Transaction Support for Log-Based Middleware Server Recovery

Rui Wang [*1], Betty Salzberg [2], David Lomet [*3]

[1]*Microsoft,* [3]*Microsoft Research*
[*]*Redmond, WA USA*
{ [1]ruiwang, [3]lomet}@microsoft.com

[2]*CCIS, Northeastern University*
Boston, MA USA
salzberg@ccs.neu.edu

***Abstract*— We have developed log-based recovery for middleware servers that access back-end transaction systems (DBMSs). Transactional consistency is provided between in-memory state stored in middleware servers and persistent state stored in transaction systems. A new logging method called *results logging* is exploited to ensure coordinated recovery of in-memory state with persistent database state. Results logging incurs low logging overhead for middleware servers and requires little or no modification to existing transaction systems. This makes our approach a practical coordinated recovery technique.**

I. Introduction

Electronic commerce has extensively adopted multitier solutions. Front end clients submit requests to middle-tier *middleware servers*, which execute the business logic and maintain *in-memory* business state. Middleware servers may interact with one another and with back-end transaction systems such as DBMSs, which manage *persistent* business state.

High availability of business state with exactly-once execution in distributed systems is frequently required for e-commerce and can be provided by replication or logging. A logging approach [1], while not as fast in restoring service as replication, lowers both the initial cost and the on-going operational cost due to fewer computers being needed and managed. Log-based recovery can thus make high availability a commodity feature.

Transaction recovery for persistent state is standard in commercial systems. However, coordinating middleware server recovery for in-memory state with transaction system recovery for persistent state remains a problem. In this paper, we present a logging and recovery method to support transactions for log-based middleware server recovery that maintains consistency of middle-tier in-memory state with back-end persistent state upon system crashes.

A *middleware server process* (MSP) handles concurrent client-initiated service requests with a thread pool. Each request requires executing a *service method*, which may access in-memory business state and call other service methods. Our prior work [2] made in-memory business state recoverable. Based on this work, our method in this paper recovers consistent business state in both middle-tier and back-end systems.

We design a special type of service method, called a *transactional method*, which can access persistent state stored in transaction systems, in addition to in-memory state. *All-or-nothing* semantics is guaranteed: either all modifications to middle-tier in-memory state and back-end persistent state survive when the transaction is committed, or none of them survive when the transaction is aborted.

We log the results produced by transactional methods so that recovering the execution effect of a transactional method can be accomplished solely from the log, without any need to re-execute the method. We call this *results logging* and this is the essence of our contribution. Avoiding re-execution of transactional methods is essential when using the existing transaction systems, which do not provide idempotent execution of previously executed transactions. Otherwise, if we permitted a transactional method, during replay, to re-execute a back-end transaction, that re-execution might lead to duplicated executions, e.g. redundant e-commerce orders.

Results logging has low logging overhead in the usual e-commerce setting where the results are modest in size. At most one log flush by middleware servers is required for a transactional method execution. This low overhead yields excellent performance for our technique. Further, little or no change is required of existing back-end transaction systems in order to provide coordinated recovery of middle-tier and back-end business state. The characteristics of low overhead and little or no change to back-end infrastructure make our approach a practical coordinated recovery technique.

Following this section, section 2 describes the middleware server architecture supporting transactions. Section 3 briefly explains logging and recovery for in-memory business state, which were the focus of our prior work [2]. Section 4 elaborates on results logging and recovery for transactional methods. We review related work in section 5 and conclude in section 6. Throughout, we use **boldface** for definitions and *italics* for emphasis.

II. System Architecture

Figure 1 illustrates MSP architecture, which is derived from our prior system [2]. An MSP provides its service through service methods. A client, either an *end client* or another MSP, is identified as a *session* at the MSP. Over a session, a client will not send the next request before the reply for the

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.45

previous request is received. Thus anytime at most one request is outstanding for a session, while requests of different sessions are processed concurrently.

An MSP maintains two types of in-memory business state: *session state* and *shared state*. A client's session state (consisting of *session variables*) is accessible only by a service method executing within the session, while the shared state (consisting of *shared variables*) is accessible by all sessions.

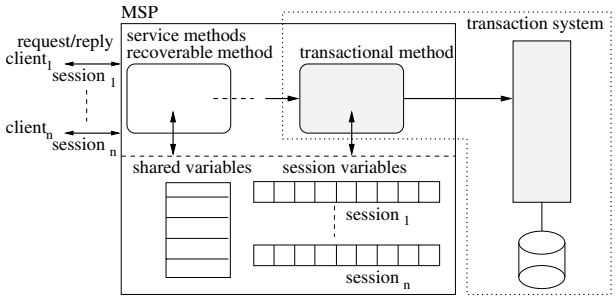

Fig. 1. Transaction-extended MSP architecture.

Transactional Methods: Within the existing framework we introduce transaction support, which is enclosed by dotted lines in Figure 1. A service method can be declared as a **transactional method**, which may interact with transaction systems such as DBMSs, but cannot call other service methods. We call a request to execute a transactional method a **transactional request**.

All operations within a transactional method execute in one transaction. If the transaction commits, all modifications to shared variables, session variables, and external persistent resources take effect. If it is aborted, no modifications take effect. If a transactional method does not interact with back-end transaction systems, the transaction is a **local** transaction; otherwise, the transaction is a **distributed** transaction and is managed by an external transaction manager.

A regular service method, which is non-transactional, is called a *recoverable method*. It is not allowed to access transaction systems directly, but can call other recoverable or transactional methods.

Shared State Locking: Shared variables may be accessed concurrently by transactional or recoverable methods. Read or write locks are required for accessing shared variables. A transactional method's lock spans the method execution, i.e. it is released at method end. This is strict two phase locking, which ensures both isolation and recovery for transactional methods. A recoverable method's lock encompasses only the variable access, i.e. it is released once the access is finished. This permits a shared variable's changes outside of transactions to be seen "immediately". Locking is implemented by the middleware infrastructure, and is transparent to middleware user programs.

III. Log-Based In-Memory State Recovery

When an MSP does not provide transactional methods, recovery of its in-memory business state relies on the now common logging of non-deterministic events together with re-execution of recoverable methods [2]. The logging of non-deterministic events, e.g. messages and shared variable accesses, captures all non-determinism and makes replay of recoverable methods deterministic and hence repeatable.

Each MSP has a single log stream, which is shared by all its sessions and shared variables. Table I shows an execution scenario for a session and the log records being generated.

TABLE I

An execution scenario and its log records.

execution scenario	*log records*
execute recoverable method 1	request for method 1 with parameters
read session variable *sev1*	
read shared variable *shv1*	value of *shv1* being read
write *sev1*	
write *shv1*	new value of *shv1* being written
execute recoverable method 2	request for method 2 with parameters
read session variable *sev2*	
write *sev2*	
write shared variable *shv1*	new value of *shv1* being written

Each request to execute a recoverable method is logged together with its parameters. During execution of a recoverable method, any value read from a shared variable is logged, and a new value written to a shared variable is also logged (for recovery of this shared variable). However, operations on session variables are not logged.

During recovery, each session's state and each shared variable's value are recovered separately. To recover a session's session state, we replay the logged requests for recoverable methods by re-executing those methods following this session's log records. During re-execution, when a method tries to read from a shared variable, it obtains the shared variable's value from the log, instead. Any access to a session variable, either read or write, is re-executed. In this way, the session state can be recovered up to the state as of the end of the stable log. During recovery, re-execution of a recoverable method can stop before the recoverable method finishes executing, when the end of the log has been reached for this session. After that point, this recoverable method switches to normal execution.

During the log scan after a crash, each shared variable is recovered by following its log records which contain the new value being written. After the end of the log is reached, a shared variable has the value which appears in this variable's most recent log record.

In addition, to speed up recovery, each session takes its own session checkpoints containing all its session variable values, and an MSP takes MSP checkpoints. Session recovery can start from its most recent session checkpoint, while the most recent MSP checkpoint contains the log scan start point for an MSP recovery after a crash.

IV. Transactional Method Recovery

When transaction support is introduced, recovery of a middleware server needs to recover in-memory business state and maintain the transaction semantics for transactional methods.

To replay a transactional request, we do not want to re-execute the corresponding transactional method, but will obtain the execution effect of the transactional method from the log. For this purpose, we design a special logging method, called **results logging**.

TABLE II
A TRANSACTIONAL METHOD EXECUTION AND RESULTS LOGGING.

execution scenario	*logging*
execute transactional method 1	**request log record** for method 1
read session variable *sev1*	
read shared variable *shv1*	
open a DBMS connection	
read/write database	
write *sev1*	
write *shv1*	
return *retv*	
	committing log record:
	retv, new values of *sev1* and *shv1*
	flush log buffer
	send commit decision to TranMan
	wait for transaction outcome
	result-status log record:
	committed or aborted
automatically close connection	

Table II shows a transactional method execution and its results logging. This transactional method accesses not only a session variable and a shared variable, but also a back-end database. All operations of this method are wrapped in a distributed transaction. At method end, a value is returned.

The transactional request is logged. Accesses to the shared variable are not logged. At method end, the method votes to commit the transaction. At this point, a **committing log record** is written to include the returned value and the new values being written to the shared variable and the session variable. Then the existing log records in the log buffer, including the committing log record, are flushed to disk. After the flush completes, the method further sends the commit decision to the external transaction manager and waits for the transaction outcome. After the transaction outcome is received, a **result-status log record** is written, which indicates either "committed" or "aborted". The result-status log record does not need to be flushed to disk immediately.

During recovery, to replay a logged transactional request, we obtain its execution effect from the log. The result-status log record may get lost due to a middleware server crash. In this case, we are unable to find the result-status log record in the middle server's log and will check with the transaction manager for the result status. In case of a "committed" result status, we install the execution effect from the committing log record, including the method returned value and new values of shared variables and session variables. The DBMS ensures that all changes to its persistent state by this method are durable. In case of an "aborted" result status, we do nothing and skip this logged request. The DBMS ensures that any change to its persistent state by this method is undone.

Why Not Re-execute Transactional Methods

Existing DBMSs do not support *idempotent* access. If we re-execute the transactional method to replay a logged transactional request, we need to re-execute interactions (read or write) with the database and thus the same write may be done twice, e.g. the same e-commerce order may be executed twice. Enabling idempotence for back-end systems frequently requires transactional queuing, adding to both implementation and execution costs.

Why Flush Log Buffer before Sending Commit Decision

If we do not flush the log buffer before the commit decision is sent to the transaction manager, the following scenario could occur. The commit decision has been sent, the transaction has been committed by the transaction manager and changes to the database by this method have become durable. Now the middleware server crashes and the committing log record gets lost, thus during recovery of the middleware server, it is unable to install the execution effect of this method on shared variables and session variables, but the execution effect of this method on the database persistent state is already in place. This violates transactional consistency.

Requirement on Transaction Manager

In case that the result-status log record is not found in the middleware server's log, we check with the transaction manager. Thus here we put a requirement on the transaction manager: *Transaction manager remembers committed transactions' status persistently until being explicitly directed to discard their status*. Aborted transactions' status can be discarded immediately, following the "presume-abort" protocol. Traditional transaction managers, such as those used in queued transaction processing, already satisfy this requirement [3]. This is the *only* requirement on existing transaction systems for them to work with our logging and recovery framework.

After a session checkpoint is taken, the middleware server is entitled to direct the transaction manager to forget all committed transactions which were initiated by this session since the session's previous checkpoint. This is because session recovery can start from this new session checkpoint and will no longer need the status of those transactions.

Results Logging within Locally Optimistic Logging

We developed *locally optimistic logging* in our prior work [2] to reduce logging overhead by reducing log flushes when multiple middleware servers interact. Each recovery unit, i.e. each shared variable or each session, needs to maintain a *dependency vector* to record the expected log persistence progress on each middleware server. Within the context of locally optimistic logging, before we send the commit decision, we need to do a *distributed log flush* according to the session's current dependency vector. This distributed log flush can result in multiple middleware servers flushing their log buffers in parallel, rather than only this middleware server flushing its local log buffer.

The example logging flow in Table II presents the longest path of results logging. There are a few other cases and Figure 2 illustrates all the cases (please refer to [4] for details).

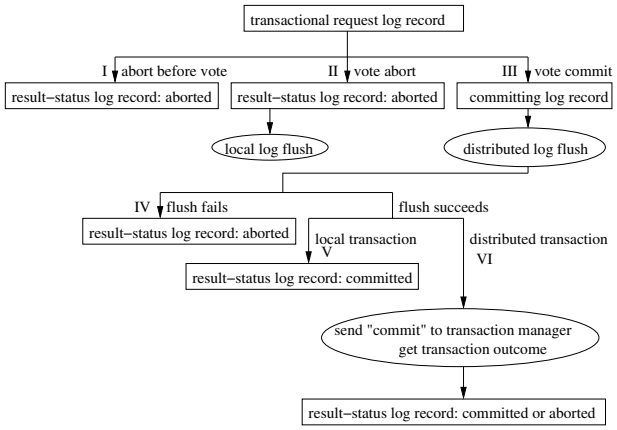

Fig. 2. Log records and actions of results logging.

Using results logging, a transactional method writes at most three log records and requires at most one local log flush or one distributed log flush. Except for the result status of a distributed transaction, we do not log any other interaction with transaction systems, such as database query results. So results logging incurs low overhead. In addition, it requires little or no change to existing transaction systems. Results logging is the *main contribution* of this paper.

V. Related Work

Fault tolerance of middleware servers incorporating transaction processing could be implemented via replication [5], [6], which requires duplicate resources and additional infrastructure support.

E-transactions [7] with the semantics of exactly-once execution in three-tier systems were implemented [8] by storing transaction results, including replies and side-effects on middleware server state, into backup servers or back-end databases. E-transactions do not support shared in-memory state, may require modification to DBMS commit processing and overload DBMS servers.

The Phoenix project [9], [10], [11] exploited message logging to enable recovery, including recovery dealing with back-end transaction systems. For recovery of ODBC sessions [10], some of this "logging" involved storing result messages in persistent tables at the DBMS server, especially when there were large results. Handling short results was optimized by logging at the client. In both cases, logging enabled replay to complete transaction result delivery when the transaction committed but the server crashed before the client had completed processing the results. Middle-tier recovery involved message logging as well. However, it did not include handling shared in-memory state in the middle-tier. Consistency with the back-end was provided via a transactional component that hid the interactions with the back-end from other components and guaranteed idempotence. The guarantee was provided by storing transaction result status in a persistent table at the DBMS server. The case of involving multiple servers was not explicitly treated.

Certain Web services infrastructures [12] provide transactional Web methods, where the transactional access semantics is applied only to the persistent state of transaction systems, not to the in-memory state of middleware servers. Neither in-memory state recovery nor coordinated recovery of in-memory state with persistent state is provided.

VI. Conclusion

We have described transaction support for log-based recovery of middleware servers. Building on our prior work of log-based recovery for in-memory state of middleware servers, our new work supports transactional methods, which enable middleware servers to access transaction systems.

Using our new notion of results logging for transactional methods, we ensure coordinated recovery of both in-memory business state and persistent business state. Results logging incurs modest logging overhead in maintaining the transaction semantics, and requires little or no change to existing transaction systems.

Utilizing our middleware server infrastructure and the existing transaction system recovery facilities, application programmers no longer need to cope with system failures and thus are able to focus on the business logic of the applications.

Acknowledgment

This work was supported by USA National Science Foundation under Grant No. 0533625.

References

[1] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A Survey of Rollback-Recovery Protocols in Message Passing Systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[2] R. Wang, B. Salzberg, and D. Lomet, "Log-Based Recovery for Middleware Servers," in *Proc ACM SIGMOD*, 2007, pp. 425–436.

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

[4] R. Wang, B. Salzberg, and D. Lomet, "Transaction Support for Log-Based Middleware Server Recovery," Microsoft Research, Tech. Rep. MSR-TR-2008-146, 2008.

[5] H. Wu and B. Kemme, "Fault-tolerance for Stateful Application Servers in the Presence of Advanced Transactions Patterns," in *Proc IEEE SRDS*, 2005, pp. 95–108.

[6] W. Zhao, L. E. Moser, and P. M. Melliar-Smith, "Unification of Transactions and Replication in Three-Tier Architectures Based on CORBA," *IEEE Transactions on Dependable and Secure Computing*, vol. 2, pp. 20–33, 2005.

[7] S. Fr"lund and R. Guerraoui, "e-Transactions: End-to-End Reliability for Three-Tier Architectures," *IEEE Trans. Softw. Eng.*, vol. 28, no. 4, pp. 378–395, 2002.

[8] ——, "A Pragmatic Implementation of e-Transactions," in *Proc IEEE SRDS*, 2000, p. 186.

[9] R. Barga, D. Lomet, and G. Weikum, "Recovery Guarantees for General Multi-Tier Applications," in *Proc IEEE ICDE*, 2002, pp. 543–554.

[10] R. S. Barga, D. Lomet, T. Baby, and S. Agrawal, "Persistent Client-Server Database Sessions," in *Proc EDBT*, 2000, pp. 462–477.

[11] D. Lomet and G. Weikum, "Efficient Transparent Application Recovery In Client-Server Information System," in *Proc ACM SIGMOD*, 1998, pp. 460–471.

[12] A. Freeman, *Microsoft .NET XML Web Services Step by Step*. Microsoft Press, 2003.