Persistent Applications via Automatic Recovery

Roger Barga, David Lomet
Microsoft Research
{barga, lomet}@microsoft.com

Stelios Paparizos
University of Michigan
spapariz@umich.edu

Haifeng Yu
Duke University
yhf@cs.duke.edu

Sirish Chandrasekaran
UC Berkeley
Sirish@cs.berkeley.edu

Abstract

Building highly available enterprise applications using web-oriented middleware is hard. Runtime implementations frequently do not address the problems of application state persistence and fault-tolerance, placing the burden of managing session state and, in particular, handling system failures on application programmers. This paper describes Phoenix/APP, a runtime service based on the notion of recovery guarantees. Phoenix/APP transparently masks failures and automatically recovers component-based applications. This both increases application availability and simplifies application development. We demonstrate the feasibility of this approach by describing the design and implementation of Phoenix/APP in Microsoft's .NET runtime and present results on the cost of persisting and recovering component-based applications.

1. Introduction

Application developers, unless instructed otherwise, tend to write stateful applications, retaining information necessary for correct and successful execution across interaction and transaction boundaries. However, stateful applications risk losing state when the system on which they execute crashes. This can create a "semantic mess" that may require human intervention to repair or restart the application, resulting in long service outages. The classic response to this is to insist that an application be "stateless", where stateless means "no meaningful information is retained across transactions". Unfortunately, stateless applications force a rather unnatural form of workflow programming. The application must, within a transaction, first read its state from, e.g., a transactional queue, execute its logic, and then commit the step by writing its state back to a queue for the next step. This programming model also has a potential performance problem due to the need for two phase commit (2PC).

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 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. This can result in lost revenues and frustrated customers. Depending on the application and point of failure, updates may have to be manually backed out to restore consistency to the system. To avoid this, middle tier systems have been designed as stateless applications. This unnatural programming style increases development costs and may reduce system throughput.

1.1 Phoenix/APP for Improved Availability

In this paper we describe Phoenix/APP, a runtime service that provides transparent state persistence and automatic recovery for component-based applications. Phoenix/APP is based on the recovery guarantees framework [7], the techniques and protocol of which offers several distinct advantages:

- Exactly-once execution semantics;
- Protocols to reduce logging cost, especially log force, enabling efficient log management;
- Recovery independence, allowing components to recover independently;

The Phoenix/APP prototype is implemented as a runtime service in Microsoft's .NET framework and supports any component based application. The Phoenix/APP approach could also be adapted to CORBA [23] or EJB [9]. All that is required is a middleware framework that supports message interception between components so the appropriate logging can occur.

Proceedings of the Seventh International Database Engineering and Applications Symposium (IDEAS'03)
1098-8068/03 $17.00 © 2003 IEEE

Phoenix/APP is so named because it persists application state across system failures. An earlier prototype, Phoenix/ODBC [4] persisted database session state across system failures. Both efforts have the intent to permit applications to survive system failures and, as such, are part of a movement in systems research to turn focus away from its performance-oriented agenda to other important aspects of computing, such as availability and maintainability. Phoenix/APP offers a unique perspective on how to achieve high availability for enterprise applications that requires very little effort from the application developers to deploy.

To achieve fault-tolerance using Phoenix/APP simply requires a programmer to identify stateful components and declare them as persistent or transactional. Components not so declared are by default external. Application code is written without the need to consider possible system failures. Thus it can be focused entirely on business logic. Phoenix/APP transparently logs component interactions and, if a failure occurs, automatically recovers all components marked persistent up to the last logged interaction1. Components that interact with a (transactional) database are marked transactional. These are recovered up to the last successfully completed transaction – in-flight transactions will be aborted by the database. Applications must be written to deal with (e.g., retry) aborted transactions, which is required in any case since transactions can fail for many reasons. 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. Thus, the application can continue execution across system failures without loss of state (market basket, orders, etc) and need not take any special actions to ensure its persistence.

Phoenix/APP hence provides two major advantages for building enterprise applications:

- It enables an application to be written naturally as a stateful program. No special measures need to be taken to persist application state. Application developers do not have to format state so it can be saved to disk, nor do they have to provide code to read and write their state. Instead, they are free to focus on writing application business logic.
- It masks and recovers from failures, which improves overall system availability. High availability is crucial to the success of businesses engaged in e-commerce. Unfortunately, system and application failures do occur. Most application failures are recoverable by Phoenix/APP because problems giving rise to most failures are "heisenbugs"[2]. [Database recovery works well for the same reason.] If an application or process crashes, Phoenix/APP intercepts errors from the failure and initiates recovery on the failed components. The recovery is transparent from the application's point of view, except for response time delay as components are recovered.

1.2 Other Enterprise Attributes

Phoenix/APP captures component state by logging interactions between components, forming an event history. This history is on the log and is used to recover, i.e. make persistent, a component's state up to the last logged interaction. This captures the execution state of a component while it is active. Hence it is possible to interrupt execution at arbitrary points and capture the state. Capturing active application state provides other benefits to enterprise applications.

- **Scalability**: Scalability requires the reuse of resources drawn from a pool of anonymous inactive resources. The middle tier wants to multiplex resources used by stateful components, keeping these resources at work. Phoenix/APP, because it captures application state at arbitrary points, allows stateful components to be passivated and the physical resources they held, e.g. process or thread, to be reused by another application at any time.
- **Load Balancing**: With a clustered server, Phoenix/APP enables stateful components to transparently failover from one machine to another, again at arbitrary execution points. So, if stateful components are running on machine A, the state for these components can be captured in the log on a shared disk. If the system or an administrator notices that machine A is overloaded, some stateful components on A can be passivated and then failover to machine B. That is, their state is recovered and the runtime updated to redirect subsequent calls for these components to Machine B.
- **Debugging**: Phoenix/APP uses a log to capture component state by recording the sequence of events that produce the state. Hence, when a system fails, the log contains the precise set of events leading to the failure. If there is a hard failure caused by the component, the failed state can be re-created. This facilitates debugging, which has been notoriously difficult in distributed systems. Even when the failure is soft (a heisenbug), the log can offer clues as to what went wrong.

[1] For middle tier applications, the majority of response time is interactions with remote components, or resource managers. During recovery, Phoenix/APP replaces these interactions with their logged effects. Hence replay is much faster than original execution, further enhancing system availability.

[2] A Gartner Group study reports that 60% of unplanned downtime in commercial systems is due to application failure [14], and 90% of errors leading to failure in production-quality code were transient "heisenbugs".

- **Programmatic response to transaction failures**: Stateless applications are not easily able to respond to transaction aborts. By definition, they have no interesting state between transactions. So reporting and responding to transaction aborts has been cumbersome. Because Phoenix/APP enables the persistence of stateful applications, such applications can respond to transaction aborts in a straightforward way, e.g., re-executing them, executing alternative logic, or reporting the error.

2. Background

2.1 Computational Model

A middle tier application is composed of a collection of components. We require that each component be piecewise deterministic (PWD), i.e., its computation is deterministic between successive messages from other components. This enables the deterministic replay of the component from an earlier saved state when the original messages are fed to it, resending the same messages to other components as were sent in the original execution, and producing the same end state.

A component can be made PWD by eliminating non-determinism from its execution. We identify and remove three types of non-determinism:

1. A multi-threaded component may access shared data in a non-deterministic order. We assume components are single-threaded and that multiple components access common data only in a data server, where non-determinism is removed by the server logging the interleaved accesses to the data.
2. A component may depend on non-repeatable events such as asynchronous messages, system clock or external interrupts. These are recorded on the log to guarantee deterministic replay.
3. A recovered component usually exploits different system elements, e.g. threads, message ids, etc. than in its original execution. To cope with this, we "virtualize" underlying resources, introducing logical ids for messages, component instances, etc., and log the mapping to the physical elements.

Phoenix/APP logs asynchronous events and the logical to physical mapping of system elements. And we require data access in middle tier applications to be via a data server. These restrictions ensure that our components are PWD.

We further assume that failures are (i) soft, i.e., no damage to stable storage so that logged records are available after a failure, (ii) fail-stop [24] so that only correct information is logged and erroneous output does not reach users or persistent databases, and (iii) the fault is caused by a Heisenbug [13]), to avoid recreation of a failure producing state.

2.2 Components, Interactions and Guarantees

Phoenix/APP transparently provides the recovery contracts of [7] to make components persistent. We describe these contracts here. However, a multi-tier application is rarely composed solely of persistent components. So, we also describe contracts for other components in their interactions with persistent components.

An interaction contract specifies the joint behavior of two interacting components in the presence of failures of one or both components. It requires each component to make guarantees, the exact form of which depend on the nature of the contract and the roles of the components. Interaction contract guarantees pertain to a mutual state transition resulting from the interaction.

Interaction contracts between component pairs are combined into a system-wide recovery constitution that provides guarantees to external users. Our recovery constitution allows arbitrary asynchronous interactions between persistent components. However, we require that external components only interact with persistent components, and transactional components only reply to persistent components.

Inability to mask failures can occur only with a failure during an external interaction. This is unavoidable without special hardware support, e.g., if an ATM for dispensing cash with a mechanical counter that records when money is dispensed, then output messages (e.g., cash) are guaranteed to be delivered exactly once.

2.3 Interaction Types

There are different contract types, depending on the type of components involved.

2.3.1 Persistent-Persistent Interactions When persistent components interact with each other, they must ensure the persistence of both state and message. A committed interaction contract is used and is fundamental to making applications persistent and masking failures.

A committed interaction contract ensures that an interaction, once it occurs, will become part of the history of both the sender and receiver of a message. It requires the sender repeatedly send the message until it knows the receiver has received it, and that the receiver eliminate any duplicate sent messages. Further, both components guarantee that their state will persist across a system failure. Finally they must agree as to how each component deals with the message being exchanged and how the obligation to resend the message is released. The exact specification of this contract is given in [7].

The sender exposes its current state and doesn't know the implications on other components or, ultimately, external users, that could (transitively) result from subsequent receiver execution. Thus the sender must ensure that its state transition is persistent when it initiates the interaction, by forced logging if there is nondeterminism that needs to be captured. Only when the receiver later becomes a sender does it need to ensure the persistence of the effects of the received message on its state. Before this, receiver forced logging is not required. When a receiver releases the sender from its contract, the sender can discard the interaction data while still guaranteeing the persistence of its own state as of interaction time.

With a committed interaction, forced logging is needed only if there are non-deterministic events not yet on the stable log of a component. If not, then forcing is not required to make the interaction persistent via replay.

2.3.2 Persistent-External Interactions External components are, by definition, outside of the part of the system in which we provide our service. Our intent is to come as close as possible to providing immediately committed interactions with external components, including users. This leads us to external interaction contracts.

An external interaction contract is between a persistent component that subscribes to the rules for an immediately committed interaction, and an external component, which does not. The persistent component must immediately log input received from an external component. And when it sends a message to an external component, it must be prepared to resend until it receives an indication the external component has received it. Because a persistent component can crash during this interaction, it may be necessary for:

- External component to resend an input message, but such a resend cannot be guaranteed.
- Persistent component to resend an output message, since it may not be sure the message arrived. But the external component is not guaranteed to eliminate duplicate messages.

Thus, a failure during an interaction may not be masked. However, in the absence of a failure during the interaction, the result is like a committed interaction, masking failures from external components.

2.3.3 Persistent-Transactional Interactions Interactions with a transactional component, e.g. a data server, are solely request/reply interactions, and they are not guaranteed to complete. A transactional component might abort and forget the transaction. There is also a risk that a transactional data server's final reply might not be delivered even though the transaction commits. We require more. Our requirements are described in a transactional interaction contract between a persistent component and a transactional component. A transactional interaction contract requires a transactional component to guarantee an atomic state transition (either committing or aborting) and a faithful and persistent message describing a commit outcome. An abort can be reported with either an explicit message or with the transactional component indicating "no memory" of the transaction, which the persistent component interprets as an abort.

With this background for our work in place, we present our design and implementation of Phoenix/APP in the following section.

3. Phoenix/APP Implementation

Our Phoenix/APP prototype implements the recovery guarantees of [7]. It is built as a runtime service on Microsoft's .NET infrastructure. In this section we first present an overview of the architectural setting for a persistent stateful application. This is followed by a description of .NET runtime services. Then we describe how a Phoenix/APP enabled application operates, both during normal execution and failure recovery to illustrate the end-to-end persistence story.

3.1 Persistent Application Architecture

Each stateful component of an application lives in a context. Contexts define a boundary at which calls for component creation and all subsequent method calls and associated responses can be intercepted. There are four main elements to providing persistent components in Phoenix/APP:

1. Interception at the context boundary is handled by the .NET interceptor, which captures all events and method calls crossing the context. Depending on the event type, the interceptor passes the call to the appropriate Phoenix/APP module, described below, before processing.
2. Logging is handled by the Log Manager (LM);
3. Error detection and masking is handled by the Error Handler (EH);
4. Component recovery and runtime update is handled by the Recovery Manager (RM);

Figure 1 illustrates the relationship of these elements to an application running on .NET using Phoenix/APP. We illustrate the client and server in different processes, which can be on the same or different machine. The highlighted boxes in Fig 1 are code modules we supply to implement Phoenix/APP, specifically one Error Handler (EH) module, one Recovery Manager (RM) module, and one Log Manager (LM) module per process.

When the runtime intercepts an event it calls the appropriate Phoenix/APP module for method call logging or error handling, as detailed in following sections.

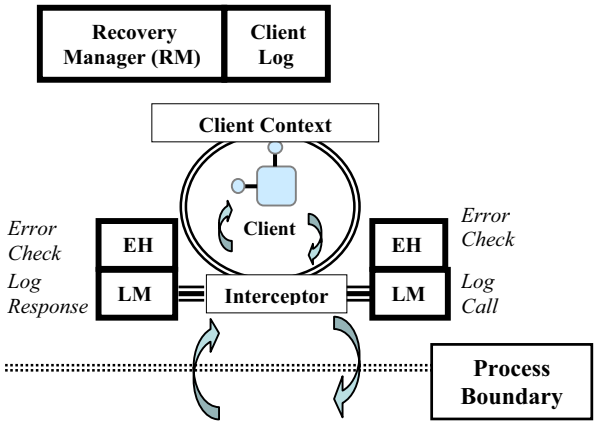

Figure 1 – Simplified overview of Phoenix/APP

3.2 Runtime Services in .NET

.NET is a runtime infrastructure for component-based software. It provides a set of runtime services to make building scalable distributed systems easier [18][3]. .NET stores the information it needs to provide runtime services in each component's context. A context is a set of properties, maintained by the .NET runtime, that describe what is required for the correct operation of a component. If a class requires a service, .NET makes sure that class instances reside in a context that provides the service. For example, if a class requires a transaction for method calls, .NET puts each class instance in a context that will provide a transaction for the component.

.NET provides services to components at runtime via interception. At component creation, .NET creates an interceptor that wraps the component's interface and contains the .NET "property" logic to provide services at runtime. When a component in one context calls a component in another context, the interceptor captures the call and processes it through a series of message sinks (see Figure 2) allowing the runtime to execute any attached property logic before forwarding the call. Runtime services like transaction management, just-in-time activation, security, object pooling, etc., are implemented as properties attached to message sinks to be called by the interceptor.

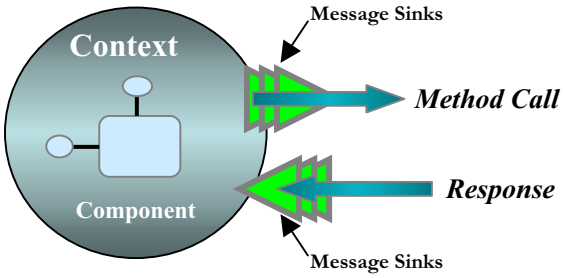

Figure 2 – Contexts and Interception

A developer specifies a runtime service by setting a declarative attribute on a class. Each service defines one or more declarative attributes that are used to control its behavior. When an object of the class is instantiated, the component creation interceptor examines the service requirements, as expressed by the class's declarative attribute values and ensures that the new component is created in a context that provides the specified runtime services. Subsequently, interceptors examine calls and provide the appropriate services for the component. The same general process applies to all services in .NET. There are no new APIs to learn, no complex code to obscure application logic, just attributes to select. Application programmers write business logic; the runtime provides the services.

3.3 Phoenix/APP as a Runtime Service

To describe our implementation of Phoenix/APP as a .NET runtime service we describe the creation of a component and follow a request from client (caller) to server (callee) and back, including error handling and recovery actions that would take place if a failure were to occur. Although we refer to clients and servers, any component can be a client, a server, or both.

Component Creation: Each stateful component lives it its own context. To make a component persistent, a developer simply declares its class as PERSISTENT, as illustrated in Figure 3. The class MarketBasket is specified to be persistent – all instances of MarketBasket will be created in contexts that provide persistence functionality.

```
[RecoverableComponent(PERSISTENT)]
 public class MarketBasket
{
   method definitions…
}
```

Figure 3 – Attribute declaring a class PERSISTENT.

[3] .NET runtime services were previously referred to as COM+ services.

The creation call (ConstructionCallMessage) for any component specified to be PERSISTENT is intercepted. The PERSISTENT property construction code called by the interceptor generates a logical identifier (LogicalID) for the instance and calls the log manager to log the creation information. The logical identifier is a string that uniquely identifies the component and encodes information useful for locating it. Component creation need not be force logged, as the component has not yet "committed" its state.

The constructor for PERSISTENT components adds a client message sink and server message sink to the context. The client message sink intercepts messages when the component acts as a client; the server message sink intercepts messages when the component acts as a server. All calls go through this chain of message sinks when they enter or exit a context and we attach our code to this chain. Our property code will extract information from each message, log it, and check for exceptions indicating component failure. The client message sink also generates a unique request id for each method call of the component and attaches the LogicalID of the component to the message so the server knows with whom it is talking.

Client sends request: During a method call the client property code will log all information pertaining to the call, including the client identity (LogicalID), server identity (LogicalID), method identity, along with the method arguments. Each method call is stamped with a new request identifier (requestID). The client identity and method identity are passed to the server side property using a call buffer, which is an out-of-band way of passing information between components.

Server receives request: Upon receiving a method call, the server property first checks if the call is a duplicate request and, if not, it records the call and passes it on for normal processing. The server property code maintains an in-memory list of the last call from every client and the response (if any) that was returned precisely to enable checking for duplicates.

Server sends response: The server property code intercepts the return and records the message in the in-memory list, force logging the response before passing the result on to the client. This makes the last call list persistent, i.e. changes to the list are logged and reconstructed when recovery takes place. When a server receives a method call after a failure, it can detect if the call is a duplicate and act appropriately.

Client receives response: The client property code intercepts the response to the original method call and logs the response before passing on the result message to the application.

Error detection and interception: Phoenix/APP detects errors resulting from component failure, masks the error from the application and initiates recovery. An error handler (EH) stub is built into the runtime and is called by the interceptor whenever an error is detected on an attempted method call or response delivery. Our EH code first determines if the error is the result of component failure, and if so it collects all information pertaining to the failed component and calls a local Recovery Manager (RM) to initiate recovery. If recovery is successful the RM returns a new reference for the recovered component to the EH. The EH updates the runtime so that subsequent calls to the failed component are redirected to the recovered instance. The EH then returns control to the runtime for normal method processing.

Automatic component recovery: At a high level, the recovery manager (RM) is given a component's LogicalID and returns a reincarnated version of that component. The RM creates a new instance of the failed component based on the log record containing ConstructionCallMessage and replays the creation call. The RM then reinstalls state into the recreated component by scanning the log and replaying method calls associated with the failed component. The RM intercepts all actions involving method invocation and response, or other non-deterministic events; the relevant information is reconstructed from the corresponding log entry and fed to the component instead of re-executing the event. Outgoing messages that the recipient is known to have successfully and stably received prior to failure are suppressed. However, if this cannot be determined (e.g. the message is the last one and a reply has not been received), the message is re-sent, and the receiver must test for duplicates. Recovery completion brings the component state up to the point of the last logged interaction (i.e. right before the failure occurred).

The RM is optimized to support concurrent recovery for multiple components. The LM will first perform a log scan for all records pertaining to the failed component and return an in-memory structure to the RM, without blocking read/write access to the log. Upon receiving the in-memory structure of log records the RM will initiate redo-recovery playback using thread safe code, enabling other concurrent threads to perform recovery for other failed components. The RM maintains a list of components (by logical ID) currently being recovered to avoid servicing multiple requests to recover a failed component.

Retransmission of messages: Because Phoenix/APP retransmits messages across crashes, and because the information needed for the recipient to eliminate duplicate messages is persistent, Phoenix/APP provides persistent exactly-once message delivery.

Unmasking Errors: An error resulting from a failure is unmasked when the error handler determines that a failed component cannot be successfully recovered. This should be rare, but in the event a component repeatedly fails without making forward progress (i.e., a Bohr bug) or the Recovery Manager is unable to recreate

the component (i.e., system on which the component resides is dead), the error handler unmasks the error by propagating a message to the application.

Recovery Service: All applications running on .NET are "hosted", which means to execute an application the systems administrator must submit a start string containing the .DLL or .EXE to execute, along with a port number for communications and associated URI (universal resource identifier). The .NET execution manager will load the .DLL or .EXE, then listen for requests (method calls) for the given URI over the specified port number.

In addition to a recovery manager, our implementation of Phoenix/APP includes a recovery service that monitors hosted applications which contain recoverable components. When a hosted application creates a persistent component, Phoenix/APP will send a message to the recovery service to register the application for monitoring and record its restart string, consisting of the .EXE or .DLL, port number and URI. The recovery service will monitor Windows events and if it detects the process running the application has terminated abnormally it will automatically restart the application using the restart string and issue a request to the recovery manager to initiate recovery on all stateful components. As a Windows runtime service, the recovery service is automatically started when the system reboots so if the system itself failed the recovery service will be restarted at reboot and initiate recovery all active applications.

4. Phoenix/APP Performance

We had two objectives in our performance evaluation of Phoenix/APP:

(i) to measure the overhead to persist stateful component interactions and its impact on application response time;
(ii) to measure how fast Phoenix/APP can recover the persistent components of a stateful application after a failure and reestablish normal application operation.

To conduct this evaluation we implemented a test program and designed controlled experiments.

4.1 Micro-benchmarks

We performed our measurements using a simple distributed application that sends book order requests of varying sizes to a middle tier server and receives responses back from the server. This application is illustrated below in Figure 4. The middle tier server maintains a shopping basket of orders for the client and communicates with two backend servers which maintain inventory and order requests. We used two versions of this distributed application.

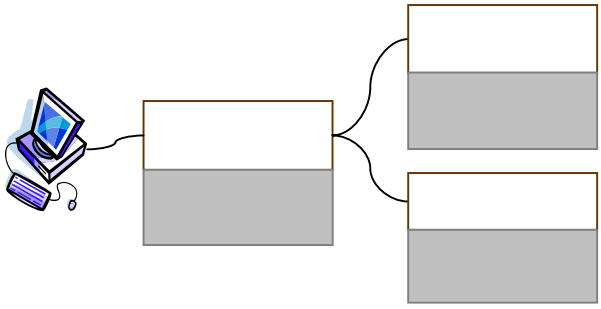

Figure 4 – Illustration of the book buyer application used in the performance evaluation of Phoenix/APP.

Our baseline system is a non-recoverable implementation of this application with volatile state. If a failure occurs in any process during a session then messages, data and state are lost as a result of the crash. In particular, the shopping basket maintained at the middle tier is lost, as are orders placed at backend bookstore servers. We compared this baseline system with a fully persistent and recoverable implementation, built using the Phoenix/APP runtime service by simply declaring selected classes as "persistent".

Both versions of the application are implemented using Microsoft's .NET platform and components are linked to .NET runtime services (i.e. inherit from the class "managed component"). Hence, by comparing round-trip request/response times between client and middle tier, we get an accurate picture of the latency caused by making an application persistent using Phoenix/APP.

4.2 Environment

We ran our benchmark on two Intel-based PC's, each a 795 MHz Pentium III CPU with 256 Megabytes of memory. The systems are connected via a switched 100 Base-T network, using an Intel Express 10/100 Fast Ethernet Switch. The client and middle tier (price shopper) processes run on separate machines.

Because the client and server are located on the same local area network (LAN), the relative overheads we present greatly overstate what a typical client would experience in a real deployment of Phoenix/APP. In a more conventional setting, where client and server connect over a wide area network, the latency for a request/response would be orders of magnitude higher, resulting in lower relative latency impact for Phoenix/APP. However, this experimental set-up still gives us an indication of logging overheads at the server and the time required to recover a stateful session, which is the focus for this discussion.

4.3 Results

Table 1 shows the results of our benchmark runs measuring the request/response cycle for each of our two test systems, averaged over 1000 trials. Each request message is roughly 240 bytes after marshalling for transfer. For reference, we also show the round-trip time to send a remote procedure call of the same size from the client to the middle tier server and back.

Table 1: Elapsed time (msecs) to process request (book order) and response between client and server.

A. Elapsed Time for Native Implementation	**3.34 msecs**
B. Elapsed Time for Phoenix/APP	**7.39 msecs**
C. Elapsed Time for RPC	**2.42 msecs**
D. Difference (B-A)	**4.05 msecs**

Table 1 shows that the Phoenix/APP based implementation of the application requires approximately four additional milliseconds to complete the request/response cycle over the non-recoverable implementation. This overhead is due largely to the time required to force write records to the log. The latency introduced by the logging is approximately half the rotation time of our 7200 rpm disk. While that latency is significant relative to LAN latencies, in a wide area network implementation where network latencies are closer to 200 milliseconds than the 2.42 milliseconds measured on our LAN, this latency is a small fraction (about 2%) of the request/response round trip time.

Next, we measured the time required to recover the persistent components of the middle tier application that manage the shopping basket. To conduct this experiment we submitted book order requests from the client to the middle tier to the server. We "crashed" the "Price Shopper" application by issuing the command "end task" from the Windows Task Manager, terminating the process running the middle tier. We then issued a request from the client to display the contents of the shopping basket. At this point the client application is left waiting for the middle tier server to respond to this request. The Phoenix/APP runtime service on the client intercepted the error message resulting from the failed RPC and initiated recovery of the middle tier application running on the server. We measured the time required for Phoenix/APP to recover the stateful components in the session and respond to the outstanding client request.

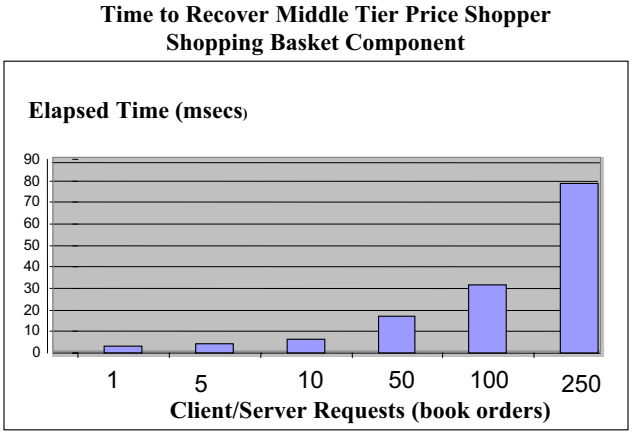

Figure 5: Elapsed time to recover a component.

Figure 5 presents results of the experiment to recover the middle tier application for a varying number of logged request/response messages. When the recovery request is received by the server, the Phoenix/APP runtime recreates the stateful components and replays all logged method calls against these components to reinstall application state. When only one order was received prior to failure, it required 3.16 milliseconds to recover the application and respond to the outstanding client request. An examination of this base case revealed that 2.41 milliseconds were spent sending the recovery request from the client machine to the server and the remaining 0.7 milliseconds were spent recreating components representing the market basket and replaying the logged method call. Since each method is replayed as an intraprocess function call, avoiding the cost of a RPC and cross process call, component state recovery is quite fast. As expected, as the number of logged method calls increases the time required to recover the application and respond to the outstanding client request increases. However, the average cost per logged interaction declines, approaching .3 milliseconds. This reflects high sequential read performance from our log disk and very small computational load for recovery.

6. Related Work

Recoverability for communicating processes has been studied in the fault-tolerance community (e.g., [1,10,11]), but the main focus has been long-running computations (e.g., scientific applications). The goal has been to avoid a failure losing too much work and failures have usually not been masked from users. Masking failures has required "pessimistic logging" (see, e.g., [16]), with log forcing for both sender and receiver upon every message exchange. Earlier, process checkpointing (i.e., copying process state to disk) upon every interaction, was used in

the pioneering fault-tolerant systems of the early eighties [2, 8, 17].

Techniques focusing solely on process-based failure detection and recovery, however, are not necessarily applicable to component-based applications for the following reasons:

- ***Overly coarse granularity:*** A process may contain several active components. If individual components fail, e.g., due to a crash or abnormal termination, a process may not terminate. More importantly, it may not be possible to resurrect the components independently. Thus, a strategy that detects only process failures cannot deal with redeployment of the finer-grained components.

- ***Inability to restore component relationships:*** Components communicate with other components and identify them via component identifiers. When a failed component is recovered, identifiers for the failed component held by other components must be remapped to the recovered component. Restoring these "relationships" is unique to distributed component middleware and is not handled well by process-based recovery strategies.

- ***Restriction on process checkpointing and recovery:*** Components often maintain state that must be checkpointed periodically to survive crash failures. Process checkpointing may incur excessive overhead or be unable to checkpoint the state for individual components. Thus, a finer granularity for persisting state is needed to permit components in a process to checkpoint their state independently of each other, and at arbitrary times.

Hence, it is necessary to have strategies specifically tuned to component-based distributed applications. The most successful prior approach uses queued transactions and in OLTP [15, 3], supported by most TP monitors, e.g., MQ Series, Tuxedo, MTS. This is the stateless application paradigm that requires transactions to enqueue a client request on a queue; to dequeue it, process and enqueue the reply; and to dequeue the reply at the client. This incurs forced-logging for three commits. It requires significant effort to implement applications in this stateless paradigm.

Some work on failure masking for stateful applications exists, but is limited in the architectures that it can support [20].

Fault tolerance is being discussed for component middleware like CORBA [23] and EJB, but that focus is also on service availability for stateless applications (i.e., restarting re-initialized application server processes). Products (e.g., BEA WebLogic, or Sun's J2EE suite) support failover techniques that do not relieve the application programmer from having to either code 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 [21], based on application-transparent replication and redirection of http requests.

The need for execution guarantees for e-services has been raised by a number of researchers (e.g., [25, 22]), but they have been concerned with specific applications such as payment protocols or mobile data exchange and do not specifically address system-wide failure masking in general multi-tier architectures. Closest to our approach in terms of objectives is the work in [12] 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 and logging optimization.

Prior work on user-transparent database application recovery was restricted to applications embedded in the data server such as stored procedures [19] and two-tier client-server systems [20, 4, 5]. A key difference between a client and an application server is that clients are trivially piecewise deterministic, while application servers typically are multi-threaded and asynchronously receive messages. It is not obvious how to extend client-server protocols. Our interaction contract is the key for the generalization to multi-tier systems.

The recovery guarantees framework [7] upon which Phoenix/APP is based improves the state of the art in a number of ways. Compared to traditional techniques based on pessimistic logging or frequent process state saving, these protocols reduce logging and state saving costs while providing very fast recovery. Our approach can handle stateful applications, removing the burden to the application programmer of making his application stateless. It can be extended to deal with clients interacting with browsers [6], enabling browser state persistence.

7. Summary

Writing distributed component-based applications that reliably manage persistent state requires application programmers to address a host of difficult issues, such as carefully logging application state, masking errors resulting from the failure, ensuring consistent recovery for various component types, ensuring exactly once-message delivery, etc.

Phoenix/APP automatically recovers component-based applications, without requiring any special modifications to the application. This increases application availability by avoiding the extended down-time that failures can produce when manual intervention

is needed to correct failed application state and restart the application.

Because Phoenix/APP transparently handles stateful applications, it removes the burden on the application programmer of making his application stateless. This substantially improves application programmer productivity and reduces the complexity of the resulting program, a side effect of which should be to reduce the number of programming errors.

In this paper we presented the design, implementation and evaluation of Phoenix/APP, our runtime service based on the notion of recovery guarantees. Our prototype is built on the .NET middleware framework that supports extensible interception, which we use to insert our logging, error handling and recovery extensions. We confirmed the feasibility of Phoenix/APP by presenting performance results on the cost to persist component state and recover from failures.

While our implementation is built on the .NET platform, the techniques described are more widely relevant. For example, the approach has been extended to deal with clients interacting with internet browsers [6], enabling browser state persistence. Moreover, functionality similar to .NET is available in both J2EE and CORBA [23] runtime environments. Thus, our general approach should be widely relevant.

8. References

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

[2] J.F. Bartlett: A NonStop Kernel. SOSP 1981.

[3] P. Bernstein, M. Hsu, B. Mann: Implementing Recoverable Requests Using Queues. SIGMOD 1990.

[4] R. Barga, D. Lomet, S. Agrawal: Persistent Client-Server Database Sessions. EDBT 2000.

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

[6] R. Barga, D. Lomet, G. Shegalov G. Weikum: Recovery Guarantees for Internet Applications. Submitted for publication (2003).

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

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

[9] Enterprise JavaBeans Technology, http://java.sun.com/products/ejb/.

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

[11] J. C. Freytag, F. Cristian, B. Kähler: Masking System Crashes in Database Application Programs, in the proceedings of VLDB 1987.

[12] S. Frolund and R. Guerraoui: Implementing E-transactions with Asynchronous Replication, in the proceedings of Int'l Conf. on Dependable Systems and Networks, 2000.

[13] J. Gray, Why Do Computers Fail and What Can We Do About It. 5th Symposium on Reliability in Distributed Software and Database Systems, 1986.

[14] Gartner Group Publication: The Zero Latency Enterprise. Roy Schulte, VP of Applications Systems and Middleware, 1999.

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

[16] Y. Huang, Y-M. Wang: Why Optimistic Message Logging Has Not Been Used In Telecommunications Systems. FTCS 1995.

[17] W. Kim: Highly Available Systems for Database Applications. ACM Computing Surveys, 16(1), 1984.

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

[19] D. Lomet: Persistent Applications Using Generalized Redo Recovery. ICDE 1998.

[20] D. Lomet, G. Weikum: Efficient Transparent Application Recovery in Client-Server Information Systems. SIGMOD 1998.

[21] M.-Y. Luo, C.-S. Yang: Constructing Zero-Loss Web Services. IEEE International Conference on Computer Communications (INFOCOM) 2001.

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

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

[24] F. Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. ACM TOCS, 2(2), May 1984.

[25] J. D. Tygar: Atomicity versus Anonymity – Distributed Transactions for Electronic Commerce, in the proceedings of VLDB 1998.