The VLDB Journal
DOI 10.1007/s00778-010-0199-1

REGULAR PAPER

Log-based middleware server recovery with transaction support

Rui Wang · Betty Salzberg · David Lomet

Received: 1 December 2009 / Revised: 23 April 2010 / Accepted: 14 June 2010

Abstract Providing enterprises with reliable and available Web-based application programs is a challenge. Applications are traditionally spread over multiple nodes, from user (client), to middle tier servers, to back end transaction systems, e.g. databases. It has proven very difficult to ensure that these applications persist across system crashes so that "exactly once" execution is produced, always important and sometimes essential, e.g., in the financial area. Our system provides a framework for exactly once execution of multi-tier Web applications, built on a commercially available Web infrastructure. Its capabilities include low logging overhead, recovery isolation (independence), and consistency between mid-tier and transactional back end. Good application performance is enabled via persistent shared state in the middle tier while providing for private session state as well. Our extensive experiments confirm both the desired properties and the good performance.

Keywords Application fault tolerance · Exactly once execution · Transaction processing · Recovery · Optimistic logging · Distributed systems

R. Wang (✉)
Microsoft, One Microsoft Way, Redmond, WA 98052, USA
e-mail: ruiwang@microsoft.com

B. Salzberg
College of Computer and Information Science,
Northeastern University, 360 Huntington Avenue
202 West Village H, Boston, MA 02115, USA
e-mail: salzberg@ccs.neu.edu

D. Lomet
Microsoft Research, One Microsoft Way,
Redmond, WA 98052, USA
e-mail: lomet@microsoft.com

Published online: 02 July 2010

1 Introduction

1.1 The current situation for enterprise Web apps

There are a number of widely used commercial infrastructures that support Web applications [20,27,33,34]. These permit integration across client, middle tier, and back-end transaction systems. In the absence of system crashes in the middle tier, these systems are quite effective in meeting requirements. They partition functionality, protect data, support a rich client experience, and provide scalability and decent availability.

The problem that they all suffer is that a crash can be a non-transparent event. That is, they do not provide exactly once execution in the presence of system crashes. A system crash can require extensive human intervention by system operators. Alternately, the error may be exposed to the user/customer, who experiences the need to re-execute the interaction with the system because the prior input has been lost. While these events are not common, even when systems do fail, they are not prevented. The impact can be very negative indeed, resulting in unhappy customers and substantial extra costs.

Classic transaction processing (TP) systems (e.g. involving a TP monitor) coped with system crashes and did provide exactly once execution. TP systems frequently dealt with client/server system configurations, though additional middle tier systems could be supported, with all system elements being typically managed within a single enterprise. Distributed transactions using two-phase commit were used to maintain appropriate middle tier state and keep it consistent with back end state.

Applying classical transaction processing to enterprise Web applications that can span multiple enterprises and almost always span protection domains encounters two primary problems.

– Enterprises are reluctant to rely on two-phase commit (2PC) across protection domains. 2PC is a blocking protocol, and no one wants to be a participant held hostage to the availability of a remote transaction manager in another organization.
– Using transactions throughout the middle tier is expensive. Transactions require an application model sometimes called a "string of beads", where state is committed, usually to a transactional queue or database, after each step (bead). This is also called the "stateless" application approach, as no important application state exists in the application outside of a transaction. All such state is stored in a transactional resource manager. This approach increases the complexity of developing the applications and introduces extra code path for the I/Os and extra latency for the frequent log forces needed to ensure that the state is captured on disk.

This has led to some leading internet sites "rolling their own" enterprise Web application infrastructure, e.g. Amazon. This is fine if you regard, as Amazon does, your infrastructure as part of your business, e.g. it sells its hosting service to a large number of other businesses. But this situation does not generalize to many enterprises, and in particular, it is a development model that is very difficult for mid-sized businesses to emulate. What they need is an infrastructure that provides a complete set of enterprise application attributes, and specifically, an infrastructure that guarantees exactly once execution.

There have been a number of research efforts to provide an infrastructure for enabling robust Web applications [16,17,23,31,32,35,36,44,45]. In particular, Melliar-Smith provides a replication-based method to assure that middle tier state can persist across system crashes. Replication is enabled by using atomic broadcast to ensure that messages arrive in the same order at all replicas. After one replica server crashes, it does not restore middle tier state by recovering the failed server using its persistent storage, instead, middle tier state is available immediately in other replica servers, thus potentially eliminating outages entirely. This is a high availability solution but requires duplicate computing resources and thus is a relatively expensive solution. We have pursued a different direction.

1.2 A recovery oriented approach

We look for a low-cost way to provide state persistence for middle tier applications and to unify these mechanisms with back-end transaction system state management to produce the desired exactly once execution property. This leads us to use recovery technology as the basis for our solution, similar to the approach taken in the Phoenix project [1,5], which provided an end-to-end exactly once execution framework for Web applications based on recovery.

A recovery oriented approach, as we have pursued it, is based on logging the information needed to rebuild the desired application state. This is very flexible. The log can be used to provide (1) conventional recovery at the site of the original application execution; (2) cold replication where the log is used to completely recreate and rebuild application state at a secondary server; (3) warm replication where the log is replayed at an already allocated secondary server; and (4) hot replication where the log is continuously replayed to keep the secondary server up-to-date.

Our solution extends the basic Phoenix approach and provides "stateful" application persistence based on recovery to enable exactly once application execution for the middle tier and back end. (Shegalov [2] showed how to empower a client so as to complete the end-to-end story.) We have focused on reducing logging overhead, especially of forced logging, and on how to handle interactions with back-end transactional systems so as to avoid the need for changing them.

1.3 Areas of improvement

Logging Cost: Logging to persistent storage determines the cost of providing application recovery, and success in reducing this cost is always beneficial. Given the rarity of crashes, we use optimistic logging to greatly reduce the number of log forces needed to provide persistence in a distributed system.

Recovery Independence: Using optimistic logging has a large performance benefit but can lead to two difficulties. The larger the number of systems involved, (1) the greater the overhead of tracking logging dependencies; and (2) the larger the number of systems that need to be involved in recovering from the crash of any system. To cope with these problems, we introduce the notion of a "service domain". Logging within a domain is optimistic, while messages crossing domain boundaries result in pessimistic logging. This makes the service domain a recovery domain as well and isolates both logging dependencies and crash effects to within a domain.

Back-End Interactions: The existing transaction systems do not provide idempotent execution of transactions previously executed. Resubmission of a previous transaction may lead to duplicated executions, e.g. redundant e-commerce orders. It is impractical to require that the existing transaction systems provide idempotence to enable exactly once execution for the middle tier and back end. Instead, our approach exploits the idempotence provided by the transaction manager (TM) coordinating two-phase commit. TM idempotence only requires that the TM remember the outcome of the

transaction, i.e. committed or aborted. Our 2PC can be restricted to the back end mostly or to within an administrative or security domain. That is, wide-scale 2PC is not required.

Application Performance via Middle Tier State Persistence: Another way that we improve system performance is via the kinds of persistent state that we support in middle tier applications. Most precursor research systems support persistent private state, usually called session state.

Shared persistent state in the middle tier has great value and has the advantage of being accessed at low cost. It has not, except at rather great cost, been made persistent. Many applications use cached out-of-date and non-persistent state to improve performance. This is very effective. What is less common is an ability to make this shared state and changes to it persistent. This is sometimes done via posting shared state changes to a back-end transaction system on every update, which is an effective but expensive mechanism. We provide this with much lower overhead, supporting both transactional and non-transactional reading and writing of this state. This allows more flexible system designs, since part of shared application state can be moved out from back-end transaction systems to the middle tier, without losing persistence and recoverability.

1.4 Our system and its technology

We have implemented our system on a commercial middle tier application framework, ASP.NET [27]. It attacks the problems enumerated above in an integrated and effective way. This has required that we invent and develop a number of new technologies that are the core of our contribution.

Locally Optimistic Logging: This greatly reduces the overhead for forcing the log, limiting it to only the places where service domains are crossed. Making optimistic logging "local" solves two problems: (a) the dependency information needed is restricted to only a small number of systems; and (b) the requirement for a system to be involved in recovery is restricted to its local service domain and does not propagate further.

Back-End Integration: We introduce a special middle tier service method called transactional method. State modified within this method has transactional isolation. Its transactions can be tied to transactions at back-end transaction systems, with commit coordinated by a transaction manager. It generalizes the back-end transaction manager idempotence for the rest of our middle tier application, so that they are only dependent upon the results produced by this transactional method and do not directly interact with the back-end transaction systems. A transactional method uses what we call "results logging" to provide idempotence, without requiring idempotence from the back-end transaction system.

Shared Application State: When a middle tier application program accesses (reads or writes) a variable of the shared application state, a read or write lock is acquired internally by our infrastructure, transparent to the application program. Shared state accesses are logged. This permits us to capture the nondeterminism introduced by the unpredictable access sequence produced by multiple concurrent sessions. We log non-deterministic accesses to shared state using state-based logging, i.e. value logging, to make shared state recovery independent of the need to recover individual sessions. This makes recovery much more flexible and reduces its latency.

Integrated Recovery Using Multiple Technologies: To realize our multi-tier infrastructure, we integrate a number of different logging and recovery technologies: locally optimistic message logging (a transition based technique); value logging where the after state is captured explicitly in the log; results logging, which captures the total effect of a transactional method; fuzzy checkpointing of a middle tier server; and parallel recovery of session states. The impact of fuzzy checkpointing on server performance is small, while parallel recovery is much faster than replaying all activities sequentially in log order.

1.5 Paper organization

Section 2 provides an overview of our middleware server (i.e. mid-tier application server) system. Section 3 elaborates on normal execution. Section 4 explains recovery. Section 5 presents performance measurements. We review related work in Sect. 6 and conclude in Sect. 7. Throughout, we use **boldface** for definitions and *italics* for emphasis.

2 System overview

Figure 1 illustrates the middleware server architecture. In this section, we begin by describing the nature of our application server and how it provides services. We next describe the correctness criteria related to distributed systems and messages. Then, we enumerate the different forms of application server state, followed by how we treat transactions and interactions with back-end transaction systems. Finally, we describe system assumptions and recovery requirements.

2.1 Services and sessions

A middleware server provides its service through **service methods**. A client, either an **end client process** or another

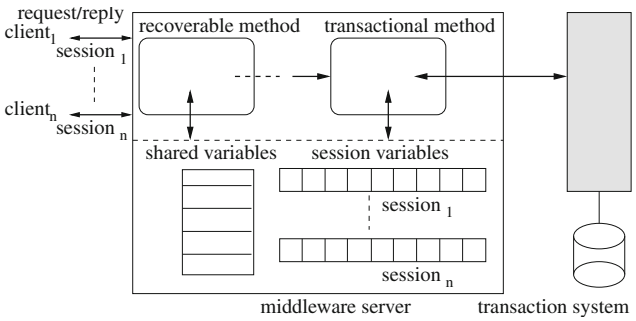

Fig. 1 Middleware server architecture

middleware server, uses the service via (synchronous) remote procedure calls to service methods.

Each middleware server maintains a request queue and a thread pool. A request message arrives first at the request queue and is served by a thread dispatched from the thread pool. The thread will execute the required service method, returning the result in a reply message. Messages between a middleware server and its clients are thus **request/reply** exchanges.

A client can start or end a **session** at a middleware server. Within a session, at most one request is processed at a time and the client will not send a new request before receiving the reply for the previous request. Requests over different sessions are processed concurrently inside a middleware server.

A service method is allowed to access (read or write) in-memory application state and can be declared either as **transactional** or **recoverable**. A request to execute a transactional method or recoverable method is thus called a **transactional request** or **recoverable request**.

A transactional method can access transaction systems such as DBMSs but cannot call any other service method. A recoverable method cannot access transaction systems but can call recoverable or transactional methods of the same or a different middleware server. In this way, a transactional method provides idempotence to its callers on behalf of the back-end transaction systems, and the middle tier service becomes composable and extensible via a recoverable method calling other service methods.

2.2 Global state consistency

Global state consistency requires that if the state of a process (a middleware server or an end client process) includes a message receive, either request or reply, the sender process's state must include the message send [13]. Figure 2 illustrates a violation of such consistency.

Process p_1 receives an input message m_1 from outside and logs it but does not flush the log record to disk before sending m_2 to p_2. Then, p_1 crashes and the log record for m_1 gets lost. Since we cannot guarantee the same message

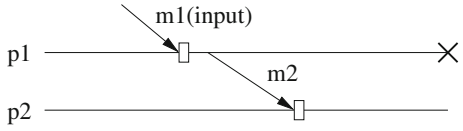

Fig. 2 Processes with message logging

m_1 will be received again, we cannot guarantee m_2 will be reconstructed and resent by p_1. However, p_2 believes that m_2 has been sent. Thus, message m_2 is an **orphan** message, and the current state of p_2 becomes an orphan. Global state consistency requires there to be no orphans after recovery.

2.3 In-Memory application state

A middleware server maintains two types of in-memory application state: **session state** and **shared state**.

For each client session, a middleware server maintains *private* session state that consists of **session variables**. A session variable can be accessed only in service methods that serve the requests of its session. A value can be saved (written) in a session variable when a request is processed and can be retrieved (read) any time later when this or a subsequent request over the same session is processed.

A middleware server maintains *global* shared state, which consists of **shared variables**. A shared variable is shared by all sessions of a middleware server, i.e. it can be accessed by any service method over any session. A value can be saved (written) in a shared variable whenever a request is processed and can be retrieved (read) later when the same or another request is processed, possibly over a different session. Shared variables may be accessed in an arbitrary non-deterministic order by concurrent threads.

Given that shared variables may be accessed at any time by transactional or recoverable methods, we acquire read or write locks when accessing shared variables. A transactional method lock spans the method execution, i.e. it is released when the method completes. This is strict two-phase locking [7], which ensures both isolation and recovery for transactional methods. A recoverable method 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 application programs.

2.4 Transactions

The transaction support here is designed for the typical short duration database style transactions, not for long running business transactions with compensations. Because of this, it does not implement the Web services Atomic Transaction specification [21], which defines contracts for Web services

to participate in distributed transactions. Rather, our transactions support more traditional 2PC with a traditional transaction manager.

2.4.1 Transactional methods

A transaction is initiated when a transactional method is invoked. All operations within a transactional method execute in this transaction. If the transaction commits, all its modifications to shared variables, session variables, and external persistent resources are guaranteed to persist. 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. The transaction can be committed when the transactional method returns to its caller within a session. Thus, only the session calling the transactional method can commit the transaction by notifying the transaction manager, while other transaction participants, including participating transaction systems, can vote to abort the transaction.

2.4.2 Distributed transactions

In our system, a transactional method is not allowed to call other transactional methods, and thus, a distributed transaction always involves only a single transactional method, possibly interacting with one or more transaction systems.

A transactional method interacts with transaction systems over a connection. Because it is costly to create and delete connections, a common practice is to maintain a *connection pool* at the middleware server and to map logical connections to connections in this pool. To access a transaction system, a transactional method obtains a connection from the pool and returns it to the pool when the method execution completes. Thus, a transactional method keeps a logical connection open only during the method execution.

Transaction managers (i.e. two phase commit coordinators), such as those used in traditional transaction processing, retain the state of the transaction until all participants have acknowledged that they have been notified of the outcome [18]. (Optimizations can be used to reduce the amount of state retained, e.g. "presumed abort".) We likewise require this from our transaction manager. This is the *only* requirement on any existing transactional infrastructure for it to work with our logging and recovery framework.

2.4.3 Transactional consistency

Transactional consistency requires a transactional method be *atomic* (all or nothing) and *durable*, i.e. when a transactional method commits, all modifications to shared variables, session variables and persistent resources become permanent and cannot be undone. If a transactional method interacts with transaction systems, its transaction outcome is determined by an external transaction manager.

The completion of a distributed transaction requires the two-phase commit protocol [18] involving the transaction manager. Once a transaction initiator acknowledges a transaction commit, the transaction manager may discard this commit. However, the transaction initiator, from the point of acknowledgement onward, needs to remember the transaction outcome. Consistency is violated if, e.g. after recovery, a transactional method is recovered as if aborted, but the transaction was instead committed by the transaction manager, which has forgotten this commit.

2.5 System assumptions and recovery

We assume fail-stop operation, i.e. once a failure occurs, the whole server crashes. We do not handle a faulty server continuing to run after a failure occurs. We do handle multiple concurrent crashes of middleware servers and transaction managers, although these crashes are rare in practice.

Message communication between a client and a middleware server is assumed to be unreliable, i.e. messages may arrive out of order, may be duplicated, or get lost. Due to possible message loss, the client may resend the same request until its reply is received. The middleware server can identify any duplicate or out-of-order request, and the client can identify any duplicate reply.

After a crash occurs, a middleware server must be recovered to a state that is at least as late as its most recent "visible" state. This state must satisfy both *global state consistency* and *transactional consistency*. These requirements, together with a client resend of the same request until the corresponding reply is received, guarantee *exactly once* execution of service requests, both recoverable and transactional.

3 Normal execution

During normal execution of a middleware server, requests are processed and nondeterminism is logged so that recovery of the middleware server after a crash can reconstruct the in-memory application state by replaying the logged requests. We elaborate on normal execution in six aspects: message handling, session handling, shared state handling, recoverable method handling, transactional method handling, and middleware server checkpointing.

3.1 Message handling

To identify duplicate or out-of-order messages, we associate a **request sequence number** with both a request and its

reply [1]. Over each session, the client maintains a **next available request sequence number** and the middleware server maintains a **next expected request sequence number**. In addition, the middleware server buffers the reply of the latest request for each session, so that this **buffered reply** can be resent should it get lost due to network failure or client crash [1]. To log request and reply messages, we exploit an integration of two classical logging methods: pessimistic logging and optimistic logging.

3.1.1 Pessimistic versus optimistic logging

Pessimistic logging guarantees that no orphans are ever created. One form of pessimistic logging [1,3] is that messages are written to a buffer upon receipt. Before this receiver sends a message, it flushes the buffer to disk. In Fig. 2, with pessimistic logging, before m_2 is sent, the log record for m_1 must be on disk. Even if p_1 crashes, it can still be recovered up to having received m_1 and re-executed to enable the resend of m_2. So m_2 never becomes an orphan. Pessimistic logging incurs disk write overhead for this log flush.

Optimistic logging does not flush the log buffer before sending a message, thus allowing orphans to be created, but orphans will be detected later via **dependency vectors** (DVs) [9,40] and eliminated via recovery. A process's DV includes a state identifier for each process on which this process depends and is attached to any message this process sends. A process's **state identifier** consists of a **state number** and an **epoch number**. Its state number is its most recent log record's log sequence number (LSN). A process's epoch number identifies a failure-free period of its execution and is incremented after recovery from a crash. A process always depends on itself at its current state identifier. (We elide the epoch number to simplify the following presentation.)

Referring to Fig. 3, when process p_1 receives a message m_1, p_1 posts it to a buffer with log sequence number 10. Before sending message m_2 to p_2, the DV is attached in m_2 to include 10 as p_1's state number. When p_2 sends m_3 to p_3, the log record written by p_2 has log sequence number 20. Hence, both 10 and 20 are in the DV sent with m_3. The DV is *transitive* as LSNs from all processes on which a sender depends are sent with its message. After m_3 is received, p_3's DV is [p_1*:10, p_2:20, p_3:30*]. When m_5 is received, m_5's DV [p_1*:11*] is **merged** (via item-wise maximization) into p_3's DV, which becomes [p_1*:11, p_2:20, p_3:31*].

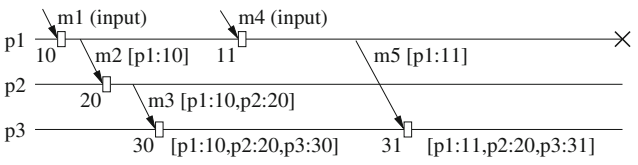

Fig. 3 Messages with dependency vectors

Later, if p_1 crashes, after **crash recovery**, p_1 broadcasts a **recovery message** indicating the state to which it has recovered (called the **recovered state number**) for the previous epoch. Other processes log and remember this recovered state number. If p_1 is unable to recover to state 10, both p_2 and p_3 will know they are orphans by checking their DVs against this recovered state number, and must do **orphan recovery** to roll back to a state before they got the orphan messages.

Note that a process R does not process any application specific message from another process's new epoch, until R has received and processed recovery messages from the sending process S's past epochs. Otherwise, were an application specific message to be processed and its DV is merged into R's DV, R's orphanhood may be undetectable.

An *output message* (to the external world) should never become an orphan. Thus, before it is sent, the sender issues a **distributed log flush** following its DV. If p_3 sends an output message after getting m_5, p_1, p_2, and p_3 are notified to flush their log up to 11, 20, and 31, respectively.

Optimistic logging reduces logging overhead by reducing log flushes. However, system complexity increases, and a process crash may cause another process to roll back. Message overhead is increased by recovery message broadcasting and the notification required for distributed log flush. When the number of processes is large, the size of DVs becomes large, increasing message size.

3.1.2 Locally optimistic logging

A middleware server may send requests to other middleware servers. Thus, there are message interactions among middleware servers. Some interacting middleware servers are provided by the same service provider and have fast and reliable communication. We exploit this to configured them into **service domains**. Less tightly associated middleware servers with less reliable communication will usually be in separate service domains. Service domains partition the middleware servers. End client processes and transaction systems are outside of all service domains.

We exploit service domains via **locally optimistic logging** for request and reply message exchanges. Message exchanges from a middleware server to systems outside of its service domain use pessimistic logging, and message exchanges *within* a service domain use optimistic logging. To reduce logging overhead, all sessions inside a middleware server share one physical log. Figure 4 illustrates this logging. Since an end client process is outside of any service domain, message exchanges between it and any middleware server use pessimistic logging.

A message (request or reply) within the service domain includes the sender's DV. When a message is sent across service domains, a distributed log flush is executed in accordance with the sender's DV. The separate local flushes

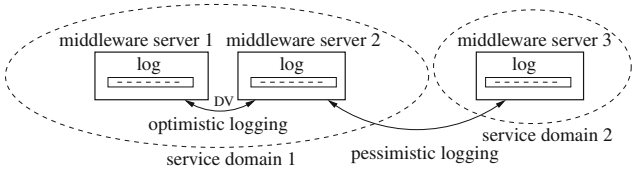

Fig. 4 Locally optimistic logging

Table 1 Actions for message M (request or reply)

	inside service domain	across service domains
before send:	attach sender's DV to M;	distributed log flush following sender's DV;
after receive:	check if M is an orphan; if yes, discard M and stop; log M and attached DV in buffer; merge attached DV into receiver's DV;	log M in buffer;

required by a distributed log flush can usually be done in parallel, since physical logs of middleware servers in the service domain will rarely share a disk controller. Once the distributed log flush succeeds, this externally directed message will never be an orphan and the sender's DV does not need to be attached to this message. Table 1 lists the actions associated with message exchanges.

Locally optimistic logging combines advantages from both optimistic logging and pessimistic logging. Optimistic logging within a service domain reduces log flushes. With pessimistic logging across service domains, the service domain becomes the boundary for dependency vector propagation. Since the DV contains only the dependency on middleware servers in the service domain, it has a limited size and adds only limited overhead to messages sent within the service domain.

A middleware server crash can cause only other middleware servers in the same service domain to roll back. Thus, recovery independence is maintained across service domains. After its crash recovery, a middleware server broadcasts recovery messages only within its service domain. Each middleware server needs to keep recovered state numbers only for middleware servers in its service domain. Finally, since middleware servers within a service domain usually have fast and reliable communication, the message overhead for recovery message broadcasting and distributed log flushes will normally be modest.

3.2 Session handling

3.2.1 Session as recovery unit

When a middleware server sends a request to another middleware server, it is a session of the sending server, say SE_c, that works as a client and starts a session, say SE_s, at this other middleware server. The request is sent over the session SE_s. In this case, SE_s is called an **outgoing session** started by SE_c.

When a middleware server crashes, another middleware server in the same service domain may become an orphan, but usually only some of its sessions are orphans and need recovery. Non-orphan sessions can continue normal execution. Thus, sessions are the unit of recovery, while middleware servers are crash units, i.e. all sessions executing on a middleware server "crash" when the middleware server crashes.

If a DV were maintained to capture dependencies for a middleware server as a whole, all its sessions would require roll back, possibly unnecessarily, whenever any one of its sessions required rollback. To avoid this, we associate a DV with each session. This enables sessions to recover separately, avoiding unnecessary rollback cost. Thus, in Table 1, *sender's DV* refers to the DV of the sending session, and *receiver's DV* means the DV of the receiving session.

Correspondingly, each session must have its own state number, which is the most recent LSN of the session. Since any session does not crash by itself but only as part of its middleware server crash, every session's epoch number is the same as its middleware server's epoch number.

3.2.2 Session checkpointing

All requests over a session are logged to enable session state recovery via replaying logged requests. To speed up session recovery, a **session checkpoint** is taken whenever its logged information since the previous checkpoint reaches a threshold. Each session is checkpointed independently, and only between requests during normal execution. Because of this, a session checkpoint contains only session variables, the buffered reply, the next expected request sequence number, and every outgoing session's next available request sequence number. It does not contain control state, e.g. stacks and program counters. New requests are held until the checkpoint is completed. As part of a session checkpoint, a distributed log flush is issued according to the session's DV to ensure that the state as of checkpoint cannot become an orphan. On completion, the session's previous log records can be discarded.

3.2.3 Session log management

All sessions of a middleware server share one physical log. To recover a session, its log records need to be extracted from the shared log. To make such extraction efficient, each session maintains a **position stream** consisting of the positions (inside the physical log) of its log records since the latest session checkpoint. An in-memory **position buffer** is associated with each position stream. When a session's log

records are written, their positions are *sequentially* written to its position buffer. *Only* when the position buffer becomes full is it flushed to disk. So the cost of writing positions is low. In case of a middleware server crash, positions that are still in the buffer get lost. Missing positions of persistent log records will be reconstructed from the physical log during crash recovery. After a session checkpoint is taken, the session's previous position stream is discarded. The position stream's maximum length is determined by the session checkpointing frequency and is usually small. When a session ends, its position stream is discarded and a log record is written to mark the session end.

3.3 Shared state handling

Like a session, a shared variable is a recovery unit and has its own DV and state number, which is the log record LSN of its most recent write. The DV indicates whether the variable's value is an orphan. Since a shared variable does not crash in isolation but only as part of its middleware server crash, every shared variable's epoch number is the same as its middleware server's epoch number. (Once again, the middleware server is the crash unit, and here a shared variable is the recovery unit.)

When a session accesses a shared variable, we need to track the dependencies introduced by that access. By considering *read/write semantics* separately, we can see how to deal with these dependencies, i.e. how to update DVs for both session and variable.

Read: For a read, the variable's dependency is passed to the reader session, but the reader session's dependency does not need to be passed to the variable. A read has no effect on the variable. Thus, reading a shared variable *merges* this variable's DV into the reader session's DV.

Write: For a write, the writer session's dependency is passed to the variable, and the variable's dependency does not need to be passed to the writer session. Since a write completely replaces the variable's existing value with a new value, the variable's existing dependency is replaced by the writer session's dependency. Thus, writing a shared variable *replaces* this variable's DV with the writer session's DV.

If a shared variable is read before it is written, then the DV processing will go in both directions as described individually.

3.4 Recoverable method handling

When a request is received by a recoverable method, it is first logged. Then, the method is executed. Within a session, at most one service method can be executed at a time. Thus, the access order to the session state is captured by the order of logged method requests. During recovery, the execution effect of the method on the session state is recovered by re-executing the method. Because of this, accesses to session variables inside a recoverable method need not be logged. However, accesses to shared variables by a recoverable method do need to be logged.

3.4.1 Value logging for shared variables

Access order logging was suggested for shared state access [37]. That is, the access order is logged, and the same access order is followed during recovery to reconstruct the shared state. However, this approach can mean that recovery for one session becomes dependent on another session's recovery. For example, if session R reads a shared variable that was written by another session W, should R become an orphan, its recovery will require the writer (whether orphan or not) to roll back. This enables the writer to recreate and rewrite the value for the variable, enabling the reader session to read this value for its recovery.

With access order logging, deadlocks involving thread pooling are possible. Each request (or logged request) is processed (or replayed) by a thread, which is dispatched from the thread pool. When a shared variable becomes an orphan, access order logging requires other orphan sessions to roll back and replay logged requests to bring the shared variable to its most recent non-orphan value. Now, if a thread processing a new request tries to read a shared variable and finds this variable an orphan, it has to be blocked until orphan sessions' recovery brings this variable to the most recent non-orphan value. It is possible that so many threads are blocked for similar reasons that the thread pool has no threads left for orphan sessions to recover the orphan shared variable. This is a deadlock and the middleware server hangs.

To overcome the above two drawbacks of access order logging, we exploit **value logging**. For a *read*, the variable's value with its DV is logged. Hence, a recovering reader session can obtain the value from the log directly. This means that reader session recovery does not depend on the writer's recovery. For a *write*, in addition to the written value and the writer session's DV, the LSN of the previous write log record for the same variable is logged, meaning that write log records are *chained backward*. If any session tries to read a shared variable and this variable's value is an orphan, session recovery can roll back this variable to its most recent non-orphan value by following this chain. In this way, no other session need be recovered to recreate the earlier value. Since shared variables are expected to have small storage sizes, value logging should incur only modest overhead compared to access order logging.

Table 2 lists actions involved with accessing a shared variable inside a recoverable method. This includes dependency tracking and value logging. There exist two more

Table 2 Actions for accessing shared variable *SV* inside a recoverable method

read	write (NewValue)
check *SV*'s DV: if *SV* is an orphan, roll *SV* back to its most recent non-orphan value; log *SV*'s value and DV; merge *SV*'s DV into reader session's DV; change reader session's state number to the new log record LSN; return *SV*'s value;	log writer session's DV, *NewValue* and LSN of previous write log record; replace *SV*'s DV with writer session's DV; change *SV*'s state number to the new log record LSN; set *SV*'s value to *NewValue*; return;

differences between read and write accesses. First, when a session writes a shared variable, it need not check whether the variable's existing value is an orphan, because this value will be replaced by a new value. (Unlike a writer session, a reader session needs to check whether the variable is an orphan so that the value returned to the reader is not an orphan.) Second, reading a shared variable causes the reader session's state number to change, while writing a shared variable causes the variable's state number to change.

3.4.2 Shared variable checkpointing

To shorten the part of the log that needs to be read to roll back a shared variable to its most recent non-orphan value, we take a **shared variable checkpoint** whenever the number of writes since the previous checkpoint reaches a threshold. Shared variable checkpoints are taken independently. To checkpoint a shared variable, a distributed log flush is issued following its DV. Then, its value is logged and this value will never become an orphan.

A checkpoint's subsequent write log record points back to the checkpoint. But a checkpoint does not point back to any previous write log record as no earlier log records are required in order to recover a shared variable. The backward chain breaks at checkpoints. Figure 5 illustrates this.

3.5 Transactional method handling

We do not want recovery of a transactional method to involve re-execution of the method. If that was required, it would be necessary to replay the method's interactions with back-end transaction systems. Transaction systems, as well as their clients, would then need modification to support such replay idempotently, and clients would need to maintain virtual connections with DBMSs across failures [4]. Since any *connection* of a transactional method with a transaction system is kept open only for the method duration, there is no control state maintained with the transaction system outside the method. This makes it possible to avoid re-executing interactions with transaction systems during recovery.

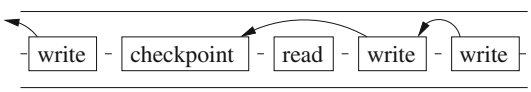

Fig. 5 A log record sequence of a shared variable

We exploit *results logging* to capture the execution effect of a transactional method by logging the reply of the method call and the values of session and shared variables written by the method. This avoids the need to re-execute a transactional method as there is no middle tier control state associated with a transaction outside the method that requires its replay. Thus, a recoverable method's *call* to a transactional method inside the same middleware server can be replayed from the log, without the need to re-execute the transactional method itself. The recoverable method's control state resumes after the transactional method call, using only the transactional reply returned together with the values of written session and shared variables, all of which we ensure are on the log.

Before we delve into results logging, we describe the preprocessing to establish the execution context for results logging.

3.5.1 Execution context

At the start of a transactional method, the session's DV is saved in-memory, and the transactional request message's DV, if it exists, is merged into the session's DV. Prior to a session variable or shared variable being written for the first time within a transaction, its value (and DV, for a shared variable) is saved. If the transaction be aborted, transaction undo replaces written variables with their saved values (and DVs, for shared variables) and sets the session's DV back to the saved one. Back-end transaction systems guarantee that modifications to their resources are undone on abort. Dependency is tracked along with shared variable access inside a transactional method. Read and write locks on a shared variable are held until transaction method end.

Figure 6 illustrates execution of a transactional method. Middleware servers *MS1* and *MS2* are in the same service domain. *MS2* sends a transactional request message *M* with a DV *[p1:1,p2:5]* to *MS1* over the session *SE1*. This DV shows dependencies on *MS1* (symbolized by *p1*) at state number *1* and on *MS2* (symbolized by *p2*) at state number *5*. During the transactional method execution, *SE1* first reads the shared variable *SVx*, then writes *SVy*. DV changes are listed in Table 3, where each column represents an occasion, "-" indicates no change and DVs indicate those *before*

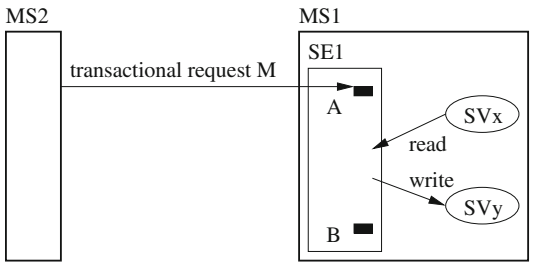

Fig. 6 A transactional method execution, where A and B are log records

Table 3 DV changes in the scenario of Fig. 6

	request	read	write	commit
M	[p1:1,p2:5]	–	–	–
SE1	[p1:2,p2:3]	[p1:3,p2:5]	[p1:3,p2:6]	[p1:4,p2:6]
SVx	–	[p1:1,p2:6]	–	–
SVy	–	–	[p1:4,p2:7]	[p1:3,p2:6]

various occasions. For simplification, epoch numbers are elided from DVs.

Before the request message *M* is received, *SE1*'s DV is *[p1:2,p2:3]*. When the request message is received, the message's DV *[p1:1,p2:5]* is merged into *SE1*'s DV. The message is logged in the request log record *A* with an LSN *3*. So before the read, *SE1*'s DV is *[p1:3,p2:5]*. Upon the read, *SVx*'s DV *[p1:1,p2:6]* is merged into *SE1*'s DV. So after the read (that is, before the write), *SE1*'s DV is *[p1:3,p2:6]*. Before the write, *SVy*'s DV is *[p1:4,p2:7]*. Upon the write, *SVy*'s DV is replaced with the session's DV *[p1:3,p2:6]*.

3.5.2 Results logging

We exploit **results logging** for transactional methods in order to avoid re-executing interactions with back-end transaction systems during recovery. When a transactional method starts to commit its transaction at method end, a **committing log record** is written (into the log buffer), which contains the transactional reply, and the final values of session variables and shared variables written by the method. In Fig. 6, upon commit, a committing log record *B* is written with an LSN *4* and *SE1*'s DV becomes *[p1:4,p2:6]*.

To reproduce the effect of replaying a committed transactional request, we replace the session variables and shared variables with their logged values, and return the logged reply. Transaction systems guarantee that the transactional modifications to their resources are durable. Results logging guarantees durability for the effects of the transactional method execution.

No matter whether a transaction is committed or aborted during normal execution, a **result-status log record** indicating the transaction outcome is *always* written for a transactional request. The result-status log record is mandatory in case that the transaction method votes to abort, and it simplifies session recovery processing in other cases.

With results logging, the execution effect of a transactional method can be obtained from its committing log record and/or its result-status log record. This reduces or eliminates logging for other occasions. First, the transactional request message's DV, if any, need not be logged with the transactional request message. Second, no logging is required when a session variable or shared variable is accessed. Further, detailed interactions with transaction systems, e.g. database query results, need not be logged. Figure 7 illustrates log records and actions of results logging during normal execution. We discuss the cases we need to deal with next.

Abort Before Vote (branch **I***):* If the session is found to be an orphan before the method votes, no committing log record is written, the transaction is aborted, and a result-status log record indicating "orphan-initiated abort" is written but not immediately flushed to disk. If this log record get lost due to a crash, this transactional request would be simply recovered as "aborted", since recovery would find neither a committing log record nor a result-status log record for this request.

Transaction Votes Abort (Branch **II***):* If a method invocation votes to abort its transaction, a result-status log record containing a specific aborting reason (related to the application logic) is written and immediately flushed to disk. This *local* log flush is necessary because recovering a transactional method does not re-execute this method but re-obtains the result, referring to a specific aborting reason, from the log. If the middleware server crash, the "abort" result can be obtained from the log after the middleware server recovers.

When a transactional method votes abort, it finishes as if it was not executed. Before the transactional reply is sent out, the session's DV is restored to the DV saved when the transaction request was received. The reply message does not have the dependencies on other middleware servers that are indicated by this restored DV. So if the reply is sent within the service domain, this DV is not attached to this reply message. Further, if the reply is sent across service domains, before it is sent, a distributed log flush following this DV need not be issued.

Transaction Votes Commit (Branch **III***):* A method can vote to commit its associated transaction only at method end (just before it returns its results). If the method votes to commit, a committing log record is written (into the log buffer). In addition to the transactional reply and the values of written session variables and shared variables, the session's

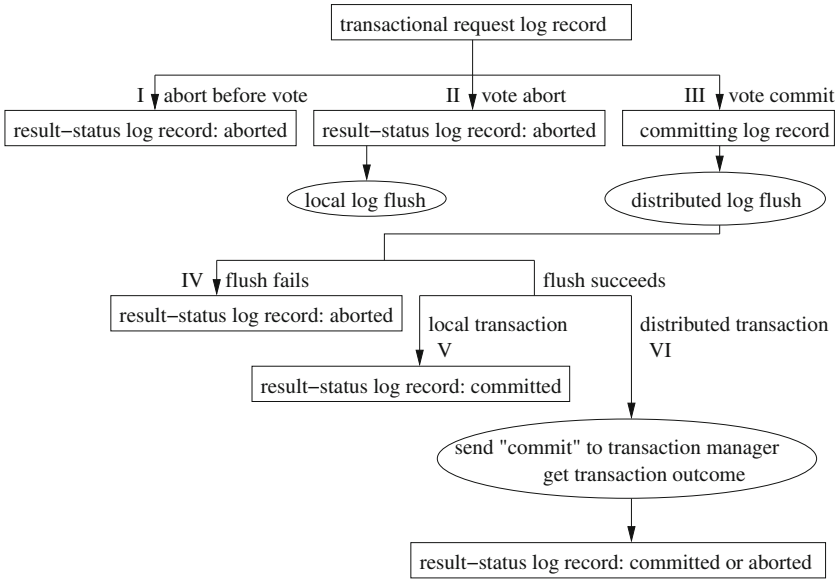

Fig. 7 Log records and actions of results logging

DV is included in the committing log record when the transaction is local, and the transaction identifier is included when the transaction is distributed.

Then, a distributed log flush is issued in accordance with the session's DV to ensure that the session's current state never becomes an orphan. This distributed log flush is required for *transactional consistency*.

When the transaction is local, i.e. not interacting with transaction systems, a transactional method independently decides whether to commit or abort the transaction. Before the committed reply is sent to the client, transactional consistency requires that all modifications to session variables and shared variables become permanent and that the session's current state never becomes an orphan.

When the transaction is distributed, the transaction outcome is decided by the transaction manager that coordinates the transaction. For transactional consistency, before this method sends its "commit" vote to the transaction manager, it must guarantee that its current state never becomes an orphan (like the state and message stability requirement before the commit request [3]). Otherwise, the transaction may be committed by the transaction manager, but later this session's current state becomes an orphan and needs rollback, which would "abort" the transactional method part of the transaction.

The distributed log flush also tries to flush the committing log record to disk. After the committing log record is flushed to disk, regardless of whether the result status is "committed" or "aborted", a *result-status log record* will be written. However, it need not be flushed to disk immediately, since the result status can be derived during recovery should this log record be lost due to a crash.

Now, we elaborate on writing a result-status log record and how to resolve the transaction outcome when the result-status log record is lost.

branch **IV**: If the distributed log flush fails because one or more middleware servers in the same service domain have crashed, making the session an orphan, the transaction needs to be aborted. A result-status log record indicating "aborted" is written but not flushed. If this log record get lost after a crash, the result status would be resolved as "aborted" during recovery.

- For a local transaction, its resolution relies on the logged DV in the committing log record.
- For a distributed transaction, with its identifier being included in the committing log record, its resolution relies on the transaction manager.

branch **V**: If the distributed log flush succeeds, when the transaction is local, a result-status log record indicating "committed" is written but not flushed. If this log record get lost after a crash, the logged DV is used during recovery to determine the result status.

branch **VI**: When the transaction is distributed, the method sends its "commit" vote to the transaction manager. Once the method receives the transaction outcome from the transaction manager, this outcome will be logged in a result-status log record but not flushed. If this log record be lost after a crash, the transaction manager is queried for the outcome.

When the outcome is "committed", the session saves the transaction identifier. After the next session checkpoint is

taken, the session tells the transaction manager to discard the status information of all committed distributed transactions that were initiated by this session since the previous session checkpoint. Session checkpointing makes all transaction results recoverable at the middleware server and permits the transaction manager to forget transaction outcomes.

If a transaction needs to abort due to a failed distributed log flush or a transaction manager's decision, transaction undo is executed, like the votes abort case, except that no *local* flush is required for the result-status log record.

An abort is required should the distributed log flush fail or the session be detected as an orphan before the vote. In this case, no reply is sent out. If the transactional request message itself is not an orphan (as indicated by its associated DV), the client will resend the same request after a timeout as it would for a lost message. If the transactional method has accessed transaction systems, the corresponding distributed transaction is aborted by the transaction manager by transaction timeout.

After a transaction, either local or distributed, is committed successfully at the method end, the distributed log flush (following the session's current DV) must have succeeded and the current session state will never become an orphan. Regardless of whether the reply message is sent within the service domain or across service domains, the session's DV need not be attached to this reply because the distributed log flush has removed all dependencies.

Dependency tracking for shared variables read or written by the method will, at the method end, also have their dependencies transitively included in the session's DV. Thus, success of the distributed log flush means that the current values of these variables never become orphans. Since the values of variables *written* by this method are included in the committing log record, the committing log record can serve as a checkpoint for these variables. However, this log record does not serve as a checkpoint for variables *read* by this method, since their values are not included in this log record.

For example, in Fig. 6, if the transaction is committed, the distributed log flush following the session's DV *[p1:4, p2:6]* must have succeeded. Thus the current values of *SVx* (with a DV *[p1:1,p2:6]*) and *SVy* (with a DV *[p1:3,p2:6]*) will never become orphans. The committing log record *B* serves as a checkpoint for *SVy*, but not for *SVx*.

3.5.3 Summary

As Fig. 7 shows, during normal execution there are four abort types: orphan-initiated abort, transaction votes abort, abort by failed distributed log flush, and transaction manager abort, while there are two commit types: local commit and transaction manager distributed commit. 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 at the middleware server. Except for the outcome of a distributed transaction, we do not log any other interaction with transaction systems. So the overhead of results logging is very low. In addition, supporting recovery requires little or no modification to existing transaction systems.

3.6 Middleware server checkpointing

To reduce crash induced outages, a middleware server periodically takes a checkpoint, which mainly contains recovered state numbers of middleware servers in the service domain, the LSN of each session's most recent checkpoint, and the LSN of each shared variable's most recent checkpoint. Taking a middleware server checkpoint does not block ongoing session activities. This is a **fuzzy checkpoint** [18] and has little impact on server performance.

The *minimum LSN* of all sessions' and all shared variables' most recent checkpoints will be the start point of the log scan during crash recovery. Similar to ARIES [29], after a middleware server checkpoint is taken, its LSN is recorded in the **log anchor**, a log block located at a specific location inside the physical log, such as the log header. After a crash, recovery will look for the most recent middleware server checkpoint's LSN inside the log anchor. Figure 8 illustrates a middleware server checkpoint and its relationship with others.

If a session remains inactive for a long period, no new checkpoint will be taken for this session, causing the minimal LSN to become very old. To advance the start point and shorten the log scan, we force a checkpoint for a session if the number of middleware server checkpoints taken since the previous session checkpoint reaches a threshold. This simply writes the checkpoint record again later in the log. Checkpoints for shared variables are similarly forced. Note that a shared variable checkpoint may be in a committing log record of a committed transaction.

4 Recovery

After a middleware server crashes, crash recovery is performed using the physical log. At the end of recovery, the middleware server broadcasts within the service domain its

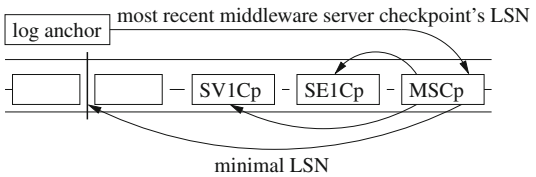

Fig. 8 *SE1Cp*, *SV1Cp* and *MSCp* are a checkpoint of session *SE1*, shared variable *SV1* and middleware server *MS*, respectively

recovered state number from the previous epoch. Based on this, sessions or shared variables of another normally executing middleware server within the service domain may find they have become orphans. Orphan recovery of these sessions or shared variables ensures *global state consistency*. Thus there are three types of recovery: session orphan recovery, shared variable orphan recovery, and middleware server crash recovery. A middleware server crash recovery will include session recovery and shared variable recovery. Session recovery, either session orphan recovery or session recovery during middleware server crash recovery, is done by replaying logged recoverable requests and transactional requests. Transactional request replay ensures *transactional consistency*.

4.1 Session orphan recovery

During normal execution, when a middleware server receives a recovered state number, the DV of any *idle* (i.e. not currently executing a method) session is checked to see whether the session has become an orphan. For a non-idle session, the session's DV is checked whenever the recovery infrastructure can intercept method execution. Interception can occur at the following occasions:

- at a service method start;
- when a service method accesses (reads or writes) a session or shared variable;
- when a recoverable method sends or receives a message (request or reply);
- before a transactional method votes;
- during the distributed log flush either when sending a request or a reply across service domains or when a transactional method votes to commit.

If the session is found to be an orphan while a transactional method is executing, transaction undo is initiated. First, a result-status log record indicating orphan-initiated abort is written (not immediately flushed). Then, the session's DV is restored to the DV saved when the request was received. If the restored DV still indicate the session is an orphan, session orphan recovery is initiated.

In other cases when the session is found to be an orphan, session orphan recovery is immediately initiated. Thus, when a middleware server is executing, some sessions may be in normal execution while others may be recovering.

To recover a session to the most recent non-orphan state, the session is initialized from its most recent checkpoint. Then, the session performs *redo* recovery by replaying the logged requests indicated by its position stream. During recovery, the session's state number and DV, the next expected request sequence number and every outgoing session's next available request sequence number are updated in the same way as they were during normal execution.

4.1.1 Recoverable method recovery

Recovery for recoverable methods relies on the logging of non-deterministic events together with re-execution of the methods. The logging of events, i.e. messages and shared variable accesses, captures all non-determinism and makes replay of recoverable methods deterministic and hence repeatable.

Replay of a logged recoverable method is done in the following way:

- Accessing a *session* variable is done as in normal execution.
- Reading a *shared* variable gets its value from the log.
- Writing a *shared* variable is skipped due to the variable's own separate recovery.
- A request to another middleware server is not sent. Rather, its reply is read from the log.

4.1.2 Transactional method recovery

We capture the transactional method execution in the same way that we capture events. That is, we log the execution effect of a transactional method and feed this effect to the recovery process in the same way that we feed logged events to the recovery process for recoverable methods. Thus, during session recovery, replay of a logged transactional request does not re-execute the transactional method. Rather, it reads the results of the transactional method from the log and installs them as appropriate. This makes the transactional request *idempotent*. The transactional method's execution effect is as if the transactional method was executed (committed) or avoided (aborted), as determined by the transaction result status. This is the essence of the "results logging" recovery process.

With results logging, we do not log the transactional methods' interactions with back-end transaction systems because we need never recreate the transactional method control state. If the transaction finishes and commits, we only need its effect, which is logged. If it finishes and aborts, we similarly only need its effect, which does not change any variable or persistent data. If a crash during the transactional method execution causes loss of the result-status log record and if we are unable to derive the transaction outcome, we abort the transaction, again avoiding any need to recover the control state.

We next describe the specific log states and how recovery responds to them in order to produce this effect.

Committed Transactions: If the result-status log record for this transactional request exists and the result status is "committed", its committing log record must also exist. Values in the committing log record are used to replay the execution effect: the logged transactional reply is returned, and the logged values for session variables written by this method are used to replace their current values. Shared variables written by this method are ignored by the replay, because shared variables are recovered separately from session recovery.

In case of a committed distributed transaction, its transaction identifier is saved so that after the next session checkpoint is taken, the session can tell the transaction manager to discard the status of all committed transactions which were initiated by the session since the previous session checkpoint.

Aborted Transactions: If the result-status log record for this transactional request exists, but the result status is "aborted", this transactional method's execution effect is to leave the state unchanged. In this case, there may or may not be a committing log record. If the result-status log record indicates that the abort is a "transaction votes abort" or a "transaction manager abort", an "aborted" reply, which may include a specific aborting reason, is returned. If the abort is due to a failed distributed log flush or due to the session being detected as an orphan before the vote, no reply is returned. Nothing else needs to be done, and session recovery continues at the session's next logged request.

Result-Status Availability: During normal execution, a result-status log record is *always* written right after a transactional method is executed. However, a crash may occur before this log record is flushed to disk. Thus, the subsequent session recovery that replays a logged transactional request may not find its result-status log record. Once the result status is resolved, a result-status log record for the request will be written at the session recovery end.

Session orphan recovery may be initiated during normal execution and re-initiated during an ongoing session recovery. With result-status log records being written during normal execution *and* during session recovery after a crash, we guarantee that when session orphan recovery is replaying a logged transactional request, it can *always* find the corresponding result-status log record.

4.1.3 Session orphan recovery end

Since session orphan recovery is initiated after the session has been found an orphan, this recovery following the session's log records will encounter an **orphan log record**, that is, a log record containing a DV, which indicates an orphan. This orphan log record may be a log record for a request or a reply from the same service domain, or a log record for reading a shared variable.

There may exist multiple orphan log records. When the session encounters the *first* orphan log record during recovery, it shows that the session became an orphan because of the action associated with this orphan log record: receiving this request or this reply, or because of reading this shared variable. At this point, the session skips this orphan log record and all subsequent log records of the session and switches to normal execution, hence terminating replay and eliminating the orphan state.

The committing log record for a local transaction also contains a DV, which may indicate an orphan. However, session orphan recovery does not handle such a committing log record as an orphan log record, i.e. it does not skip this log record and all subsequent log records of the session, but replays this transactional request as aborted and recovery continues with the subsequent logged request.

Before switching to normal execution, the session *truncates* its position stream to remove the positions of all those skipped log records. After switching, the session continues the action occurring at the recovery end, i.e. waiting for a new request or a reply, or reading the shared variable.

Those skipped log records are left in the physical log (shared by all sessions). However, their positions are removed from the session's position stream. Since session recovery follows the session's position stream, even if the session becomes an orphan again due to another crash of other middleware servers, those log records will be invisible to (skipped without being read during) the subsequent session orphan recovery.

If this middleware server crashes, the session's position stream gets lost and has to be reconstructed from the physical log during crash recovery. To ensure that those skipped log records can be identified after a crash, at the end of session orphan recovery, in addition to truncating its position stream, the session writes an **end-of-skip** (or **EOS**) log record, which contains the LSN of the orphan log record just found. In other words, the EOS log record *points* back to the orphan log record.

This EOS log record does not need to be flushed to disk immediately. If it gets to disk before the crash, the session's log records beginning with this orphan log record until this EOS log record can be identified and will be skipped by the session's recovery. However, this session's log records after the EOS log record will still need to be read after the crash. In case the EOS log record does not get to disk before the crash, *all* the session's log records beginning with this orphan log record and thereafter will be skipped.

4.1.4 Session orphan recovery upon multiple crashes

During session orphan recovery, the session's DV is also checked in case that the session has become an orphan again due to another middleware server crash within the service

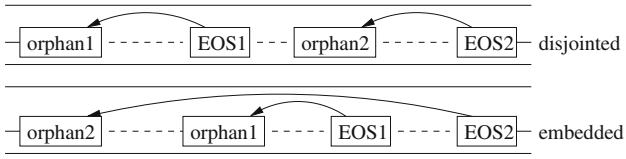

Fig. 9 Combinations of (orphan, EOS) pairs upon multiple crashes

domain. Session orphan recovery can be initiated during an ongoing session orphan recovery. This permits us to deal with *multiple concurrent* crashes promptly. However, no matter how many concurrent crashes there are, one crash can cause one session to initiate orphan recovery at most once. Figure 9 illustrates the only two possible combinations of (orphan log record, EOS log record) pairs for a session upon multiple crashes. Log records between *orphanx* and *EOSx* will be skipped during any subsequent session recovery.

Specially, the *embedded* combination could occur in the subsequent scenario: the session conducts orphan recovery with *EOS1* written, then before a session checkpoint is taken, it becomes an orphan again due to another middleware server crash, and finally the subsequent orphan recovery finds the orphan log record *orphan2* and writes the corresponding *EOS2*. Any future orphan recovery of this session will skip all its log records between *orphan2* and *EOS2*, which include those between *orphan1* and *EOS1*.

4.2 Shared variable orphan recovery

During normal execution, before a session reads a shared variable, the session checks the variable's DV to see if this variable has become an orphan. When a shared variable is to be checkpointed, during the distributed log flush following this variable's DV, this DV will also be checked. So read and checkpointing during normal execution are the *only* two occasions to trigger an orphan check for a shared variable. Once a variable is detected as an orphan, orphan recovery of this variable will be initiated. Only after this variable's most recent non-orphan value is recovered, will this value be returned to the reader session.

To do orphan recovery for a shared variable, the reader session or the checkpointing thread will follow the backward chain of write log records of this variable and roll this variable back to the most recent non-orphan value (based on the middleware server's current knowledge of other middleware servers' recovered state numbers). So, shared variable orphan recovery can be considered as *undo* recovery. Due to such separate recovery of shared variables, session *redo* recovery does not need to recover shared variables. This simplifies session recovery.

Table 4 Actions during middleware server crash recovery

re-initialize from most recent middleware server checkpoint;
scan persistent log:
a. reconstruct position streams;
b. roll forward shared variables;
c. update knowledge about recovered state numbers;
broadcast a recovery message;
resolve crash-interrupted transactions;
make a middleware server checkpoint;
recover sessions in parallel and accept new sessions;

4.3 Middleware server crash recovery

After a middleware server crashes, crash recovery is started following the procedure in Table 4.

First, the middleware server is re-initialized from its most recent checkpoint, whose LSN is in the log anchor. Then, a single-threaded analysis scan of the physical log is started at the *minimum LSN* as recorded in this checkpoint, to reconstruct position streams for all sessions and to update all shared variables to their most recent logged values. When log records containing recovered state numbers of other middleware servers in the service domain are encountered, the scan will update this middleware server's knowledge about those recovered state numbers.

When the scan is finished, the largest persistent LSN before the crash has been determined and it is broadcast in a *recovery message* within the service domain as the middleware server's recovered state number of its previous epoch.

Next, for all transactions that were interrupted by the crash, their result status is resolved and after resolution, the middleware server makes a checkpoint. Finally, all sessions start to recover *in parallel* following their reconstructed position streams, and the middleware server can start accepting new sessions.

Now, the middleware server crash recovery can be considered as finished. From this point on, recovering sessions and new sessions in normal execution may coexist. New requests of a recovering session are held until recovery of this session finishes.

4.3.1 Shared variable roll forward

During the scan, checkpoints and write log records of shared variables, and committing log records are used to roll forward shared variables.

When a checkpoint of a shared variable is encountered, the variable is updated with the checkpointed value. This value will never be an orphan. For any shared variable that appears in a committing log record, its *current* value is saved and the variable is updated with the value from this log record.

Table 5 Crash-interrupted transaction resolution

no committing log record	Aborted
distributed in-doubt	query transaction manager with logged transaction ID
local in-doubt	1 re-issue distribute log flush following logged DV; 2 *committed* if flush succeeds and *aborted* otherwise

If the result-status log record for this transactional request encountered later in the scan indicates that the transaction was aborted, this variable is reset to the saved value. Otherwise, if the transaction was committed, this committing log record serves as a checkpoint for this variable and the saved value is discarded.

When a write log record of a shared variable is encountered, both the variable's value and DV are updated with the logged value and DV. The logged DV cannot indicate that the logged value is an orphan. This is because this logged DV was indeed the writer session's DV right before the write during normal execution. This DV was checked then to see whether the session had become an orphan and it did not indicate an orphan. So the logged value was not an orphan then. After a crash, the middleware server scans its physical log to recover. During this scan, at the point when this write log record is encountered, the middleware server has no more knowledge about recovered state numbers than it had when the write occurred during normal execution. So at this point, this logged DV cannot indicate an orphan.

At the scan end, each shared variable has been updated to the most recent logged value. Since all log records containing recovered state numbers have been encountered, the middleware server has built up all its knowledge about recovered state numbers from its physical log. If a shared variable's most recent value was obtained from a write log record, this value may be an orphan according to the middleware server's current (more recent) knowledge about recovered state numbers. However, orphan recovery for this variable is not initiated immediately, but later when a session in normal execution tries to read this variable or when this variable is to be checkpointed.

4.3.2 Crash-interrupted transaction resolution

At the log analysis scan end, there may exist a transactional request log record for which the result-status log record has not been found in the log. This occurs when the crash occurred during transaction execution. Table 5 lists the result-status resolution for crash-interrupted transactions.

Transactions without Committing Log Records: If the committing log record for a transactional request does not exist, the associated transaction has not been committed. Hence, whether the transaction is local or distributed, we resolve its result status as "aborted" and the transactional method execution has no effect. For a distributed transaction, where the transactional method interacts with transaction systems, the transactional method must have not sent its vote ("commit" or "abort") to the transaction manager before the crash. The transaction manager aborts the transaction due to the transaction timeout (either already or in the near future). Transaction system participants in the aborted transaction guarantee that modifications to their own resources are undone eventually.

In-Doubt Transactions: If the committing log record for a transactional request does exist, we do not know whether the associated transaction has been committed or aborted. We call such a transaction an **in-doubt** transaction.

Note that the committing log record contains information about which shared variables were written by the transactional method. Before the result status is resolved, new sessions (which may be started after the scan end) cannot access these variables. We choose to resolve any in-doubt transaction at the log analysis scan end, rather than at a later stage (e.g. at the session recovery end). This makes these variables immediately accessible to new sessions, and also simplifies the subsequent session recovery, which thus does not need to handle (e.g. lock) these variables. How to resolve the result status of an in-doubt transaction depends on whether the transaction is distributed or local.

Distributed In-Doubt Transactions: In case of a distributed in-doubt transaction, whose transaction identifier is included in the committing log record, the transaction manager is queried for this transaction's outcome.

If the transaction manager knows the outcome, which is "committed", it replies with a result status "committed". Otherwise, the transaction manager replies with a result status "aborted". It is possible that the required distributed log flush succeeded for this transaction during normal execution, but the middleware server crashed before the vote "commit" was sent to the transaction manager. In this case, the transaction manager has no result status for this transaction, and will reply with a result status "aborted" as required for the "presumed abort" protocol.

Queries for the outcome of all distributed in-doubt transactions are combined so that the transaction manager is contacted just once, thus reducing the message overhead.

Local In-Doubt Transactions: In case of a local in-doubt transaction, we do not know whether the distributed log flush issued before the end of the transaction succeeded or not. If it succeeded, the transaction should be committed; otherwise,

the transaction should be aborted. Note that we could force to abort the transaction no matter the distributed log flush succeeded or not, since the client of this transactional request is not aware of the result status yet. But we want to avoid unnecessary aborts.

The logged DV in the committing log record is utilized to determine the transaction outcome. We *re-issue* a distributed log flush following this DV. If the flush succeeds, the transaction is resolved as "committed", otherwise, as "aborted". The distributed log flushes for all local in-doubt transactions at this middleware server are combined (in a batch) so that any other middleware server in the service domain is contacted only once. This reduces the overhead of both messages and flushes.

When a middleware server is doing a distributed log flush at the end of the log analysis scan, although it has not started to accept new sessions or to process new requests of existing recovering sessions yet, it *must* be ready to process requests (sent from other middleware servers in the service domain) of flushing its log. Since all log records of the recovering middleware server are already persistent (only those log records survived the crash), this middleware server will simply return its largest persistent LSN in response to distributed flush requests from other middleware servers.

Figure 10 illustrates this *necessity*. The two middleware servers, *MS1* and *MS2*, are in the same service domain. Both have crashed and have come to the log analysis scan end. *MS1* has a persistent log up to LSN *20*. To resolve the result status of a local in-doubt transaction, it has issued a distributed log flush following the DV *[p1:15,p2:30]*. *MS2* has a persistent log up to LSN *40*. To resolve the result status of a local in-doubt transaction, it has issued a distributed log flush following the DV *[p1:21,p2:35]*.

The distributed log flush issued by *MS1* results in a flush request (up to LSN 30) sent to *MS2*. The distributed log flush issued by *MS2* results in a flush request (up to LSN 21) sent to *MS1*. So *MS1* is waiting for the reply of its flush request to *MS2*, and *MS2* is waiting for the reply of its flush request to *MS1*. If *MS1* and *MS2* were unable to process flush requests from each other when they themselves are doing a distributed log flush, a deadlock would form between *MS1* and *MS2*, and both *MS1* and *MS2* would hang.

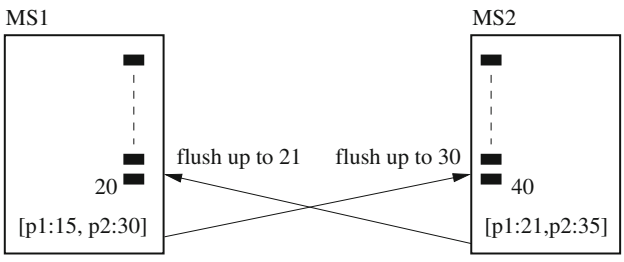

Fig. 10 Two middleware servers send flush requests to each other at the scan end. A *solid box* indicates a log record

Here, *MS1* returns *20* for the flush request (up to *21*) from *MS2*. The distributed log flush issued by *MS2* thus fails. *MS2* returns *40* for the flush request (up to *30*) from *MS1*. The distributed log flush issued by *MS1* thus succeeds.

Discussion: Resolution for in-doubt transactions may require other middleware servers in the same service domain and the transaction manager to be online. If there are multiple concurrent crashes, recovery blockage among middleware servers may occur. If the transaction manager is offline, crash recovery of a middleware server may be blocked until the transaction manager is brought online. Due to rareness of crashes and reliability of transaction managers, such blockage is usually short and not noticeable.

4.3.3 Session recovery after scan

Session recovery after the scan is similar to session orphan recovery. Each session recovers by replaying its logged requests. Replaying a recoverable request re-executes the recoverable method following the log. With results logging for a transactional method, instead of re-executing the method, the method's execution effect is supplied from the log. Besides, during recovery, the session's DV is also checked to see whether the session has become an orphan due to another middleware server crash in the same service domain.

The main difference lies in the recovery end condition. Unlike session orphan recovery, session recovery after the scan may not encounter an orphan log record. In case that a recovering session does not encounter an orphan log record after all its log records are consumed, the session simply switches to normal execution immediately, unless the session's last persistent log record is a transactional request log record or a committing log record, in which case the result status of the transaction has already been resolved at the scan end and the session will write a result-status log record, before switching to normal execution.

In case an orphan log record *O* is encountered, similar to session orphan recovery, the session skips those log records beginning with *O* until the end-of-skip log record *EOS* which points back to *O*, or until the session's last persistent log record, if such an *EOS* does not exist.

EOS Found: In case such an *EOS* is found, the session removes from its position stream the positions of all skipped log records beginning with *O* and ending with *EOS*. If there are no more log records after *EOS*, the session switches to normal execution; otherwise, the session continues to recover following those subsequent log records.

EOS Not Found: In case such an *EOS* does not exist, the session writes an EOS log record pointing back to *O*, truncates its position stream to remove positions of skipped log records beginning with *O*, and switches to normal execution.

5 Performance discussion and measurement

We implemented a prototype for recovery of Web services with transaction support using the Microsoft .NET middleware infrastructure. The prototype demonstrates the feasibility of our logging and recovery methods for middleware servers.

5.1 Prior performance work

In our prior work [41], we measured and analyzed locally optimistic logging and pessimistic logging in terms of normal execution overhead and recovery speed. We present an overview of that here, prior to describing our new experiments that include measuring performance when transaction support is also included.

With Microsoft .NET, session state can be stored at a standby in-memory state server or in a DBMS. Our prior measurements confirmed that making session state persistent at the middle tier, either by pessimistic logging or locally optimistic logging, incurs less overhead than storing it into a DBMS, but more overhead than storing it into a standby state server. It is important to note, however, that this later method does not provide state persistence. When the program implementing business logic does not access the disk, the logging cost for recovery is significant, compared to not providing recovery and hence avoiding logging. However, multi-tier applications usually access the disk, either to use files or databases. In this setting, the relative overhead of making shared state or session state persistent is modest.

With locally optimistic logging, the larger the fraction of messages confined within a service domain compared to messages crossing service domains, the lower the logging overhead compared to purely pessimistic logging. Configuration of service domains is a tradeoff between logging cost and failure isolation, which business can use to create the appropriate balance.

Recovery performance is affected by both the crash frequency and the checkpoint frequency. Our measurements showed that replaying a logged request is faster than processing a request during normal execution, since a portion of the execution effect can be obtained from log, e.g. getting a reply. When crashes are rare, which is the usual case, and especially for enterprise servers, locally optimistic logging produces higher throughput than pessimistic logging by reducing log flushes. However, when a crash occurs, recovery with locally optimistic logging takes longer, since orphan rollback may be required. The more frequent checkpointing is, the faster recovery becomes. Hence, the checkpoint frequency can be used to reduce the recovery time to a specified range, while incurring some modest normal execution overhead.

Replicating mid-tier server state during normal execution can reduce or eliminate outage time, since recovery and even failover are not required after a crash. However, this both doubles the resources needed for middle tier applications and does not preserve state should overlapping failures of the replicas occur, e.g. as in a power failure.

5.2 Performance impact of transaction support

The prior work did not measure the impact of transaction support. Here, we focus on evaluating the impact of "results logging", our technique for recovering transactional methods, when either locally optimistic logging or pessimistic logging is used. Our new experiments show that although transactions incur additional forced logging, the locally optimistic logging still preserves its performance advantage.

Figure 11 shows our experimental configuration. We have one or more end clients, two middleware servers (*MS1* and *MS2*) hosted by Web servers, and one DBMS. An *end client* starts a session *SE1* with *MS1*, then sends *request1* to execute *RecoverableMethod1* a number of times. *SE1* at *MS1* further starts a session *SE2* with *MS2*. *RecoverableMethod1* reads and writes shared variable *SV1*, sends *request2* to execute *RecoverableMethod2*, then reads and writes *SV2*, next sends *request3* to execute *ServiceMethod3* and reads and writes *SV3*, finally writes *SE1*'s session variables. Over *SE2*, *RecoverableMethod2* reads and writes *SV4*, and writes *SE2*'s session variables. *ServiceMethod3* reads and writes *SV5*, and writes *SE2*'s session variables. *ServiceMethod3* may be a recoverable method, or a transactional method which may access the back-end database.

Both the parameter and the returned value of a request are 100 bytes. The session state size for each of *SE1* and *SE2* is 8K bytes. All service methods write 512 bytes of their session state. Each shared variable is 128 bytes. Such a configuration is consistent with the observation that the shared in-memory state is relatively small. A database access includes a read of 3K bytes and a write of 150 bytes of persistent business data.

Our hardware consists of four computers: one client, two Web servers and one DBMS server, connected via Ethernet. Table 6 lists the hardware parameters. Each computer has one disk. The disk on *MS1* is used for *MS1*'s log and the operating system virtual memory. The disk on *MS2* is used for *MS2*'s log, the operating system virtual memory and the distributed transaction manager's log. Since the servers have large memories, the operating system virtual memory did not incur disk I/Os during our experiments.

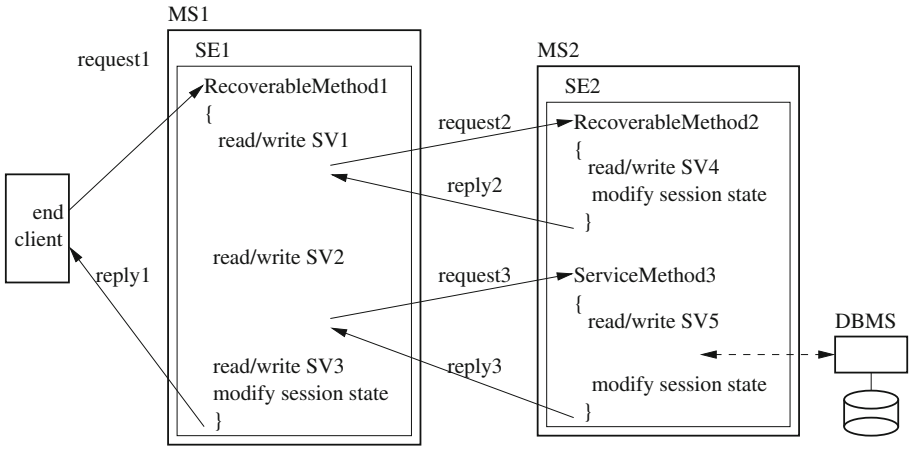

Fig. 11 Experimental configuration

Table 6 Hardware parameters

network bandwidth	100 Mbps
client	CPU 2.33 GHz, memory 512 MB
Web servers	CPU 2.66 GHz, memory 1 GB
DBMS server	dual core 2.16 GHz, memory 2 GB
server hard disks	7200 RPM
	avg seek 10.5 ms(W)/9.5 ms(R)
	track/track seek 1.2 ms(W)/1 ms(R)
	default sectors per track 63

We first measured and analyzed the response times of transactional methods, when using results logging, with a single end client for both locally optimistic logging and pessimistic logging, respectively. We then measured and analyzed the response time and throughput with multiple concurrent end clients. In both cases, the transactional method results logging preserves the performance advantage of locally optimistic logging, with the appropriate configuration of service domains, when compared with pessimistic logging.

5.3 Performance with single client

We measured the response time of requests from a single end client for the experimental configuration shown in Fig. 11. We explored six different scenarios. We compared locally optimistic logging (labelled *LoOptimistic*), where *MS1* and *MS2* are configured into the same service domain, to pessimistic logging (labeled *Pessimistic*), where *MS1* and *MS2* are configured into different service domains, and thus pure pessimistic logging is used. We tested these logging strategies using three alternatives for *ServiceMethod3*: (1) a recoverable method (labeled *Rec*); (2) a transactional method that does not access the DBMS (labeled *TraLoc*); and (3) a transactional method that does access the DBMS (labeled *TraDis*).

Table 7 Response time (ms) with a single end client

	Rec	*TraLoc*	*TraDis*
Pessimistic	62.84	62.75	70.85
LoOptimistic	29.82	41.59	48.73

Table 7 displays the average response times over 20,000 end client requests.

5.3.1 Recoverable methods case

For case *Rec*, messages flow between *MS1* and *MS2*, which are both recoverable methods. With pessimistic logging, each end client request requires five log flushes in sequence: (1) at *MS1* before sending *request2*; (2) at *MS2* before sending *reply2*; (3) at *MS1* before sending *request3*; (4) at *MS2* before sending *reply3*; and (5) at *MS1* before sending *reply1*. Each log flush writes 2 sectors to disk. With locally optimistic logging, each end client request requires only one distributed log flush at *MS1* before sending *reply1*. This distributed flush incurs two log flushes *in parallel*: one 4-sector flush at both *MS1* and *MS2*.

The log is written in units of log blocks, aligned at sector boundaries. When a log block is flushed, its last sector may not be full. More forced logging hence results in more partially filled sectors. This accounts for our observation that locally optimistic logging saves log space due to less frequent log flushes.

The following simplified analysis explains the response time difference between locally optimistic logging and pessimistic logging. We assume a constant time for each log flush of T_F because seek time is much larger than data transfer time determined by log write size, and hence is the dominant factor in response time. The response time difference between pessimistic logging and locally optimistic logging is thus approximately equal to $5 \cdot T_F - max(T_M + T_F, T_F))$, where T_M is the time of the message round trip to request

MS2 to flush during the distributed log flush. The difference is equal to $4 \cdot T_F - T_M$.

The average disk seek is about 10.5 ms. Considering that disk access for log flushes is not completely random, but more sequential, we crudely estimate T_F to be 9 ms for our analysis. The message round trip time T_M was measured as 4 ms. The response time difference is thus calculated as 32 ms ($= 4*9-4$), which accounts for most of the measured response time difference 33.02 ms ($= 62.84 - 29.82$).

5.3.2 Local transactional case

For case *TraLoc*, *ServiceMethod3* is a transactional method and uses results logging. With pessimistic logging, each end client request requires five log flushes in sequence, the same as for case *Rec*. The difference is that one more disk sector is written by the fourth flush at *MS2*, because slightly more data needs to be logged by results logging for a transactional method.

When locally optimistic logging is in use, each end client request requires two distributed log flushes. The first distributed flush occurs at *MS2* to commit the transaction. This first flush incurs two log flushes in parallel, i.e. one 5-sector flush at *MS2* and one 3-sector flush at *MS1*. The second distributed flush occurs at *MS1* before sending *reply1*. This second flush incurs two log flushes in parallel, i.e. one 0-sector flush at *MS2* and one 2-sector flush at *MS1*. Here 0-sector flush indicates that there is no log to be flushed, but a message round is still required in order to discover this.

The time difference in log flushes between pessimistic logging and locally optimistic logging is thus equal to $5 \cdot T_F - \max(T_M + T_F, T_F) - \max(T_F, T_M)$, i.e. $3 \cdot T_F - T_M$, which, using the same estimates as before, is calculated as 23 ms, very close to the measured response time difference 21.26 ms (62.75 − 41.59). The measured response time difference is smaller, partly because when locally optimistic logging is adopted transactional method processing has extra overhead on execution context.

The response time difference, i.e. saving by locally optimistic logging compared to pessimistic logging, in case of *TraLoc* (21.26 ms) is smaller than case *Rec* (33.02 ms), exactly because of the extra distributed log flush required to commit the transaction. This extra log flush would be needed for transaction commit as long as a logging-based approach is adopted.

5.3.3 Distributed transactional case

Compared to case *TraLoc*, in case of *TraDis*, the response time is 7 or 8 ms longer, for both pessimistic logging and locally optimistic logging. This increase in response time occurs as a result of the DBMS access and is very close to the time of one disk log flush. Each DBMS access results in one database transaction, which requires one DBMS log flush at transaction commit.

5.4 Performance with multiple clients

To observe the performance of transactional method handling with multiple concurrent end clients, we measured the response time and throughput for case *TraLoc* when there are multiple clients, each of which continuously submits requests. In this set of experiments, we enabled batch flushing, i.e. a request to flush the log is not executed immediately, but rather after a specified timeout, providing a chance to batch several flush requests into a single write. Batch flushing is a log write optimization similar to group commit [18], adopted commonly by commercial DBMSs. With batch flushing, recovery may require more rollbacks, both for crash recovery and for orphan recovery. However, given the low incidence of crashes, batch flushing is a good strategy. It reduces the amortized logging overhead and improves the throughput. We chose 8 ms as the batch flushing timeout, which is close to the time of a disk log write.

Figures 12 and 13 show the response time and the throughput change as the number of concurrent clients increases. With increasing clients, the response time becomes longer. However, locally optimistic logging always has a shorter response time than pessimistic logging.

With increasing end clients, the throughput increases, because the system resources including CPU, disk I/O and network bandwidth become more heavily utilized. When the number of clients reaches 4, the throughput reaches a maximum for locally optimistic logging, while the throughput reaches its maximum for pessimistic logging when the number of clients reaches 5. Further increase in concurrent end clients leads to lower throughput.

These experimental results should not be interpreted as the system being limited to supporting only a few clients.

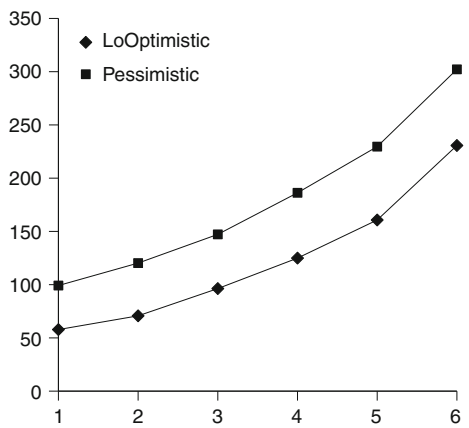

Fig. 12 Response time (ms) versus number of end clients

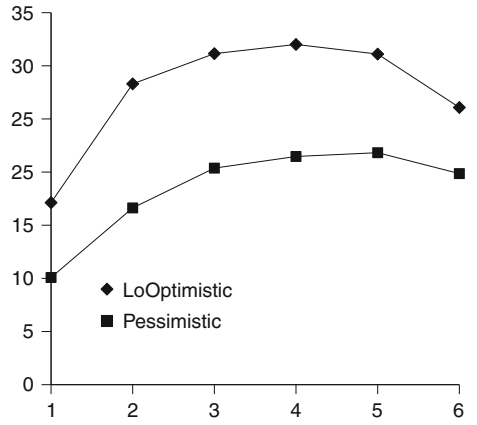

Fig. 13 Throughput (requests per second) versus number of end clients

With a real workload, end clients do not continuously submit requests, but rather are idle most of the time, which permits many more clients to be supported simultaneously.

With locally optimistic logging, the distributed log flush issued at "transaction votes commit" takes $T_M + T_F$ $(= max(\ T_M + T_F, T_F))$, while with pessimistic logging, only a local log flush T_F is required at "transaction votes commit". Since a transactional method (using strict two phase locking) holds locks on shared variables until end of transaction, the lock duration with locally optimistic logging is longer than with pessimistic logging, reducing concurrency. In spite of this negative impact, the performance results reported in Fig. 13 reveal that the highest throughput supported by locally optimistic logging is still 47% higher than the highest throughput supported by pessimistic logging.

Determining the timeout value for batch flushing should take into account the specific system configuration and the workload, e.g. the number of concurrent clients. The timeout (8 ms) used in our experiments might not lead to the highest throughput supportable by our recovery system.

6 Related work

6.1 Commercial providers

Commercial Web servers [6,19,28] make session states resilient to failures by replicating session states to a standby process or Web server, or saving session states to disk files or databases synchronously. They do not support fault tolerance for shared in-memory state.

Certain Web services infrastructures [14] 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. Recovery of in-memory state is not provided, nor is coordinated recovery of in-memory state with persistent state.

6.2 Research efforts

6.2.1 Persistence approach

Replication: Fault tolerance of middleware servers incorporating transaction processing could be implemented via replication [23,31,32,35,36,44,45], which requires duplicate resources and additional infrastructure support, such as totally ordered multi-cast communication infrastructure.

In case of replication with transaction support [23,36], both middleware components and back-end transaction systems had replicas, changes to back-end transaction systems were captured and replicated to backup middleware components and applied to redundant transaction systems.

Logging: Log-based techniques are less expensive yet effective, though typically with longer outages. Log-based recovery for general application processes over a Java virtual machine was partially implemented [30], but no consistency among interacting processes was considered. Pessimistic logging and optimistic logging were invented in the fault tolerance community [13] and were used for consistent recovery of message passing systems, where entities interacted with one another via message exchanges only. Individual threads were considered as separate recovery units with optimistic logging [10]; however, log management of a multi-threaded process with optimistic logging was not explored.

The Phoenix project [1,3,4,25] exploited message logging to enable recovery, including interactions with back-end transaction systems. Middle tier recovery [3] involved message logging as well.

E-transactions [17] with exactly once execution semantics in three-tier systems were implemented [16] by storing transaction results, including replies and side-effects on middleware server state, into backup servers or back-end databases. This might require modification to DBMS committing processing and overload DBMS servers. This is an implicit (degenerate) case of logging where the need for a log was removed from the middle tier by restricting functionality. Logless components [26] generalized this, explicitly revealing the value of client logging in supporting a richer set of capabilities while retaining the absence of logging in the middle tier.

None of the above approaches directly supported concurrent accesses to shared state by multiple threads.

6.2.2 In-memory state

In-memory state is usually located inside the same process as the middleware server.

Private State without Concurrent Access: Private in-memory state for single-threaded "sessions" is not subject to race conditions on access, and can be recovered via message logging alone, or the usual replication techniques.

Transactional access to in-memory state within a single-threaded process was implemented [22,43]. If the business logic could be captured in a chain of transactions [22], each of which accessed the in-memory state, the state simply consisted of the finishing status of those transactions. From the log of those transactions, the state could be easily recovered. More complicated business logic was supported [43] with the in-memory state access being implemented as part of the two phase commit protocol. In this case, a distributed transaction coordinator was needed.

Shared State with Concurrent Access: When the state is accessible by multiple threads, one needs to use concurrency control to protect such state, e.g. by using locks to control access. Access to in-memory state needs to be logged for recovery. Lock acquisition can be used to trigger appropriate logging for logging-based approaches.

Transactional access to shared in-memory state concurrently by multiple threads was implemented with all the in-memory objects of an object-oriented program being shared and implicitly protected by read or write locks [38]. Since the number of objects and threads might be large, it was critical to implement the lock manager efficiently [11,12].

6.2.3 Resource manager interactions

Persistent application state stored in disk files, database systems and transactional message queues is usually managed by a separate process, called a transactional resource manager. There was an exception [43] of having no separate manager process for disk files, where transactional access to disk files was supported only for a single-threaded process and the disk files could be accessed only by this single process.

When the transactional resource manager is a separate process, messages are exchanged for accessing transactional persistent state. Usually, before any request message is sent to another process, the client process needs to flush its log records to ensure deterministic replay. However, since a transaction can be canceled later, flushing is required only when the committing request message is sent [3].

Messages from a transactional resource manager back to a client process, such as query results from a DBMS, were logged by the client process [3,4,15,25]. On the other hand, intermediate results from a transactional resource manager within a transaction might not be logged by the client [39]. Instead, only the transaction outcome (either committed or aborted) was logged, thus resulting in less logging.

To interact with back-end transactional resource managers, Phoenix/ODBC used ODBC (Open Database Connectivity). For recovery of ODBC sessions [4], this sometimes 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, storing complete results prior to delivering them to a client ensured that all transaction results were delivered when the transaction committed but the server crashed before the client had completed processing the results. Update consistency with the back end was provided via idempotence. The idempotence guarantee was realized by storing transaction result status in a persistent table at the DBMS server. Involving middle tier servers in a transaction was not explicitly treated.

6.2.4 End-to-end story

With traditional transaction processing, when a client process accessed a server such as a DBMS, the client requests could be guaranteed to be processed exactly once by the server via recoverable queues [8]. However, the client business logic was only expressible as a chain of transactions and no more complicated business logic was allowed.

Recovery for middleware components such as CORBA components [31,32], Microsoft .NET components [1], J2EE components [23,36] and PHP serverlets [2] have been studied. Some work used replication [23,31,32,36], while others used optimized pessimistic message logging [1,2] to approach exactly once execution of requests.

The Phoenix/App effort [1,5] provided an end-to-end exactly once execution framework for Web applications based on recovery. It supported a stateful application programming model, realizing persistence via recovery based on application replay. It recognized the importance of reducing the overhead of logging, developing a number of techniques. It compartmentalized recovery via "contracts" that required forcing the log, providing, in some cases, failure isolation in which parts of the system that did not fail were isolated from having to deal with the failure. The Phoenix approach to interacting with a back-end transaction system was to require that it provide idempotence. Idempotence is, indeed, a requirement for recovery [2]. The notion of a transactional component was introduced to provide idempotence for the rest of the distributed system. Such a component can be located at the back end or at the middle tier. However, there exist several ways in which idempotence might be provided naturally within an application. Phoenix did not describe a particular approach. The transactional method in our work is a form of transactional component located at the middle tier. It relies on the transaction manager for distributed commit idempotence and builds on that to provide idempotence for the rest of the middle tier.

In a Web services setting, idempotent services were advocated to enable recovery of applications via replay if they did appropriate (optimized pessimistic) message logging [24].

6.2.5 Our work

For recovery of interacting middleware servers, locally optimistic logging was introduced [41] to reduce logging overhead. For interactions with back-end transaction systems, we have introduced results logging (described briefly in [42]) where what is logged is the result of executing a transactional method (its reply and its impact on shared state and session state). Our paper here further develops results logging and recovery in the context of locally optimistic logging and measures its performance. It presents a complete persistent state solution for middleware servers interacting with transactional back ends.

7 Conclusions

We have described our system for log-based recovery of middleware servers with transaction support. By using locally optimistic logging for message exchanges, we reduced logging overhead and maintained recovery independence across service domains. Value logging for shared variables increased recovery independence inside a middleware server. It also kept logging overhead modest in the usual case where shared variables are small. Fuzzy checkpointing incurred only minimal impact during normal system execution. We enabled parallel recovery of multiple sessions after a crash while using a common physical log for all sessions.

Using results logging for transactional methods ensured coordinated recovery of both in-memory state and persistent state. Results logging incurred modest logging overhead in maintaining transactional consistency and required little or no modification to existing transaction systems. Our performance measurements of our prototype system showed that our transactional method processing has low overhead and preserved the performance advantage of locally optimistic logging over pessimistic logging.

Our system has demonstrated the feasibility of using results logging to provide coordinated recovery for both in-memory state and transactional persistent state, and having the system infrastructure transparently provide persistence guarantees for middleware servers. Our technology thus supports the high performance and availability needed by transactional business applications, without the need for major modification to existing transactional system infrastructure. 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.

Acknowledgments This work was supported by the National Science Foundation under Grant No. IIS-0533625. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Barga, R., Chen, S., David, L.: Improving logging and recovery performance in Phoenix/App. In: Proceedings of IEEE ICDE, pp. 486–497 (2004)
2. Barga, R., Lomet, D., Shegalov, G., Weikum, G.: Recovery guarantees for internet applications. ACM Trans. Interet Technol. **4**(3), 289–328 (2004)
3. Barga, R., Lomet, D., Weikum, G.: Recovery guarantees for general multi-tier applications. In: Proceedings of IEEE ICDE, pp. 543–554 (2002)
4. Barga, R.S., Lomet, D., Baby, T., Agrawal, S.: Persistent client-server database sessions. In Proc EDBT, pp. 462–477 (2000)
5. Barga, R.S., Lomet, D.B., Paparizos, S., Yu, H., Chandrasekaran, S.: Persistent applications via automatic recovery. In *IDEAS* pp. 258–267 (2003)
6. BEA Corp.: Deploying and configuring web applications. http://edocs.bea.com/wls/docs60/ (2000)
7. Bernstein, P.A.: Vassos Hadzilacos, and Nathan Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)
8. Bernstein, P.A., Hsu, M., Mann, B.: Implementing recoverable requests using queues. In: Proceedings of ACM SIGMOD, pp. 112–122 (1990)
9. Damani, O.P., Garg, V.K.: How to recover efficiently and asynchronously when optimism fail. In: Proceedings of IEEE ICDCS, pp. 108–115 (1996)
10. Damani, O.P., Tarafdar, A., Garg, V.K.: Optimistic recovery in multi-threaded distributed systems. In: Proceedings of IEEE SRDS, pp. 234–243 (1999)
11. Daynes, L., Czajkowski, G.: High-performance, Space-efficient, automated object locking. In: Proceedings of IEEE ICDE, pp. 163–172 (2001)
12. Daynes, L., Czajkowski, G.: Lightweight flexible isolation for language-based extensible systems. In: Proceedings of VLDB, pp. 718–729 (2002)
13. Elnozahy, E.N. (Mootaz), Alvisi, L., Wang, Y.-M., Johnson, D.B.: A survey of rollback-recovery protocols in message passing systems. ACM Comput. Surv. **34**(3), 375–408 (2002)
14. Freeman, A.: Microsoft. NET XML Web Services Step by Step. Microsoft Press, Redmond (2003)
15. Freytag, J.C., Cristian, F., Kaehler, B.: Masking system crashes in database application programs. In: Proceedings of VLDB, pp. 407–416 (1987)
16. Frølund, S., Guerraoui, R.: A pragmatic implementation of e-Transactions. In: Proceedings of IEEE SRDS, pp. 186 (2000)
17. Frølund, S., Guerraoui, R.: e-Transactions: end-to-end reliability for three-tier architectures. IEEE Trans. Softw. Eng. **28**(4), 378–395 (2002)
18. Gray, J., Reuter, A.: Transaction Processing: Techniques and Concepts. Morgan Kaufmann, San Fransisco (1993)
19. Hines, B., Alcott, T., Barcia, R., Botzum, K.: IBM WebSphere Session Management. http://www.informit.com/articles/article.asp?p=332851 (2004)
20. IBM: WebSphere Application Server (Distributed platforms and Windows), Version 7.0. http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.base.iseries.doc/info/welcome-base.html (2009)

21. IBM, BEA Systems, Microsoft, Arjuna, Hitachi, and IONA: Web Services Transactions Specifications. http://www-128.ibm.com/developerworks/library/specification/ws-tx/ (2005)
22. Jeong, K., Shasha, D.: PLinda 2.0: A transactional/checkpointing approach to fault tolerant linda. In: Proceedings of IEEE SRDS, pp. 96–105 (1994)
23. Kistijantoro, A.I., Morgan, G., Shrivastava, S.K., Little, M.C.: Enhancing an application server to support available components. IEEE Trans. Softw. Eng. **34**(4), 531–545 (2008)
24. Lomet, D.: Robust Web Services via Interaction Contracts. In: VLDB Workshop on Technologies for E-Services (2004)
25. Lomet, D., Weikum, G.: Efficient transparent application recovery in client-server information system. In: Proceedings of ACM SIGMOD, pp. 460–471 (1998)
26. Lomet, D.B.: Persistent middle tier components without logging. In: IDEAS, pp. 37–46 (2005)
27. Microsoft: Web Devlopement ASP.NET. http://msdn.microsoft.com/en-us/default.aspx (2009)
28. Microsoft Corp.: ASP.NET. http://msdn.microsoft.com/ library (2000)
29. Mohan, C., Haderle, D.J., Lindsay, B.G., Pirahesh, H., Schwarz, P.M.: ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. Database Syst. **17**(1), 94–162 (1992)
30. Napper, J., Alvisi, L., Vin, H.: A fault-tolerant Java virtual machine. In: Proceedings of IEEE DSN, pp. 425–434 (2003)
31. Narasimhan, P., Moser, L.E., Melliar-smith, P.M.: Enforcing determinism for the consistent replication of multithreaded CORBA applications. In: Proceedings of IEEE SRDS, p. 263 (1999)
32. Narasimhan, P., Moser, L.E., Melliar-Smith, P.M.: State synchronization and recovery for strongly consistent replicated CORBA objects. In: Proceedings of IEEE DSN, pp 261–270 (2001)
33. Oracle.: Enterprise JavaBeans Technology. http://java.sun.com/products/ejb/ (2009)
34. Oracle.: Oracle WebLogic Server. http://www.oracle.com/us/products/middleware/application-server/index.htm, (2009)
35. Perez-Sorrosal, F., Patino-Martinez, M., Jimenez-Peris, R., Kemme, B.: Consistent and scalable cache replication for multi-tier J2EE applications. In: Middleware 2007, pp. 328–347. Springer Berlin, Heidelberg (2007)
36. Perez-Sorrosal, F., Patino-Martinez, M., Jimenez-Peris, R., Vuckovic, J.: Highly available long running transactions and activities for J2EE applications. In: Proceedings of IEEE ICDCS, p 2, (2006)
37. Ronsse, M., Bosschere, K., Mark, C., Jacques, C.K., Dieter, K.: Record/replay for nondeterministic program executions. Commun. ACM **46**(9), 62–67 (2003)
38. Rudys, A., Wallach, D.S.: Transactional rollback for language-based systems. In: Proceedings of IEEE DSN, pp. 439–448 (2002)
39. Salzberg, B., Tombroff, D.: DSDT: Durable scripts containing database transactions. In: Proceedings of IEEE ICDE, pp. 624–633 (1996)
40. Strom, R.E., Yemini, S.: Optimistic recovery in distributed systems. ACM Trans.Comput. Syst. **3**(3), 204–226 (1985)
41. Wang, R., Salzberg, B., Lomet, D.: Log-based recovery for middleware servers. In: Proceedings of ACM SIGMOD, pp. 425–436 (2007)
42. Wang, R., Salzberg, B., Lomet, D.: Transaction support for log-based middleware server recovery. In: Proceedings of IEEE ICDE, pp. 353–356 (2009)
43. Wang, Y.-M., Chung, P.Y., Huang, Y., Elnozahy, E.N.: Integrating checkpointing with transaction processing. In: Proceedings of IEEE FTCS, pp. 304–308 (1997)
44. Wu, H., Kemme, B.: Fault-tolerance for stateful application servers in the presence of advanced transactions patterns. In: Proceedings of IEEE SRDS, pp. 95–108 (2005)
45. Zhao, M.-W., Moser, L.E., Melliar-Smith, P.M.: Unification of transactions and replication in three-tier architectures based on CORBA. IEEE Trans. Dependable Secur. Comput. **2**(1), 20–33 (2005)