techniques based on pessimistic message logging require $2u + 4u + 4ux + 12uxy + 12uxyz$ forced log writes. Our protocol, using XIC's between user and client, TIC's between client and its local data server, CIC's between client and Expedia, ICIC's between Expedia and external application servers, and TIC's between external application servers and their local data servers, would require only $u + u + 0 + 12uxy + 3uxyz$ forced log writes, a saving of $4u + 4ux + 9uxyz$ disk I/Os.

5 Concluding Remarks

We have developed a framework for recovery guarantees in general multi-tier applications. Our notion of committed interaction contract with exactly-once execution and failure masking is particularly useful for e-services, which exhibit many idiosyncrasies due to failures that are exposed to users. Interaction contracts avoid these problems and can contribute to the construction of more dependable e-services. For practical viability, it is important that our protocols have been designed to minimize logging overhead.

Our framework applies to a spectrum of multi-tier e-services, whether underlying components are database systems, mail servers, message queues, workflow servers, or other kinds of specialized application component. Our key contribution of recovery guarantees for persistent components complements and is orthogonal to the more established notion of transactional component. We have shown how to orchestrate persistent and transactional components into a system. We have started work implementing of recovery guarantees as a runtime service for the COM+ middleware environment [Kirt97], in a project that we call Phoenix/COM+.

References

[**AlMa95**] L. Alvisi, K. Marzullo: Message Logging: Pessimistic, Optimistic, and Causal, Int'l Conf. on Distributed Computing Systems, 1995.

[**BLSW01**] R. Barga, D. Lomet, G. Shegalov G. Weikum.: Recovery Guarantees for Internet Applications. Submitted for publication (2001).

[**BLAB00**] R. Barga, D. Lomet, S. Agrawal, T. Baby: Persistent Client-Server Database Sessions, EDBT'2000.

[**BL01**] R. Barga, D. Lomet: Measuring and Optimizing a System for Persistent Database Sessions, ICDE'2001

[**Ba81**] J.F. Bartlett: A NonStop Kernel, SOSP'81.

[**BHM90**] P. Bernstein, M. Hsu, B. Mann: Implementing Recoverable Requests Using Queues, SIGMOD'90.

[**BeNe96**] P. A. Bernstein, E. Newcomer: Principles of Transaction Processing, Morgan Kaufmann, 1996.

[**Bo89**] A. Borg, W. Blau, W. Graetsch, F. Herrmann, W. Oberle: Fault Tolerance under UNIX. ACM TOCS 7,1, 1989.

[**Cr91**] F. Cristian: Understanding Fault-tolerant Distributed Systems. CACM 34, 2, 1991.

[**EJW96**] E.N. Elnozahy, D.B. Johnson, Y.M. Wang: A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report, Carnegie-Mellon University, 1996.

[FBHS01] X. Fu, T. Bultan, R. Hull, J. Su: Verification of Vortex Workflows, Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems, 2001.

[**FCK87**] J. C. Freytag, F. Cristian, B. Kähler: Masking System Crashes in Database Application Programs, VLDB'87.

[**FrGu00**] S. Frolund, R. Guerraoui: Implementing e-Transactions with Asynchronous Replication, Int'l Conf. on Dependable Systems and Networks, 2000.

[**GrRe93**] J. Gray, A. Reuter: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993.

[**HaGe97**] D. Harel, E. Gery: Executable Object Modeling with Statecharts, IEEE Computer 30, 7, 1997.

[**HuWa95**] Y. Huang, Y-M. Wang: Why Optimistic Message Logging Has Not Been Used In Telecommunications Systems. FTCS'95.

[**JoZw87**] D. B. Johnson, W. Zwaenepoel: Sender-based Message Logging. FTCS'87.

[**Kim84**] W. Kim: Highly Available Systems for Database Applications. ACM Computing Surveys 16. 1.

[**Kirt97**] M. Kirtland: Object-Oriented Software Development Made Simple with COM+ Runtime Services. Microsoft Systems Journal, 12, Nov. 1997.

[**Lo98**] D. Lomet: Persistent Applications Using Generalized Redo Recovery. ICDE'98.

[**LoWe98**] D. Lomet, G. Weikum: Efficient Transparent Application Recovery in Client-Server Information Systems, SIGMOD'98.

[**LoTu99**] D. Lomet, M. Tuttle: Logical Logging to Extend Recovery to New Domains, SIGMOD'99.

[**Ma99**] P. Maes, R. Guttman and A. Moukas. Agents that Buy and Sell: Transforming Commerce. CACM, March 1999.

[**MaRa99**] C.P. Martin, K. Ramamritham: Recovery Guarantees in Mobile Systems, ACM Int'l Workshop on Data Engineering for Wireless and Mobile Access, 1999.

[MaRa01] C.P. Martin, K. Ramamritham: Guaranteeing Recoverability in Electronic Commerce, 3rd Int'l Workshop on Advanced issues of E-Commerce and Web-Based Information Systems, 2001.

[**Mo92**] C. Mohan, et al: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollback Using Write-Ahead Logging, TODS 17, 1, 1992.

[**OMG00**] Object Management Group: Fault Tolerant CORBA Spec, http://cgi.omg.org/cgi-bin/doc?ptc/00-04-04

[**SPS00**] H. Schuldt, A. Popovici, H-J. Schek: Automatic Generation of Reliable E-Commerce Payment Processes, 1st Int'l Conf. on Web Information Systems Engineering, 2000.

[**Ty98**] J. D. Tygar: Atomicity versus Anonymity – Distributed Transactions for Electronic Commerce, VLDB'98.

[**UML99**] OMG Unified Modeling Language (UML) Version 1.3, http://www.rational.com/uml.

[**WV01**] G. Weikum, G. Vossen: Transactional Information Systems. Morgan Kaufmann, 2001.

Proceedings of the 18th International Conference on Data Engineering (ICDE'02)
1063-6382/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY