Phoenix Application Recovery Project

Roger S. Barga
Database Research Group, Microsoft Research
One Microsoft Way, Redmond, WA 98052

Abstract

High availability for both data and applications is rapidly becoming a business requirement. Yet even after decades of software engineering research, computer systems and production applications still fail [11], primarily due to nondeterministic bugs that can typically be resolved by simply rebooting the system or restarting the application [6]. The Phoenix project takes the position that such hardware faults and software failures are facts to be coped with, not problems to be solved. This position is supported both by historical evidence and by recent studies on the primary sources of outages in production systems [11]. Conceding that Heisenbugs will remain a fact of life in computing, the goal of the Phoenix project is to enable applications to survive system crashes, without requiring application developers to take special measures for state persistence and application recovery. This simplifies application programming, reduces operational costs, masks failures from users, and increases application availability. In this paper we describe the principles behind the Phoenix project and outline some research areas and recent projects.

1 Introduction

Availability is as important to e-commerce as breathing in and out is to human beings.
- Kal Raman, CEO Drugstore.com.

High availability is crucial to the success of any company engaged in e-commerce and e-business. There is a growing consensus within parts of the systems and research communities that the traditional focus on performance has become misdirected in enterprise computing, where availability has become at least as important as peak performance, if not more so [7]. One need only open a recent issue of the *New York Times* or *Wall Street Journal* to find evidence of this – the number of articles on recent outages of big e-commerce providers and the economic impact of those outages is striking. In April 2002 when Ebay suffered a 22-hour sustained outage affecting most of its online auctions the loss of revenue was estimated at five million US dollars [17] but their stock value fell by 26% during this downtime, representing a loss of over five billion dollars in market capitalization. This and similar outages are highly visible, noteworthy, and affect customer loyalty and investor confidence [9]. While the immediate result of key application downtime is lost revenue, in the longer term customer frustration, damaged reputation and investor confidence may lead to even greater losses.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Unfortunately, system outages and application failures do occur. The Gartner Group [11] estimates that over 40% of unplanned downtime in business environments is due to application failures; 20% is due to hardware faults, of which 80% are transient [8, 15], hence resolvable through system restart. Most bugs in production quality software are Heisenbugs [8, 12, 1]. Heisenbugs are difficult to reproduce, or depend on the timing of external events, and often there is no better way to work around them other than to simply restart the application. While restart will work around transient hardware failures and Heisenbugs in software, the volatile state of user sessions and running applications is lost, exposing the failure to end users. Hence, automatic restart alone does not fully address the problem.

Transaction processing monitors, exploiting a database or durable queue, and "stateless" applications, have long been the preferred solution for constructing failure-resilient enterprise applications. Stateless applications are not truly 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 back into a queue or database. 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 electronic services, however, such prior solutions have not carried over to multi-tier web architectures. Indeed, how they might be readily adapted to deal with middle-tier web application servers is complex and still speculative. Clearly, the enterprise computing world is different today than it used to be. With traditional back-office transaction processing systems being supplanted by multi-tier, Internet-based service delivery, and component-based application servers, the assumptions behind system design have changed, and the traditional techniques for providing high availability have lost much of their power. Since high availability is as important today as it has always been, we need new techniques and approaches for improving application availability, ones based on the realities of modern server environments.

2 The Phoenix Project

The goal of the Phoenix project, a project within the Database Group at Microsoft Research, is to enable applications to survive system failures, in particular Heisenbugs in software and transient hardware failures. Current members of the Phoenix project include David Lomet and Roger Barga. Our approach is to enable the underlying system to take responsibility for application state persistence across failures. It is the system that acts to ensure that volatile session and application state is recoverable by transparently logging application interactions, and should a failure occur it is the system that will automatically recover the application on restart. This enables applications to be written naturally as stateful programs and simplifies application development by eliminating the need to write explicit code to persist application state and to deal with failures. This approach also enhances overall application availability by avoiding the extended down-time that failures can produce when manual intervention is needed to make sense of failed application state and restart the application. A secondary goal of our work is to be general enough to enable the implementation of other useful state management services, such as resource management and administrative utilities.

To realize this goal for multi-tier Internet based applications with communicating components, a comprehensive form of data, component state, and message recovery is required, going beyond traditional database recovery and transaction processing. In designing protocols to accomplish this we have addressed a number of issues, such as:

- To what extent can failures be masked from end users, and which logging actions are necessary for this.
- Which system 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, e.g., database servers, kept from being held "hostage" to other components (applications servers, clients), which may interfere with their independent recovery or normal operation?

To realize our goal, we have explored logging and recovery principles, resulting in a framework for recovery guarantees in multi-tier applications, and have built a number of prototype systems to demonstrate the practicality of our approach. In the remainder of this article we outline our framework for recovery guarantees and then describe three of our more recent projects based upon this framework.

3 Selected Research Areas and Projects

Our recent project work is based on a framework for recovery guarantees in general multi-tier applications that makes system component state persistent, as described in [5]. To a user, the framework masks all failures such that the user's initial request has exactly-once execution semantics. To provide this guarantee for applications we require piecewise deterministic applications and identify the needs for logging specific non-deterministic events. This ensures that after a failure, a system component can be replayed from an earlier installed state and arrive at the same state as its pre-failure incarnation. There are three types of components considered in our original version of this framework: eXternal components (**XCom**) modeling human users and parts of the system outside the framework, Persistent components (**PCom**) representing mid-tier applications, and Transactional components (**TCom**) modeling resource managers such as database servers and transactional queues.

The framework introduces *interaction* contracts between two persistent components. For example, a committed interaction contract between persistent components requires guarantees related to persistence of sender and receiver state and messages. Contracts exist for persistent component interactions with external components (including users) and with transactional components that provide all-or-nothing state transitions (but not exactly-once executions).

Finally, 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 forced logging and expensive measures.

3.1 Persistent Middleware via Phoenix/APP

Phoenix/APP is a runtime service that provides transparent state persistence and automatic recovery for component-based applications [4], and was the first prototype system based on the recovery guarantees framework. Building highly available enterprise applications using web-oriented middleware is difficult. Consider an enterprise middle tier application. It is typically made up of one or more server components that implement business logic and expose a set of interfaces. A handful of the components in the application may access persistent data, typically stored in a relational database. Many components perform a specific task (calculation, data formatting, etc) on behalf of server components, possibly modifying data and returning a result but they retain no state. Finally, there are a small number of critical components which maintain state for the application during the session. A classic example is a middle tier e-commerce system for shopping and price comparison. A client session begins when a customer logs in and customer information is read from a database into a component. As the customer shops, purchases are recorded in the stateful component representing the market basket. When the customer checks out, items in the market basket are written to databases (e.g., orders and billing) and the customer database is updated to reflect recent activity. If a failure occurs during the session, all volatile state is lost and the session must be restarted.

To achieve fault-tolerance using Phoenix/APP simply requires the application programmer to identify the stateful components and declare them as *persistent*, *transactional* or (by default) *stateless*. The runtime service transparently logs volatile state and, if a failure occurs, automatically recovers all components marked *persistent* up to the last logged interaction[1]. Components marked *transactional* are recovered up to the last successfully

[1]The time required to recover failed components is a consideration. For middle tier applications, the major response time overhead

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. The remaining components in the application, which by default are treated as stateless, are simply restarted (no state is recovered). The application can continue with the session without loss of state (market basket, orders, etc) and having to take any special actions for it own recovery or state persistence.

Phoenix/APP is implemented on the Microsoft .NET framework and supports any component based application. Our implementation is based on a mechanism called *contexts*, a component wrapper mechanism that transparently intercepts object events, such as creation, activation, and method calls. New component services can be added to the .NET runtime by implementing "handlers", referred to as *policies*, for object events and method calls, and including them "in the context". Phoenix/APP runtime service for automatic component recovery is the composition of context policies that work together to mask failures from the application and initiate recovery to reconstruct impacted components. These policies involve:

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

Post failure execution requires a different process and thread than used by the original execution. Therefore, Phoenix/APP virtualizes components, providing each with a logical id that is independent of its mapping to a physical process or thread. This logical id is used to identify the program and the persistent state of a component. During execution, this id is then mapped to the specific threads and/or processes that realize the object. Further details on the design and implementation of Phoenix/APP can be found in [4].

The Phoenix/APP effort has benefited from the efforts of several graduate students interning with our group. Sirish Chandrasekaran (UC Berkeley, Summer 2000) contributed to the initial design of Phoenix/APP. Stelios Paparizos (Northeastern University, Winter 2000 and Summer 2001) extended the functionality of the prototype during his first internship and during his second internship, together with Haifeng Yu (Duke University, Summer 2001), they ported the prototype to the .NET runtime and performed a detailed performance evaluation.

3.2 New Component Types for Improved Logging and Recovery Performance

In Phoenix/APP, logging, in particular forced logging, is the largest performance cost associated with the normal execution. Thus, reducing forced logging is the most important aspect to reducing the overhead required to provide application state persistence. In our paper describing Phoenix/APP [4] we introduced three new component types, *subordinate components*, *functional components*, and *read-only components*. Each of these new component types has reduced logging overhead during interactions between it and other component types. As with our previous component types, *persistent*, *transactional* and *external*, an application builder simply declaratively states the type of the component in their application and the Phoenix/APP runtime service will tailor its logging actions based on the type of the component involved.

During his internship with our group, Shimin Chen (CMU, Summer 2002) examined the logging and recovery requirements in Phoenix/APP to eliminate unnecessary logging and log forces, and took advantage of these new component types and methods to improve runtime logging overhead. In addition, he devised mechanisms to checkpoint component state so that the log could be truncated, reducing processing costs during recovery,

is communication with remote components or resource managers. During recovery, the service replaces these interactions with logged effects of the original interactions. Hence replay is much faster than original execution.

and introduced methods to coordinate the state saving of multiple components in a process. These enhancements were implemented in a new version of the Phoenix/APP research prototype, which is now the subject of a performance evaluation that will be presented in a forthcoming paper [3].

3.3 EOS - Persistent State for Internet Browsers

Gerhard Weikum and German Shegalov have implemented a prototype system at the University of the Saarlands that extends our recoverable components work 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 [16]. Specifically, they have built the prototype using Microsoft's IE5 as the browser, Apache as the http server, and PHP as the servlet engine. The prototype transparently provides an external interaction contract XIC between the browser and user, and a CIC between the browser and mid-tier application server. This permits both browser and mid-tier application to be persistent components. This should prove to 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, modifications of the source code of the PHP session management in the Zend engine, and modifications of the ODBC-related PHP functions as well as stored procedures in the underlying database, as described in [16].

Additional Information on the Phoenix Project
The Phoenix project home page is http://www.research.microsoft.com/db/phoenix. This webpage is updated regularly and includes links to relevant references.

References

[1] E. Adams. Optimizing preventative service of software products. IBM Journal of Research and Development, 28(1):2-14, 1984.

[2] Nick Allen. Gartner Group Research Report, 2000.

[3] S. Chen, R.S. Barga and D. Lomet. Improving Logging and Recovery Performance in Phoenix/APP. In preparation, 2002.

[4] R.S. Barga, D. Lomet, S. Paparizos, H. Yu and S. Chandrasekaran, Persistent Applications via Automatic Recovery. Submitted 2002.

[5] R.S. Barga, D. Lomet and G. Weikum, Recovery Guarantees for General Multi-Tier Applications. ICDE 2002.

[6] G. Candea and A. Fox, Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel. HotOS-VIII, 2001.

[7] D. Patterson, et. al., Recovery Oriented Computing(ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB CSD-02-1175, U.C. Berkeley, March 2002.

[8] T.C. Chou. Beyond fault tolerance. IEEE Computer, 30(4):31-36, 1997.

[9] Chet Dembeck. Yahoo cashes in on Ebay's outage. E-commerce Times, June 18, 1999. http://www.ecommercetimes.com/perl/story/545.html

[10] http://www.forrester.com/research/cs/1995-ao/jan95csp.html.

[11] http://www.gartner.com/hcigdist.htm.

[12] J. Gray. Why do computers stop and what can be done about it? In Proc. Symposium on Reliability in Distributed Software and Database Systems, pages 3-12, 1986.

[13] J. Gray. Locally served network computers. Microsoft Research white paper. February 1995. Available from http://research.microsoft.com/ gray.

[14] D. Scott. Making smart investments to reduce unplanned downtime. Tactical Guidelines Research Note TG-07-4033, Gartner Group, Stamford CT, 1999.

[15] D. Milojicic, A. Messer, J. Shau, G. Fu and A. Munoz. Increasing relevance of memory hardware errors - a case for recoverable programming models. In ACM SIGOPS European Workshop “Beyond the PC: New Challenges for the Operating System”, Kolding, Denmark, Sept. 2000.

[16] G. Shegalov, G. Weikum, R. Barga, D. Lomet: EOS: Exactly-Once E-Service Middleware (Demo Description), Proceedings of International Conference on Very Large Data Bases, Hong Kong, China, 2002 (pp. 1043-1046).

[17] D. Scott. Operation Zero Downtime, a Gartner Group report, Donna Scott, 2000.