Phoenix Project: Fault-Tolerant Applications

Roger Barga   David Lomet
Database Research Group
Microsoft Research
One Microsoft Way,
Redmond, WA 98052

Abstract

After a system crash, databases recover to the last committed transaction, but applications usually either crash or cannot continue. The Phoenix purpose is to enable application state to persist across system crashes, transparent to the application program. This simplifies application programming, reduces operational costs, masks failures from users, and increases application availability, which is critical in many scenarios, e.g., e-commerce. Within the Phoenix project, we have explored how to provide application recovery efficiently and transparently via redo logging. This paper describes the conceptual framework for the Phoenix project, and the software infrastructure that we are building.

Introduction

The Problem

High availability is crucial to the success of mission critical applications of many businesses, especially those engaged in e-commerce. Unfortunately, system outages do occur, which frequently results in added costs or lost revenue. In the longer term, customer frustration with an e-commerce site may lead to even larger losses.

Database systems deal with system crashes in a robust manner, recovering to the last committed transaction. This technology is mature and cost effective. Unfortunately, applications do not share this robust behavior. It is very difficult for an application programmer to deal well with a system crash. Either programming style must be tightly controlled, e.g. stateless applications that have no meaningful state between transactions, or subtle and complicated programming is needed to deal with failures. In either case, the application programmer needs to be aware that system crashes are possible, and write the application accordingly. A common outcome is that applications fail when a system crashes. The burden then manifests as user frustration and operational headache as ad hoc manual efforts are made to restore an application's state and restart it.

Application programmers tend to write stateful applications, retaining information necessary for correct execution across transaction boundaries. This natural style of programming is reflected in standard SQL with its session state. The problem here is losing the state when the system crashes. This may create a "semantic mess" that frequently requires human intervention to repair. Outages can be very long because of this. Classical TP monitors [BHM90, GrRe93] insist that applications be stateless, a rather unnatural programming style. Programming stateless "workflows" then requires a multi-transaction application in which each step commits and passes its "state" in a transactional queue to the next step of the workflow.

Phoenix Project

The Phoenix goal is to enable robust applications to be written "naturally" as stateful programs. Importantly, the application programmer is not required to take special measures to ensure the persistence of application state. Phoenix deals with system failures "transparently", logging component interactions and checkpointing state to ensure that applications can be automatically recovered should a crash occur. This enhances application availability by avoiding the extended down-time resulting from manual intervention, and simplifies application programming by avoiding the need to deal with such failures.

To realize our goal, we have explored recovery principles and have built two prototype systems to demonstrate the practicality of our approach. We start with our work on recovery principles, and then devote two sections, one for each of our prototype systems. A final section describes how our prototype systems can be brought together in a system that provides an end-to-end exactly once execution guarantee.

Recovery Foundations

We have worked to develop the underlying technology that enables recovery. This body of work includes both redo recovery principles and optimizations and characterization of recovery requirements for distributed systems.

Redo Recovery Principles

The task of redo recovery after a system crash is to reconstruct the state of the database at the time of the crash by redoing some subset of logged operations in some order. Supporting recovery is hard because it requires careful coordination of changes to the state and the log by many system components, e.g., cache manager, log manager, installation procedure, checkpoint procedure, and recovery procedure. Our recent work [LoTu02] is a theory in which we can state the property (invariant) that this coordination must implement. It builds on our prior Phoenix recovery work [LoTu95,Lo98, LoTu99].

We define a simple, abstract system model, which includes state, changes to the state, installing these changes into the state, and recovering the state. We relate operation sequences to state sequences and operation graphs to state graphs.

Our theory describes the states that are ***potentially recoverable***. An ***installation graph*** characterizes the sets of operations that can "appear" in recoverable states. Only some of the variables have values that are ***exposed*** to the recovery process, and only these exposed values need be ***explained*** by the operations in the state.

We give a general recovery procedure with a ***redo test*** that is invoked as it reads the log to determine whether it should or should not redo a logged operation. We prove the recovery procedure correct, assuming that the redo test selects a set U of operations from the log to redo, and that the complement I of this set explains the exposed values in the state.

Many real redo recovery algorithms can be modeled in our theory, e.g. LSN based techniques [MHLPS92]. These support recovery exactly by keeping invariant the correctness condition for our recovery procedure. This invariant becomes the contract that must exist between state installation and recovery in order for the phases of normal operation before a crash and system recovery after a crash to interact seamlessly.

Distributed Application Recovery

Most directly relevant to application persistence is our framework for recovery guarantees in general multi-tier application that makes application state persistent [BLW02]. To provide this strong guarantee:

- We require piecewise determinism [EJW96] and identify the needs for logging specific non-deterministic events. This ensures that after a failure, an application can be replayed from an earlier installed state and arrive at the same (abstract) state as in its pre-failure incarnation.
- We introduce **interaction contracts** between communicating pairs of application components. For example, a committed interaction contract (**CIC**) between two persistent components requires guarantees related to persistence of sender and receiver state and messages. Contracts exist for persistent component interactions with external components (**XIC**) (including users) and with transactional components (**TIC**) that provide all-or-nothing state transitions (but not exactly-once executions).
- We compose contracts into system-wide agreements such that persistent components are provably recoverable with exactly-once execution semantics. We strictly separate the obligations of a contract from its implementation in terms of logging. We can thus give strong guarantees to the external users while frequently avoiding expensive measures such as forced logging [LoWe98].

Persistent Database Sessions via Phoenix/ODBC

Phoenix/ODBC [BL99, BL2000, BL2001] was our first prototype system. It insulates applications from database server failures. It does not provide recovery from client (application) system failures.

ODBC Background

ODBC (Open Database Connectivity) is a client application API to SQL database servers based on the X/Open SQL Call Level Interface standard. Applications use ODBC to access data in any of the major commercial DBMS's, all of which support ODBC. An application makes data accesses via ODBC using SQL statements written in either ODBC SQL or DBMS-specific SQL. The application exploits the following standard client side software when using ODBC.

ODBC Driver: a DBMS vendor provided module that responds to all client application calls to the ODBC API. The driver translates SQL statements into DBMS-specific SQL that it passes to the server and reformats results returned from the DBMS into ODBC format.

ODBC Driver Manager: a platform component that manages communications between the application and vendor provided ODBC drivers. The driver manager loads the ODBC driver appropriate to the database being accessed and passes all application requests to it.

An Example ODBC Database Session

Our example ODBC database session involves three tables: a master customer table, a detail orders table, and summary invoice table. The task is to extract the appropriate records for a customer with the last name "Smith," find that customer's current orders, and aggregate the order totals into the invoice summary table. This client application might be coded as in Figure 1:

1. Create an ODBC session by opening a connection to the server, log on the database and set session specific attributes
2. Request the server create a result set from the customer table (**A**) consisting of records with a last name of 'Smith'
3. Fetch customer records from the result set, until the appropriate customer record is found.
4. Open a cursor on the orders table (**B**) for orders matching this customer's ID
5. Fetch all matching order records for this customer ID.
6. Calculate the aggregate of the order records.
7. Send a command to update the invoices table (**C**) with the calculated aggregate.
8. Close connection to database, terminating the session.

Figure 1: Steps of a "typical" ODBC application.

Consider what would happen were the database server to fail during this ODBC database session. The main problems are:

- **Server Failure:** ODBC functions can have undefined behavior when the server is down this can require that the application be terminated.
- **Application Availability**. The application's session with the database is lost. The resulting partial session execution can leave the application state confused, requiring long outages in order to reconstruct it manually.
- **Loss of Transient State.** If a failure occurs after the application has created volatile state, e.g. result sets from SQL statements, this state is lost.
- **Extra Complexity.** For an application to cope with database server failure requires additional code to deal with these problems. This increases application complexity, delays deployment, contributes to bugs, and can further reduce overall application availability.

Phoenix/ODBC Implementation

To provide session availability, Phoenix/ODBC, a Phoenix-enabled ODBC Driver Manager, wraps any native ODBC driver, intercepting application requests going to the database as well as responses returned from the server. This architecture is illustrated in Figure 2 below.

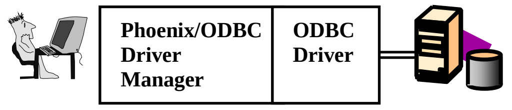

Figure 2: Illustration of the main components in the Phoenix/ODBC architecture.

Phoenix/ODBC Actions

Phoenix/ODBC creates a virtual ODBC database session for an application (step 1 of our example) and maps it to one or more real ODBC database sessions. It detects server failures and recovery by timing out requests and pinging the server until it recovers. It then re-associates the virtual session and its associated state to a new ODBC session when it reconnects to the server. Finally, when the session terminates it cleans up any persistent session state that was created (step 8).

Phoenix/ODBC makes transient session state persistent. It logs statements that alter session context (statement 1). It rewrites SQL statements to create persistent database tables that capture application session state, before passing the request on to the native ODBC driver (result sets of statements 2 and 4 will be made to persist). Phoenix/ODBC intercepts server responses, variously caching, filtering, and reshaping result sets, and synchronizing with state materialized on the database server (partially delivered result sets in statements 3 and 5 are synchronized to provide seamless delivery).

Decomposing Application State

We decompose server session state into elements, each of which has a different lifetime and recovery requirements. These include:

ODBC Session Context – All client settable attributes of a session, including connection request and user login information.

Environment, connection, and statement attributes – Context, not associated with attributes, includes user identification, current database, user temporary objects, and unacknowledged messages sent by the server to the client.

Result Generating SQL Statement – SQL statement that will return one of following:

- A result set for a SELECT statement.
- A global cursor that can be referenced outside the SQL statement.
- A return code, which is an integer value.
- Output parameters, which can return either data or a cursor variable.

Delivery of SQL Statement Results

A challenging aspect of masking server failures is seamless delivery of results to the client. If the statement generates a result, Phoenix/ODBC takes the following steps to ensure it will be recoverable in case of server failure.

1. Access the result set metadata that describes the columns in the result set.

2: Use the metadata to generate a CREATE TABLE statement that is sent to the server to create an empty persistent table for the result.

3: Execute the SQL statement to insert its result in this persistent table at the server.

4: Keep track of the current location in the now persistent result set. After a failure, access is resumed at the remembered location of the last access before the failure.

At step 3, Phoenix/ODBC creates a stored procedure that encapsulates the application's SQL statement. The advantage of using a stored procedure is that all the data is moved locally at the server. This procedure, using only ANSI-standard SQL, is:

```
CREATE PROCEDURE P (@T string) AS
INSERT <original application SQL
statement>
INTO T
```

Procedure execution is an atomic SQL statement. Phoenix/ODBC then issues the SQL statement SELECT * FROM **T** to open the table and returns control to the application.

Summary

Phoenix/ODBC relieves the application developer from coping with the programming complexity of handling server failures, increases the availability of the application, and in many cases avoids the operational task of coping with an error. Indeed, a user of the application may not be aware that a database server crash has occurred, except for some delay.

Persistent Components via Phoenix/COM$^+$

Phoenix/COM$^+$ is a runtime service that provides transparent state persistence and automatic recovery for component-based applications.
Many transient system failures are recoverable in this way because failures are frequently so-called "Heisenbugs".

Phoenix/COM$^+$ is implemented as a service in Microsoft's .NET runtime and supports any component based application. The implementation is based on the recovery guarantees framework [BLW02], the techniques and protocol of which offers several distinct advantages:

- Exactly-once execution semantics;
- Protocols that minimize logging cost, especially log forces, and enable efficient log management;
- Recovery independence, allowing components to recover independently;

To achieve fault-tolerance using Phoenix/COM$^+$ requires the application programmer to identify the stateful components and declare them as **persistent**, **transactional,** or **external**. The service transparently logs component interactions and, if a failure occurs, automatically recovers all components marked **persistent** up to the last logged interaction. Components marked **transactional** are recovered up to the last successfully completed transaction – transactions in-flight during the failure will be aborted by the database and it's assumed the application is written to deal with (retry) failed transactions – a reasonable assumption for transactional applications. **External** components are outside of the boundary of the system, and cannot be recovered. However, we limit our dependence upon them by prompt logging of interactions with them.

Developing Fault-tolerant Applications

Phoenix/COM$^+$ offers three distinct advantages to application programmers in developing a fault tolerant enterprise application:

Stateful Applications

Phoenix/COM$^+$ enables component-based applications to be written "naturally" as stateful programs. Importantly, the application programmer is not required to take any special measures to ensure the persistence of application state. Phoenix/COM$^+$ deals with system failures by logging component interactions and checkpointing state to ensure that application state can be automatically recovered.

Error Handling

Stateless applications have difficulty dealing with aborted transactions. If an application executing a transaction were to fail, there is no easy way for aborted transactions to return error status. Nor can the application provide, in response to the error, a programmatic way of dealing with the error that is specific to the nature of the error. There is simply no place to put the program logic, which must exist outside of the failed transaction.

Phoenix/COM$^+$ permits stateful application components to "begin" and "end" transactions, and, because these components have state outside of transactions (and state that persists across system failures), they are able to see transaction error codes, and act on them as appropriate. For example, such an application can test the error code to decide whether to simply re-execute the transaction or change input parameters before re-execution, and can decide at what point to abandon the effort and itself return an error code describing what has happened.

Debugging

The log captures the precise set of events that led a system to a failure. If this is a hard failure (a "Bohrbug"), the failed state will be re-created. This facilitates the debugging of distributed enterprise systems. Even when the failure is soft (a "Heisenbug"), the log should offer clues as to what when wrong.

System Attributes (the "ities")

Phoenix/COM$^+$ captures component state by logging interactions between components, forming an *event history*. This history is on the log and can be used to recover, i.e. make persistent, a component's state up to the last logged interaction. That is, Phoenix/COM$^+$ captures the ***execution state*** of a component while it is active. This ability to make application state persistent at a fine execution granularity provides important benefits to enterprise applications.

Availability

Phoenix/COM$^+$ provides high availability by performing database style recovery for application components using its own logging and recovery infrastructure. We (in the longer term) avoid double logging, and exploit logical logging as much as possible. This minimizes normal execution cost.

For most middle tier applications, the majority of the response time is the result of communication with other remote applications or resource managers. Replay of an application replaces those interactions with the logged effects of the interactions that took place originally. Hence replay is much faster than original execution. The fast replay enhances application availability.

Scalability

Scalability comes largely from reusing resources drawn from a pool of anonymous resources, each in a suspended state. For example, a database connection is multiplexed among many applications. When an application is finished with it, the connection is returned to the pool and is reused by the next application. Similarly, application components themselves can be multiplexed between user requests when components are stateless between requests.

Phoenix/COM$^+$, however, permits components to retain state between requests, whether the requests are from users or other components. It also permits components to have state between transactions. We can "pickle" the state: between requests by taking a checkpoint; or at an arbitrary execution point because Phoenix/COM$^+$ captures the logged interactions from the point at which the state first became important up to the current execution point. Logical identification of COM$^+$ components permits us to re-instantiate the state using a process or thread from a pool of such resources. Since replay is fast, this is a feasible way of dealing with reclaiming a thread or process and permitting another request to utilize this underlying system resource.

Load Balancing

In a clustered server environment with shared disks, Phoenix/COM$^+$ allows transparent fail over of stateful components from one machine to another. For example, a set of stateful components may be running on machine A, where the state for these components has been recorded in the log on a shared disk. If an administrator notices A is overloaded, he can selectively terminate some stateful components on A and direct their failover to Machine B, where subsequent client requests can pick up where they left off. This works because the component state is automatically recovered and the runtime updated to redirect subsequent calls for these components to B. Importantly, this failover can occur at any time, not just between requests, because Phoenix/COM$^+$ captures state at any execution point.

Phoenix/COM$^+$ Implementation

Phoenix/COM$^+$ exploits the component wrapper mechanism in Microsoft's .NET runtime that transparently intercepts component events, such as creation, activation, and method calls. This architecture is illustrated in Figure 3. While Phoenix/COM$^+$ is implemented using the Microsoft .NET runtime, the techniques should be relevant to other middleware architectures, e.g., CORBA and Enterprise Java Beans (EJB).

The Phoenix/COM$^+$ runtime masks failures from the application and initiates recovery to reconstruct impacted components. Phoenix/COM$^+$ mechanisms do the following:

1. **Logging** – Intercept interactions between components and logs information sufficient to recreate components and recover their state to the last logged interaction.
2. **Error Detection** – Detect errors arising from component failure and masks the error from the client, and initiates component recovery.
3. **Component Recovery** – Recreate an instance of a failed component and *re-install* its state by replaying intercepted calls of the failed component from the log. After recovery, COM$^+$ runtime tables are updated to direct future calls to the new component.

Post failure execution requires a different process (or thread) than used by the original execution. Therefore, Phoenix/COM$^+$ virtualizes components, providing each with a logical id independent of its mapping to a process. This logical id identifies the program and state of a component. During execution, this id is mapped to the specific threads and/or processes that realize the component.

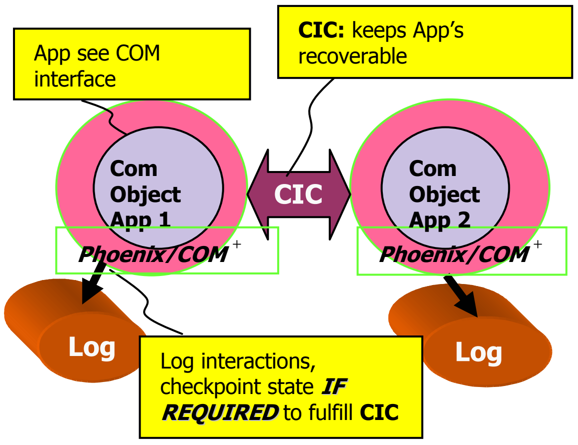
App see COM interface
CIC: keeps App's recoverable
Com Object App 1
CIC
Com Object App 2
Phoenix/COM+
Phoenix/COM+
Log
Log
Log interactions, checkpoint state IF REQUIRED to fulfill CIC

Figure 3: Illustration of the main Phoenix/COM$^+$ elements involved with making components persistent.

Exactly Once Semantics

The overall Phoenix goal is to provide "exactly once" semantics. That is, a user should perceive the behavior of the system as if no failure has ever occurred and his request was executed seamlessly, from the time it was submitted until he receives a reply. This requires a complete, end-to-end story. Phoenix work, along with other work done with colleagues, can provide this seamless exactly once execution. The previous section described how Phoenix/COM$^+$ provides persistent components in the middle tier. In this section we describe how we can provide transactional components for database interactions and extend persistence to the desktop via enhancing the an internet browser.

Transactional Components

Transactional components are components that only promise that state and messages will persist should the transaction they are engaged in commit. If the transaction aborts, all information, including even knowledge of the existence of the transaction, can disappear. Note that because a SQL database does not automatically persist session state or result messages, that the database by itself is not a transactional component. Hence, normal interactions with such a database do not provide the required guarantee.

Phoenix/ODBC provides exactly the persistent state and messages required of a transactional component when interacting with a SQL database. Only the syntactic nature of its interface prevents it from being a COM$^+$ component, and hence a Phoenix transactional component. It supports the transactional interaction contract (TIC) between persistent and transactional components. Unlike the committed interaction contract, the strong guarantees of the contract are only required should the transaction commit. If the transaction aborts, amnesia is possible for either transactional or persistent component, without sacrificing any recovery guarantees. Our stateful persistent components can safely engage in Phoenix/ODBC "sessions" extending over several transactions. If a transaction aborts, the persistent component's application program can take its own remedial action.

Persistent Browser State

Gerhard Weikum and German Shegalov have implemented a prototype system at the U. of the Saarland [BLSW2002] that extends recoverable components to 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, they have built the prototype using IE5 as browser, Apache as http server, and PHP as servlet engine. The prototype transparently provides an external interaction contract XIC between the browser and the user, and a CIC between the browser and the mid-tier application server. This permits both browser and mid-tier application to be persistent components. This should be very relevant for Internet-based e-services. 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 [Zend], and modifications of the ODBC-related PHP functions as well as additional stored procedures in the underlying database.

End-to-End Story

A system design based on these prototypes can provide a user with the exactly-once semantics that is the Phoenix goal. A user ("external component") interacts with an enhanced browser (a persistent component). The browser in turn interacts with middle-tier web and application servers (more persistent components). And some middle tier persistent components subsequently interact with "augmented" database systems (transactional components). All enforce the appropriate contracts and in a way that is transparent to the application programmer. The result is exactly once execution of user requests.

Discussion

Our Phoenix effort is a work in progress. We expect to continue to work on increasing the robustness of applications. An important element of this work will also be to reduce performance penalty in providing robustness. So stay tuned! The Phoenix project home page is http://www.research.microsoft.com/db/phoenix. This page includes links to references and a regularly updated view of the project status.

Acknowledgments

We have benefited greatly from collaboration with exceptional colleagues. Gerhard Weikum worked with us on persistent applications, both client/server and multi-tier. German Shegalov, Gerhard's student, built the internet persistence system. Mark Tuttle worked with us on redo recovery principles. Excellent interns spent summers (in one case several months) working with us to build our prototype systems: Sanjay Agrawal, Thomas Baby, Sirish Chandrasekaran, Stelios Paparizos, and Haifeng Yu.

References

[BL99] R. Barga, D. Lomet. Phoenix: Making Applications Robust. (demo paper) *SIGMOD Conf.***,** Philadelphia, PA (1999)

[BL2000]R. Barga, D. Lomet, T. Baby, S. Agrawal. Persistent Client-Server Database Sessions, *Conf. on Extending Database Technology*, Lake Constance, Germany (2000)

[BL2001] R. Barga, D. Lomet. Measuring and Optimizing a System for Persistent Database Sessions. *Int'l Conf. on Data Engineering*, Heidelberg, Germany (2001)

[BLW2001] R. Barga, D. Lomet, G. Weikum. Recovery Guarantees for Multi-tier Applications. *Int'l Conf. on Data Engineering*, San Jose, CA (2002)

[BLSW2002] R. Barga, D. Lomet, G. Shegalov, G. Weikum. Recovery Guarantees for Internet Applications. (submitted)

[BHM90] Bernstein, P.A., Hsu, M., Mann, B.: Implementing Recoverable Requests Using Queues. *SIGMOD Conf.*, Atlantic City, NJ (1990).

[EJW96] E. Elnozahy, D. Johnson, Y-M Wang, A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *Tech. Rept*, CMU, Pittsburgh, 1996.

[GrRe93] Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA (1993).

[**Lo98**] D. Lomet. Persistent Applications Using Generalized Redo Recovery. *Int'l Conf. on Data Engineering*, Orlando, FL (1998).

[**LoTu99**] D. Lomet, M. Tuttle. Logical Logging to Extend Recovery to New Domains. *SIGMOD Conf.*, Philadelphia, PA (1999).

[LoTu2002] D. Lomet, M. Tuttle. Principles of Redo Recovery. (submitted)

[LoWe98] D. Lomet, G. Weikum. Efficient Transparent Application Recovery in Client-Server Information Systems, *SIGMOD Conf.*, Seattle, WA (1998).

[MHLPS92] C.Mohan,et al. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM TODS 17,1 (Mar. 1992)

[Zend] Zend Engine, www.zend.com