Log-Based Recovery for Middleware Servers

Rui Wang
Microsoft
Redmond, WA
ruiwang@microsoft.com

Betty Salzberg [*]
CCIS, Northeastern University
Boston, MA
salzberg@ccs.neu.edu

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

ABSTRACT

We have developed new methods for log-based recovery for middleware servers which involve thread pooling, private in-memory states for clients, shared in-memory state and message interactions among middleware servers. Due to the observed rareness of crashes, relatively small size of shared state and infrequency of shared state read/write accesses, we are able to reduce the overhead of message logging and shared state logging while maintaining recovery independence. Checkpointing has a very small impact on ongoing activities while still reducing recovery time. Our recovery mechanism enables client private states to be recovered in parallel after a crash. On a commercial middleware server platform, we have implemented a recovery infrastructure prototype, which demonstrates the manageability of system complexity and shows promising performance results.

Categories & Subject Descriptors: D.4.5 Reliability, D.2.4 Software/Program Verification

General Terms: Reliability, Performance

Keywords: Application Fault Tolerance, Exactly-Once Execution, Recovery, Optimistic Logging, Distributed Systems

1. INTRODUCTION

Multitier systems have been extensively adopted in electronic commerce. The front end clients submit requests to the middle tier *middleware servers*, which execute the business logic and maintain *in-memory* business state. Middleware servers may further interact with one another. As an example of middleware servers, Web services are becoming the standard for middleware services, especially for the interactions among heterogeneous software provided by several business organizations.

Mission-critical middleware servers require high availability. Ideally, server crashes are masked and clients experience at worst a slower response time rather than a service failure. Further, when crashes are masked by automatic recovery, human intervention is avoided, reducing both unavailability and the possibility of human mistakes that can lead to very long outages.

High availability can be achieved via several different techniques. Two common techniques are:

Replication: The *business state* is replicated in more than one computer nodes. When one node fails, the business state is available in other nodes. Replication requires duplicate computing resources and is a relatively expensive solution, though with the potential of eliminating outages entirely.

Log-Based Recovery: The business state and its changes are logged on disk. When a node fails and restarts, the business state is reconstructed by applying the logged changes. Log-based recovery is a relatively cheap yet effective technique of making high availability a commodity feature.

In this paper, we present our methods for log-based recovery for *Middleware Server Processes* or **MSP**s. Our implemented prototype will restore a failed MSP's in-memory business state before the crash, bring it online quickly, mask the crash from its clients and keep consistency among MSPs. MSPs may further interact with back end transactional systems, such as DBMSs. Coordinating recovery of MSPs with transactional systems' recovery, part of our prototype, is not covered in this paper.

1.1 Log-Based Recovery Basics

The purpose of log-based recovery [7] is to restore the last business state before a crash. When given the same pre-state, a deterministic re-execution (called *redo*) will end with the same post-state. However, a program may have nondeterministic events, e.g. receiving an external message. The nondeterministic events must be logged persistently. During recovery, the program re-execution is fed this logged information as each nondeterministic event is encountered. To speed up recovery, the system can take checkpoints of the business state so that re-execution can start from the most recent checkpointed state.

1.2 Fundamental Challenges

An MSP handles a large number of concurrent client-initiated service requests with a thread pool. Request and reply messages are exchanged between an MSP and its clients. Each client is identified as a *session* at the MSP. The clients can be other MSPs. An MSP maintains two types of in-memory business state: *session state* and *shared state*. A client's session state is private to the client while the shared state is shared by all (client) sessions. Upon a crash, MSP

[*]This work was supported by National Science Foundation under Grant No. 0533625.

log-based recovery replays the logged requests to recover shared and session states up to the point of the crash. This guarantees *exactly-once* execution semantics for requests.

Like all failure masking techniques, log-based recovery faces some fundamental challenges. In our case these challenges include:

Recovery Independence: In a distributed system, it is important to isolate the effects of crashes to the failed parts of the system. Recovery independence means that one MSP crash does not affect other MSPs and that one session's recovery is isolated from other sessions of the same MSP.

Logging Overhead: It is essential to minimize logging overhead for high performance. This can be difficult when dealing with distributed recovery where recovery independence is essential.

Shared State: Shared state can be accessed simultaneously by multiple sessions on behalf of concurrent clients. These accesses result in nondeterminism and need to be logged. They also result in dependency among sessions. Thus, supporting shared state impacts both logging overhead and recovery independence.

Lightweight Checkpointing: Checkpointing reduces recovery time, which is important for high availability. However, it needs to be lightweight so as to keep its impact low on normal MSP execution.

Recovery Parallelism: Server computers usually have multiple processors. We need to exploit these processors by making recovery parallel to speed up recovery and reduce service outages.

1.3 Overview of Our Solution

We introduce the concept of *service domain*, which consists of multiple tightly associated MSPs with fast and reliable network communication among them. Our log-based recovery is *layered*, with different logging methods at each of three different layers: client sessions/shared state, MSPs, and service domains. In addition, the checkpointing method is lightweight and parallel recovery is supported.

We make all session states and shared state recoverable. This can lead to more flexible system designs. Most existing recoverable systems do not support shared in-memory state, but rather require that different client sessions share data through the back end databases. However, round trip accesses to databases have a high runtime overhead. With shared in-memory state support, MSP programs can boost system performance. For example, an MSP program can now cache shared state retrieved from a database, enabling later requests to have speedy access to it [11].

The logging and recovery methods implemented in our prototype and described here guarantee exactly-once execution semantics for requests. Our *contribution* is the introduction of service domains and the integration of multiple techniques to address the fundamental challenges faced by log-based recovery for MSPs. The main features of our solution are:

Locally Optimistic Logging: Because system crashes are rare, we use *locally optimistic logging*. This entails both pessimistic logging [1, 2] involving frequent log flushes and optimistic logging [5, 17] which reduces log flushes but complicates recovery. For message exchanges between MSPs in different service domains, pessimistic logging is used, while optimistic logging is used if they are in the same domain. Locally optimistic logging balances the conflicting needs for low overhead and recovery independence. Optimistic logging reduces overhead while pessimistic logging increases recovery independence.

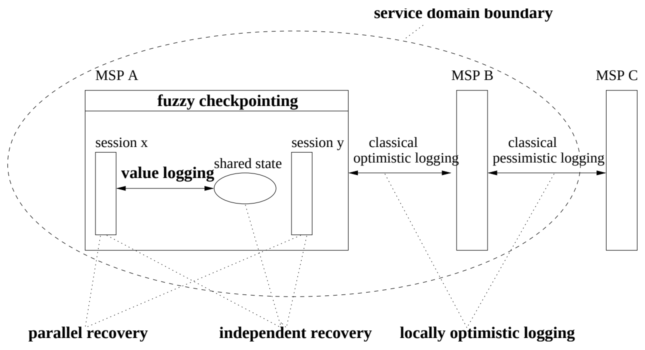


Figure 1: Contributions (terms in bold fonts).

Value Logging: Typically in a multitier system, most business state is stored persistently in transactional systems that already provide crash recovery. Some business state for each client is stored as session state, which usually does not need logging since there is no nondeterminism involved in its access. The remaining small portion of business state is stored as shared state, requiring logging to capture its nondeterministic access. This typical scenario leads us to exploit a *value logging* method (sometimes called physical logging) for both reading and writing shared state by sessions. Value logging provides recovery independence between sessions in an MSP. With value logging for *reads*, any value read from shared state can be obtained from the log, without involving other clients. With value logging for *writes*, shared state can be recovered from the log without relying on session *redo* recovery. Because of the small size and infrequent access to shared state, value logging overhead is modest.

Fuzzy Checkpointing: Sessions and shared state are checkpointed independently. An MSP is checkpointed *fuzzily* to include only positions of its session's and shared state's most recent checkpoints. This fuzzy checkpointing does not block other activities inside the MSP and has modest impact on server performance. After a crash, the log scan starts from the lowest position as recorded in the MSP checkpoint.

Parallel Recovery: We enable *parallel recovery* of session states. Activities inside an MSP are logged in a single physical log. After a crash, we first identify the log records for each session so that the session states of all clients can be recovered in parallel. This results in faster recovery than replaying all activities sequentially in log order.

Each MSP has a single physical log shared by all its sessions. This sharing lowers the amortized log flush overhead, but makes log management more challenging. Our log management is capable of accommodating our logging and checkpointing methods.

No log-based recovery work of which we are aware, either research or commercial, balances the conflicting needs of low logging overhead and high recovery independence in the effective way that we report here. In addition, we support shared in-memory state access with low logging overhead and provide lightweight checkpointing and parallel recovery to shorten recovery times. Figure 1 summarizes the contributions of this paper.

We implemented a log-based recovery infrastructure prototype using a commercial Web services platform. The recovery infrastructure is transparent to middleware programs and can deal with multiple concurrent crashes. Our implementation demonstrates that system complexity is manageable and that performance gains are real. Locally optimistic logging is the backbone of our other methods. The experimental results show that the overall performance of a recoverable middleware server is significantly improved with locally optimistic logging.

1.4 Paper Organization

Section 2 describes the architecture of MSPs. Section 3 elaborates on normal execution processing. Section 4 explains recovery processing. Section 5 presents the experimental results on performance measurements. We review related work in section 6 and conclude, in section 7 with a short discussion, including future work. Throughout, we use **boldface** for definitions and *italics* for emphasis.

2. SYSTEM ARCHITECTURE

Figure 2 illustrates an MSP's internal architecture. We focus on two aspects: message exchanges and business state, and then wrap up with the recovery correctness requirement.

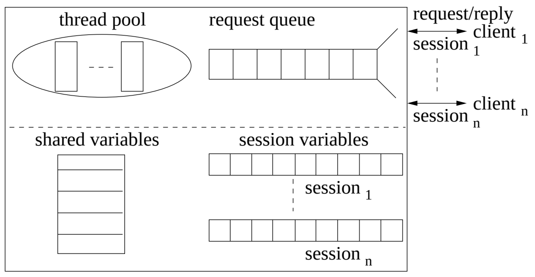


Figure 2: Internal architecture of an MSP.

2.1 Message Exchanges

An MSP provides its service through its **service methods**. A client, either an **end client process** or another MSP, uses the service via (synchronous) remote procedure calls to service methods. Messages between an MSP and its clients are **request/reply** exchanges.

Each MSP 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.

A **session** with an MSP is started or ended by a client request. 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 the MSP.

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

In order to process a request, a session SE_c of an MSP MSP_c, may further send a new request to another MSP, say MSP_s. This request must be sent over a session of MSP_s, say SE_s, which is started by SE_c. In this case, SE_c is the client of SE_s and SE_s is an **outgoing session** started by SE_c. Figure 3 illustrates this relation.

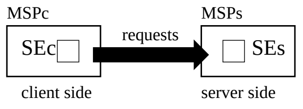


Figure 3: SE_s is an outgoing session of SE_c.

Thus there are message interactions among MSPs. Some interacting MSPs are provided by the same service provider and have fast and reliable communication. These MSPs can be configured to be in a **service domain**. Less tightly associated MSPs with less reliable communication will usually be in separate service domains. Service domains are disjoint and end client processes are outside of all service domains.

2.2 Business State

The lower half of Figure 2 illustrates the business state of an MSP, including two types of in-memory state: **session state** and **shared state**. For each client session, an MSP maintains *private* session state consisting of **session variables**. An MSP maintains shared state consisting of **shared variables**.

A session variable can be accessed only in service methods which 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 shared variable is shared by all sessions of an MSP, i.e., it can be accessed by any service method over any session. So multiple threads may access a shared variable concurrently. 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 simultaneously by concurrent threads. Read or write locks are issued when accessing shared variables. The duration of a lock spans the variable access, i.e., locks are released once the access is finished.

2.3 Recovery Correctness

After a crash occurs, an MSP must be recovered to its most recent logged business state which consists of its shared variables and all session variables, and which satisfies **inter-MSP consistency**. Inter-MSP consistency requires causal consistency of messages, i.e., if the business state of a process (an MSP or end client process) includes a message receive, either request or reply, the sender process's business state must include the message send [7]. Figure 4 illustrates a violation of such consistency.

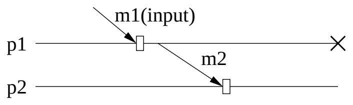


Figure 4: Processes with message logging.

Process p_1 receives an input message m_1 from outside and

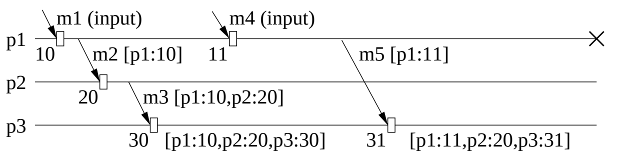


Figure 5: Messages with dependency vectors.

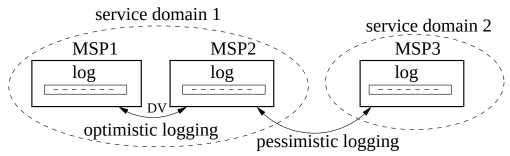


Figure 6: Locally optimistic logging.

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 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 (process). Inter-MSP consistency requires there to be no orphans after recovery. Inter-MSP consistency with a client resend of the same request until the corresponding reply is received guarantees **exactly-once** execution semantics for a request.

3. NORMAL EXECUTION PROCESSING

During normal execution, nondeterminism is logged so that re-execution can reconstruct the business state from the log. We elaborate on four aspects: message processing, session processing, shared state processing and MSP fuzzy checkpointing.

3.1 Message Processing

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 MSP maintains a **next expected request sequence number**. In addition, an MSP buffers the reply of the latest request for each session, so that this **buffered reply** can be re-sent should it get lost due to network failure or client crash [1].

Next we first review classical pessimistic and optimistic logging of message exchanges. Our innovation in message processing is to combine these well-understood methods in a practical way, which we call *locally optimistic logging*.

Pessimistic vs Optimistic Logging

Pessimistic logging guarantees that no orphans are ever created. One form of pessimistic logging [1, 2] is that messages are written to a buffer upon receipt. Before this receiver sends a message, it flushes the buffer to disk. In Figure 4, 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. So m_2 never becomes an orphan. Pessimistic logging incurs logging overhead for this log flush. (This is called *pessimistic* logging since crashes are rare.)

Optimistic logging allows orphans to be created, but orphans will later be detected by **dependency vectors** (DVs) [5, 17] and recovery will eliminate orphans eventually. 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 discussion.)

Referring to Figure 5, when process p_1 receives a message m_1 and posts it to a buffer with log sequence number 10 before sending message m_2 to p_2, the DV attached in m_2 includes 10 as p_1's state number. When p_2 sends m_3 to p_3, and the log record written by p_2 has log sequence number 20, 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*].

Later, if p_1 crashes, after **crash recovery**, p_1 broadcasts a **recovery message** indicating the state to which it is able to recover (called the **recovered state number**). Other processes log and remember this recovered state number. If p_1 is not able to recover to state 10, both p_2 and p_3 will know they are orphans by checking their DVs with this recovered state number, and must do **orphan recovery** to roll back to a state before they got the orphan messages.

Since an *output message* (to outside) should never become an orphan, before it is sent, the sender does a **distributed log flush** according to 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.

Locally Optimistic Logging

In the place of either classical optimistic logging or classical pessimistic logging, we use **locally optimistic logging**. In this new method, message exchanges *across* service domains use pessimistic logging, and message exchanges *within* a service domain use optimistic logging. To further reduce logging overhead, all sessions of an MSP share one physical log. Figure 6 illustrates this logging. Since an end client process is outside of any service domain, message exchanges between it and any MSP 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 initiated dictated by the sender's DV. The separate local flushes required by a distributed log flush can be done in parallel, unless the physical logs of MSPs in the service domain share a disk controller. Once the distributed log flush completes, this message will never be an orphan and the sender's DV does not need to be attached to this message. Figure 7 lists the actions associated with message exchanges.

Figure 7: Actions for message M (request or reply).

	inside service domain	*across service domains*
before send:	attach sender's DV to M;	distributed log flush with 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;

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

An MSP crash can cause only other MSPs in the same service domain to roll back. But recovery independence is maintained between service domains. After crash recovery, an MSP broadcasts recovery messages only within its service domain. Each MSP needs to keep recovered state numbers only for MSPs in its service domain. Finally, since MSPs within a service domain usually have fast and reliable communication, the message overhead for recovery message broadcasting and distributed log flushes is usually modest.

3.2 Session Processing

Sessions inside an MSP interact via shared variables, and accesses to shared variables need to be logged. However these interactions are relatively infrequent because each session executes the relatively independent task [6] of servicing a single user. Importantly, accesses to session variables need not be logged, as recovery will re-execute service methods to reconstruct such private session state.

When one MSP crashes, another MSP 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. A session does not crash by itself but only as part of its MSP crash. So sessions are recovery units, while MSPs are crash units.

If only one DV is maintained to capture dependencies for an MSP as a whole, all its sessions will roll back, possibly unnecessarily. To avoid this, we associate a DV with each session. This enables sessions to recover separately, avoiding unnecessary rollback cost. Correspondingly each session must have its own state number, which is the most recent LSN of the session. This is shown in Figure 7 where *sender's DV* means the DV of the sending session, and *receiver's DV* means the DV of the receiving session.

Session Checkpointing

To speed up recovery of a session, a **session checkpoint** is taken whenever its logged information since the previous checkpoint reaches a threshold. Each session is checkpointed independently, 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. Prior to a session checkpoint, we do a distributed log flush as dictated by the session's DV to ensure that the state as of checkpoint completion is never an orphan. On completion, the session's previous log records can be discarded.

Position Stream

All sessions of an MSP 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 buffer becomes full are they flushed to disk. So the cost of writing positions is low. In case of an MSP crash, positions which 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 checkpoint, previous positions are discarded by *truncating* the position stream to zero length. 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 end of the session's log records.

3.3 Shared State Processing

Like a session, a shared variable is a recovery unit. Unlike a session, a shared variable is *passive*, accessed by sessions.

Shared Variable Locking

Our infrastructure locks shared variables on access *transparent* to middleware programs. The lock is directly associated with the variable and we do not maintain a *separate* lock table. Read and write locks are held only for the duration of the access and thus no deadlocks are involved. Since the number of shared variables is limited, the total memory space required for locks is limited.

Dependency Tracking

A shared variable 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.

Access to a shared variable by a session was modeled as two messages [6], one from session to variable, and another from variable back to session. Such modeling requires that the session's DV is first merged into the variable's DV, then the merged DV is merged back into the session's DV. This results in both the session and the variable having the same dependency on other MSPs after the access.

However, by considering *read/write semantics* we can see that this results in *false dependency*. For a read, the variable's dependency should be passed to the reader session, and the reader session's dependency does not need to be passed to the variable. For a write, the writer session's dependency should be passed to the variable, and the variable's dependency does not need to be passed to the writer session. In addition, since a write completely replaces the variable's existing value with a new value, the variable's existing dependency can be removed and its dependency will become the same as the new value's dependency, which is also the writer session's dependency.

Exploiting the above considerations, we refine the dependency tracking as follows: reading a shared variable *merges* this variable's DV into the reader session's DV, and writing a shared variable *replaces* this variable's DV with the writer session's DV.

Value Logging

Access order logging was suggested for shared state access [16]. That is, the access order is logged and the same access order is followed during recovery to reconstruct the shared state. However, this approach increases recovery dependence among sessions. For example, if a session (reader) reads a shared variable which was written by another session (writer), should the reader later become an orphan, its recovery will require the writer (whether orphan or not) to roll back. This roll back enables the writer to re-create and re-write the value for the variable, enabling the reader session to read this value for its recovery.

Deadlocks involving thread pooling are also possible. Each request (or logged request) is processed (or replayed) by a thread newly 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 MSP hangs.

To overcome the drawbacks of access order logging, we introduce **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 increases recovery independence of the reader session. 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 finds this variable an orphan, it can itself roll this variable back to its most recent non-orphan value by following this backward chain. In this way, deadlocks involving thread pooling are avoided. In addition, due to the relatively small size of shared variables and infrequency of accesses, value logging incurs only modest overhead compared to access order logging.

Figure 8 lists actions involved with accessing a shared variable. This includes dependency tracking, value logging and two more 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.

Figure 8: Actions for accessing shared variable SV.

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;

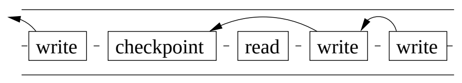


Figure 9: A log record sequence of a shared variable.

Shared State 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 initiated as indicated by 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. The backward chain breaks at checkpoints. Figure 9 illustrates this.

3.4 MSP Fuzzy Checkpointing

To reduce crash induced outages, we take an MSP checkpoint, which mainly contains recovered state numbers of MSPs in the service domain, the LSN of each session's most recent checkpoint, and the LSN of each shared variable's most recent checkpoint. In order to take an MSP checkpoint, ongoing session activities are not blocked. This is called **fuzzy checkpointing** and has little impact on server performance.

The *minimal 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 [12], after an MSP checkpoint is taken, its LSN is recorded in the **log anchor**, a 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 MSP checkpoint's LSN inside the log anchor. Figure 10 illustrates an MSP checkpoint and its relationship with others.

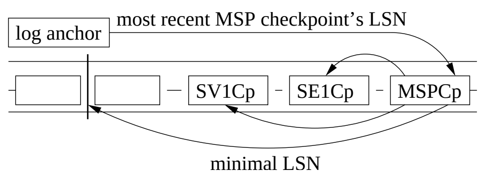


Figure 10: *SE1Cp* **and** *SV1Cp* **are a checkpoint of session** *SE1* **and shared variable** *SV1*, **respectively.**

If a session is 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 MSP checkpoints taken since the previous session checkpoint reaches a threshold. Checkpoints for shared variables are similarly forced.

4. RECOVERY PROCESSING

After an MSP crashes, MSP crash recovery is conducted following its physical log. At the end of this crash recovery, the MSP broadcasts within the service domain its recovered state number, based on which, sessions or shared variables of another normally executing MSP within the service domain may find they themselves have become orphans. Orphan recovery of these sessions or shared variables are required to ensure *inter-MSP consistency.* Thus there are three aspects to recovery processing: session orphan recovery, shared state orphan recovery and MSP crash recovery.

4.1 Session Orphan Recovery

During normal execution, when an MSP receives a recovered state number, any *idle* (i.e. not executing a method) session's DV is checked to see if the session has become an orphan. For a non-idle session, whenever the recovery infrastructure is able to intercept the method execution, the session's DV is checked. This interception occurs when the executing method sends or receives a message (request or reply), or accesses (reads or writes) a session or shared variable. In addition, during the method execution, a distributed log flush according to the session's DV may fail with the session found to be an orphan.

Session orphan recovery is immediately initiated once the session is found to be an orphan. So when an MSP is running, 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 does *redo* recovery by replaying the logged requests following its position stream.

Logged Request Replay

To replay a logged request, the requested method is re-executed. Re-execution follows the subsequent rules:

- Accessing a *session* variable is done in the same way as 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.
- Requests to other MSPs are not sent, and their reply is read from the log.

During re-execution, 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.

Orphan Recovery End

Since the session was found an orphan before session orphan recovery starts, this recovery must eventually 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.

When a session encounters such an orphan log record during recovery, it shows that the session became an orphan because of 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.

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 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 MSPs, those log records will be invisible to (skipped without being read during) the subsequent session orphan recovery.

If the MSP 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 of the MSP, at orphan recovery end, 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 be read after the crash. On the other hand, if the EOS log record does not get to disk, *all* the session's log records beginning with this orphan log record will be skipped.

Orphan Recovery upon Multiple Crashes

During session recovery, the session's DV is still checked in case the session has become an orphan due to another MSP crash within the service domain. Session orphan recovery can be initiated during an ongoing session 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 11 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 its session checkpoint is taken, it becomes an orphan again due to another MSP crash, finally the subsequent orphan recovery finds the orphan log record *orphan2* and writes the corresponding *EOS2*.

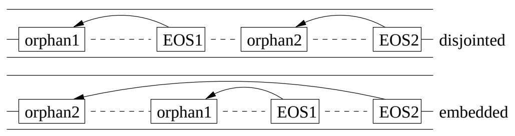


Figure 11: Combinations of (orphan, EOS) pairs upon multiple crashes.

Figure 12: Actions for MSP crash recovery.

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

Any future orphan recovery of this session will skip all its log records between *orphan2* and *EOS2*, including those between *orphan1* and *EOS1*.

4.2 Shared State 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 according to this variable's DV, its DV will also be checked. So read and checkpointing during normal execution are the *only two events* 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.

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. So shared state 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.

4.3 MSP Crash Recovery

To recover an MSP after a crash, the MSP is re-initialized from its most recent checkpoint, whose LSN is in the log anchor. A single-threaded analysis scan of the physical log is started at the *minimum LSN* as recorded in the MSP checkpoint, to reconstruct position streams for all sessions and to update all shared variables to their most recent logged values. When log records on recovered state numbers of other MSPs in the service domain are encountered, the scan will update the MSP'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 MSP's recovered state number. Next the MSP makes an MSP checkpoint. Finally, all sessions start to recover *in parallel* following their reconstructed position streams and the MSP can start accepting new sessions.

Now the MSP 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. Figure 12 lists the actions for MSP crash recovery.

Shared State Roll Forward

During the scan, checkpoints or write log records for a shared variable are used to roll forward this variable. When a checkpoint of a shared variable is encountered, the variable is updated with the checkpointed value. This value will never be an orphan.

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 if 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 MSP relies only on its physical log to recover. During this scan, at the point when this write log record is encountered, the MSP 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 about recovered state numbers have been encountered, the MSP has built up all its knowledge about recovered state numbers from its physical log. If a shared variable's most recent value is from a write log record, this value may be an orphan according to the MSP's current (more recent) knowledge about recovered state numbers. However, orphan recovery for this variable is not initiated immediately, but only possibly later if a session in normal execution tries to read this variable.

Session Recovery after Scan

Session recovery after the scan is similar to session orphan recovery. 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 a recovering session does not encounter an orphan log record after all its log records are consumed, the session simply switches to normal execution. In case an orphan log record O is encountered, the session skips those log records beginning with O until the EOS 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 with those subsequent log records.

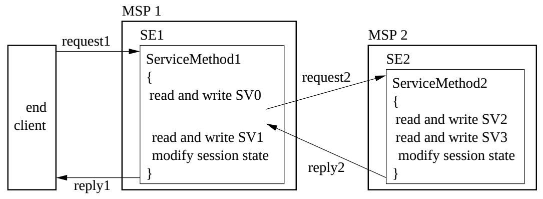
EOS Not Found: In case such an EOS does not exist, the session writes an EOS log record pointing back to O, truncates its position stream to remove positions of skipped log records beginning with O, and then switches to normal execution. This case is the same as session orphan recovery.

5. PERFORMANCE MEASUREMENTS

We implemented a Web services prototype with the framework of Microsoft ASP.NET [11] on Windows XP. The prototype demonstrates the feasibility of using our recovery methods. We executed a number of performance experiments. We report results on the comparative performance of locally optimistic logging versus pessimistic logging, the impact of checkpointing on normal execution, the recovery performance when crashes are introduced, and finally the performance of a more fully loaded system. The results are promising. We first describe our experimental setting.

5.1 Experimental Setting

Figure 13 shows our experimental configuration and hardware parameters. We have one end client and two MSPs hosted by web servers. The *end client* starts a session *SE1* with *MSP1*, then sends *request1* to *ServiceMethod1* a number of times. *SE1* at *MSP1* further starts a session *SE2* with *MSP2*. *ServiceMethod1* reads and writes shared variable *SV0*, sends *request2* to *ServiceMethod2*, then reads and writes *SV1*, and finally writes *SE1*'s session variables. *ServiceMethod2* reads and writes *SV2* and *SV3*, and writes *SE2*'s session variables. Both parameter and returned value of a request are 100B. The total session state size of each of *SE1* and *SE2* is 8KB. *ServiceMethod1* and *ServiceMethod2* write only 512B of their session state. Each shared variable is 128B. Such a configuration is consistent with the observation that the shared in-memory state is relatively small. Our hardware consists of one client and two server computers in an Ethernet network.


MSP 1
SE1
request1
end client
ServiceMethod1 { read and write SV0 read and write SV1 modify session state }
reply1
request2
reply2
MSP 2
SE2
ServiceMethod2 { read and write SV2 read and write SV3 modify session state }

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

Figure 13: Configuration and hardware parameters.

5.2 Locally Optimistic Logging Performance

To measure the comparative performance of locally optimistic logging versus pure pessimistic logging, we measured the response time of requests in two different system configurations. Configuration *LoOptimistic* has two MSPs in the same service domain and hence uses optimistic logging locally within this domain. Interactions with the *end client* are always logged pessimistically. Configuration *Pessimistic* has each MSP in a different service domain and hence uses only pessimistic logging.

In order to compare log-based recovery with alternative approaches, we also measured the response time in three other configurations. Configuration *NoLog* has no logging and recovery infrastructure, and hence does not provide recovery. Configuration *Psession* provides persistent sessions via the web server storing session states inside a local DBMS. When a request is processed, the session state is fetched from the database, and after processing, the session state is written back to the database. In configuration *StateServer*, session states are stored in-memory at a state server on a different computer. Both *Psession* and *StateServer* are common commercial approaches for recovery of session states, though they do not support shared in-memory state. The table in Figure 14 lists the average response times for 20K end client requests.

LoOptimistic	*Pessimistic*	*NoLog*	*Psession*	*StateServer*
24.746	35.227	8.697	48.617	16.658

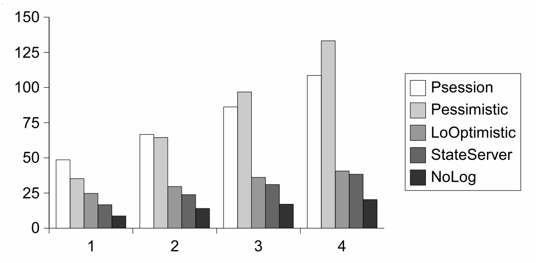


Figure 14: Table: response time (ms). Chart: response time versus number of calls to ServiceMethod2 inside ServiceMethod1.

Response Time Analysis

Psession takes a session checkpoint after every request and requires two database transactions (read and write) at both MSPs for each request. This is very costly. *StateServer* has a much shorter response time, but session states are not persistent and will not be recovered if the state server crashes.

Locally optimistic logging has fewer log flushes than pessimistic logging, which results in a shorter response time. For each *end client* request, pessimistic logging requires three log flushes in sequence: at *MSP1* before sending *request2*, at *MSP2* before sending *reply2*, and at *MSP1* before sending *reply1*. Locally optimistic logging requires one distributed log flush before sending *reply1*. This distributed flush incurs two flushes in parallel, at *MSP1* and at *MSP2*.

The log is written in blocks whose size varies from 1 to 128 disk sectors of 512B. Large log blocks can improve log performance for heavily-loaded servers. Log blocks are aligned at sector boundaries and when a log block is flushed, its last sector may not be full. On average, a half sector is wasted on every flush. Locally optimistic logging, with its fewer log flushes, results in less log space wasted. In pessimistic logging, each of the two flushes at *MSP1* writes 2 sectors and the flush at *MSP2* writes 3 sectors. In locally optimistic logging, the flushes at *MSP1* and *MSP2* each write 3 sectors. Thus, locally optimistic logging uses one less sector per *end client* request.

Locally optimistic logging requires an extra message round to request *MSP2* to flush as part of the distributed log flush and it has to do dependency tracking. We can express the response time difference between the two logging methods as $\Delta_{response} = 2 \cdot T_{F2} + T_{F3} - max(T_{F3}, T_M + T_{F3}) - T_{DV} = 2 \cdot T_{F2} - T_M - T_{DV}$, where T_{Fn} is the time to flush n sectors, T_M is the time of a message round trip, and T_{DV} is the time on dependency tracking.

On a server hosting an MSP, a hard disk is mostly dedicated to the MSP as a log, so writing does not require a random disk seek of about 10.5ms. Excluding this seek time, the log flush time is mostly the average disk rotational latency plus the data transfer time together with the amortized track to track seek time. This suggests that $T_{Fn} = (60000/7200/2 + n/63 \cdot 60000/7200 + n/63 \cdot 1.2)$, yielding T_{F2} of about 4.5ms. However, the operating system also uses the hard disk and this incurs occasionally random disk seeks for log writing. So the actual flush time is slightly more than 4.5ms, but much less than 15ms (= 10.5+4.5). For our analysis, we crudely estimate T_{F2} to be 8ms(= 4.5+10.5/3).

We measured the message round time (T_M) between two MSPs at 3.596ms, which is less than the time of a log flush. Moreover since T_{DV} is relatively small, $\Delta_{response}$ should have a positive value. Hence locally optimistic logging has a lower response time than pessimistic logging. $\Delta_{response}$ is calculated as 12.404ms $-T_{DV}$, which is close to the measured difference of 10.481ms (= 35.227 − 24.746). Thus, compared to pessimistic logging, locally optimistic logging reduces the response time by about 30%. Locally optimistic logging still incurs considerable overhead compared to *NoLog*. When more business logic code is incorporated, however, this overhead becomes a smaller fraction of the total response time.

Response Time with Intra-Service-Domain Interactions

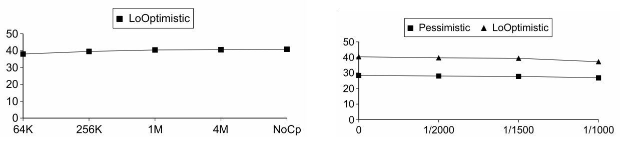
Suppose now that *ServiceMethod1* calls *ServiceMethod2* m times before it accesses *SV1*. If we, for simplicity, assume every log flush requires the same time T_F, the response time difference would then grow to $2 \cdot m \cdot T_F - T_M - T_{DV}$. So the more interactions per request inside the service domain, the larger the response time difference. To confirm this, we varied the number of calls to *ServiceMethod2* inside *ServiceMethod1*, and measured the response times for all five configurations.

Figure 14 charts the result. As the number of calls becomes larger, the response time becomes larger and the response time difference between locally optimistic logging and pessimistic logging increases. The number of flushes is the decisive factor, not the size of the flushed records. While the number of flushes required by pessimistic logging increases with the number of calls, locally optimistic logging always requires only one flush at both MSPs (in parallel).

The response time of pessimistic logging increases faster than that of *Psession*, because each call increases the number of flushes in pessimistic logging by two, while the number of flushes in *Psession* increases only by one (due to the write transaction). Further, the response time difference between locally optimistic logging and *NoLog* tends to increase (slowly) because of the increased log size for locally optimistic logging. Finally, the response time of *StateServer* increases faster than that of locally optimistic logging. When the number of calls is 4, the response time of *StateServer* is close to that of locally optimistic logging.

5.3 Checkpointing Overhead

Checkpointing speeds up recovery at the cost of extra overhead during normal execution. We measured the session checkpointing overhead for locally optimistic logging by measuring the throughput at the *end client.* Figure 15 (a) shows the result, where the checkpointing threshold is the amount of log consumed by a session between checkpoints. The higher the checkpointing frequency (i.e., the lower the



(a) (b)

Figure 15: (a) Throughput (number of requests per second) versus checkpointing threshold. (b) Throughput versus crash rate (one crash per certain number of end client requests).

checkpointing threshold), the lower the throughput. Session state size is small (8KB), so even a low checkpointing threshold of 64KB, leads to only a small throughput reduction. With a threshold of 4MB, throughput is close to the no checkpointing case. In the previous section, we set the threshold to 1MB, which requires with locally optimistic logging, a session checkpoint after about every 682 end client requests for both *MSP1* and *MSP2*.

5.4 Recovery Performance

We introduced forced crashes to measure recovery performance. To generate orphans, in *ServiceMethod1* with locally optimistic logging, when the reply from *ServiceMethod2* is received by *MSP1*, *MSP2* is instructed to kill itself. This causes the buffered log records of *MSP2* to be lost. Thus, the distributed log flush initiated at the end of *ServiceMethod1* will fail, making session *SE1* at *MSP1* an orphan. We fixed the session checkpointing threshold at 1MB and measured the throughput using different crash rates (one crash every certain number of end client requests). To force performance differences, the crash rates were set much higher than a real system would likely encounter.

Throughput Upon Crashes

Figure 15 (b) shows the result. When the crash rate is 0, there are no crashes. When the crash rate is 1/1000, it is actually very high (equivalent to about *one crash every half a minute*). Locally optimistic logging always has higher throughput than pessimistic logging. As the crash rate increases, throughput decreases for both logging methods. The throughput decrease for locally optimistic logging is a bit larger than for pessimistic logging because locally optimistic logging has more crash recovery overhead. Pessimistic logging requires only crashed *MSP2* to recover. Locally optimistic logging also requires *SE1* at *MSP1* to perform orphan recovery after *MSP2* is recovered.

Maximum Response Time Upon Crashes

When a client knows that an MSP is checkpointing or recovering, it sleeps for 100ms and resends the request. Such delayed requests have a larger response time. For both logging methods, we recorded the maximum response time without crashes with a checkpointing threshold of 1MB, and the maximum response time with a checkpointing threshold of 1MB when the crash rate is 1/1000, 1/1500 or 1/2000. For comparison, we also recorded the maximum response time when there are no crashes for five other cases: locally optimistic logging without checkpointing, pessimistic logging without checkpointing, *NoLog*, *StateServer* and *Psession*.

	Crash	*NoCrash*	*NoCp*	
LoOptimistic	3,245	490	123	*NoLog:* 217
Pessimistic	2,360	150	133	*StateServer:* 544
				Psession: 660

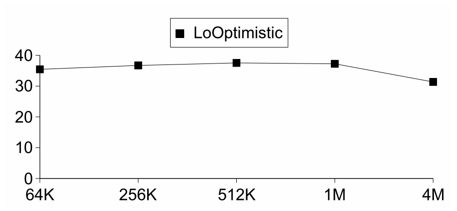


Figure 16: Table: maximum response time (ms). Chart: throughput for crash rate 1/1000 versus checkpointing threshold.

The table in Figure 16 shows our results. Even with no logging, the maximum response time (217ms) is much larger than the average response time (8.697ms), indicating some instability. The maximum response time in both *StateServer* and *Psession* is relatively high even without crashes (544ms and 660ms, respectively). Checkpointing increases the maximum response time (column *NoCrash*) in both logging methods compared with not checkpointing (column *NoCp*). The increase for locally optimistic logging is more obvious. With locally optimistic logging, a distributed log flush is required before a session checkpoint, while pessimistic logging requires only a local log flush.

When crashes are introduced, maximum response time for both logging methods increases substantially due to recovery. However only a small number of requests experience large response times. The average response time remains low (26ms for locally optimistic logging, 36ms for pessimistic logging). Locally optimistic logging has a larger maximum response time because of *SE1*'s orphan recovery at *MSP1*. The difference, 885ms (= 3245 − 2360), results from replaying logged requests by *SE1*'s orphan recovery (about 682 requests in the worst case, with 1MB threshold). So replaying a logged request takes only 1.30ms (= 885/682).

The average response time during normal execution for locally optimistic logging was measured at the *end client* as 24.746ms. So processing a request at *MSP1* during normal execution takes 20.846ms (= 24.746 − 3.9), where 3.9ms is the measured message round trip time between *MSP1* and the *end client*. Hence replaying a logged request is much faster than processing a request during normal execution. Both require the same amount of CPU time at *MSP1* for the method execution. However, normal processing requires execution of *ServiceMethod2* at *MSP2*, distributed log flushes and message exchanges between the two MSPs, while replaying involves only *efficient* local log reads.

Log reads are 128 sectors (= 64KB) and can contain multiple log blocks. Each log flush is one log block, usually 3 sectors for *MSP1*. So log reads during recovery are larger and more efficient than log flushes during normal execution. *SE1*'s orphan recovery at *MSP1* may read 1MB of log in the worst case, which is mostly sequential and takes 370ms (= $1M/64K \cdot (60000/7200/2 + 128/63 \cdot 60000/7200 + 128/63 \cdot 1)$), following a formula similar to that used for flushes. So, on average, only 0.54ms (= 370/682) is required to read the log for each replayed logged request.

Maximum Throughput with Fixed Crash Rate

With the introduction of crashes, throughput decreases. The Checkpoint frequency trades throughput for recover speed. It incurs extra overhead during normal execution, which can decrease throughput, but enables faster recovery. As checkpointing threshold increases, throughput increases but eventually reaches a maximum after which further threshold increases lead to increased recovery time and decreased throughput. The Figure 16 chart shows an optimal checkpointing threshold (yielding the highest throughput) for a crash rate of 1/1000 is between 256KB and 1MB, with 512KB near the maximum. Since the crash rate in real systems is much lower, a higher threshold should be used.

5.5 Performance with Multiple Clients

With a single end client, the system is underutilized, even when the client *continuously* submits requests. To load the system more heavily, we measure the throughput using multiple clients. We also implemented batch flushing. A request to flush the log is not executed immediately, but rather after a specified timeout, providing a possibility to process several flush requests with a single write. With batch flushing, the amortized logging overhead (both disk and CPU loads resulting from logging) will decrease and throughput will improve.

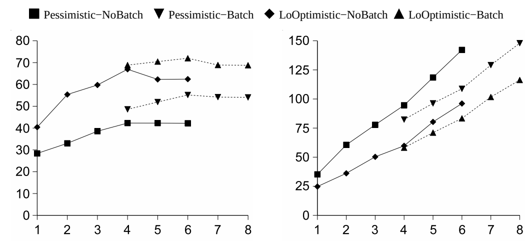


Figure 17: Performance (with/without batch flushing) versus number of end clients: throughput (left) and response time (ms, right).

Figure 17 shows performance with and without batch flushing for both logging methods when the session checkpointing threshold is 1MB and there are no crashes. Without batch flushing, the throughput is highest when the number of clients reaches 4. To pick a timeout value for batch flushing, we focused on pessimistic logging, fixed the number of clients at 4 and tried several different timeout values. When the timeout is 8ms, throughput is highest for pessimistic logging with 4 clients. This timeout, roughly the time of a log write, is used in all subsequent experiments with batch flushing. With batch flushing, throughput is highest when the number of clients reaches 6.

For pessimistic logging, batch flushing increases the highest throughput by about 30%, while for locally optimistic logging, batch flushing increases the highest throughput by about 8%. So pessimistic logging benefits more from batch flushing than locally optimistic logging. However, with batch flushing, the highest throughput supported by locally optimistic logging is still about 30% higher than that supported by pessimistic logging.

With more clients, the response time becomes larger in all cases. Batch flushing reduces the response time for both logging methods with more than 3 clients. With 4 clients, batch flushing reduces the CPU utilization of the *MSP1* computer from about 90% to about 60% for both logging methods. This shows that batch flushing can reduce both CPU and disk utilization simultaneously.

The timeout value for batch flushing should take into consideration the specific system configuration and the workload, e.g. the number of concurrent clients. The timeout (8ms) used in our experiments might not lead to the highest throughput supportable by our recovery system. In addition, the experimental results should not be interpreted as the system being limited to support at most several clients. With a real workload, end clients do not continuously submit requests, but rather are idle most of the time.

6. RELATED WORK

Pessimistic logging and optimistic logging were invented in the fault-tolerance community [7] and are used for message passing systems, where entities interact with one another via message exchanges only. Individual threads were considered as separate recovery units with optimistic logging [6], however, log management of a multi-threaded process with optimistic logging was not explored. Log-based recovery for general application processes over a Java virtual machine was partially implemented [13], but no consistency among interacting processes was considered.

Requests from a client are guaranteed to be processed exactly once by a server via recoverable queues [4]. However, the client application must be structured as a "string of beads" style workflow where each bead is a transaction. More complicated structures are not allowed. Application recovery in a client/server system was studied with message logging incorporated with the server internal logging [10].

Recovery for middleware components such as CORBA components have been studied [14, 15]. Their work uses replication for fault-tolerance and utilizes an underlying totally ordered multi-cast communication infrastructure. Recovery methods for Microsoft .NET components were studied based on optimized pessimistic message logging [1, 2]. None of these component recovery mechanisms directly supports concurrent accesses to shared state by multiple threads. In a Web services setting, idempotent services were advocated to enable recovery of applications via replay if they do appropriate (optimized pessimistic) message logging [9].

Commercial web servers [3, 8, 11] 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 state.

7. DISCUSSION

We have described our prototype system for log-based recovery of middleware servers. By using locally optimistic logging for message exchanges within service domains, we reduced logging overhead. We maintained recovery independence between service domains via pessimistic logging. Value logging for shared in-memory state (shared variables) increased recovery independence inside a middleware server. It also kept logging overhead modest in the usual case where shared state is small size and infrequently accessed. 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. Finally, we reported the promising performance produced by our prototype.

In on-going work, we handle middleware server interactions with transactional systems within our recovery infrastructure. To support consistent recovery in this case, we continue our pursuit of efficient logging and recovery that involves only modest modification to existing transactional systems.

We believe that our prototype has demonstrated both the feasibility and the desirability of having the system infrastructure transparently provide persistence guarantees for middleware servers. By relieving application programmers from the problem of coping with system failures, we greatly simplify their task, permitting them to focus on the business logic of the application. By providing a highly efficient infrastructure for this, we enable enterprises to achieve the fault tolerance that they need at an acceptable cost.

8. REFERENCES

[1] R. Barga, S. Chen, and D. Lomet. Improving Logging and Recovery Performance in Phoenix/App. In *Proc IEEE ICDE*, pages 486–497, 2004.

[2] R. Barga, D. Lomet, and G. Weikum. Recovery Guarantees for General Multi-Tier Applications. In *Proc IEEE ICDE*, pages 543–554, 2002.

[3] BEA Corp. Deploying and Configuring Web Applications. http://edocs.bea.com/wls/docs60/, 2000.

[4] P. A. Bernstein, M. Hsu, and B. Mann. Implementing Recoverable Requests Using Queues. In *Proc ACM SIGMOD*, pages 112–122, 1990.

[5] O. P. Damani and V. K. Garg. How to Recover Efficiently and Asynchronously When Optimism Fail. In *Proc IEEE ICDCS*, pages 108–115, 1996.

[6] O. P. Damani, A. Tarafdar, and V. K. Garg. Optimistic Recovery in Multi-threaded Distributed Systems. In *Proc IEEE SRDS*, pages 234–243, 1999.

[7] E. N. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message Passing Systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[8] B. Hines, T. Alcott, R. Barcia, and K. Botzum. IBM WebSphere Session Management. http://www.informit .com/articles/article.asp?p=332851, 2004.

[9] D. Lomet. Robust Web Services via Interaction Contracts. In *VLDB Workshop on Technologies for E-Services*, 2004.

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

[11] Microsoft Corp. ASP.NET. http://msdn.microsoft.com/ library, 2000.

[12] C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. 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.

[13] J. Napper, L. Alvisi, and H. Vin. A Fault-Tolerant Java Virtual Machine. In *Proc IEEE DSN*, pages 425–434, 2003.

[14] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing Determinism for the Consistent Replication of Multithreaded CORBA Applications. In *Proc IEEE SRDS*, page 263, 1999.

[15] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. State Synchronization and Recovery for Strongly Consistent Replicated CORBA Objects. In *Proc IEEE DSN*, pages 261–270, 2001.

[16] M. Ronsse, K. Bosschere, M. C., J. C. K., and D. K. Record/Replay for Nondeterministic Program Executions. *Commun. ACM*, 46(9):62–67, 2003.

[17] R. E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.