Recovery Guarantees for Internet Applications

ROGER BARGA and DAVID LOMET
Microsoft Research
and
GERMAN SHEGALOV and GERHARD WEIKUM
Max-Planck-Institut für Informatik

Internet-based e-services require application developers to deal explicitly with failures of the underlying software components, for example web servers, servlets, browser sessions, and so forth. This complicates application programming, and may expose failures to end users. This paper presents a framework for an application-independent infrastructure that provides recovery guarantees and masks almost all system failures, thus relieving the application programmer from having to deal with these failures—by making applications "stateless." The main concept is an interaction contract between two components regarding message and state preservation. The framework provides comprehensive recovery encompassing data, messages, and the states of application components. We describe techniques to reduce logging cost, allow effective log truncation, and permit independent recovery for critical components. We illustrate the framework's utility via web-based e-services scenarios. Its feasibility is demonstrated by our prototype implementation of interaction contracts based on the Apache web server and the PHP servlet engine. Finally, we discuss industrial relevance for middleware architectures such as .Net or J2EE.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; C.4 [**Performance of Systems**]—*Fault tolerance, Reliability, availability, and serviceability*; H.2.4 [**Database Management**]: Systems—*Transaction processing*

General Terms: Design, Management, Performance, Reliability

Additional Key Words and Phrases: Exactly-once execution, application recovery, interaction contracts, communication protocols

1. INTRODUCTION

1.1 Motivation and Problem Statement

Internet-based e-services, ranging from B2C (business-to-consumer) e-commerce and B2B (business-to-business) supply chains to advanced services

Authors' addresses: R. Barga and D. Lomet, Microsoft Research, One Microsoft Way, Redmond, WA 98052; email: {barga,lomet}@microsoft.com; G. Shegalov and G. Weikum, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, Saarbrüken, FRG; email: {shegalov,weikum}@mpi-sb.mpg.de.

such as web-based auctions, tele-teaching, collaborative authoring, or electronic tax preparation, require a sophisticated software infrastructure that includes user browsers, web application servers with servlet engines, workflow engines, and backend database servers [Debull 2001]. Since all of these components are failure-prone, application developers have to take measures for dealing with failures. Database recovery does not mask failures to applications and users. Transaction atomicity guarantees all-or-nothing but not **exactly-once execution**, where the latter means 1) no output to the user is duplicated (to avoid confusion), 2) the user provides input only once (to avoid irritation), and 3) the user intent, for example, buying airline tickets, is carried out exactly once. To achieve this exactly-once behavior, application programs need explicit code for retrying failed transactions. Often such code is incomplete or missing, exposing failures to users. Or even worse, a failure occurs with no notice provided. For an e-commerce service, for example, this can lead to user inconvenience and lost sales when this happens during shopping cart checkout. However, the application program or 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.

The current approach to coping with these problems is to combine explicit failure handling code with the notion of stateless applications. This puts a major burden on the application programmer, and generally hampers application development productivity and maintainability. Rather it is much more desirable to factor failure handling out of the applications and provide a generic solution for "universal recovery" in the surrounding run-time infrastructure. This way application programs could be written as if there were no failures in the Internet and the components of an e-service.

TP monitors, exploiting transactional message queues or a database, and "stateless" applications, have long been the preferred solution for failure-resilient business applications. Stateless applications are not without state, however. Rather, they are written so that each application step executes entirely within a transaction. A step begins by retrieving prior state from a queue or database, executes, and then stores the resulting end state into a queue or database. Thus, such an application does not have "control state" outside of a transaction, though of course, the state saved in the queue or database lives across transaction boundaries.

With the advent of web-based e-services, however, prior solutions have not been fully carried over to multitier web architectures. Indeed, how they might be adapted to deal with middle-tier web application servers (e.g., Apache/Tomcat or IIS) is complex and still speculative. 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. Another example is one-stop shopping services that

provide a unified view of a diversity of individual e-commerce sites. A somewhat futuristic e-service of this kind would be an integrated real-estate purchase service that automatically handles the entire workflow for the necessary steps with notaries, land registry offices, banks, insurances, and so on.

For multitier Internet applications with communicating components, a comprehensive form of data, component state, and message recovery is needed, going beyond traditional database recovery. Designing protocols to accomplish this entails a number of issues:

- To what extent can failures be masked to the end users, and which logging actions are necessary to this end?
- Which component logs which messages or state to be recoverable, mask failures, and provide exactly-once semantics to a user?
- How are logs managed, when is a log force written to disk, and how are logs coordinated for log truncation, crucial for fast restart and thus for high availability?
- How are critical components, for example, database servers, kept from being "hostage" to other components (applications servers, clients) that may hamper or block their independent recovery or normal operation?

1.2 Our Framework for Recovery Guarantees

We have developed a framework for multitier applications in the context of the Phoenix project [MSR PHOENIX] that answers the above questions [Barga et al. 2002]. It masks from users all failures of clients, application servers, or data servers (e.g. database servers) such that all user requests can have exactly-once semantics (see Subsection 1.1). We identify the logging required for nondeterministic events 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 prefailure incarnation.

Our framework exploits **interaction contracts** between two components. For example, a committed interaction contract between persistent components requires sender and receiver guarantees to ensure that the interaction persists at both components across system failures. There are also contracts for persistent component interactions with external components (including users) and transactional components, which provide all-or-nothing state transitions (but not exactly-once executions). The bilateral contracts are composed to make persistent components provably recoverable with exactly-once execution semantics.

We separate contract obligations from their implementation in terms of logging. As a result, many internal interactions can avoid forced logging or other expensive measures. We present implementation techniques to a) minimize logging cost, especially forcing the log to disk, b) allow effective log truncation to limit restart cost and thus provide high availability, and c) permit independent recovery of certain components.

We have built a prototype implementation of our framework as a proof of concept. Our prototype directly implements interaction contracts for an

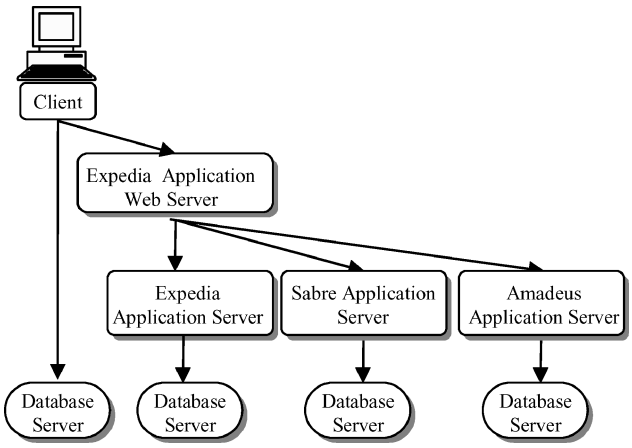

Fig. 1. Components in a four-tier electronic travel service.

application server based on Apache and PHP, via changes to the PHP interpreter. It provides browser extensions to enable browser session state to also be included in our framework, thus extending our recovery guarantees to the client desktop. The result is a system that can provide an end-to-end exactly-once execution guarantee, successfully recovering from and masking system failures from the user. And it achieves this without the application programmer needing to take any explicit programming steps to deal with failures.

1.3 Sample Scenario

We illustrate here how our framework deals with an advanced multitier e-services scenario, the travel services provided by companies such as Expedia.com and Travelocity.com. The system architecture, illustrated in Figure 1, 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 database servers.

A client sends a travel request to an upper-tier Expedia web server. The client, whose state is extended via cookies or applets for personalization (e.g., frequent flyer numbers, etc., which could be stored locally at a client-side database or file server), sends the request and related personalization information to the Expedia web server. The web server runs workflow-style servlets in sessions on behalf of client requests. This level hosts business logic and is in charge of building and maintaining tentative itineraries for users' travel plans. To this end, it keeps user state that spans conversational interactions with the client for the duration of a user session, typically using session objects whose only job is to hold shared data on the web server. For querying flight fares, hotel rates and availability, and so on, the web server interacts with lower-tier application servers. These include servers operated by autonomous travel companies with their own application servers and database servers, for example, Amadeus and Sabre. One lower-tier application server is an integrated part of

Expedia, again a server running servlets that communicate with a database for long-term information about customers.

In this scenario, client, Expedia web server, application server, Amadeus and Sabre application servers, and database servers all support sessions that retain state. These sessions are established as a result of user initiation. While servers are typically multi-threaded, the sessions they support are single-threaded. Thus, all nondeterminism (outside of data accesses within database servers, which is captured by database logging) resides in the interactions between sessions. Our contracts are targeted precisely at these interactions. Each component in this picture provides guarantees to the other components via our framework. This permits them all to be jointly recoverable should a system failure interrupt execution of a user request. And, importantly, this recovery ensures that a user will see exactly-once execution of his request. We return to this application scenario in Subsection 5.4, showing the excellent performance of the framework in providing its guarantees.

The rest of the article is organized as follows. Section 2 provides background on transactional recovery, as used by database systems and TP monitors, and discusses related work. Section 3 introduces the system architecture and computational model that underlie our work, and it provides an overview of the key concepts of our approach. Section 4 introduces the fundamental notion of interaction contracts between two components and their four flavors, and discusses its ramifications. Section 5 elaborates the implementation techniques for realizing interaction contracts, and provides a performance analysis for our sample application scenario. Section 6 describes our prototype implementation of interaction contracts for an application server based on the Apache web server and the PHP servlet engine. To emphasize the industrial relevance of this work we discuss, in Section 7, its applicability to widely used middleware architectures such as .Net or J2EE. Section 8 provides our conclusions.

2. BACKGROUND AND RELATED WORK

2.1 Recovery Basics

In database systems and TP (transaction processing) monitors, which form the backbone of most mission-critical business applications (both traditional client-server as well as Internet-based multi-tier), recovery after hardware and software failures is based on the fundamental notion of an **atomic transaction** [Gray and Reuter 1993, Weikum and Vossen 2001]. Transactions guarantee all-or-nothing **atomicity** in the sense that a failure in the middle of an ongoing transaction does not leave any traces and **persistence** in the sense that the effects of a committed (i.e., successfully completed) transaction will survive future failures. These system-provided guarantees simplify the task for application developers whose code simply needs to specify the begin and end of a transaction during program execution.

Atomicity and persistence are usually implemented by **logging** data modifications on stable storage, typically, one or more dedicated disks. For efficiency, database updates in the server's database page cache are not written

to disk immediately; rather log entries are generated that contain sufficient information for either undoing a modification, if the transaction aborts or a failure occurs before the corresponding transaction's commit, or redoing it, if the transaction commits and a failure occurs afterwards. The implementation may choose to record the state of a data object before and after the modification, or may merely record the change to the object in a log entry. Care must be taken to avoid excessive disk I/O for writing log entries onto disk. To this end, log entries are first collected in a log buffer and the log buffer is written to disk only when it is necessary for recoverability (e.g., for a transaction's commit log entry). When correctness requires that a log entry be written, it is referred to as a **forced log write**, otherwise as a non-forced log write.

During recovery after a failure, log entries are sequentially read from the stable log file. In a **redo pass,** all updates that are not reflected in database pages on the disk are redone by replaying log operations. Then an **undo pass** considers all log entries that belong to incomplete transactions and removes their effects. These procedures are complicated by the fact that, when reading a log entry, the recovery method does not know whether this update is already contained in the stable database. This uncertainty is a problem because updates and the corresponding redo/undo steps are not necessarily idempotent; redoing an update that is already in the database or redoing a lost update twice may lead to incorrect data. To handle these situations, the recovery method implants **log sequence numbers** (**LSN**s) in both the log entries and the data objects to be recovered. The LSN of an object (e.g., in the header of a disk page) tracks the state of the object and provides **testable state** to the recovery method: a simple comparison of the LSNs in a log entry and the corresponding data object tells us whether the log entry's operation should be executed or not.

For **fast restart** (and hence high availability) the recovery system limits the number of log entries to be read from the stable log file and the number of data objects to be fetched from disk during recovery. Database systems periodically **truncate** the log based on bookkeeping that identifies which pages may require recovery (as opposed to those that are known to be up-to-date in the disk-resident database). Some of this bookkeeping information is periodically written **asynchronously** to the log in a **checkpoint**. This does not require writing any cached data pages onto disk, rather it is combined with a background process that writes back modified pages in an asynchronous and I/O-optimized manner. This method allows continuous log truncation, a particularly effective way of garbage collecting unneeded log entries.

Figure 2 depicts the key components that underlie an efficient data recovery method as discussed above. In the figure b, q, and z denote database pages, t_{17}, t_{19}, and t_{20} are transactions, and the other numbers are LSNs.

2.2 Application Recovery

The database recovery approaches outlined above provide an efficient, industrial-strength solution for many systems requiring data recovery: file systems, mail servers, and so on. However, they do not immediately carry over to recovering applications after failures. In an Internet-based three-tier

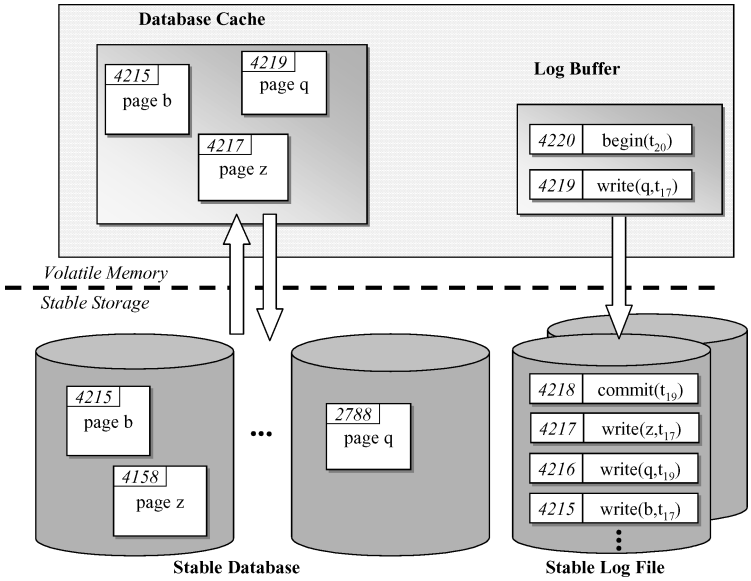

Fig. 2. State-of-the-art data recovery.

(or a simpler client-server) application, with an Internet browser as a client, an HTTP server and a servlet engine at the middle tier, and a database system as backend, database recovery only ensures the consistency of the data at the backend. It is then up to the application programs running at client and middle-tier server to test and handle return codes that may signal a transient failure (e.g., a code like "unknown transaction" from the database system), to reinitiate requests, and, most critically, to ensure idempotence by implementing some form of testable state. To simplify the implementation of robust applications requires that failures be masked from application programs (and also from end users). This requires a more comprehensive approach to overall application recovery.

The most successful technique for application recovery in mission-critical situations uses **queued transactions**, invented for OLTP (online transaction processing) [Bernstein et al. B. 1990, Gray and Reuter 1993, Bernstein and Newcomer 1996, Weikum and Vossen 2001] and supported by most TP monitors (e.g., IBM MQ Series, BEA Tuxedo, Microsoft MTS) and associated web application servers (e.g., IBM WebSphere, BEA WebLogic, Microsoft IIS). Messages between clients, the TP monitor acting as an application server, and the database server are held in **transactional message queues** whose operations, enqueuing or dequeuing messages, are embedded in an atomic transaction with its all-or-nothing guarantee. The queue manager usually is a separate resource manager with its own log and needs to support the two-phase commit protocol for distributed transactions that access both messages in queues and permanent databases. An application program can then read an input message from a queue, process the message by querying and updating one or more databases, and finally place a reply message into a queue—all in one atomic transaction.

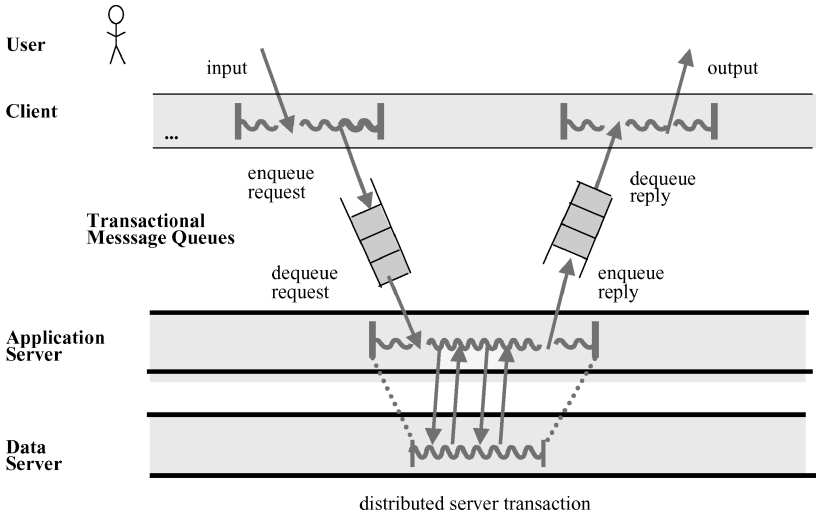

Fig. 3. State-of-the-art message and process recovery based on transactional queues.

Should the database transaction abort, the message that was originally dequeued from the input queue is automatically placed back into the queue (based on log entries that undo the transaction), so that the input message will eventually be processed even after an arbitrary number of failures. Further, the message will be processed exactly once (for the commit of the transaction also commits the dequeueing of the input message and the enqueuing of the output message). This principle is illustrated in Figure 3.

While queued transactions have the desired exactly-once execution guarantee, they also have severe disadvantages that make them unattractive as a general-purpose solution. The queue-based message interaction between client and e-service involves three transactions per user request: to enqueue the request on the queue; to dequeue it, process it in the database server, and enqueue the reply; and to dequeue the reply. This incurs the high overhead of **forced-logging for three commits**. Even more importantly, queued transactions require **"stateless"** applications where the only application state between transactions is in a database or queue. For a multi-step application "session", all information that needs to be kept across session steps is either kept in the database or encoded in the messages that are sent back and forth between client and application. Both options are expensive in terms of disk I/O. The second option resembles the web programming technique of maintaining session-related information in cookies, except that cookies are a much less reliable intermediate store than transactional queues.

The queued transaction approach, with its shortcomings, may be acceptable for some e-commerce services but it can pose substantial difficulties. It requires significant programming effort and thus high costs to cast rich stateful applications into this stateless paradigm.

There is some prior work on masking failures and providing recovery for **stateful** applications, but only in limited contexts. The papers Freytag et al.

[1987], Lomet and Weikum [1998], Barga et al. [2000] are focused only on client-server systems, and do not consider multi-tier Internet applications. Other work is restricted to applications embedded in the database server, like stored procedures [Lomet 1998]. It is not obvious how these protocols can be generalized to apply to multi-tier systems. The notion of interaction contracts, developed in the current paper, is the key for this generalization.

Recovery for general systems of communicating processes has been extensively studied in the fault-tolerance community (e.g., Johnson and Zwaenepoel [1987], Cristian [1991], Alvisi and Marzullo [1995], Elnozahy et al. [2002]), where the main focus has been to avoid losing too much work in a long-running computations (e.g., scientific applications), usually using distributed checkpointing. Most of this work does not mask failures. Methods that do mask failures exploit "pessimistic logging" (see, e.g., Huang and Wang [1995]), with forced log I/Os at both sender and receiver on every message exchange. More expensive techniques, such as process checkpointing (i.e., writing process state to disk) upon every interaction, were used in the fault-tolerant systems of the early eighties [Bartlett 1981, Borg et al. 1989, Kim 1984]. So failure masking has been considered a luxury affordable only by mission-critical applications (e.g., stock exchanges).

Fault tolerance is also being discussed for component middleware like CORBA [OMG: CORBA 2000] and EJB [SUN 2001], 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, or Sun's J2EE suite) at best support simple failover techniques that do not relieve the application programmer from having to either write failure handling logic or structure his application as "stateless," and are not geared for masking process or message failures to users. More recently, failover techniques for web servers have been presented in Luo and Yang [2001], based on application-transparent replication and redirection of HTTP requests.

The need for execution guarantees for e-services, raised in Tygar [1998]; Pedregal-Martin and Ramamritham [1999, 2001]; Dutta et al. [2001]; Fu et al. [2001]; Popovici et al. [2000]; Schuldt et al. [2000]), has been concerned with specific applications such as payment protocols or mobile data exchange and does not specifically address failure masking in general multi-tier architectures. Closest to our approach in terms of objectives is the work in Frølund and Guerraoui [2000] that presents a multi-tier protocol for exactly-once transaction execution based on asynchronous message replication and a distributed consensus protocol. However, this work focuses on stateless application servers and does not address the autonomy requirements of components, the optimization of logging, and the need for effective log truncation.

The recovery framework and protocols we introduce 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 reduce logging and state saving costs while providing very fast recovery. In contrast to solutions provided by TP monitors and CORBA- or EJB-oriented application servers, our approach handles stateful applications, removing the application programmer burden of making his application stateless. In addition, our method is unique in

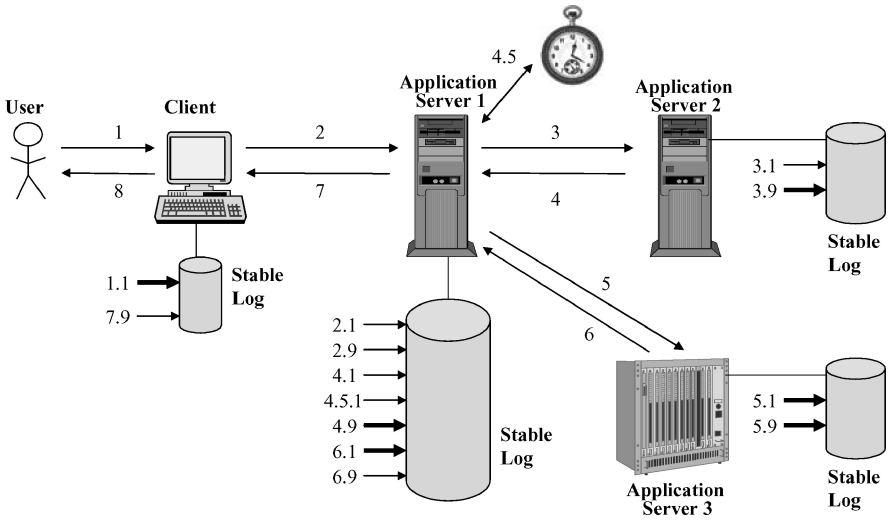

Fig. 4. Multi-tier system architecture and component interactions.

providing an end-to-end solution for masking failures across an entire multi-tier federation of application and data servers while, to a large extent, preserving the autonomy of these servers.

3. SYSTEM ARCHITECTURE AND KEY CONCEPTS

3.1 Components

We consider a group of interacting **components**: clients, application servers, data servers, and users (viewed as components). In a typical scenario a client (e.g., an Internet browser) submits a request on behalf of a user to a mid-tier application server (e.g., HTTP server, workflow server, or web service broker) that processes the request and generates further requests to other application servers or backend data servers. When the mid-tier server has received all necessary information from its subsidiary servers (including return messages for data updates that it may have initiated), it sends a reply message to the client, which in turn displays the result to the user. This either concludes the entire round-trip of a stateless interaction or is continued with another client request in a stateful conversational session (e.g., an extended workflow). The system architecture and the message exchange between the various components are illustrated in Figure 4 (for the time being ignore all events with non-integral numbers, they will be explained later). In the figure, application servers 1 and 2 are assumed to belong to the same domain and thus consider each other as trustworthy, whereas server 3 merely belongs to the same loosely coupled federation.

We require that components be **piecewise deterministic (PWD).** A PWD component is deterministic between successive messages from other components. Thus it can be **deterministically replayed** from an earlier state by resending it the original messages, producing the same component end state. Replay starts from a previous component state on disk, perhaps the initial state.

We call saved states **"installation points"** (**IP**s), not checkpoints, as they are often called in the fault-tolerant computing field, to avoid confusion with the different notion of checkpoint used in database recovery (see Section 2.1). 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., files or a database), message buffers, and information about active sessions. After a failure, to replay a component from a previous installation point, the component may need access to a **stable log** with log entries that contain sufficient information to recreate all incoming messages.

Components are mapped to processes or threads of the underlying runtime system (e.g., operating system or Java VM). Thus we can model a multithreaded server as a set of single-threaded components interacting with each other via shared data or messages. Alternatively, a server might be treated as a single component, but then we need to pay specific attention to the potential nondeterminism that may arise from thread interference on shared data.

3.2 Interactions, Contracts, and Logging

We assume that component failures are 1) soft, that is, no damage to stable storage so that logged records are available after a failure; 2) fail-stop so that only correct information is logged and erroneous output does not reach users or persistent databases; and 3) the result of "Heisenbugs" [Gray and Reuter 1993] such as timing-related race conditions or overload conditions so that replay does not reproduce the failure deterministically. We make no assumptions about multiple failures; more than one component may fail at any time.

Consider the scenario in Figure 4, with message-based interactions among user, client, and three application servers. We want the message flow and resulting states of the involved components to persist in any possible failures. When a component fails and is restarted, we want recovery to recreate exactly the state that it was in before the failure, including all received messages. We call a component with this ability a **persistent component** (**Pcom**). We will introduce two additional kinds of components in Section 4. The current discussion concentrates on Pcoms. A baseline algorithm to ensure persistence and thus failure masking might employ pessimistic logging (see Section 2.2). This requires that Pcoms log all incoming and outgoing messages and immediately force them to their stable logs before taking any subsequent actions. For example, in Figure 4, application server 1 dealing with messages 2, 3, 4, 5, 6, and 7, force-writes log entries numbered 2.1, 2.9, 4.1, 4.9, 6.1, and 6.9 for them. The numbering reflects the ordering between messages and log writes, that is, 2, 2.1, 2.9, 3, 4, 4.1, 4.9, 5, 6, 6.1, 6.9, and 7. Should the server fail, it can completely replay its prior execution starting from its initial state or some intermediate installation point (e.g., 4.5.1 after having received message 4). When replay reaches a point where the original execution received a message, the logged message is fed to the component's message buffer, and replay is resumed. In this way, the replayed process will generate exactly the same outgoing messages as in its prior incarnation.

Pessimistic logging can be inefficient, leading to unnecessarily many forced-log-writes. Overhead is at least as high with communication based on reliable message queues. This inefficiency is illustrated by considering the logging for a single message, as shown in Figure 4. For a sender-receiver pair, it suffices for the sender to force-log the message. Consider message 4. Suppose server 1 failed after receiving message 4; replay simply asks server 2 to provide the missing log entry. This shows the I/O saving potential.

There is a substantial advantage to considering what is needed for successful exactly-once execution in the abstract, without assuming any specific technique for implementing the requirements. We use the following observations to guide us in defining **interaction contracts (ICs)** between pairs of components.

Committed state: When a component sends a message, it externalizes, and thus commits its state. The message serves to reveal what state the sender is in. This is akin to committing a transaction, and requires that the sender state become persistent (durable). Since the sender must make its state durable, there is low cost of also taking responsibility for ensuring that the message is likewise persistent, as both message and state persistence can be recovered with the same deterministic replay.

Deterministic replay: Replaying components requires that all nondeterministic events be captured so that these events can be replayed during recovery. We have already indicated that messages between components are logged. We need to ensure that all sources of nondeterminism are removed. Such nondeterministic events as system interrupts, and nonidempotent interactions with the system environment, need to be captured so that they can be replayed deterministically. For example, in Figure 4 the event 4.5.1 could indicate reading the system clock or a sensor's real-time data stream. It is not necessary to make these events stable until we "commit" component state as described in (1) above.

Duplicate detection: During replay, we may not know whether a message generated immediately preceding a system crash was sent or not. Even if we knew the message had been sent, we may not be sure that it arrived successfully. To guarantee that a message will eventually be received, we may need to send it multiple times. To ensure exactly-once semantics, we thus need to detect duplicate messages and respond appropriately, either by ignoring them or by replying to them with the same reply message as was generated originally. In all cases, we must avoid multiple executions of any nonidempotent receiving component.

Autonomous recovery: If components are managed in an autonomous fashion, we cannot necessarily depend on (trust) other components to resend previously sent messages. In this case, we need to more promptly make stable the events (messages) that we receive from these undependable components.

With these observations, it is possible to define what we call a **committed interaction contract (CIC).** This contract is sufficiently abstract to enable its satisfaction with techniques that exploit the optimization illustrated in the previous section: where server 1 asks server 2 to resend the message rather than forcing the message to its log immediately. Details will be given in Section 4. In particular, we will discuss when the obligations that the contract parties agree upon can be safely released, thus enabling garbage collection on log files.

When autonomous recovery is an issue, we will incur higher logging costs. Returning to our example, when server 1 does not trust that server 2 will resend the original message, it will need to force log message 4. We refer to a contract covering this situation as an **immediately committed interaction contract (ICIC)**, for it allows both interacting parties to recover independently and to perform garbage collection independently right after the message exchange.

For the scenario of Figure 4, this suggests using CICs between the client and server 1 and also between server 1 and server 2, the latter two being mutually considered trustworthy. On the other hand, an ICIC would be in order between server 1 and the external server 3. Furthermore, if application server 1 did not want to become in any way dependent on the client, then an ICIC would also be used between client and server 1. Between user and client a special kind of contract is needed to be discussed in Section 4. This overall setup of interaction contracts translates into a specific choice of which log entries need to be forced to disk and which ones can be written to log buffers and lazily flushed to disk in the background. In Figure 4, the log-writes that require forcing are denoted by thick arrows.

3.3 Piece-Wise Deterministic Behavior

Ensuring that a component is indeed PWD, is subtle. Application servers with multiple threads serving multiple clients that communicate with it and that asynchronously access shared data clearly have multiple sources of nondeterminism. But even simple single-threaded, synchronous components, for exapmle clients, can have "hidden" sources of nondeterminism that need to be expunged. We identify three sources of nondeterminism and indicate how the nondeterminism can be removed.

- A component, for exapmle database or application server, may execute on multiple **concurrent threads**, accessing shared data such that access interleaving order is essential for successful replay (e.g., SAP-style ERP systems or eBay-style web sites). We assume there is no shared state among different components. Multiple components accessing common data require that data be in a component, for exapmle, a database server. Nondeterminism is removed by logging the interleaved accesses to the data. For a transactional database, the order of interleaving is the order in which transactions are serialized.
- Component execution may depend on **asynchronous events**, for exapmle, received messages or interrupts that prompt component execution at arbitrary points. These events are not reproducible during replay. The order of asynchronous events needs to be logged to guarantee deterministic replay. Often short logical log entries are sufficient, for exapmle message receive order, if message contents can be recreated by other means (for example, from the sender). However sometimes, physical logging is inevitable, for example, when reading the real-time clock.
- Component re-creation after a crash does not usually exploit the same system elements as the original execution, a form of nondeterminism. Ids for

messages sent before a failure may differ from ids of replayed messages. Ids for processes, threads, or users may also change. To remove **system resource mapping** nondeterminism, we "virtualize" these resources, introducing logical ids for messages, component instances, and so on. These logical ids are mapped to different physical entities after a crash, but at the abstract level, the "logically identified" component becomes PWD. We log these mappings.

4. INTERACTION CONTRACTS

4.1 Components and Interactions

4.1.1 *Component Guarantees.* Guarantees specify the behavior of individual components upon failure and are the basis for interaction contracts between components. Mostly we are concerned about persistent components (**Pcom**s), for which we guarantee persistent state. To guarantee state persistence will also require persistence guarantees for messages. However, a multi-tier application is rarely composed solely of persistent components. We need to treat other components and their interactions with Pcoms. Some components are transactional (**Tcom**s), where state and messages are only guaranteed to persist when transactions commit. A transaction abort resets the Tcom state to the beginning of the transaction, losing intratransaction updates and messages. Finally, external (user) components (**Xcom**s) cannot usually provide any of the above guarantees. For example, when prompted to provide a previous input, a user does not necessarily deliver identical input.

To implement persistence guarantees, we exploit a **log** and a**recovery manager** in the run-time environment to capture the order of all nondeterministic events and record the ones that cannot be replayed. During **normal operation** log entries are created in a log buffer for received messages, sent messages, and other nondeterministic events. The log buffer is written to the stable log on disk at appropriate points (forced) or when full. In addition, component state may be periodically "installed" (saved) to disk in an **installation point** to facilitate log truncation, frequently making log records preceding the installation point unnecessary. A data server that needs a stable log for the recoverability of persistent data can also use the log to hold message-related and other log entries.

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

At this point we can establish an important property that relates the persistence guarantees of a component to the underlying implementation mechanisms.

THEOREM 1. *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 nondeterministic events, such that these events can be replayed,*
- *forces the log upon each message send (before actually sending it) if there are nondeterministic 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.*

PROOF. By ensuring that all prior nondeterministic events are stable on the log upon each message send, the component can be replayed at least up to and including the point of its last send. This replay can be done for all nondeterministic events because the last installation point can be reconstructed from the log, and received messages can be accessed, perhaps locally, perhaps by request to their senders. Note that the latter implies that the component has not necessarily logged the contents of its received messages. Finally, all outgoing messages can be recreated during the component replay. Note that this does not require that the message send is itself logged; rather outgoing messages can be deterministically reconstructed provided all preceding nondeterministic events are on the log or already installed in the component state. □

4.1.2 *Interaction Contracts.* An interaction contract specifies the joint behavior of two interacting components. It 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 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, for example, to capture human users, where there is usually no recovery guarantee.

4.2 Persistent-Persistent Component 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.

4.2.1 *Committed Interaction Contract.* 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:

S1: Persistent State. Sender promises that its state at the time of the message send or later is persistent.

S2: Persistent Message.

S2a: Sender promises to send the message repeatedly (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.

S3: Unique Messages. Sender promises that its messages have unique contents (including all header information such as timestamps, HTTP cookies, etc.).

Sender obligations ensure that an interaction is **recoverable**, it is guaranteed to occur, though not with the receiver guaranteed to be in exactly the same state. Note that message uniqueness is required so that the receiver has a chance to detect duplicates (i.e., resent messages) and does not confuse them with another message with exactly the same content as a previous one (e.g., the same shopping cart contents with the same timestamp and the same cookies etc.).

Receiver:

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

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 periodically resending. After S2a release, receiver must explicitly request the message from sender should it be needed. The interaction is **stable**, 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**, 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.

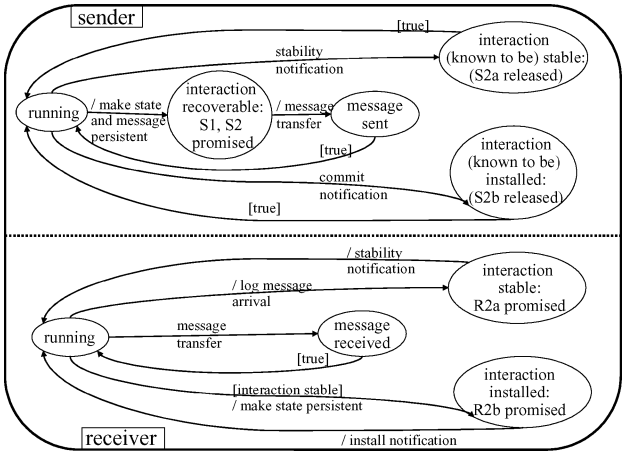

Fig. 5. Statechart for committed interaction contract.

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 resend 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. However, to fully discharge the CIC may require several messages. The receiver need not immediately expose to any other component that it received the message. Only when the receiver itself 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 to provide persistence for previously sent messages, not only in-memory data structures, 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.

Sender and receiver CIC behavior is depicted as a statechart [Harel and Gery 1997, OMG: UML 1999] in Figure 5. 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 that 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 their running states before making further steps towards a stable interaction. 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 resends (e.g., sender's periodic resend 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, are 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 precondition 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 need only resend the reply on request, which in this case is in response to resends of Q.

4.2.2 *Immediately Committed Interaction Contract.* In some situations, it is desirable to release a sender from its obligations all at once. This can be useful not only to the sender, but also the receiver, 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. Receiver's announcement thus makes the interaction both stable and installed simultaneously.

An ICIC can be seen as a package of two CICs, the first one for the original message and the second one for a combined stability-and-install notification sent by the receiver component. 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 as well. With an ICIC for the entire interaction, the sender waits synchronously for this notification (rather than resuming other work in its "running" state), and the receiver's part of the committed interaction is no longer deferrable. This is similar to an optimized form of two-phase commit between the components: it involves only two participants with the sender being the coordinator, and making its commitment right away, a form of "first agent optimization" (i.e. a "dual" version of the well known last agent optimization [Gray and Reuter 1993; Bernstein and Newcomer 1996; Weikum and Vossen 2001]). This a priori commitment is feasible here because the sender guarantees that it will resend the message

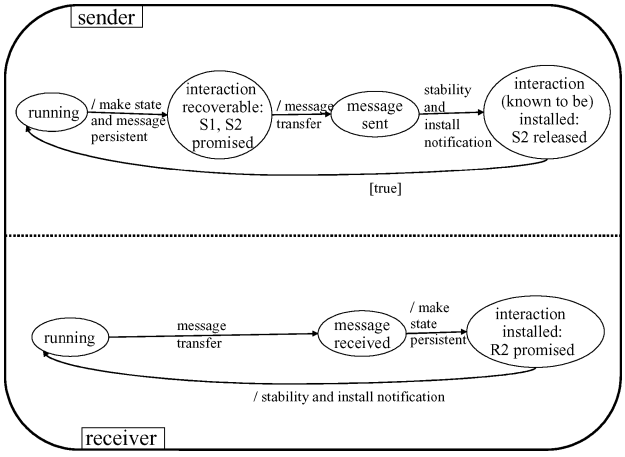

Fig. 6. Statechart for immediately committed interaction contract.

until it eventually gets the receiver to also commit the interaction. Figure 6 depicts the ICIC behavior as a statechart.

With a committed interaction, whether either party requires logging depends on whether there are nondeterministic events that need to be made repeatable. If not, then no logging is required, as the interaction, including the message contents, is made persistent via replay. With an immediately committed interaction, the receiver is required to make stable the message contents so that its state, which includes the receipt of the message, is persistent without needing the sender to resend the message. Thus, an immediately committed interaction 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 ICICs 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 ICICs have been the method of choice. CICs substantially reduce the overhead of a recovery contract. ICICs 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 5.3.

4.3 Persistent-External Component Interactions

Sender and receiver must be Pcoms to engage in committed interactions. External components (Xcoms) may not be persistent, 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 Xcoms, including users. This leads us to introduce **external interaction contracts (XICs)**.

Definition 3. An **external interaction contract** is a contract between a persistent component that subscribes to the rules for an immediately committed interaction, and an external component, that does not. The impacts on external sender or receiver (or users) of Pcom interactions with it are described below. Note that these are impacts on, not obligations of, the external component.

X1: Output Message Send. 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 resend the message. Because an Xcom might not eliminate duplicates, a user may see a duplicate message.

X2: Input Message Receive. 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 resend the message. But the user (an Xcom) has not promised to resend the message automatically; rather if 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. Importantly, a Pcom failure between the logging of the input message and the output message is known to be masked from the external component.

4.4 Persistent-Transactional Component Interactions

Another form of contract covers interactions between a Pcom and a transactional component (Tcom), usually a data server. These are request/reply interactions, where either a) a Pcom request message initiates the execution of a transaction (e.g., invoking a stored procedure) against the server's state and produces a reply reporting the transaction outcome, or b) a sequence of Pcom request/reply interactions (e.g., SQL commands) occurs, the first initiating a transaction and the last being the server's final reply to a commit-transaction or rollback-transaction request. The Tcom's state is transactional. During transaction execution, the changes made by the transaction are isolated, visible only to transaction participants. The Tcom's state transition is atomic, all-or-nothing, based on whether the transaction commits (all) or aborts (nothing).

Persistence makes no requirements on interactions between Pcom and Tcom during a transaction. And transaction isolation keeps intratransaction state changes from propagating elsewhere in the system. The critical interaction is when the Pcom sends a transaction-commit request message to the Tcom, and the Tcom sends a reply to this message. To provide persistence and exactly-once semantics for interactions between Pcoms and Tcoms, we place requirements on both Pcom and Tcom, as described below in the **transactional interaction contract (TIC):**

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

Pcom:

PS1: Persistent reply-expected state. The Pcom's state as of the ***time at which the reply to the commit request is expected***, or later, must persist without its having to contact the Tcom to repeat its earlier sent messages. The persistent state guarantee thus includes the installation of all earlier Tcom replies within the same transaction, for example SQL results, and return codes.
Persistence by the Pcom of its reply-expected state means that the Tcom, rather than repeatedly sending its reply (under TS1), need send it only once. The Pcom explicitly requests the reply message should it not receive it by resending its commit request message.

PS2: Persistent commit request message. The Pcom's commit request message must persist and be resent, driven by timeouts, until the Pcom receives the Tcom's reply message.

PS3: Unique message. The Pcom promises that its commit request message has unique contents (including all header information such as timestamps, etc.).

PR1: Duplicate message elimination. The Pcom promises to eliminate duplicate reply messages to its commit request message (which the Tcom may send as a result of Tcom receiving multiple duplicate commit request messages sent by Pcom because of PS2).

PR2: Persistent reply installed state. The Pcom promises that, before releasing Tcom from its obligation under TS1, its state at the time of the Tcom commit reply message receive or later, is persistent without the need to again request the reply message from the Tcom.

Tcom:

TR1: Duplicate elimination. Tcom promises to eliminate duplicate commit request messages (which Pcom may send to satisfy PS2). It treats duplicate copies of the message as requests to resend the reply message.

TR2: Atomic, isolated, and persistent state transition. The Tcom promises that before releasing Pcom from its obligations under PS2 by sending a reply message, it has proceeded to one of two possible states, either committing or aborting the transaction (or not executing it at all—equivalent to aborting), and that the resulting state is persistent.

TS1: Persistent (faithful) reply message. Once the transaction terminates, the Tcom replies, acknowledging the commit request, and guarantees persistence of this reply until released from this guarantee by the Pcom. The Tcom promises to resend the message upon explicit Pcom request, as indicated in TR1 above. The Tcom reply message identifies the transaction named in the commit request message and faithfully reports whether it has committed or aborted.

TS2: Unique message. The Tcom promises that its commit reply message has unique contents (including all header information such as timestamps, etc.).

In current practice, for example with current database systems, the most difficult requirement is to ensure the delivery of the transaction outcome message. The reply to a commit request message might not be delivered even though the transaction successfully commits, as a crash may occur between transaction commit and message send. We require a stronger guarantee for Tcoms, as expressed in TS1, namely that the reply must persist and be resent upon request from the Pcom. Furthermore, when the transaction aborts, a database may forget the transaction, which could pose difficulties for making the transaction outcome message persistent.

When a transaction commits, as directed by a commit request message, the Tcom must ensure that the commit message is stable until it is released from this requirement by the Pcom. It is this stable reply message (reporting that the transaction committed) that provides the testable state preventing the transaction from multiple executions. We do not need the faithful reply guarantee, in which the Tcom is obligated to correctly report on the commit/abort outcome of the transaction, to ensure that Pcom state is persistent. This guarantee is necessary, however, to ensure exactly-once execution semantics, including the updates to databases.

Note that PS1 applies only for commits, not aborts. PS1 removes the need for a Tcom to retain earlier messages of the transaction. TS1, in conjunction with PS1 means that the Tcom need only capture the transaction's effects on its database and its reply to the commit request message, 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 was indeed committed. This is important as persistent committed state and persistent commit reply are what are usually guaranteed by queued transactions TP monitors and by Phoenix/ODBC [Barga et al. 2000; Barga and Lomet 2001]. Thus, our recovery contracts work with existing infrastructure.

When a transaction aborts, there are no guarantees except that 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. The Tcom fails or autonomously aborts the transaction for other reasons. If a transaction aborts following a sequence of request/reply interactions within the transaction, abort is signaled to the Pcom in reply to the next request (e.g., through a return error code). Abort can also result from a commit or a rollback request message. When the Tcom has no record of a transaction, this means that the transaction has been aborted. So persistence of an abort reply message is trivially achieved. The Pcom may choose to re-initiate the transaction, but the Tcom treats this as a completely new transaction, a standard practice in transaction processing. If the abort response is lost, a repeated commit request message either returns the abort response, or, if

the entire transaction is defined by the commit request message (e.g., when invoking a stored procedure with a single SQL call), causes the re-execution of the transaction, with a new outcome that is reported to the Pcom.

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 (on a new connection), the Tcom will respond with, for example, a "transaction/connection unknown" return code, and the Pcom proceeds as in the first case.

In addition to commit and abort, a third scenario arises when the transaction is still active at the Tcom and the Pcom repeats the sending of the commit request message, its way of asking about the transaction status. This can happen if the Pcom times out waiting for a reply to its commit request, or fails after having sent a commit request but before receiving the commit reply. Should the Pcom require restart, it would be able to recreate the commit request (by PS1). In any event, the Tcom must detect a duplicate message and, if the activity of the original commit request is not finished, the Tcom must await the completion of this work and then report back the outcome as if it had received only a single message.

4.5 System-Wide Composition of Contracts

Our aim is to transform from black art to engineering the method for guaranteeing system-wide recoverability for multi-tier systems. Our approach is to combine bilateral interaction contracts between pairs of components into a system-wide agreement that provides the desired guarantees to external users. The key to such a **recovery constitution** is the observation that the behavior of a multi-tier system 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. Interaction contracts provide the following general solution:

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

Note: Tcoms are not allowed to call Pcoms (violating isolation and breaching the transactional all-or-nothing paradigm) or Xcoms.

Our recovery constitution allows arbitrary interaction patterns between Pcoms, including, for example, asynchronous message exchanges, callbacks from a server to a client or among servers (e.g., to signal exceptions), 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 Tcoms not call Pcoms or Xcoms but only reply to requests from Pcoms. Then the following very general theorem holds:

THEOREM 2. *Consider an arbitrary (possibly cyclic) graph of message exchange relationships among a set of components with an arc from component A to component B if A sends a message that B receives. The graph must have no edges between Tcoms and Xcoms, and the only edges from Tcoms to Pcoms are Tcom replies to Pcom transactional requests. Then the following holds: If there is a CIC (or ICIC) for each pair of Pcoms 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 external interactions.*

PROOF. We first derive a total ordering of all messages in the entire system's history from what is known as causal order in the distributed computing literature:

1. If a component receives message A and later sends message B, then A is before B;
if a component sends message A and later sends message B, then A is before B;
if A is before B and B is before C, then A is before C (i.e., we consider the transitive closure of the orderings according to (i) and (ii)).
All messages that are unordered according to (i) through (iii) are arbitrarily ordered by some topological sorting of the partial order obtained from (i) through (iii).

This yields a total message order that preserves message send order in each component. In addition, we assume that at each component, any nondeterministic events (including message receives) are totally ordered among each other and with message sends. This total message order does not guarantee that messages are delivered to the receiving components in causal order. We make no assumptions about the underlying communication system. This send message order merely serves as a basis for the following induction: we prove our claim by induction on the number of message exchanges in the system history, up to the last failure.

Basis: The first sent message in the system history must be an input message as all other messages are causally dependent on input. This message is subject to an XIC. Thus, it is guaranteed to be recreatable once the contract is completed. However, since failures can occur at any time and repeatedly, there is no guarantee that this input might not need to be entered repeatedly, as permitted by our claim.

Induction: Assume that the entire system history consists of n+1 sent messages and that our claim holds for the first n messages, where we ignore and discard any messages sent on behalf of uncommitted or aborted transactions between a persistent component (Pcom) and a transactional component (Tcom). (These transactional messages become relevant only upon the commit of the transaction.) We further assume that external interactions up to this point also satisfy our induction hypothesis. Thus all n messages have been received and processed with exactly-once semantics. Further assume that it is either the sender or the receiver of the $n+1^{st}$ message that fails after the completion of the $n+1^{st}$ committed interaction. Note that this does not rule out multiple failures

of one or both of these components, or other components, but we can assume that the recovery is performed for each failure separately and sequentially. All guarantees up to and including the n^{th} message are satisfied by the induction assumption; so it is indeed only a failure by the $n+1^{st}$ sender or receiver that needs closer inspection.

We can verify our claim by a case analysis with regard to the type of the $n+1^{st}$ message (i.e., internal versus external and persistent versus transactional component) and the component that fails (i.e., sender versus receiver):

1. *Internal interaction among persistent components*: If the $n+1^{st}$ message is between two persistent components then the following holds. The failed component is reincarnated from its last installation point and replayed from there. During replay all nondeterministic events that precede the last sent message are reconstructed from a local log. This is guaranteed because of the component's state recreatability guarantee that is part of the last send's committed interaction contract. All received messages, that precede the component's own last send in the causal order, can be recreated by the induction assumption.
   a. *Failed component is sender of* $n+1^{st}$ *message*: The CIC for the send ensures that we can recreate and repeat the send. Since all receivers in a CIC can test for duplicate messages, this restart behavior preserves exactly-once semantics.
   b. *Failed component is receiver of* $n+1^{st}$ *message*: We know that, in its prior incarnation, it has not sent any messages after receiving the $n+1^{st}$ one, for then the $n+1^{st}$ message would then not be the last in the global order. So the receiver has not committed its state to any other component after its own last message send, which is recreatable by the induction assumption. Thus the fact that the received message was sent again and received (perhaps multiple times) in a state that possibly differs from that of the prior incarnation but has not been exposed to other components, does not unmask the failure.
2. *External Interaction*: The $n+1^{st}$ message is an external input or output message. Then the failed component must be either the internal receiver of external input or internal sender of external output. These interactions are subject to an XIC. Hence, once the interaction completes, the message is stably delivered (in either direction). Hence, no subsequent failure can affect the system's ability to mask a failure in this XIC. For failures within the external interaction, arbitrary repeating of the message send or receive may be required. This is inevitable in any recovery scheme, and is permitted in the behavior for our claim.
3. *Internal interaction between a persistent and a transactional component*: The $n+1^{st}$ message is a transactional request from a Pcom to a Tcom or a transactional reply from a Tcom to the requesting Pcom. If this message is neither the Pcom's commit request nor the Tcom's commit reply then this $n+1^{st}$ message does not require any guarantee as the transaction is not (yet) committed. If it is the Pcom's commit request, then the Pcom guarantees its state to be persistent as of the point when the commit request is

sent; so the commit request message can be repeated if necessary. Recall, however, that the Tcom may still abort the transaction; then it is up to the Pcom to handle this outcome and the Pcom is able to do so because of its state being persistent. The final case to consider is when the $n+1^{st}$ message is the Tcom's commit reply. At this point the Tcom promises that both its state and the reply message are persistent. So the guarantee in our claim is extended to the $n+1^{st}$ message and covers also, implicitly through the persistent state of the Pcom, all reply messages that may have been sent within the transaction boundaries (in the case that the transaction comprises multiple request/reply steps). □

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, and garbage collection and log truncation can be performed at the components. Before we describe these, we must discuss our underlying implementation techniques. We will return to this issue in Section 5.

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. An example of possible hardware support (providing testable state) for output messages is an ATM for dispensing cash in which a mechanical counter records when money is dispensed. Output messages (e.g., cash) are guaranteed to be delivered exactly once, that is only when the counter is in the correct state.

5. IMPLEMENTING RECOVERY CONTRACTS

Interaction contracts and implementation measures are separate layers in our framework. A system can provide 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 CICs. Here we outline one such way to do this; more details can be found in Barga et al. [2002].

5.1 Log Management

Data servers have the hardest logging requirements because they are usually heavily utilized, support many concurrent "users," maintain valuable enterprise data, and are carefully managed for high availability. In addition to the usual logging for persistent data (see Section 2.1), the data component, a Tcom, needs to log only the final reply message for a caller's commit request, with enough information so that it can recognize duplicate commit request messages, and the server log needs to be forced before sending this final reply. Aborted transactions require no log forcing. Because SQL session states such as cursors or temporary tables can span transaction boundaries, we also provide persistent state for session components. The server maintains this information as state that is covered by interaction contracts. Phoenix/ODBC [Barga et al. 2000; Barga and Lomet 2001] retain this state to provide persistent sessions.

Asynchronous message receives require logging (but no log-forcing), logical logging being sufficient for CIC interactions. Logical log entries capture the nondeterministic interleaving and uniquely identify sender and message, but do not contain message contents. Logging for sent messages can be either physical, including message contents, or logical. CICs require 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, when received messages have arrived in nondeterministic order.

The advantage of CICs versus ICICs in reducing recovery overhead shows up with application servers and clients. For these components, often (but not necessarily) the only nondeterminism is the result of user input or data server interactions. Further, these components usually have little reason for using ICICs. 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 nondeterminism, 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., Tcoms), Pcoms (application servers or clients) need to ensure their state persistence as of the time 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 earlier Tcom replies 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 nondeterministic events that have to be tracked. If the Pcom issues a roll-back request, no force logging is needed. The Pcom needs application logic for aborted transactions in any event.

5.2 Component Restart after Failure

After a failure, each Pcom performs local recovery that reincarnates the component at its most recent installation point and replays the component log from there. The log is scanned in order, interpreting log entries to recreate persistent data and component state. To recreate component state, data reads, and other nondeterministic 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 and other events. All Pcoms (data and application servers and clients) use this procedure to recover both component state and sent messages.

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

5.3 Recovery Independence and Garbage Collection

We want 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, that is, no cascading restarts typical of many "optimistic" fault-tolerance algorithms [Alvisi and Marzullo 1995]. Interoperating components providing cross-organizational e-services are frequently autonomous, and cascading restarts are absolutely unacceptable. Nonetheless, an isolated component must resend messages as long as its contracts are not released. A solution is the volatile **message lookup table (MLT)** [Lomet and Weikum 1998] that records in main memory all uninstalled sent messages. These messages can be resent without component replay or reading the log. The MLT is rebuilt during recovery if the component fails; so it can always be present during normal execution.

With complex multi-tier systems spanning autonomous organizations, components must be able to recover without reliance on other potentially less reliable or less trusted components. This **autonomous recovery** [Lomet and Weikum 1998] for the server in a client-server setting can be generalized to **component ensembles**. A component of an ensemble may rely on trusted ensemble components, but wants to be autonomous of components outside the ensemble. One example ensemble is a data server and application server at an e-commerce site, clients being outside the ensemble. An immediately committed interaction contract with immediate forced logging, similar to the Pcom side of an external interaction contract, produces this autonomy.

Garbage collection is important for all components because they need to discard information from the MLT to reclaim memory and truncate the log to reclaim log space. It is critical for server components to ensure 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 in a reasonable time.

5.4 Performance Impact

In this section we illustrate the potential performance of interaction contracts using the advanced multi-tier e-services scenario illustrated in Figure 1 (Subsection 1.3). In this example, client, Expedia web server and application server, and Amadeus and Sabre application servers, support sessions that are Pcoms, while all database servers support sessions that are Tcoms. Amadeus and Sabre interactions are handled via ICIC forced logging. But messages leading to a purchase that are directed to the lower-tier Expedia application server are treated as CICs, and are not forced. Interactions with database servers are treated as TICs. The bilateral interaction contracts for our e-service are set up as follows:

> [**user** $\Leftrightarrow$ **client**] The client handles user input and output with XICs via prompt forced logging. Current Internet browsers do not provide native support for logging, but can be enhanced with a plug-in or applet.

[**client ⇔ database server**] Interactions between the client and the database server are handled with TICs. The database server commits modifications to the permanent and shared database when sending its final reply to the client, and must force log this final reply message to ensure its persistence.

[**client ⇔ Expedia web server**] Between client and upper tier web server, client request and server reply are handled with CICs. Forced logging is not required as client XIC logging captures all nondeterminism.

[**Expedia web server ⇔ Expedia application server**] Between Expedia web server and application server, requests and replies are handled with CICs. No forced logging is required as, again, client XIC logging has captured all nondeterminism.

[**Expedia web server ⇔ external application server**] Between the upper tier application web server and lower tier external application servers, ICICs that require forced logging by both the Expedia web server and external application servers are used to capture the potential nondeterminism as these application servers belong to other organizations and are thus autonomous.

[**application server → database server**] Requests from application server to database server are transactional, and require a TIC. Because the application server is without nondeterminism, forced logging of individual requests is not required. A commit request exposes the effects of application server execution via changes to database server state, hence this state must persist. However, since prior ICICs with Expedia server or client have already captured all nondeterminism, forced logging is not required.

[**database server → application server**] Finally, a database 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. Hence, this final reply (i.e., the return value for the SQL "commit work") must be force logged, which can also capture the database server's committed transaction updates.

The contracts identified above are necessary for system-wide recoverability. The database server may also require effective garbage collection and independent recovery. Specifically, the database server can treat its transaction ending reply to the application server as an immediately committed interaction so that it can discard messages once it knows that the application server has received them, and therefore subsequently freely truncate its log.

The number of forced log writes dominates the cost of our protocols in the above scenario. Let the user session consist of u input messages and u output messages, and let the client generate *one* request to its database 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 create z requests per

incoming request to its database server. Under these assumptions:

- Pessimistic message logging requires a total of $2u + 4u + 4ux + 12uxy + 12uxyz$ forced log writes, a forced log write for each message by both sender and receiver.
- Our protocol with the system configured as above, permits a potential reduction in the number of forced log writes to only $u + u + 0 + 12uxy + 3uxyz$, a saving of $4u + 4ux + 9uxyz$ disk I/Os.

6. IMPLEMENTATION OF INTERACTION CONTRACTS FOR INTERNET E-BUSINESS SESSIONS

This section describes a prototype system coined EOS (for Exactly-Once E-Service) [Shegalov et al. 2002] that implements the interaction contract protocols for a three-tier web service architecture: an Internet browser as client, an HTTP server with a servlet engine as middle-tier application server, and a database system as backend data server. Specifically, we have built the prototype using IE5 as browser, Apache as HTTP server, and PHP as servlet engine; the data server can be any ODBC-compliant database system (e.g., SQL Server, Oracle, or DB2/UDB). The prototype currently supports an XIC between the user and the browser, a CIC between the browser and the mid-tier application server, and a TIC between the application server and the data server (where the implementation of the last part is not yet completed). Building the prototype required extensions to the IE5 environment in the form of JavaScript code in dynamic HTML pages (DHTML), modifications of the source code of the PHP session management in the Zend engine [PHP; ZEND], and modifications of the ODBC-related PHP functions as well as additional stored procedures in the underlying database. The main emphasis of the following description is on implementing (i) the XIC behavior at the IE browser, and (ii) the CIC behavior in the middle-tier Internet application server, which are the most innovative aspects of our work.

6.1 Browser Extensions

To implement an XIC between the user (an Xcom) and her Internet browser session (a Pcom), one needs to extend or modify browser behavior so that certain types of user interactions (and also HTTP *get* or *post* requests to application servers and their replies) can be logged to stable storage. This raises issues of how to intercept the relevant events and how to embed the extra XIC code in the run-time environment of the browser. The solution that we came up with embeds special JavaScript code in a normal HTML page, and this DHTML (dynamic HTML) code realizes the XIC behavior. Our usage protocol requires that the user start interacting with an e-service by first visiting an initialization web page of that service, for example, a greetings page. The reply from this HTTP request contains our JavaScript code. The code is inserted into the HTML page returned by the server transparently to the application PHP program that

constructed the HTML page; this is done by our modifications to the Zend engine (see Subsection 6.2).

The inserted JavaScript code exploits *extensible event handling* of IE5 [MSDN LIBRARY: PERSISTENCE]. It registers for various kinds of events, most importantly, the filling in of form entries, clicking on buttons, and positioning of scrollbars (and, optionally, the opening or closing of additional windows); IE5 automatically invokes the associated event-handling code upon these events. Note that our code is invoked only after the events have been processed by the browser itself and any other JavaScript code that may be already present in the application's original HTML page. Figure 7 shows fragments of our browser extension code.

The main function of our event-handling code stubs is to log the updates to the browser state (e.g., the user having clicked a button or filled in a form entry). This logging is done by modifying a so-called *XML store*, which is an XML structure managed by IE on the client's disk in a way similar to a persistent cookie. This feature is provided by IE with a default persistence behavior called "*userData Behavior*." HTML elements with attached *userData behavior* provide the methods for accessing the individual elements of the XML store. The XML object associated with the XML store can also be accessed and manipulated using the XML DOM parser, which is natively supported by IE [MSDN LIBRARY: PERSISTENCE]. To force our log entries in the XML store to stable storage, we simply call the "*pagestate.save*" method, which triggers IE to write the XML store to disk.

For the CIC between browser and web application server, all relevant state information (i.e., each input field, whose value is passed to the web server, so that a completely identical HTTP request can be generated or "replayed" again) and rendering details (the window area viewed or edited by the user) are logged to an XML store associated with the current session step. Replies to HTTP requests are also logged, and they will be reflected in the browser state that is reconstructed upon recovery.

Client state survives failures as follows. When the browser fails and the user restarts it and revisits the same e-service initialization page, she will be automatically redirected by the web server to the last visited (i.e., most recent) page of her interrupted conversation and our JavaScript code will be reloaded. The JavaScript code is set up to first look for an XML store previously saved on the client machine. If the XML store exists, its contents are used to recover input field values and replay all relevant events on windows, buttons, and forms, so that the user would not see any difference to the state immediately before the failure.

6.2 Application Server Modifications for CIC

To implement a CIC for the web application server, the main issues to be addressed were the virtualization of message ids and the logging of HTTP requests and replies as well as session state information at the server side. We were able to realize this while modifying only the session management module of the

```
1.  <html>
2.    <head>
3.      <!-- EOS 1.0: BEGIN AUTOMATICALLY INSERTED RECOVERY CODE -->
4.      <!-- the following element is not rendered -->
5.      <sdk:logger style="behavior:url(#default#userData);"
6.       id="pagestate" recovered="false"></sdk:logger>
7.      <script>
8.        function recover() {
9.          //recover page look and input
10.         document.body.innerHTML = pagestate.getAttribute('body');
11.         //persistent cookies are persistent per se; no recovery needed
12.
13.         //now check whether user hit a submit button before crash
14.         for (var i=0; i < htmlForms.length; i++) {
15.            logged_value = pagestate.getAttribute('FORM_' +
    htmlForms[i].name);
16.            if (logged_value == 'submitted') {
17.               setRequestID(LastRequestID);
18.               htmlForms[i].submit(); // request is resent and we are done
19.            }
20.         } //end for loop
21.         //recover scrollbar position
22.        ....
23.       }// end recover
24.       function updatePageState() {
25.         logEventNr++;
26.         //log the current page look along with user input
27.         pagestate.setAttribute('body', document.body.innerHTML);
28.         pagestate.setAttribute('URL', document.URL);
29.         //force-log when more than 5 pending events or after timeout
    (5*100ms).
30.         if((flushTimeout!=null) && (logEventNr % 5 !=0))
31.           clearTimeout(flushTimeout);
32.         flushTimeout=setTimeout(flushPageState, 100);
33.       }
34.       function logFormSubmission() {
35.         pagestate.setAttribute('FORM_' + event.srcElement.name,
    'submitted');
36.         pagestate.setAttribute('LastRequestID', currentRequestID);
37.         pagestate.save(ip_name);  //force-log
38.       }
39.       function periodicResend() {
40.         formID = event.srcElement.id; //id of the form being submitted
41.         setInterval (formID + ".submit()", TIMEOUT);
42.       }
43.     </script>
44.     <script for='window' event='onLoad'>
45.       var ip_name = getCookieValue( '<?php echo session_name(); ?>' );
46.       var inputFields = document.all.tags('input');
47.       var htmlForms = document.forms;
48.       pagestate.load(ip_name);
49.       var log = pagestate.XMLDocument.documentElement;
50.       var recovery_needed = ( log.attributes.length > 0 ) ||
51.                             ( log.childNodes.length > 0 ); //log isn't
                                                               //empty
52.       if (recovery_needed)
53.           recover();
54.       //intercept property changes of all html tags in the HTML DOM
55.       for(var i=0; i < allElements.length; i++)
56.           allElements[i].attachEvent('onPropertyChange', updatePageState);
57.       //intercept form submissions and proceed analogously with links
58.       for (var i=0; i < htmlForms.length; i++) {
59.           htmlForms[i].attachEvent('onSubmit', logFormSubmission);
60.           htmlForms[i].attachEvent('onSubmit', periodicResend);
61.       }
62.     </script>
63.     <!-- EOS 1.0: END OF AUTOMATICALLY INSERTED RECOVERY CODE -->
64.
65.      <-- continue original output
66.   </head>
67.  </html>
```

Fig. 7. Fragments of the DHTML code embedded in the reply from the recovery-enabled web server.

PHP Zend engine, hence avoiding source code modifications to the regular infrastructure on the server.

For virtualization, we used our own message sequence numbers (MSNs) that are unique and consecutive within an application session. An application session, which is independent of any TCP session, might consist of several steps, for example adding items to a shopping cart, steps that constitute a "logical (and stateful) session." MSNs are added to the HTTP request and reply messages in the form of additional cookies.

The modified Zend engine force-logs all outgoing HTTP replies, tagging each of them with the MSN included in its header. This is done by writing this information as additional session variables to the PHP session state file. Note that there is no need to force-log the incoming HTTP request, as the client already promises the persistence of this message as the CIC sender. The session state file is accessible to all clones of PHP server processes that are controlled by the Apache web server. If multiple Apache servers run in a computer cluster for the same IP address, the file must be shared among all nodes in the cluster. This technique ensures that we do not depend on "sticky" connections between HTTP clients and PHP server processes.

For a server to proactively recover servlet results after a server crash, as opposed to having servlet replays only when prompted by resent client requests, we may optionally log incoming HTTP requests along with the PHP variables filled by HTTP *get* or *post* parameters and all session variables that have been registered up to this point. The log record for an HTTP request already contains the name of the invoked PHP program and its input parameters, which are either form variables or encoded in the URL. This captures the initial state of the servlet execution. Since the servlet is PWD, no other logging is needed for correctness.

We can now describe how the modified Zend engine handles the various exceptions and recovery situations:

- Upon receiving an HTTP *get* or *post* request carrying a cookie with an MSN, we test for a duplicate request by checking the log. If it is a resent request and the corresponding servlet has already terminated and produced an HTTP reply, the reply is retrieved from the log and sent to the client. If the servlet has died and no reply is available, it is restarted. If we had logged the state of session variables during the prior incarnation of the servlet, its replay would start from the last completed installation point.
- When an HTTP reply is sent to an unresponsive client (e.g., the client does not send a TCP ACK), the server simply ignores this but is prepared to receive a duplicate HTTP request at a later point. When this client-initiated prompting happens, the server resends the reply. As the reply itself is stably logged at the server, it is guaranteed to persist across server failures. Once the client acknowledges the receipt of the HTTP reply by issuing another HTTP request within the same application session or invokes a servlet with a *session_destroy* function call, the server can garbage-collect the previous step's log entry. To alleviate the potential danger that the server cannot safely discard log entries for clients with users who have intentionally aborted sessions

```
1.  sess_id = getSessonID(); // extract session id from HTTP request
2.  msn = getMSN(); //extract MSN from HTTP request
3.  if( HTTP_reply = lookUpReplyLog(sess_id, msn + 1) ) {
4.     sendToBrowser(HTTP_reply);
5.     exit(0); // nothing to do anymore
6.  } //end if
7.  //there is no reply log entry yet, but we want to proactively
8.  //replay the servlet based on the request log entry
9.  if( !loadSessionData(session_id, msn) ) //load and decode serialized data {
10.    //no state data available: when the user hits the "Back" button,
11.    //put her back to the most recent valid page
12.    new_url = lookUpLastURL(sess_id);
13.    sendRedirectReplyToBrowser(new_url);
14.    exit(0);
15. } //end if
16. //execute servlet, add new msn and JavaScript code for XIC
17. HTTP_reply = executeScript(HTTP_request, sess_id, ++msn, JSCode);
18. saveSessionData(session_id, msn); // create new installation point
19. saveLastUrl(sess_id, HTTP_request->url);
20. addToReplyLog(HTTP_reply, msn);
21. sendToBrowser(HTTP_reply);
```

Fig. 8. Pseudocode for application server logging and recovery.

or simply walked away, we can enhance the JavaScript code that we embed in the HTTP replies to react to "onAbort" browser events and to timeouts: the code would simply send an explicit "abort session" message as a final HTTP request to the web application server.

When a PHP process fails while executing a servlet on behalf of a client's HTTP request, recovery is initiated when the client repeats its request (as part of the CIC behavior) or the user hits the "Refresh" button. The resent request is handled by the next available PHP process, which first performs a duplicate elimination test, possibly replays the servlet execution, and finally (re-)sends the reply. When Apache or the entire computer fails, the same thing happens after the restart of Apache. So recovery is automatic, but we rely on the client re-initiating the request rather than on server initiative. This behavior carries over to a web server farm on a computer cluster; failover to another node in the cluster is automatic as long as clients resend requests.

The logging and recovery logic of our modified Zend engine are summarized in pseudocode form in Figure 8.

6.3 Performance

To evaluate the run-time overhead of our failure-masking techniques and exactly-once guarantees, we performed measurements with Apache/1.3.20 and the Zend engine (PHP/4.0.6) running on a PC with a 1 GHz Intel Pentium III and 256 MB memory under Windows2000. The load on this web application server was generated by a synthetic HTTP request generator (Microsoft Web Application Stress Tool). The generator simulated conversations with n steps, each of which simply sent three string parameters as form fields, and a simple PHP program incremented a counter registered as a session variable and returned its value to the client. For simplicity we did not involve any data server in this setup. There were no human user interactions or simulated think times in this experiment. Table I shows the total elapsed time, between the first request

Table I. Elapsed Time and CPU Time for n-step Conversations With and Without CIC

	Original Zend engine		Modified Zend engine with CIC	
N	Elapsed time [sec]	Server CPU time [sec]	Elapsed time [sec]	Server CPU time [sec]
1	0.01893	0.01412	0.01921	0.01435
5	0.09378	0.07052	0.09507	0.07156
10	0.18183	0.14380	0.18610	0.14363

Table II. Multi-User Response Time and Throughput for n-Step Conversations With and Without CIC

	Original Zend engine		Modified Zend engine with CIC	
N	Response time[sec] for n-step session	Server throughput [sessions/sec]	Response time [sec] for n-step session	Server throughput [sessions/sec]
1	0.07872	62.35	0.08946	54.96
5	0.39475	12.434	0.44865	10.96
10	0.79597	6.168	0.80234	5.522

and the last reply as seen by the client, and the CPU time on the server side for $n = 1, 5, 10$ steps, comparing the original Zend engine to the modified Zend engine with CIC behavior for exactly-once guarantee. The figures show that the overhead of our CIC implementation is almost negligible, with respect to both user-perceived latency and increased CPU time.

We also performed multi-user measurements where the HTTP request driver was replicated on 5 different client machines, each of which generated requests to the same web application server without simulating any think times (i.e., using a closed system model). Table II shows the measured average response time and throughput in terms of the simulated n-step user sessions. The figures show that the performance degradation is less than 10 percent and thus well within the range of acceptable overhead.

6.4 Further Considerations

The CIC exactly-once guarantee between web browser and application server is very useful when impatient users click a commit/buy/checkout button multiple times. This particular difficulty could, to a large extent, also be avoided by better design of the user interface in the browser. Applications could and, in our opinion, should be written so that a clicked button is deactivated, possibly highlighted in a special color, until the corresponding HTTP reply is received. With a XIC-enabled browser and a CIC with the application server, the request would be guaranteed to be executed exactly once, yet the user would be prevented from creating unnecessary "noise." For the human user such a deactivated button that is reactivated upon receiving the reply serves as "eyeball-testable state." Note that such a better GUI alone would not eliminate all kinds of problematic situations caused by browser or application server failures in the middle of an application session, and surely does not obviate the need for interaction contracts. Our interaction contract protocols developed in this paper are orthogonal and complementary to more robust user interfaces.

Another dimension that is mostly orthogonal to interaction contracts is security. When secure communication channels are used, say HTTP over SSL (known as HTTPS), the underlying encryption is transparent to our protocols. In particular, as our resource virtualization never refers to any actual TCP session ids, the message sequence numbers and application session ids used by our protocol are not at all affected by the physical transport layer underneath.

The situation is not quite that obvious with regard to authentication. While standard password-based authentication is no problem, authentication schemes with digital signatures based on keys that must not be used more than once could possibly interfere with the replay techniques of our recovery methods. If replaying a lost message involved using the same key a second time, this could present a security or privacy leak. The solution again lies in a flexible mapping of virtual to physical resources. Our log records refer only to virtual message and session ids; when a message is replayed its virtual id is mapped to a new physical id so that the underlying signature scheme would automatically use a new key as needed.

Finally, very advanced cryptographic protocols, for example, for anonymous payments or legally binding electronic contracts (with mathematically provable tracking of the various parties' behavior including any attempts of unfair behavior), can again be viewed as part of the Internet application itself, and our interaction contracts are orthogonal to the security issues. Overall, we believe that by layering the interaction contracts on top of whatever security and privacy measures are used, failure-masking remains unaffected by the security protocols and, likewise, no security leaks are introduced by the failure-handling protocols.

7. INDUSTRIAL RELEVANCE

7.1 Phoenix/App

We have also implemented the recovery guarantees framework, as part of the Phoenix project on robust applications [MSR PHOENIX], in a system we call Phoenix/App [Barga et al. 2003]. In Phoenix/App we integrate interaction contracts into the Microsoft .Net runtime environment [MSDN LIBRARY: .NET; Williams 2002], allowing programmers to build persistent component-based applications without requiring modifications to their applications.

To use Phoenix/App, an application programmer simply registers component classes as Pcoms or Tcoms. At runtime, applications are embedded into the .Net runtime and the .Net interceptor calls Phoenix/App code that captures all method calls and returns between components. Calls and replies between Pcoms are treated as CICs, while calls and replies between Pcom and Tcom are treated as TIC's, and calls between a Pcom and any other component (an Xcom) are treated as XICs. Phoenix/App uses a log manager to create log records, flushing them to disk as necessary, and to handle log truncation. In the event of a system failure, Phoenix/App masks the error from the application, which can be written in any language executing on the Microsoft CLR (common language runtime), and automatically recovers any failed components from log records using redo recovery. The result is a persistent component-based application

that can survive system failures, without any special application code or operator intervention. While our current implementation of the recovery guarantees framework is .Net specific, the techniques are relevant to other object middleware environments such as CORBA or J2EE.

7.2 Comparison to EJB

Another component framework to which interaction contracts and recovery guarantees could potentially be added would be J2EE with its concept of Enterprise Java Beans (EJB). EJB includes the concepts of entity beans, which allow persistent data from relational databases to be mapped into an object view and manipulated through this view (i.e., the bean interface), and session beans, which combine multiple method invocations on encapsulated objects into a stateful session. These two types of beans would be natural components to enhance with recovery guarantees in order to simplify the failure handling code that has to be written by the bean implementer. To a first approximation, session beans are candidates for the CIC or ICIC protocol, turning a bean into a persistent component, and entity beans are candidates for the caller side of the TIC protocol, turning a bean into a persistent component that interacts with a transactional component, namely, the underlying database server. An open issue, however, is to what extent this could be done without changing any of the EJB interfaces, and how such conceivable extensions compare to other component models (such as CORBA or .Net).

As of now [SUN 2001], the recovery capabilities stated in the EJB specification are very limited and involve significant amounts of explicit programming. For session beans, EJB includes a method, ejbPassivate, for saving the conversational state of a stateful bean onto persistent storage. However, this is merely a mechanism: the bean implementer must provide code for it, and it is up to the surrounding "beans container" when to invoke it. Furthermore, the use of this method is restricted; for example, it must not be called when program control resides in the bean nor when the bean has an open transaction with a data server (or, equivalently, when it is called, the bean implementer has to make sure that all transactions and JDBC sessions are closed). For entity beans, atomicity and persistence of the underlying database operations are guaranteed through the corresponding database systems and the Java Transaction Service (JTS) for the coordination of distributed transactions. However, when a bean implementer caches persistent data and manipulates it in the bean for efficiency, all updates on cached data are not subject to transactional control and it is the implementer's responsibility to include explicit code for writing back data to the underlying databases. So the capabilities currently provided by EJB are still far from being able to mask failures to application programmers and end users.

8. CONCLUSIONS

In this article we have developed a general framework for recovery guarantees in multitier Internet applications. The novel notion of committed interaction contracts, with exactly-once execution and best possible failure masking, is particularly useful for e-services, which currently may exhibit unexpected and

undesirable behavior due to failures, the consequences of which can greatly inconvenience the human user. 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 and to provide fast recovery. Our prototype implementation for a PHP-based web application server is a proof of concept and shows that the overhead of our protocols is acceptable. The fact that this implementation required surprisingly few changes to the source code of the PHP Zend engine (and relatively small special DHTML code on the browser side) indicates that our conceptual framework provides the "right" abstractions and can be easily adapted to real software environments.

Our framework applies to a wide spectrum of general multi-tier architectures, whether underlying components are database systems, other forms of data managers such as mail servers, application servers, message queues or workflow servers, or any application component. Our implementation of Phoenix/App shows that interaction contracts can be integrated with existing middleware architectures. In such a setting, recovery guarantees for persistent components complement the established notion of transactional components and provide value-added failure masking. In principle, our framework can be adapted to other middleware architectures such as J2EE and would provide additional benefits in these environments.

The key benefit of our contribution lies in masking failures not only to end users but also, to a large extent, to application programmers, thus largely relieving them of the need to write explicit code to cope with system failures. This simplifies application development, makes code more easily maintainable, and generally reduces the cost of application software lifecycles.

Automatic and efficient application recovery as provided by our protocols also improves the availability of e-services as perceived by end users. With very fast recovery, all transient failures and temporary outages would ideally be masked to internet users, but the internet infrastructure exhibits other idiosyncrasies and barely understood phenomena, such as load bursts, queueing delays, and timeouts, that affect the users' perception of whether a service is working well or not. Our long-term vision is to provide comprehensive quality-of-service guarantees for Internet-based e-services that include availability and responsiveness, as well as "world-wide" failure masking.

REFERENCES

ALVISI, L. AND MARZULLO, K. 1995. Message logging: Pessimistic, optimistic, and causal. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, Vancouver, Canada, May30–June 2, 1995. IEEE Computer Society, Los Alamitos, CA, 229–236.

BARGA, R., LOMET, D., AND WEIKUM, G. 2002. Recovery guarantees for general multi-tier applications. In *Proceedings of the 18th International Conference on Data Engineering*, San Jose, CA, February 26–March 1, 2002. IEEE Computer Society, Los Alamitos, CA, 543–554.

BARGA, R., LOMET, D., AGRAWAL, S. AND BABY, T. 2000. Persistent client-server database sessions. In *Proceedings (Lecture Notes in Computer Science, 1777) of the 7th International Conference on Extending Database Technology*, Constance, Germany, March 2000, C. Zaniolo, P. C. Lockemann, M. H. Scholl and T. Grust, Eds. Springer, Berlin and Heidelberg, Germany, 462–477.

BARGA, R. AND LOMET, D. 2001. Measuring and optimizing a system for persistent database sessions. In *Proceedings of the 17th International Conference on Data Engineering*, Heidelberg, Germany, April 2001. IEEE Computer Society, Los Alamitos, CA, 21–30.

BARGA, R., LOMET, D., PAPARIZOS, S., YU, H., AND CHANDRASEKARAN, S. 2003. Persistent applications via automatic recovery. In *Proceedings of the 17th International Database Engineering and Applications Symposium*, Hong Kong, China, July 2003. IEEE Computer Society, Los Alamitos, CA, 258–267.

BARTLETT, J. F. 1981. A NonStop kernel. In *Proceedings (Operating System Review 15(5)) of the 8th Symposium on Operation Systems Principles*, Asilomar, CA, December 1981. ACM, New York, 22–29.

BERNSTEIN, P. A., HSU, M., AND MANN, B. 1990. Implementing recoverable requests using queues. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, Atlantic City, NJ, June 1990, H. Garcia-Molina and H. V. Jagadish, Eds. ACM, New York, 112–122.

BERNSTEIN, P. A. AND NEWCOMER, E. 1996. *Principles of Transaction Processing*, Morgan Kaufmann, 1996.

BORG, A., BLAU, W., GRAETSCH, W., HERRMANN, F., AND OBERLE, W. 1989. Fault tolerance under UNIX. *ACM Transactions on Computer Systems 7*, 1, 1–24.

CRISTIAN, F. 1991. Understanding fault-tolerant distributed systems. *Comm. ACM 34*, 2, 56–78.

DEBULL 2001. *IEEE Bulletin of the Technical Committee on Data Engineering 24*, 1. Special Issue on Infrastructure for Advanced E-Services.

DUTTA, K., VANDERMEER, D., DATTA, A., RAMAMRITHAM K. 2001. User action recovery in internet SAGAs (iSAGAs). In *Proceedings (Lecture Notes in Computer Science 2193) of the 2nd International Workshop on Technologies for E-Services (TES)*, Rome, Italy, September 2001, F. Casati, D. Georgakopoulos and M.-C. Shan, Eds. Springer, Heidelberg and Berlin, Germany, 132–146.

ELNOZAHY, E. N., ALVISI, L., WANG, Y., AND JOHNSON, D. B. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv. 34*, 3, 375–408.

FREYTAG, J. C., CRISTIAN, F., AND KÄHLER, B. 1987. Masking system crashes in database application programs. In *Proceedings of 13th International Conference on Very Large Data Bases*, Brighton, UK, September 1987, P. M. Stocker, W. Kent, and P. Hammersley, Eds. Morgan Kaufmann, 407–416.

FRØLUND, S. AND GUERRAOUI R. 2000. Implementing e-transactions with asynchronous replication. In *Proceedings of 2000 International Conference on Dependable Systems and Networks*, New York, NY, June 2000. IEEE Computer Society, Los Alamitos, CA, 449–458.

FU, X., BULTAN, T., HULL, R., SU, J. 2001. Verification of vortex workflows. In *Proceedings (Lecture Notes in Computer Science 2031) of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Genoa, Italy, April 2001, Tiziana Margaria and Wang Yi, Eds. Springer, Berlin and Heidelberg, 143–157.

GRAY, J. AND REUTER A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

HAREL, D. AND GERY, E. 1997. Executable object modeling with statecharts. *IEEE Comput. 30*, 7, 31–42.

HUANG, Y. AND WANG, Y.-M. 1995. Why optimistic message logging has not been used in telecommunications systems. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing Systems*, Pasadena, CA, June 1995. IEEE Computer Society, Washington, D.C., 459.

JOHNSON, D. B. AND ZWAENEPOEL, W. 1987. Sender-based message logging. In *Proceedings of the 7th International Symposium on Fault-Tolerant Computing*, Pittsburgh, PA, July 1987. IEEE Computer Society, 14–19.

KIM, W. 1984. Highly available systems for database applications. *ACM Comput. Surv. 16*, 1, 71–98.

LOMET, D. 1998. Persistent applications using generalized redo recovery. In *Proceedings of the 14th International Conference on Data Engineering*, Sydney, Australia, Orlando, FL, February 1998. IEEE Computer Society, Los Alamitos, CA, 154–163.

LOMET, D. AND WEIKUM, G. 1998. Efficient transparent application recovery in client-server information systems. In *Proceedings of 1998 ACM SIGMOD International Conference on Management of Data*, Seattle, WA, June 1998, L. M. Haas and A. Tiwary, Eds. ACM, New York, NY, 460–471.

LOMET, D. AND TUTTLE, M. 1999. Logical logging to extend recovery to new domains. In *Proceedings of 1999 ACM SIGMOD International Conference on Management of Data*, Philadelphia, PA, June 1999, A. Delis, C. Faloutsos, S. Ghandeharizadeh, Eds. ACM, New York, NY, 73–84.

LUO, M.-Y. AND YANG, C.-S. 2001. Constructing zero-loss Web services. In *Proceedings IEEE INFOCOM 2001 of the 20th Joint International Conference of the IEEE Computer and Communication Societies on Computer Communications*, Anchorage, AK, April 2001. IEEE, Los Alamitos, CA, 1781–1790.

MOHAN, C., ET AL. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollback using write-ahead logging. *ACM Trans. on Database Syst. 17*, 1, 94–162.

MSDN LIBRARY: PERSISTENCE. Microsoft Internet Explorer Persistence Overview. http://msdn.microsoft.com/workshop/author/persistence/overview.asp.

MSDN LIBRARY: .NET. .NET Remoting Overview. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconnetremotingoverview.asp.

MSR PHOENIX. Phoenix: Making Applications Robust. http://www.research.microsoft.com/research/db/phoenix/.

OMG: CORBA 2000. Fault Tolerant CORBA Spec V1.0. http://cgi.omg.org/cgi-bin/doc?ptc/00-04-04.

OMG: UML 1999. OMG Unified Modeling Language (UML) Version 1.3.http://www.rational.com/uml.

PEDREGAL-MARTIN, C. AND RAMAMRITHAM, K. 1999. Recovery guarantees in mobile systems. In *Proceedings of the ACM International Workshop on Data Engineering for Wireless and Mobile Access*, Seattle, WA, August 1999. ACM, New York, NY, 22–29.

PEDREGAL-MARTIN, C., RAMAMRITHAM, K. 2001. Guaranteeing recoverability in electronic commerce. In *Proceedings of the 3rd International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems*, San Juan, CA, June 2001. IEEE Computer Society, Los Alamitos, CA, 144–155.

PHP. PHP Documentation and Downloads. http://www.php.net.

POPOVICI, A., SCHULDT, H., AND SCHEK, H.-J. 2000. Generation and verification of heterogeneous purchase processes. In *Proceedings of the 1st International Workshop on Technologies for E-Services*, Cairo, Egypt, September 2000, 5–22.

SCHULDT, H., POPOVICI, A., AND SCHEK, H.-J. 2000. Automatic generation of reliable e-commerce payment processes. In *Proceedings of the 1st International Conference on Web Information Systems Engineering*, Hong Kong, China, June 2000, Q. Li, Z. M. Özsoyoglu, R. Wagner, Y. Kambayashi, and Y. Zhang, Eds. IEEE Computer Society, Los Alamitos, CA, 434–441.

SHEGALOV, G., WEIKUM, G., BARGA, R., AND LOMET, D. 2002. EOS: Exactly-once E-Service Middleware (Demo Paper). In *Proceedings of the 28th International Conference on Very Large Data Bases*, Hong Kong, China, August 2002, P. A. Bernstein, Y. E. Ioaninidis, R. Ramakrishnan, D. Papadias, Eds. Morgan Kaufmann, 1043–1046.

SUN 2001. Enterprise Java Beans Specification, Version 2.0, http://java.sun.com/products/ejb/docs.html.

TYGAR, J. D. 1998. Atomicity versus anonymity—Distributed transactions for electronic commerce. In *Proceedings of the 24th International Conference on Very Large Data Bases*, New York, NY, August 1998, A. Gupta, O. Shmueli, and J. Widom, Eds. Morgan Kaufmann, 1–12.

WEIKUM, G. AND VOSSEN, G. 2001. *Transactional Information Systems—Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, San Francisco, CA, 2001.

WILLIAMS, M. 2002. *Microsoft Visual C# .NET*. Microsoft Press, Redmond, WA.

ZEND. Zend Engine. http://www.zend.com.

Received February 2003; revised April 2003; accepted July 2003