Improving Logging and Recovery Performance in Phoenix/App

Roger Barga
Microsoft Research
Redmond, WA 98052, USA
barga@microsoft.com

Shimin Chen
Carnegie Mellon University
Pittsburgh, PA 15213, USA
chensm@cs.cmu.edu

David Lomet
Microsoft Research
Redmond, WA 98052, USA
lomet@microsoft.com

Abstract

Phoenix/App supports software components whose states are made persistent across a system crash via redo recovery, replaying logged interactions. Our initial prototype force logged all request/reply events resulting from inter-component method calls and returns. This paper describes an enhanced prototype that implements: (i) log optimizations to improve normal execution performance; and (ii) checkpointing to improve recovery performance. Logging is reduced in two ways: (1) we only log information required to remove non-determinism, and we only force the log when an event "commits" the state of the component to other parts of the system; (2) we introduce new component types that provide our enhanced system with more information, enabling further reduction in logging. To improve recovery performance, we save the values of the fields of a component to the log in an application "checkpoint". We describe the system elements that we exploit for these optimizations, and characterize the performance gains that result.

1. Introduction

1.1 Persistent Stateful Components

Component-based programming is widely used in enterprise applications, such as web services and middle-ware systems, where high availability is usually a requirement [8,9,11,14]. In contrast to fault tolerant operating systems [2,3], a common practice here to achieve high availability without loss of critical work is to enforce a form of "workflow" programming model in which stateless components communicate with each other through recoverable stateful message queues [4,10]. At every invocation, a component must read state information from a queue before processing and write it back after processing, which is an unnatural model. And distributed commits for the distributed message queues are potentially expensive. This stateless programming model is the same one supported by traditional TP monitors [4].

To support natural, stateful components and avoid distributed commits, we built Phoenix/App based on the recovery guarantees framework in [6], which generalized the client-server protocols of [12]. Programmers simply specify a component to be persistent. The Phoenix/App runtime transparently intercepts and logs incoming and outgoing messages of the component in a local log with pessimistic logging [1,7]. If the component fails, the runtime automatically recovers its state via redo recovery, replaying the messages from the local log. Thus, Phoenix/App guarantees exactly-once execution for persistent components.

Our first prototype [5], referred to as the baseline system, demonstrated the feasibility of the solution using the simplest approach, immediately forcing the log for every message between persistent components. Moreover, recovering a component state required replaying all the messages from the creation of the component, leading to potentially high recovery cost for long-lived components.

1.2 This Paper's Contributions

In this paper, we study log optimizations to improve normal execution performance and checkpointing to improve recovery performance in Phoenix/App.

We consider two ways to reduce logging cost.

1. For persistent components in general, we only log information required to remove non-determinism and only force the log when an event "commits" the state of the component to other parts of the distributed system. This is usually at an outgoing message.
2. We introduce three special types of components (subordinate, read-only, and functional components) that provide enhanced knowledge about the states of the components and their interactions with other components in the system. This permits us to further reduce logging, sometimes dramatically.

To improve recovery performance, we save the values of the fields of a component to the log in an application "checkpoint". We study how to transparently save component states, how to coordinate the saving operations for component states and the in-memory data structures of Phoenix/App runtime, and how to recover from a crash.

The remainder of the paper is organized as follows. We describe the baseline system in section 2. Then we discuss log optimization in Section 3 and checkpointing in Section 4. We evaluate our approaches with experiments in Section 5. Section 6 provides conclusions.

Proceedings of the 20th International Conference on Data Engineering (ICDE'04)
1063-6382/04 $ 20.00 © 2004 IEEE

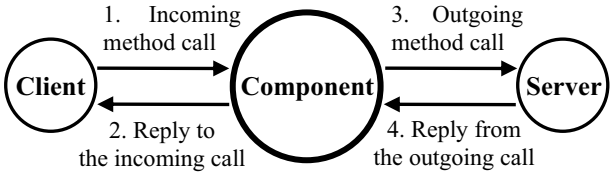

Figure 1 Messages of a component

2. Achieving Persistence in Baseline System

We first explain the component model and the sufficient conditions for persistence. Then we describe how these conditions are satisfied in the baseline system.

2.1 Component Model

A component is typically a (C++, Java, or C#, etc.) object instance, as in CORBA [14], Enterprise Java Beans [8], and .NET [9]. Component states are held in object fields; operations are provided through methods.

Components communicate via (remote) method calls. As shown in Figure 1, a component sees four kinds of messages: an incoming method call from a client component; the subsequent reply message to the client; an outgoing method call to a server component; and the response from the server.[1]

2.2 Exactly-once Semantics- Sufficient Conditions

Phoenix/App is based on .NET remoting [9]. Programmers specify a component as persistent using a customized attribute. Phoenix/App detects the attribute and transparently logs interactions. Unspecified components are external components by default, for which we take no actions and make no guarantees. For persistent components only making method calls to persistent components, we guarantee "exactly-once" semantics in case of failures, i.e. state changes are exactly the same as if there were no failures. "Exactly-once" semantics is achieved if the following are satisfied [6]:

1. When sending a message (message 2 or 3 in Figure 1), a persistent component ensures that its state as of the send and the message are persistent.
2. When making a method call, a persistent component attaches a globally unique ID to the outgoing method call message which is deterministically derived.
3. When receiving a method call from another persistent component, a persistent component checks the globally unique ID. It executes normally if it has not seen the ID before. Otherwise, it returns the reply of the previous method call with the same ID.
4. A persistent component repeats an outgoing method call to a server component until it gets some response.
5. When recovering from a failure, if a persistent component was responding to an incoming method call before the failure, it does not send the reply (message 2). Rather, this message is sent upon request from the client as in *condition 3*.

[1] Components must be piece-wise deterministic (PWD) to be replayable in [6]. To ensure PWD, Phoenix/App requires components to be single-threaded, serving one incoming method call at a time. This avoids the non-determinism caused by interleaved thread accesses to local data in a component. But of course, there can be multiple threads executing in multiple different components in a process.

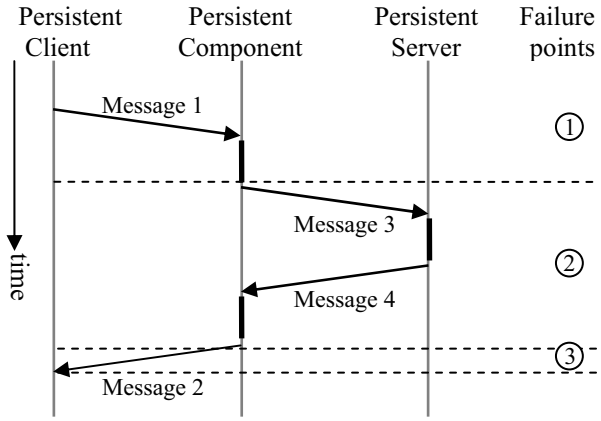

Figure 2 Failure situations

We assume all the above conditions are satisfied and examine how the system ensures "exactly-once" semantics for persistent components when external components are not servers. Without loss of generality, we consider the failures as shown in Figure 2. Suppose a persistent component receives an incoming method call from a persistent client component. In serving the method call, it sends an outgoing call to a persistent server component and receives the response for this call. After that, it sends the reply for the incoming call to the client. As shown in Figure 2, the persistent component may fail at three failure points. (In all cases, the boundaries of the failure situations are defined by the interactions that the recovering component finds on the log.) We discuss the recovery processing for each failure point:

- *Failure before message 3 is sent:* Component state before message 1 can be recovered because either it is the state after creation or the component has finished a previous incoming call. In the latter case, it has sent a reply message and can be recovered according to *condition 1*. If the component has remembered message 1, it performs the method call. By *condition 4*, the client resends message 1 in case the component has not remembered the message. Duplicates are eliminated by *condition 3*.
- *Failure after message 3 is sent but before message 2 is sent:* By *condition 1*, the component recovers message 3 and its state at the send of message 3. By *condition 4,* it resends message 3 if it has not yet remembered message 4. The ID is the same by *condition 2*. The

server eliminates duplicates by *condition 3*, returning the same message 4, even if message 3 had previously been received. Component execution then continues.

- *Failure after message 2 is sent:* By *condition 1*, the component can recover message 2 and the state at the send of message 2. By *condition 5*, the component does not resend message 2 to the client. If the client has not received message 2, it retries the method call by *condition 4*. The component detects the duplicate message by checking its globally unique ID and returns message 2 to the client.

The task of the baseline system is to enforce the five conditions to achieve "exactly-once" semantics. Section 2.3 discusses *condition 2* and *3*. Section 2.5 describes how the other three conditions are satisfied.

2.3 Runtime Logging

In .NET remoting, a component resides in a structure called a "context". Within a context, method calls are local calls. Across context boundaries method calls are remote procedure calls: method names and parameters are packaged (marshalled) into messages and sent to the remote contexts, where the messages are unmarshalled, etc. After processing, replies are sent back through messages in the similar fashion. Message interceptors at context boundaries can intercept all the four kinds of messages described in Section 2.1.

In our baseline system, every persistent component resides in its own context. So we can intercept all messages of persistent components as shown in Figure 3. The baseline system logs and immediately forces every message to the log, implementing Algorithm 1.

A process may host multiple contexts and a machine can run many processes. So the globally unique ID of a method call consists of: caller's machine name; unique process "logical" ID on that machine, assigned by Phoenix/App; unique caller component "logical" ID in the process, also assigned by Phoenix/App runtime; and local method call ID which is incremented for every outgoing method call of a component. Phoenix/App ensures that the logical process IDs and the component IDs remain the same in spite of failures. The last local method call ID before a failure is obtained from the log.

On an outgoing method call, a message interceptor attaches the globally unique ID to the message, satisfying *condition **2***. On an incoming method call message, a message interceptor checks for the ID. If the ID does not exist, the caller must be an external component.

To enforce *condition **3*** for method calls from persistent clients, method call IDs and their replies are stored in a *last call table* indexed by the first three parts of the ID. On an incoming method call message from a persistent client, a message interceptor checks the last call table and compares the new ID to the last call ID from the same component. If equal, the stored reply is sent back to the client. Otherwise, the call is delivered normally to the persistent component. Before sending a reply message, the message interceptor updates the last call table.

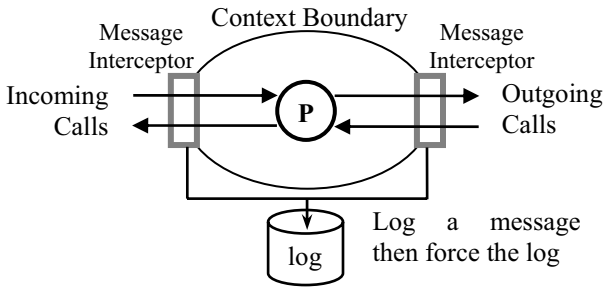

Figure 3 Intercepting and logging messages

```
Message 1: Incoming method call
       Log message 1
       Force Log
Message 2: Reply to incoming call
       Log message 2
       Force Log
Message 3: Outgoing method call
       Log message 3
       Force Log
Message 4: Reply from outgoing call
       Log message 4
       Force Log
```

Algorithm 1 Baseline system logging

We only keep information on the *last* method calls from persistent clients. If we receive a method call from a persistent client, the client can independently recover its state to the send message time by *condition 1*. So its earlier last call entry is no longer needed.

2.4 Failure Detection

Failures are detected in two ways. (1) All processes that host persistent components register at start time with the Phoenix/App recovery service running on their machine, as shown in Figure 4. The recovery service monitors the abnormal exits of the registered processes and restarts those processes. It keeps the information of registered processes in a table and force writes updates to the table to its log to make the table persistent. (2) In .NET, message interceptors can detect exceptions raised by outgoing calls. Phoenix/App handles particular exceptions that indicate a component failure. (Not all exceptions indicate failures, e.g., an invalid argument exception indicates an error, but the remote component is still alive.)

2.5 Automatic Recovery

To recover a failed component, the baseline system replays all logged incoming method calls from component creation until failure. Since all messages are forced to the

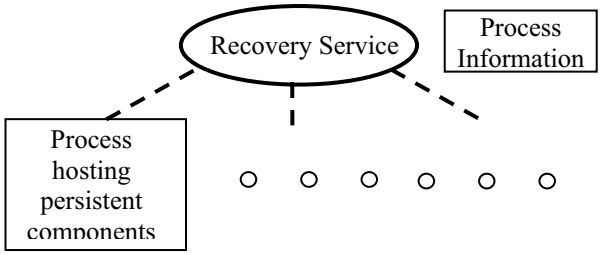

Figure 4 Recovery service monitors registered processes that host persistent components.

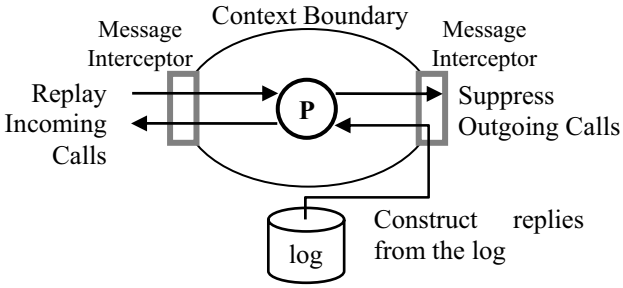

Figure 5 Replaying method calls during recovery.

log, we can recover the component state to the time of the last send message and execution can continue from there.

As shown in Figure 5, to replay an incoming method call, Phoenix/App runtime calls the logged method with the logged parameters. An outgoing call is suppressed by the message interceptor if a reply to the call is found in the log. Instead, the reply is read from the log and returned. Otherwise, either the previous send message or the last logged outgoing call must be the last send message and component state has already been recovered to the time of the last send message. We begin normal processing and send the outgoing call as in normal execution.

During recovery, the message interceptor rebuilds the last call table from incoming call log records. Hence both component state and last call table are recovered to the time of the last send message, which satisfies *condition 1*.

Condition ***5*** is satisfied because the replies of the replays are not sent to any components. Moreover, when a message interceptor detects a recognized exception for an outgoing method call message, it waits for a while and retries the call using the same method call ID. This satisfies *condition 4*.

In summary, the baseline system provides sufficient conditions for "exactly-once" semantics. It logs and forces every message. To recover a failed persistent component, it replays all the method calls from the creation of a component until the failure.

3. Improving Logging Performance

Every method call in the baseline system incurs at least two log writes and forces at a persistent component, even if no outgoing calls are made. Here we focus on reducing log writes and forces to improve performance.

3.1 Log Optimizations for Persistent Components

We discuss two situations based on whether external components are involved or not.

3.1.1. Interacting components are both persistent. The purpose of logging messages is to satisfy *condition 1* (recover a persistent component to the point of the last send message). A send message is important because it may change the states of other components, hence "*committing"* the component state to other parts of the distributed system. Such "commits" usually need a log force. However, forcing receive messages, as the baseline system does, is unnecessary. We can log receive messages without forcing. The send message log force ensures all previous messages (including receive messages) are stable in the log. Losing messages received after the last send message does not affect correctness since component state has not been committed after the last send.

Moreover, since the last send message can be re-created in the replay of all the previous messages, it is not necessary to write a send message to the log. But we still need to force all the previous log records before sending a message. This optimization is important as it allows more opportunities to combine log forces from multiple components that share the same log.

The improved logging algorithm, shown in Algorithm 2, saves a log force for every receive message and saves a log write for every send message.

3.1.2. Client component is an external component. No logging strategy fully masks failures when external components are involved. However, we may be able to mask failures when persistent and external components do not fail simultaneously. For this purpose, we force log messages promptly, as in the baseline system. Below we analyze when persistent component failures are masked:

- If a persistent component fails after it receives and immediately force logs a call (message 1) from an external component, it recovers and finishes the incoming call, masking the failure from the external component. But a call from an external component may have to be repeated since the persistent component may fail before logging.
- If a persistent component fails before force logging a reply (message 2) to an external component, it recovers, realizes that message 2 has not been sent, and proceeds to force log and send it. But we cannot know whether the external component has received the output message or not, even if it is logged. What we can mask is persistent component failures before the log force. Failures after log forcing, including message delivery failures, may not be masked.

```
Message 1: Incoming method call
       Log message 1
Message 2: Reply to incoming call
       Force all the previous messages
Message 3: Outgoing method call
       Force all the previous messages
Message 4: Reply from outgoing call
       Log message 4
```

Algorithm 2 logging for persistent components

```
Message 1: Incoming method call
       Log message 1 (long record)
       Force all messages
Message 2: Reply to incoming call
       Log message 2 (short record)
       Force all messages
```

Algorithm 3 persistent component logging for external client

```
At a functional component:  do nothing

At a persistent component:
Message 3. Outgoing method call
       IF (server is functional)
         Do nothing
Message 4. Reply from outgoing call
       IF (server is functional)
         Do nothing
```

Algorithm 4 logging for functional components

Hence there is a "window of vulnerability", i.e. if a persistent component fails *during an interaction*, before the log is forced for message 1, or after the force but before the send for message 2, the failure may not be masked. Persistent component failures at other times are known to be masked, i.e., after logging message 1, we know that persistent component failures are masked until message 2 is logged. Masking begins again with the next message 1 received from the same external component.

We can nonetheless improve on the baseline algorithm a bit. Since message 2 can be regenerated via replay of the component, we do not need to save the whole send message contents. We only need to save the fact that the message was sent (attempted to be sent). We call a full message with all its content a long record, and a message with only identity information a short record. Algorithm 3 shows this logging discipline.

3.2 Additional Kinds of Components

We introduce three additional types of specialized components: *subordinate*, *functional*, and *read-only* components exploit them to further reduce logging.

3.2.1. Subordinate Components. A subordinate component is a persistent component associated with a persistent parent component and restricted to only service method calls from its parent and other subordinates of its parent. Only the parent accepts calls from outside callers.

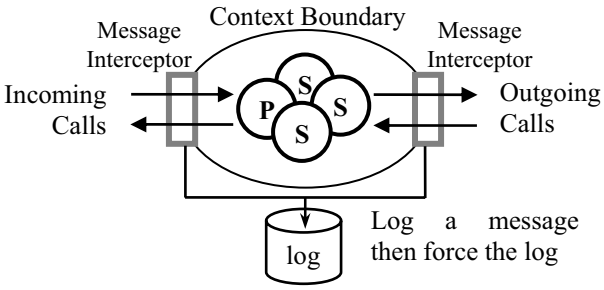

Figure 6 Subordinate components in the same context as their parent component

Hence execution is single-threaded within a parent and its subordinate components. Only incoming calls are restricted; subordinate components can call any component.

Since a persistent component services method calls from many persistent and/or external components, we must log the calls to capture their non-deterministic arrival order. For subordinate components, this non-determinism is not present. We do not need to log the messages among a parent and its subordinate components because these messages are fully determined given the incoming method call messages to the parent, and reply messages for outgoing method calls.

We could intercept parent-subordinate and subordinate-subordinate calls, check component types and use different logging algorithms. But to minimize overhead, we instead put a subordinate into the same context as its parent component, as shown in Figure 6. In this way, method calls among a parent and its subordinates do not cross context boundaries and messages are neither seen by message interceptors nor logged. This avoids interception overhead as well as logging overhead.

3.2.2. Functional Components. Functional components are stateless components that either make no calls or call only functional components. They do not call components of other kinds. Methods of functional components are purely functional: given the same parameters, a call always returns the same value.

We needn't log messages nor maintain last call tables at functional components. Moreover, at a persistent component, if the server of an outgoing call can be determined to be a functional component, we do not need to force the log or log the return message. This algorithm is shown in Algorithm 4.

3.2.3. Read-only Components. Read-only components are also stateless, but unlike functional components, they can also call other persistent components. These calls read the states of persistent server components and persistent component state can change between method calls. So generally the reply from calling a read-only component is

```
At a read-only component: do nothing

At a persistent component:
Message 1. Incoming method call
       IF (client/method is read-only)
          Do nothing
Message 2. Reply to incoming call
       IF (client/method is read-only)
          Do nothing
Message 3. Outgoing method call
       IF (server/method is read-only)
          Do nothing
Message 4. Reply from outgoing call
       IF (server/method is read-only)
          Log message 4
```

Algorithm 5 logging for read-only components and read-only methods

unrepeatable. Motivating examples are statistics collectors and meta-search engines.

Since read-only components are stateless, we do not need to recover their states and hence we do not log any messages at read-only components. Further, outgoing method calls from read-only components do not change server component states. Thus at a persistent component, we do not log calls from read-only components. However, persistent component callers of a read-only component must log (but not force) the reply message, as it is not guaranteed to be recreated via replay.

A method call to a read-only component does not change any state. So a call message from a persistent component does not "commit" its state to other components. Thus, we do not force the log when calling a read-only component.

Finally, it is not necessary to detect duplicate calls to or from a read-only component because the calls do not change any states. Hence read-only components do not have last call table entries at persistent components and we do not maintain outgoing call IDs and last call tables at read-only components.

Algorithm 5 describes the logging associated with read-only components.

3.3 Read-only Methods

A stateful persistent component may provide methods to query its state (without changing it). We call such methods read-only methods. More strictly, a read-only method is a method that neither changes any field of the component nor makes a non-read-only outgoing method call. Programmers can specify a read-only attribute on a method, and message interceptors can check for it.

We treat read-only method calls like method calls to read-only components. For a read-only method call, the client does not need to force the log, and the server does not need to log. Algorithm 5 also describes this logging.

3.4 Detecting Component Types

Programmers specify subordinate, functional, or read-only components just as they specify persistent components, via a declarative attribute accessible within a context. Hence, a message interceptor can obtain information about the components inside its context.

To detect the component types of remote components, a message interceptor attaches information about the (parent) component of its context to the messages being sent. When receiving messages, a message interceptor obtains the component information from the messages. In this way, client component types can be determined.

To determine server component types, we keep a remote component type table. Initially, the types of server components (targets of outgoing calls) are unknown, and the most conservative logging algorithms are used. From reply messages, we gradually learn server component types and store them in the remote component type table, which is checked when sending an outgoing method call.

3.5 Multi-call Optimization

Other optimization opportunities exist if we are permitted to take into account the effects of multiple calls, rather than examining each call in isolation. The optimization described here is not currently supported by our Phoenix/App system, but including it is straightforward.

Consider a persistent component that calls multiple server components. Without context information, each of these calls commits component state, and hence must force the log. The first force at the first outgoing server call captures the nondeterminism associated with the incoming method call. After this first call, our component does the force to capture the nondeterminism resulting from having read the earlier server replies.

Each server is responsible for the persistence of its reply message. Thus, the nondeterminism for our persistent component that results from reading the replies from the outgoing calls is already captured at the respective servers in their last call tables. Recovery can exploit those persistent replies to recover our component.

We can then choose to force the log only when a component itself replies to its caller, or when it invokes the same server a second time during its method execution. This requires that we remember not only the last call for each component, but each server that a component has called so far in the execution of a method. But saving a log force is a large reward for this extra bookkeeping.

4. Improving Recovery Performance

The baseline system recovers a failed component by replaying all its method calls from component creation until the failure. However, a long-lived component may

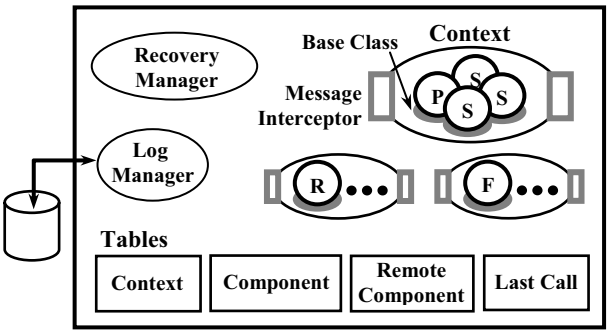

Figure 7 Intra-process architecture

handle many method calls before failure, leading to very high recovery cost. In this section, we describe how to take checkpoints to keep recovery cost low.

4.1 Intra-process Architecture

Both context states and process data structures must be saved in order to recover from a process crash. Figure 7 shows the Phoenix/App runtime structures inside a process. There can be multiple contexts hosting components. In a context, there can be a parent persistent component and zero or more subordinates, or a functional or read-only component.

Phoenix/App maintains a set of data structures outside of all contexts in a process. The c*ontext table* contains an entry for every context hosting Phoenix/App components. The c*omponent table* has an entry for every Phoenix/App component in the process. A context entry and entries for components of the context are associated through pointers. The r*emote component table* contains component type information for remote components seen so far. As described in Section 3.4, this table saves remote server component types so that optimal logging algorithms will be chosen. The *last call table* maintains information on the last incoming method calls indexed by client component ID, which is used to detect and answer duplicate method calls. The last call table is shared among all the contexts in a process so that the entry for a client is updated even if the client calls two different components in the same process. Moreover, the last call table also keeps the list of last call entries associated with every context, which is used in context saving. Descriptions of the table entries are shown in Table 1.

Message records and checkpoints are stored in disk based log files. We manage disk files on a per-process basis to simplify file access. Logging is performed through a log manager in a process.

There is a *recovery manager* in every process. At process start, the recovery manager registers the process with the *recovery service* of the machine to obtain the virtual process ID, which is part of a method call ID. If the recovery service notifies the recovery manager that the process exited abnormally, the recovery manager first recovers the process tables, contexts and components.

Table 1 Global tables in a process

Component table entry	component ID, component type (persistent, read-only, etc.), object type, pointer to the object instance, and pointer to its context table entry
Context table entry	a list of pointers to the component table entries for the components in the context, the (parent) component ID and URI, a log sequence number (LSN) of the latest context state record, and the last outgoing method call ID of the context
Remote component table entry	Remote component type information indexed by remote component ID
Last call table entry	method call globally unique ID, a pointer to the reply message and/or an LSN for the reply message log record

In the next two subsections, we study how to save context states and process data structures, respectively.

4.2 Saving Context States

States of components in a context (parent and subordinates) must be saved together because method calls inside a context are not intercepted or logged. Component state normally includes stack, and virtual memory of the process, which could be very expensive to save. To avoid this overhead, context states are saved only when the context is not "active", i.e., after an incoming method call to a parent finishes and before another incoming call is delivered to it. At this moment, component state consists only of field values. So it is sufficient to save only component fields and context related data structures to recover the context.

To save or restore the internal fields of a component, we use the .NET reflection mechanism to obtain its field types and values. (Serialization and deserialization utilities exist in .NET to save and restore fields of non-context objects. But we had to implement the support for objects in contexts.) Because component fields may be private and invisible from outside of the component, we implemented a "persistent" base class and required all Phoenix/App components to inherit from this class. A base class can visit all fields in a derived instance and we implement the support for saving and restoring a component in the base class.

We specially handle pointer fields referencing Phoenix/App components. For a remote component reference, we save the component URI; for a local component reference (to a component in the same context), we store the component ID. When restoring a pointer field, we re-obtain the pointer using the saved URI or component ID.

Moreover, we save relevant information from the global component and context tables so that the component table entries and the context table entry can be reconstructed. We combine this information with component field values into a context state record. We use the parent component ID to identify a context in log

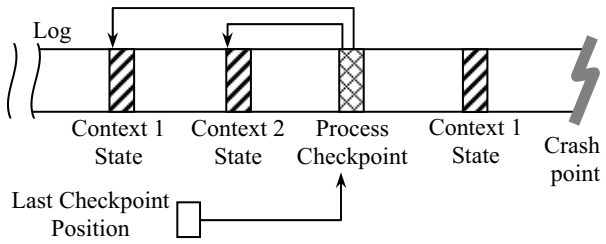

Figure 8 Context states, process checkpoints

records for messages and context state. Then the saved context states can be associated with call message records.

Reply messages (message 2) are not usually logged. So the LSNs in the last call table are often empty. We need the reply messages of last calls to recover the last call table because, after restoring a saved context state, we may not be able to re-create the replies of earlier incoming method calls. Therefore before we save a context state, we must write the replies of the last call table entries associated with the context, which are retrieved from the last call table as described in Section 4.1. We then fill in the LSNs in the last call entries. Next time we save the context state, if an LSN is not empty, we know the reply message is in the log and needn't save it again. Note that most reply messages are not logged because their last call table entries have been replaced by later calls from the same clients before we save context states.

In summary, during normal execution, a message interceptor can save context state before sending a reply message to a client (after processing). It first saves the context's reply messages in the last call table. Then it retrieves component fields and meta-information from global data structures, combines them into a state record, and writes it to the log. After that, it updates the state record LSN in the context table entry, which is saved as process states and used to retrieve the context state record during recovery. If no state record has yet been saved, the LSN of the context table entry corresponds to the creation record of the (parent) component.

4.3 Taking Process Checkpoints

In addition to saving context states, we save process state in *process checkpoints*, which includes the state record LSNs from the context table. These LSNs are akin to the recovery LSNs for pages in ARIES [13]. The relationship of process checkpoints and context state records is shown in Figure 8.

To allow concurrent accesses to the global tables, we log a begin checkpoint record before taking the checkpoint and log an end checkpoint record after we finish. We use sub-range locks to incrementally save global tables. When reading a process checkpoint, we examine all the log records between the begin checkpoint and end checkpoint record.

There is no need to force the log immediately after either a state record or a process checkpoint is written, since we can replay all the method calls from the creation record or the last forced states. Once a process checkpoint has been flushed to the log (possibly by a later send message), the log manager writes and forces the LSN of the begin checkpoint record into a well-known file. This LSN always points to a process checkpoint (if exists).

We take process checkpoints periodically. Context state records are saved independently of other contexts and process checkpoints. We will study the runtime overhead and the recovery performance in Section 5. From the experiments, we will estimate how frequent context states should be saved.

4.4 Recovery Processing

Figure 8 shows the process checkpoint and context state records at the time of a process crash. The well-known file's LSN identifies the last process checkpoint. The process checkpoint contains the LSNs of each context's last state record saved before the process checkpoint. However, more recent context states may exist after the last process checkpoint. We can recover contexts from these more recent context state records.

When a process starts, it registers with the recovery service. If it has been started to recover a prior failure, the recovery service sends back the original process identity information and directs the recovery manager in the process to recover. Its task is to recover the process data structures and the states of all contexts in the process.

The recovery manager first reads the LSN from the well-known file. This LSN is used as the *start point* to examine the log. If the LSN does not exist, the log is examined from the very beginning.

In the first pass, the log is examined from the start point to the end of the log. The recovery manager finds all the contexts that existed when the process crashed. It obtains the latest state record LSNs (or creation record LSNs) of the contexts by reading the context table records in the process checkpoint and by reading the context state records and creation records found during the log scan. After the first pass, the recovery manager restores the context states of all the contexts that have state records. Ordinary field's values are restored. References for remote and local components are kept in a list in every context, to be resolved later. (Newly created contexts without state records are processed in the second pass because component construction methods are allowed to make method calls to other components.)

Then the recovery manager reads the log in a second pass. It scans from the minimum LSN found in the context table to the end of the log. If a message log record

occurs earlier than the latest state record of the same context, *it is ignored* since we have already restored the latest context state records. Process checkpoint records are also ignored. All the context states up to the failure point are recovered by replaying incoming messages and suppressing regenerated outgoing messages. During the log scan, we buffer message records or creation records for every context until an incoming method call record is encountered. At this point the log records for all the messages related to the previous incoming method call (or creation call) are in the buffer. This previous call is replayed with outgoing method calls answered by the records in the buffer, similar to replay in the baseline system. If this is the first replay for the context, we check and resolve pointer fields that are component references before the replay.

After this pass, the recovery manager replays the remaining buffered method calls, which are the last incoming calls. If the last incoming call is successfully replayed to finish, then we set the context interceptors to normal execution states and the context begins to wait for incoming calls. If the reply to an outgoing call is missing from the log, the outgoing call is not suppressed by the message interceptor and normal execution begins. Note that the last incoming call's return value is returned to the caller, here the recovery manager, which satisfies condition 5.

Furthermore, the last call table is reconstructed from the last call records in the two passes. Only LSNs are filled in; actual reply messages are only read when they are required to reply to a duplicate call. The replays of the final incoming calls update the last call table accordingly, for the reply messages may not be in the log.

The above describes how we recover from a process crash. Recovering from a context failure is easier. The state record LSN can be found in the context table and the state record (or creation record) can be read from the log and the context restored to the state of the state record. Then the log after the state record (creation record) is read and incoming method calls for the context are replayed.

5. Experimental Results

We evaluated logging and recovery performance through experiments. In our new prototype, log optimizations and checkpointing can all be turned on or off via switches. Log records accumulate in a buffer and are written at a log force or full buffer. Unless otherwise noted, we use unbuffered writes with disk write cache disabled to ensure writes actually go to stable media.

We first describe the experimental setup. Then we show the experimental results of micro-benchmarks and an example application.

Table 2 Test Machines

CPU	One 2.20 GHz Pentium 4
L1/L2 cache/RAM	20KB/512 KB/512MB
OS	Microsoft Windows XP professional
.NET framework	Version 1.0.3705

Table 3 MAXTOR 6L040J2 Disks

Formatted Capacity	40,027 MB
Nominal RPM	7200
Average Read Seek Time	8.5 ms
Average Write Seek Time	10.5 ms
Track-to-track Seek Time	0.8 ms
Disk to Read Buffer Transfer Rate	236~472 Mb/s
Read Buffer to ATA Bus (Ultra ATA mode)	133 MB/s maximum

5.1. Experimental Setup

Our experiments were run on two identical Compaq Evo D500 machines connected via 100MB Ethernet. Machines and disks are described in Tables 2 and 3.

We mainly used micro-benchmarks to show the performance of individual operations. The micro-benchmark setup consisted of a client component making method calls to a server component. We measured the round trip elapsed time of a method call to the server component from inside the client component (i.e. from inside the client object instance). We turned on or off log optimizations and checkpointing and we changed the component types of the client and the server to show the runtime performance under various situations. Since the operating system timer is quite coarse (~15ms resolution), we batched multiple calls, measured total elapsed time, and divided total time by number of method calls. Total elapsed time was at least 1000 times the timer resolution.

For recovery performance, we killed the server component and measured its recovery time. Because of the timer resolution issue, which we found hard to deal with here, our results had rather large variances.

Besides studying individual operations with micro-benchmarks, we measured the performance of an example application — an online bookstore. This application is described in Section 5.5.

We ran every experiment 30 times and report the mean value. Standard deviations are less than 5% of the means except for recovery and results marked with *, where standard deviations are within 12% of the mean.

5.2 Runtime Logging Performance

5.2.1. Log Optimizations for Persistent Components. Table 4 shows the effect of our optimizations for persistent components. The local column shows the performance when both client and server components are on the same machine, while the remote column shows the results when they are on separate machines.

Table 4 Log Optimizations for Persistent Components (ms)

Client / Server Component Types	Local	Remote
External ➔ MarshalByRefObject	0.593	0.798
External ➔ ContextBoundObject	0.598	0.804
ContextBoundObject ➔ContextBoundObject	0.585	0.808
ContextBoundObject ➔ContextBoundObject(interception)	0.674	0.870
External ➔ Persistent (baseline)	17.0	17.3*
External ➔ Persistent (optimized)	17.1	17.0
Persistent ➔ Persistent (baseline)	34.7*	28.4
Persistent ➔ Persistent (optimized)	17.9*	10.8*

Table 5 New Components and Read-only Methods (ms)

Client / Server Component Types	Local	Remote
External ➔ Read-only	0.689	0.887
External ➔ Functional	0.672	0.875
Persistent ➔ Read-only	1.351	1.495
Persistent ➔ Functional	1.194	1.414
Persistent ➔ Subordinate	3.44×10^{-5}	
Persistent ➔ Persistent (Read-only methods)	1.407	1.547
Read-only ➔ Persistent	1.218	1.404

We compare the performance of a native .Net system, the baseline system, and our optimized system. In .NET, a server component must be a *MarshalByRefObject* or a *ContextBoundObject*. However, any component can be a client. We use "External" to indicate such a simple component. Phoenix/App persistent components are all derived from *PersistentObject*, which is in turn derived from *ContextBoundObject*.

The first four rows in Table 4 show the performance with basic .NET components for client and server. We see that the performance of using *MarshalByRefObject* and *ContextBoundObject* are similar. However, installing message interceptors (without doing any work in the interceptors) incurs a ~0.08ms overhead. The round-trip messages through network add ~0.2ms per call.

The last four rows in Table 4 show the performance of the baseline system and the optimized system. We can see their overhead is much higher than the native .NET objects because of logging. When the client is external, our optimized system performs the same as the baseline system since the same logging algorithm is employed under this situation. However, when the client is a persistent component, optimized logging achieves about a two fold speedup. The optimized system only forces the log when sending messages, saving two log forces per method call. In addition, we do not write log records for send messages because they can be re-created from previous messages during recovery.

5.2.2. Understanding the Performance Numbers. For *External*➔*Persistent* interactions, the server performs a sequence of log writes and forces, resulting in a sequence of unbuffered disk writes (because we make multiple method calls and take their average).

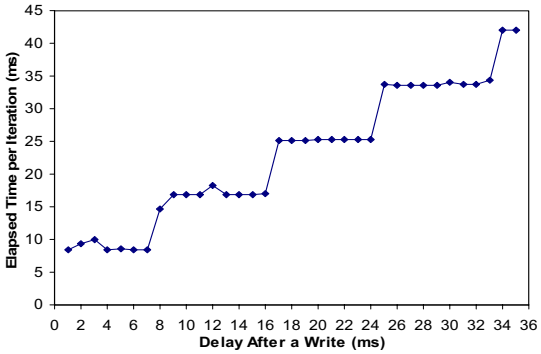

Figure 9 Unbuffered disk write performance

Figure 9 shows the performance of 1KB unbuffered disk writes in a loop (log message size in our experiments is less than 1KB). We insert some delay after every write in the loop body and report the elapsed time per iteration. The time per write is about 8.5 ms with no inserted delays. This is a little more than a full rotation time (8.33 ms/rotation for 7200 RPM). When delay time increases, the elapsed time jumps in discrete steps corresponding to the number of missed rotations. Thus unbuffered writes indeed miss a full rotation.

Therefore, the elapsed time of *External*➔*Persistent* is roughly two unbuffered disk writes (missing two full rotations). The local *Persistent*➔*Persistent* cases show four unbuffered disk writes for the baseline system and two for the optimized system, as expected.[1] Note that the first log write in the above cases sees a full rotational delay because of the way we performed the experiments: successive client requests interfere. Were client calls spaced out, e.g., with think time inserted, we would expect to see, on average, half a rotational delay for the first write.

For remote *Persistent*➔*Persistent* cases, log writing and forcing is done at both client and server machines. To understand the performance, we inserted delays at both client (between method calls) and server (in a method call). For the optimized remote *Persistent*➔*Persistent* case, we did not see discrete steps. So the results are not caused by full rotational delays. 10.8ms means a delay of 5~6ms per disk write, which could be explained by the average rotational delay of 4.17ms plus some small seek times. For the baseline remote *Persistent*➔*Persistent* case, there are four writes per method call. The timings for the two writes at the same machine are different. The first disk write may follow the 5~6ms as in the optimized remote *Persistent*➔*Persistent* case. The second disk write

[1] In our "local" experiments, client and server are in different processes, using different log files. But we believe newly allocated disk blocks for the two files are close enough to incur only small disk seek times.

Table 6 Checkpointing Performance (ms)

Client / Server Component Types	Write cache disabled	Write cache enabled
Persistent➔*Persistent*	10.8*	2.62
Persistent➔*Persistent* (save state on call)	11.8*	3.82

Table 7 Recovery Performance (ms)

Recovery Cases	Number of method calls replayed					
	0	1000	2000	3000	4000	5000
Empty log	492					
From creation	575	728	868	1007	1100	1199
From state	638	794	875	1162	1252	1507

misses a full rotation and costs 8~9ms. Therefore, altogether four writes cost ~28ms.

5.2.3. New Component Types and Read-only Methods. Table 5 shows the performance when subordinate, read-only, or functional components are used, or the read-only method optimization is enabled. Log forces are then eliminated, leading to much better performance than the last four rows in Table 4. The best performance is in the *Persistent*➔ *Subordinate* case, where method calls are local, without logging and context-crossing overhead.

In *External*➔*Read-only*, *External*➔*Functional*, *Persistent*➔*Functional*, and *Read-only*➔*Persistent,* logging is fully eliminated. As expected, the performance with external clients is similar to the performance of *ContextBoundObject* with interceptors (4th row in Table 4). However, there is ~0.5ms more overhead for the other two cases. This is due to the attachment to the message of information showing the sender's component type. In our initial experiments, the costs were even higher since we sent attachments with all messages. But we made an optimization. A client interceptor includes a field in its message attachment saying whether it knows the identity of the server. If this field is true, the server interceptor can omit the attachment in the reply message. The performance given includes this optimization.

For *Persistent*➔*Read-only* and *Persistent*➔*Persistent* with read-only methods, reply messages are still written to the log buffer (without forces), incurring additional overheads of 0.15~0.2ms versus *Persistent*➔*Functional*.

5.3 Runtime Checkpointing Overhead

Table 6 shows the runtime checkpoint performance. The result of *Persistent*➔*Persistent* with write cache disabled is the same as the remote *Persistent*➔*Persistent* (optimized) result in Table 4. To measure context saving overhead, we save the server context state after every method call. For the results in the right column, we performed the same experiments with disk write caching enabled, which removes the disk media costs.

In both situations, saving context state incurs an additional ~1ms overhead, quite reasonable compared to the disk media cost as shown in the left column and the computational cost (including the delay from memory to disk cache) as shown in the right column.

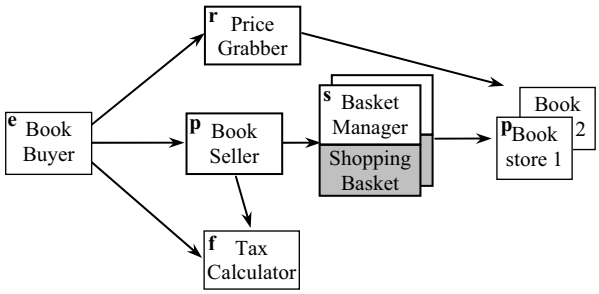

Figure 10 Online Bookstore Application

In our experiments, the size of an incoming message record is 186B and a state record 468B. For many components, the states could be substantially larger. Our small state in this example was responsible for the small computational overhead of saving the state.

5.4 Recovery Performance

Table 7 shows the recovery performance with and without context states. The elapsed time is measured inside the recovering process. But it does include the initialization of all the Phoenix/App runtime structure in the process. Recovery cost when the log is empty is ~0.5s.

Recovery processing starts by creating an object when no context state record is found. Otherwise the latest context state record is used. The recovery performances for both cases are shown. We vary the number of method calls replayed after the object creation or state restoration. The cost of replaying a method is roughly 0.15ms. Reading the creation records, creating an object, running the object constructor, and registering the object with Phoenix cost ~80ms. Restoring the state record costs ~60ms more. This 60ms is the cost of the checkpoint during recovery. Once the replay cost exceeds 60ms, recovery will be faster with a checkpoint. 60ms are approximately equal to the cost of replaying about 400 method calls, which means context states should be saved every 400 calls or more in the micro-benchmark.

5.5 Application Performance

5.5.1. On-Line Bookstore. In the previous subsections, we measured the runtime and recovery performance of Phoenix/App through micro-benchmarks. In this subsection, we show the performance improvements of an online bookstore (from [5]).

Figure 10 shows the architecture of the online bookstore application. There are six kinds of components. The arrows show the directions of method calls between components. A Bookstore component maintains the inventory of a store. The PriceGrabber component supports keyword searches on all the bookstores. The TaxCalculator computes sales tax based on total price and user information. The BookSeller manages a set of Basket Managers, each maintaining a shopping basket for a Book

Table 8 Performance of Online Bookstore Application

Optimization levels	Elapsed Time	Number of Forces
Baseline	589 ms	64
Optimized logging for Persistent Components	382 ms	46
Specialized components and read-only methods	296 ms	34

Buyer. BookBuyer runs in a console. It displays text menus and communicates with the PriceGrabber, BookSeller, and TaxCalculator to fulfil user requests.

For baseline system performance, all components are persistent except BookBuyer, which is external. For the performance of the optimized system, we specify component types with the leading letter of the type in the upper-left corners of component boxes in Figure 11. We specify read-only methods where appropriate.

To test performance, we rewrote the BookBuyer client to automatically generate inputs. Console outputs from the demo are redirected to files. The BookBuyer is run on one machine and all server components are run on the other machine. We repeatedly run the following set of operations: i) Search books with the keyword "recovery"; ii) Add a book from each bookstore to the shopping basket; iii) Show the shopping basket and compute total price including tax; iv) Remove all the books from the shopping basket. The performance is shown in Table 8.

Elapsed times are listed along with the numbers of log forces. With the baseline system, there are a total of 64 log forces. Optimizing logging for persistent components cuts 18 forces. Employing specialized components and read-only methods save another 12 forces. Note that since logging is only on the server machine and the methods are all simple, the elapsed times can be well explained by full rotational latencies plus small seek times.

Overall, we cut response time approximately in half for this small sample application. We think that this is indicative of the kind of performance gains that should be possible using the optimized logging and new component types and methods. This kind of performance gain is hard to achieve in the transaction processing world, and reflects the very real utility of our optimizations.

5.5.2. Multi-call Optimization. The PriceGrabber queries a number of Bookstores before rolling up the results and returning them to the BookBuyer. In our current prototype, the log is forced by the PriceGrabber at every Bookstore reply. With the multi-call optimization in section 3.5, the log would be forced only when the PriceGrabber itself returned. Hence, the PriceGrabber forces the log only once, regardless of the number of Bookstore's it queries.

6. Summary

In this paper, we have shown that the logging and recovery performance of Phoenix/App can be greatly improved. We re-examined the logging requirements for persistent components and took advantage of special component and method types to reduce logging overhead. Further, we implemented a checkpointing mechanism to reduce the recovery cost. Although our prototype was implemented in the .NET framework, our schemes to improve logging and recovery performance are applicable to other component-based programming environments. Importantly, our optimizations apply to a programming model that supports persistent stateful applications, a more natural model than the stateless, "string of beads" model supported by traditional TP monitors and workflow systems.

7. References

[1] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, and Causal. *ICDCS*, 1995.

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

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

[4] P. Bernstein, M. Hsu, and B. Mann. Implementing Recoverable Requests Using Queues. *SIGMOD* 1990.

[5] R. Barga, D. Lomet, S. Paparizos, H. Yu, and S. Chandrasekaran. Persistent Applications via Automatic Recovery. *IDEAS* 2003.

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

[7] E.N. Elnozahy, L. Alvisi, Y. Wang, and D.B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys,* 34(3), 2002.

[8] Sun Microsystems. Enterprise JavaBeans Technology. http://java.sun.com/products/ejb/.

[9] D. Esposito. .NET Remoting: Design and Develop Seamless Distributed Applications for the Common Language Runtime. *MSDN Magazine*, Oct. 2002.

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

[11] M. Kirtland. Object-Oriented Software Development Made Simple with COM+ Runtime Services. *Microsoft Systems Journal*, 12(11), 1997.

[12] D. Lomet and G. Weikum. Efficient Transparent Application Recovery in Client-Server Information Systems. *SIGMOD* 1998.

[13] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS,* 17(1), 1992.

[14] Object Management Group. Fault Tolerant CORBA Specification (V1.0). http://cgi.omg.org/cgi-bin/doc?ptc/00-04-04, 2000.