

Title :

Explorer: A Recovery Framework for Platforms with Commercial Features

Authors :

Rui Wang, Betty Salzberg

(This work is supported by a grant from Microsoft.)

Institution :

College of Computer and Information Science

Northeastern University

Emails :

{bradrui, salzberg}@ccs.neu.edu

Address :

161 Cullinane Hall

Northeastern University

360 Huntington Ave

Boston, MA 02115

Tech Report Number:

NU-CCIS-03-06

Explorer: A Recovery Framework for Platforms with Commercial Features

Abstract

Currently availability for applications on commercial platforms is only provided at great additional expense, usually involving duplicate stand-by systems. Less expensive alternatives are needed. One alternative is to provide the capability for transparent automatic recovery from system failures. This paper describes an object-based optimistic recovery framework, called Explorer, for platforms with commercial features. It abstracts a conceptual model, which contains message exchanging, multithreading and shared in-memory data access, from current commercial platforms and develops a complete solution for guaranteed recovery of systems in that model. It combines a transitive dependency tracking method, a direct dependency tracking method and a logging method in a seamless way. Collections of processes which share data are grouped into recovery units which we call autonomous process groups (APGs). The overhead for dependency tracking within an APG is reduced. Explorer requires only small extension to the underlying operating system. After describing Explorer, we demonstrate its correctness in accordance with our extended definition for consistency.

1 Introduction

Current applications have high requirements for availability. But no matter how reliable a computer is, failures are always possible. For high availability, expensive solutions have been provided, such as standby computers [4, 5]. Existing work on fault-tolerant computing focuses on message passing systems [10]. In a message passing system, each recovery unit is single-threaded and interacts with other recovery units through message exchanging. Message receiving is the only source of nondeterminism. But current commercial computer systems allow multi-threads, multi-processes and shared data access within a recovery unit. Solutions considering these features are needed.

Fault-tolerant solutions advanced in component software areas allow only stateless components [13]. However, most applications require middleware components to maintain state[3].

With the above requirements in mind, we abstract a

conceptual model from current commercial computer systems such as Windows 2000, Linux and Unix and design an object-based optimistic recovery framework, called **Explorer**, for that model. Upon any stop failure, applications can automatically recover to the most recent consistent point. Recoverability is transparent to the application.

The **contributions** of this paper are: **1.** A conceptual model containing multithreading and shared in-memory data access, in addition to message passing, is advanced. This is a closer approximation to commercial platforms than the purely message passing model. **2.** A complete solution including logging mechanisms is provided for the transparent recovery of that model. In our method, a log record won't be flushed to disk until all predecessor log records are flushed, so all persistent log records are orphan-free. **3.** An autonomous process group (APG) is identified as both the failure unit and the rollback unit. Dependency among APGs and dependency within an APG are differentiated, and a transitive dependency tracking method, a direct dependency tracking method and a logging method are combined in a seamless way. Tracking the dependency within an APG is further optimized in a lightweight way, which is important because interactions within an APG occur more often than interactions among APGs.

The rest of this paper is organized as follows. Section 2 talks about related work. Section 3 presents the conceptual model based on current commercial platforms, analyzes the sources of nondeterminism and advances the extended definition of consistency. Section 4 elaborates the recovery framework Explorer, by extending the functions of the conceptual model, and demonstrates the correctness of the framework in accordance with the definition of consistency. The last section concludes this paper and suggests some issues to be attacked in the future.

2 Related Work

Fault-tolerant computing has been researched extensively and most of it uses message passing systems. Recovery with pessimistic logging is presented in [4, 5]. The optimistic recovery method is introduced in [28], then [17, 25, 15, 27, 26, 8] continue to improve certain as-

pects of optimistic recovery. Optimization by exploiting message semantics is described in [19]. Causal message logging is used in [9, 22]. Sender based logging is presented in [16] and family-based logging is advanced in [1]. The implementation techniques for checkpointing are presented in [23, 6]. Customized checkpointing by programs to reduce loss of work is discussed in [31]. In [29], the number of messages to be logged and the checkpoint space overhead are reduced for the combination of optimistic recovery and independent checkpointing. A survey on recovery methods for message passing systems is given in [10].

In [14], a partial solution is provided to reduce the nondeterminism from software/hardware interrupts and thread scheduling. It uses an “Instruction Counter” to allow the reexecution of a program to follow the exact execution history. Thus it has the same access order on shared data as the failure-free execution (here access order to shared data is reduced to thread scheduling order). But it works only for uni-processor computers, because on a multiple-processor computer, different processors schedule different threads to run in parallel. The same thread scheduling order as the failure-free execution doesn't imply the same access order to the shared data. In addition, it requires hardware support, and since the recovery follows the exact execution history, recoverability and fault-tolerance is reduced because Heisenbugs might reappear [12].

Both multi-threading and shared data access are considered in [7]. The separation of rollback units and failure units is advocated. A shared data object is taken as a fictitious thread, and access to it can be taken as two messages: the request message from the caller thread to the fictitious thread and the reply message from the fictitious thread to the caller thread. The caller may need to wait for the reply message. In this way, the nondeterminism from the access order to shared data objects could be reduced to message exchanging.

From the theoretical aspect, fictitious threads are elegant. From the implementation view, there are three problems: first, currently access to an shared data object is not inherently implemented as message exchanging. Conversion to message-exchanging-based implementation will be awkward. Second, if it is implemented as message exchanging, one access to a shared data object involves two messages. That's not good, because more messages mean more overhead. Third, for more recent programming languages like Java and C#, where automatic memory management is used, all objects could be shared by all threads within the same process, thus resulting in a large number of fictitious threads and large overhead. For these reasons, we have rejected the message-only approach.

In [11], shared data access is protected by locks and

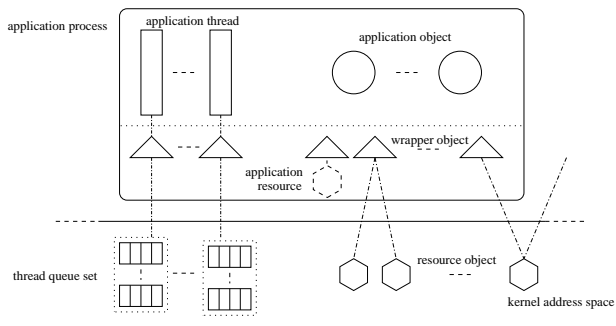


Figure 1: Application process structure of CM.

the nondeterminism from shared data access is reduced to recording access to the locks.

Masking underlying system failures, [18] provides transparent application recovery for client/server architectures, especially database applications. [3] further designs a contract to allow guaranteed recovery for multi-tier applications with consideration of transactional data service providers, but does not really consider multi-threading and shared in-memory data access.

3 Conceptual Model

Based on current commercial operating systems such as Windows, Unix and Linux, we abstract a *conceptual model (CM)*. We **assume** that the programming languages used by application programs are object-oriented and if pointers are allowed in application programs, they are used only for object references.

3.1 Model Description

Many *application processes* (Figure 1) run on each computer. One or more *application threads* run within each application process. Each application thread can access *application objects* located in the address space of its application process. The *kernel address space* belongs to the operating system.

Resource objects are operating system resources such as disk files, network channels, and primitives used for synchronization such as semaphores, which are called *synchronization resources*. Resource objects are located in the kernel address space and shared by multiple application processes. We **assume** that all application threads within an application process can access resource objects only via *wrapper objects*, a special kind of application object, located in the application process address space. A wrapper object for a synchronization resource is called a *synchronization wrapper*. Each application process has one *handle* (which has a numerical value and is valid only in its own address space) for each referenced resource ob-

ject. The handle is a private field in the wrapper object and invisible to the application program.

To give a more concrete picture, interfaces for accessing resource objects are system call functions in Unix or Linux, or Win32 APIs in Windows. Wrapper objects are instances of classes in a class library such as MFC (Microsoft Foundation Classes) which expose the functions of the operating systems and whose methods wrap the system calls or Win32 APIs, and application objects are instances of classes defined in the application program or other class libraries.

One wrapper object refers to only one resource object in most cases, but it could refer to more than one resource object. A simple example is a class in MFC, `CMultiLock`, whose method `Lock()` is to wait until any or all instances of type `CSemaphore`, `CMutex`, or `CEvent` have signaled or timed out [21]. It can be implemented as the reference of one wrapper object to multiple resource objects. Meanwhile one resource object can be referred to by several wrapper objects, which could be located in different application processes.

Some resource objects are network channels, which have two types: *reliable* ones (exactly once delivery in order) like TCP and *unreliable* ones like UDP. A reliable one is called a *session resource*, whose corresponding wrapper object is called a *session wrapper*. An unreliable one is called a *messagebox resource*, whose corresponding wrapper object is called a *messagebox wrapper*. We **assume** that reliable channels provide network communication based on message units, rather than on continuous data byte streams, and that when the sender message-box wrapper and the receiver message-box wrapper are on the same computer, the network channel between them is reliable.

A *thread queue set* is attached with each application thread, located in the kernel address space and used to store asynchronous events happening to that application thread. An application thread can read from only the queues in its own thread queue set via the wrapper object *queueset wrapper*. It works as a loop reading a message from one of the queues, then handling the message. It can put messages into the queues of its own or other application threads on the same computer via *queueset wrappers*. A thread queue set is not considered as a resource object. We **assume** that an application thread putting a message into a thread queue is reliable. Different operating systems have different sets of thread queues. Take the one in Windows 2000 as an example [24, 2].

Posted-,Send-,Reply- Message Queue: The posted-message queue is used to store the messages sent to it in *asynchronous mode*. The send-message queue is used to store the messages sent to it in *synchronous mode*. The reply-message queue is used

to store the replied messages for messages sent in synchronous mode.

Virtualized Input Queue: Device drivers of input devices like the keyboard and the mouse will put the data as messages from these devices into the queues of a special thread called Raw Input Thread (RIT) through hardware interrupts. The RIT will dispatch them further to the virtualized input queues of the destination application threads.

APC Queue: An APC (asynchronous procedure call) queue is used to store the returned message when an APC has finished. The finishing of an APC will trigger a software interrupt, whose handling puts the result message into the APC queue of the caller application thread. Later the caller application thread may actively check the results of its APCs from the APC queue.

Application resources are resources located in the address space of the application process, which might just be the proxies of objects located in other application processes, such as transactions. For a transaction, the corresponding wrapper object is called a *transaction wrapper*.

Access to synchronization resources can be either *recovery-relevant* or *recovery-irrelevant*. Recovery-relevant access can be further divided into two types: *order-sensitive* and *order-insensitive*. Suppose we have a synchronization resource called `ReadWriteLock`. The method "lock" for a write-lock is order-sensitive, the method "lock" for a read-lock is order-insensitive, and the method "unlock" is recovery-irrelevant. Access to synchronization resources in Windows 2000 can be identified as follows (the method names used in MFC are adopted)[21]. For `Mutex`, `CriticalSection` and `Semaphore`, the method "lock" is order-sensitive, and the method "unlock" is recovery-irrelevant. For `Event`, the methods "SetEvent", "ResetEvent" and "PulseEvent" are order-sensitive, the method "lock" is order-insensitive, and the method "unlock" is recovery-irrelevant.

To access shared in-memory data (either resource objects or application objects), both synchronization resources and message exchanging can be used by multi-threads or multi-processes for synchronization.

The execution of application threads may rely on some *transient environment properties* such as current system time and mutable properties of persistent data such as the current size of a disk file, which are returned when some operations on those persistent data are issued. The output of an application process may be *unrollbackable* such as screen output, printer output and output to non-transactional persistent resources, or *rollbackable* such as transactional requests to a transactional resource manager.

The operating system may have some *system threads*. System threads and application threads could share resource objects via wrapper objects, but the state of those resource objects is recovery-irrelevant to the application's logic. In such a case, those resource objects are created by system threads. System threads are hosted in *system processes*.

3.2 Nondeterminism Reduction

Sources of nondeterminism include: inputs from outside, asynchronous procedure calls, software or hardware interrupts, transient environment properties, inter thread/process message exchanging via thread queues or network channels, and shared in-memory data access.

The inputs from outside are implemented as input messages to input thread queues via certain hardware interrupts. Asynchronous procedure calls are implemented via sending messages to thread queues in software interrupts. Transient environment properties are understood as input messages returned in function calls via the interface to resource objects. For activities in software or hardware interrupts, we **assume** that applications concern about only messages input by those interrupts (In [5], converting asynchronous signals (Unix equivalents to interrupts) to messages is discussed).

Multi-thread scheduling is not an issue, because our concern is only on input, output, message exchanging and shared data access. If two application threads run simultaneously without any input, output, message exchanging or shared data access, their relative execution order is unimportant.

So all sources of nondeterminism except shared in-memory data access are reduced to message exchanging. In past work all nondeterminism was abstracted only as message exchanging [10]. But the fact is that normally shared in-memory data access is not inherently implemented as message exchanging. The solution in [7], for example, with fictitious threads, is not practical. We believe that shared in-memory data access is an essential element in any realistic model of modern systems. In our model we consider both message exchanging and in-memory data sharing and characterize one access to a shared object as one event, and we abstract their relations into a common dependency graph. **This is the essence of our method.**

For shared in-memory data access, we have an *observation*: in a **Well-Written-Program**, if multithreads access shared application objects or resource objects, the accesses are synchronized via message exchanging or synchronization resources (accesses to queue/set wrappers need not to be synchronized by application programs). *For recovery, logging message receiving and the access order to synchronization resources is enough.* A similar

idea is found in [11]. Thus nondeterminism from shared in-memory data access is reduced to message exchanging and access to synchronization resources. We **assume** Well-Written-Programs in this paper.

3.3 Definition of Consistency

Extending the definition of consistency in message passing systems [10], we get the definition of consistency for our conceptual model:

C_M1 If a receiver has received a message from a sender, the sender should eventually reach a stage with that message having been sent.

C_M2 If the communication channel between a sender and a receiver is reliable, then if the sender has sent a message, that message should be received by the receiver eventually, even in case of failures. Also observed from the channel, messages are received in the sending order.

C_SD All operations on the shared data being redone during recovery should work on the same version of the data as the previous failure-free execution.

C_M1 is violated when a receiver receives a message and continues to run, while the sender of that message rolls back and recovers only up to a point of having not sent the message yet. For an output message, **C_M1** means that once an output message has been sent, the sender should be able to recover to a point of having sent that message no matter what failures have happened to the sender. For a reliable communication channel, **C_M2** is violated when a sender sends a message and gets acknowledged from the receiver, then the sender discards that message, while the receiver fails before logging that message. If the sender is unable to reconstruct the message through redoing, that message will get lost. (For an unreliable channel, message loss because of failures at the receiver is allowed by the nature of the channel's unreliability). **C_SD** means that during recovery, redoing of application threads should basically follow the same access order to the shared in-memory data as the previous failure-free execution, unless the logic of accesses allows different orders.

4 Recovery Framework

We elaborate our recovery framework **Explorer** in this section. We **assume** only "stop-failures" might occur.

The method we use is log-based optimistic recovery with checkpointing. After restart of a failed application process, the most recent checkpoint will be accessed to reestablish the application process, and from there the

application threads will redo operations following the log records which occur after the checkpoint.

4.1 Autonomous Process Group

On each computer, there are three possible types of application processes following [3]:

Recoverable Process (RP) An application process whose in-memory state is brought to a consistent point after a failure.

Non-Recoverable Process (NRP) An application process which doesn't have any recovery procedure after a failure. An NRP doesn't share data with any RP. But it could communicate with an RP via session wrappers or messagebox wrappers.

Transactional-Resource Process (TRP) An application process which manages transactional persistent resources. Examples include database servers and message queue servers. After a failure, a TRP's in-memory state won't be brought to a consistent point, but just be initialized and a data-oriented recovery will be conducted to guarantee the transactional consistency of data. A TRP accepts multiple concurrent transactional requests from RPs or NRPs.

When an application process is created, its type should be specified by the creator explicitly. **This paper focuses only on RPs.** System processes don't do the same recovery as RPs.

Inter thread/process communication can be either *loosely-coupled* or *tightly-coupled*. Loosely-coupled communication includes communication via messagebox resources or session resources. Tightly-coupled communication includes message exchanging via thread queues, and resource object sharing.

Based on the communication types, we define an *Autonomous Process Group (APG)*. An APG is a transitive closure of RPs with tightly-coupled communications. Let A, B and C be three RPs. If A and B have tightly-coupled communication, and B and C have another tightly-coupled communication, then A and C should also be in the same APG. Two RPs in the same APG may also have loosely-coupled communication.

For convenience of presentation, we assume that an APG, along with its first RP, is created only by a system process and that other RPs within that APG are created by the first RP. Once the first RP is created by the system process, all RPs within the APG run independently without interaction with that system process. Before an APG is deleted, all RPs within it should have already ended. If an RP sends a message to a thread queue of another RP in a different APG, to a thread queue of an NRP, TRP,

or even a system thread, or if an RP accesses a resource object which is created by an RP in a different APG or by an NRP or TRP, this should be detected and reported as an error. When an RP is created, its *main application thread* is implicitly created.

In [7], application threads are the rollback units while application processes are the failure units. Taking application threads as the rollback units means that application threads should be checkpointed separately and rolled back separately. Because of the excessive complication in implementation, instead, we will take an APG as both the rollback and failure unit. All RPs within an APG will be checkpointed together with all resource objects created by them, and all RPs within an APG will be rolled back together.

4.2 Virtualization

During failure-free execution, all resource objects and application objects created by RPs must be assigned logical identifiers. During recovery, those resource objects and application objects should be recreated and mapped to the corresponding objects of the previous failure-free execution. This is called *virtualization*.

Virtualization of Objects: Each resource object or application object has a logical ID *ObjectID*, which should be globally unique. Each APG or application thread is also a resource object and their IDs are denoted as APGID and ThreadID, respectively.

During recovery, after a process is reincarnated with a checkpoint, its whole address space is replaced with the one in the checkpoint. Pointers to application objects are unchanged for the new instance of the application process. But each wrapper object must update the handle of the underlying resource object, because a handle may contain the pointer to the resource object in the kernel address space [24] and after the resource object is re-created, its address in kernel address space may have changed. Also if some resource objects or application objects are created after that checkpoint, the handles or pointers during recovery may be different from the previous failure-free execution. Thus it's necessary to ensure that when put into log records, handles or pointers be replaced by the corresponding ObjectIDs. We also **assume** that any handle or pointer is replaced by the corresponding ObjectID when being put into messages (sent via session or messagebox resources or to thread queues).

After a message is received or a log record is read, when required and feasible, the ObjectIDs appearing in the message or the log record can be replaced by handles or pointers which can be reobtained (given ObjectIDs) during the running time.

Virtual Sessions: When the two session resources of a communication channel are located in different APGs,

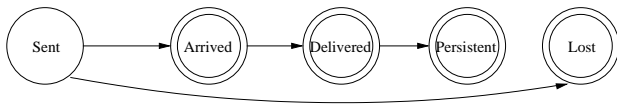


Figure 2: State transitions of a message.

or located in an RP and a TRP respectively, the corresponding session wrappers should maintain a *virtual session*, identified by IDs of the two session resources at both sides. After a virtual session is set up, the ID of the session resource at the other side is logged. When one side is down and comes up later, the session between two sides will be recovered transparently to the application programs.

If the two session resources at both sides are in the same APG, between an RP and an NRP, or between a TRP and an NRP, a *non-virtual session* is maintained instead, where no recoverability is enforced.

4.3 Message Classification

The possible state transitions after a message is *Sent* to a thread queue or over a network channel is shown in Figure 2. States in double circles are end states. A message sent over an unreliable network channel may get *Lost*. A message reaches *Arrived* eventually if it's over a reliable network channel or to a thread queue. The current underlying commercial network layers provide only *Sent-Arrived-Lost* logic. *Delivered* means the application program has read it from the receiving buffer, or from a thread queue, and possibly written its log record into the log stream. *Persistent* means that its log record has been flushed to disk.

For recovery purposes, messages can be divided into four types:

InterAPG Messages: are sent between APGs via session wrappers (virtual sessions) or messagebox wrappers. The log records for them contain the content so that the receiver side APG can get the message from the log records during recovery. Thus message sending during recovery is suppressed.

IntraAPG Messages: are sent within an APG via thread queues, session wrappers (non-virtual sessions) or messagebox wrappers. Log records don't contain the content of messages. During recovery those messages will be resent.

Input Messages: are sent to an APG from a system thread to thread queues, from an NRP via session wrappers (non-virtual sessions) or messagebox wrappers, or from a TRP via session wrappers (virtual sessions), or passed in as transient environment properties. Log records for them contain the content. During recovery those messages will be obtained from the log records directly. If an failure

happens before an input message is *Persistent*, that input message will get *Lost* and thus might need to be re-input.

Output Messages: are sent from an APG to an NRP via session wrappers (non-virtual sessions) or messagebox wrappers, to a TRP via session wrappers (virtual sessions), or to either screen, printers, or disk files via the corresponding wrapper objects. The acknowledgement for an output message may be logged as an input message. If the acknowledgement has been *Persistent*, the output message is not resent during recovery. On the other hand, if the acknowledgement does not reach the persistent log, a duplicate output message is possible.

Without loss of generality, only synchronous message receiving methods (blocked until receiving a message) are considered in this paper.

4.4 Definition of Dependency

During the process of an RP's execution, some *Non-deterministic Events (NEs)* will happen and they should be logged so that recovery could follow the execution path and reach a consistent point after it rolls back or fails. Nondeterministic events include: 1)getting a message from a thread queue, 2)reading a message from the buffer of a session or messagebox resource, 3)making a recovery-relevant access to a synchronization resource, 4)acquiring a transient environment property, 5)creating a resource object or an application object (it's an NE because resource or application objects should be assigned ObjectIDs for virtualization).

We define *directly depend* among NEs:

- D1** An NE X *directly depends* on the most recent NE which happened to the same application thread and precedes X.
- D2** For recovery-relevant access to a synchronization resource: suppose the NE X is an order-sensitive access (e.g. a write lock request), if there exist some order-insensitive accesses (e.g. read lock requests) from other application threads after the latest order-sensitive access, then X *directly depends* on all NEs of those order-insensitive accesses, otherwise, X *directly depends* on the NE of the latest order-sensitive access (if it exists) from a different application thread. Suppose the NE X is an order-insensitive access, then X *directly depends* on the latest order-sensitive access (if it exists) from a different application thread. (Creating a synchronization resource is counted as an order-sensitive access and such an NE doesn't directly depend on any other recovery-relevant access.)

D3 If the NE X is reading a message from the buffer of a session or messagebox resource or from a thread queue, and that message is sent from a different application thread, then X *directly depends* on the most recent NE which happened to the sender and precedes the sending operation.

D4 The first NE of the main application thread of an RP *directly depends* on the NE of creating that RP; the first NE of a non-main application thread of an RP *directly depends* on the NE of creating that application thread.

Since **D1** covers the case for all NEs happening to the same application thread, **D2** and **D3** don't need to cover the special cases such as accesses to a synchronization resource from the same application thread, or the receiver and the sender of a message are the same application thread. When an NE X directly depends on an NE Y , Y is called a *direct predecessor* of X .

Depend among NEs is defined as the transitive closure of the directly depend relation. If an NE X depends on an NE Y , Y is called a *predecessor* of X . If the log record of X is persistent but Y 's is not when some failure happens so that the log record of Y gets lost, then after recovery X becomes an *orphan*. A message becomes an orphan when the NE right preceding the sending operation becomes an orphan.

Figure 3 is an example showing the *directly depend* relations. NE0, NE1 and NE3 are NEs of application thread T1; NE2 is the NE of receiving M1 from a thread queue of T2, M1 is sent after NE1; NE4 is the NE of receiving M2 via a session resource, M2 is sent after NE3; NE5 and NE8 are order-sensitive (denoted by W) and NE6 and NE7 are order-insensitive (denoted by R) accesses to the same synchronization resource. Figure 3(b) is the graph of the directly depend relations. The arrow from NE x to NE y shows that NE y *directly depends* on NE x .

4.5 Dependency Tracking

The APG is both the rollback and failure unit. Dependency across APGs is tracked by a transitive dependency tracking method, and dependency within an APG is tracked by a direct dependency tracking method. Since both message exchanging and synchronization resources are used to synchronize the access to shared in-memory data, it is required that during redo, all application threads follow the dependency order within an APG to reconstruct the state of shared in-memory data.

Each application thread's execution can be in one of two different modes: failure-free mode or recovery mode. After recovery is done, all application threads will convert from recovery mode to failure-free mode. To implement Explorer, we must extend methods of

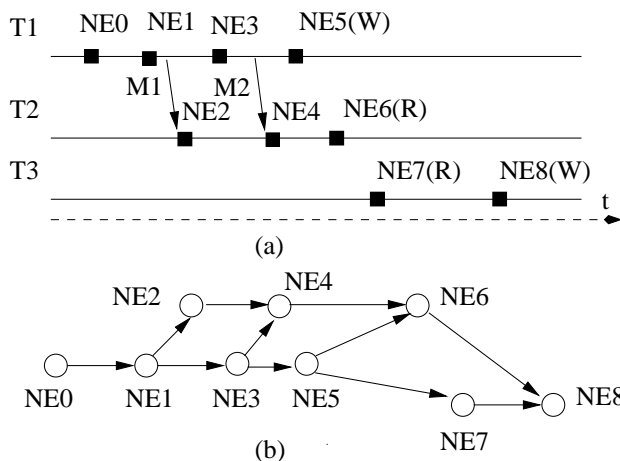


Figure 3: A sample of dependency: (a)NEs happening to application threads T1, T2, and T3; (b)is the corresponding graph showing *directly depend* relations.

wrapper objects to include logging and recovery functions. Thus each method of a wrapper object may have three versions: a failure-free-version, a recovery-version and an unextended-version. The failure-free-version is used during the failure-free mode execution, the recovery-version is used during the recovery mode execution, while the unextended-version is the original one. Recovery-irrelevant access to synchronization resources doesn't have failure-free-versions or recovery-versions. For wrapper objects created by system threads, only the unextended-versions are used. Basically a failure-free-version first does its work then makes a log record, and a recovery-version first reads the log record then does its work. NEs are always associated with methods in wrapper objects. All dependency tracking and preserving presented in this section are done by synchronization wrappers (access to synchronization resources), queue-set wrappers (access to thread queues), and session or messagebox wrappers (access to session or messagebox resources).

4.5.1 Data Structures

Each computer has a system process, called the *Distributed Recovery Manager(DRM)*. The DRM is in charge of the management related to logging and recovery. DRMs on different computers have reliable channels to communicate with each other. Underlying network communication should allow efficient broadcast. When an APG or an application thread begins or ends, the DRM will be notified. The following data structures are maintained in the DRM. *For each APG on the computer,*

- **Log Stream (LS)** A separate log stream shared by all application threads within the APG. Log records in LS have identifying numbers called *Log Sequence Numbers (LSNs)*, which are monotonically increas-

ing integers. The LS has a current *Maximum LSN* (**MLSN**). Each log record contains the ThreadID of the thread it is associated with.

- **Epoch (E)** Every time an APG recovers from the crash of the computer on which it is located, its current epoch will be incremented by 1 (the initial number is 1).
- **Group State (GS)** the current MLSN of the APG.
- **Group Dependency Vector (GDV)** A variable length vector of (apgid, (e,gs)) pairs indexed on apgid which shows the current dependency of this APG on other APGs. We use uppercase (APGID, E, GS) to refer to this APG and lower case (apgid, e, gs) to refer to other APGs. Note $GDV[apgid] = (e,gs)$, $GDV[APGID] = (E,GS)$. When there is no entry for apgid, $GDV[apgid] = null$.

For each application thread of an APG on the computer,

- **Log Buffer (LB)** A separate log buffer which is a logical part of the LS. Flushing log records follows the order of consecutive LSNs.
- **Thread State (TS)** The LSN of the log record for the most recent NE of this thread.
- **Thread Dependency Vector (TDV)** A variable length vector of (apgid, (e,ts)) pairs indexed on apgid which shows the current dependency of this thread on other APGs after the most recent NE of this thread gets logged. Note $TDV[apgid] = (e,gs)$, $TDV[APGID] = (E,TS)$. When there is no entry for apgid, $TDV[apgid] = null$.

For the whole system (and duplicated on each computer),

- **Maximum Recovered State History (MRSH)** A two dimensional **persistent** variable length vector indexed on apgids and epochs. $MRSH[apgid,e] = gs$ indicates that the maximum recovered state of APG (apgid) at epoch e is gs. When there is no entry for (apgid,e), $MRSH[apgid,e]=null$, indicating it's not yet known. This information is maintained for all finished epochs of all APGs in the system.
- **Persistent State Vector (PSV)** A **persistent** variable length vector of (apgid, gs) pairs indexed on apgid showing the LSN of the latest log record of APG (apgid), which the DRM has known to be persistent. When there is no entry for apgid, $PSV[apgid]=null$, indicating it's not yet known. This information is maintained for all APGs in the system.

Following the above definitions, both *state dependency* and *log record dependency* mentioned in later sections refer to the corresponding dependency of NEs, and *state flushing* refers to the corresponding log record flushing.

4.5.2 Dependency Tracking within an APG

Dependency within an APG comes from **D1**, **D2**, **D4**, and the subcase of **D3** for intraAPG messages. Neither **D1** nor **D4** needs explicit tracking, because they are always observed during recovery. For **D2** and that subcase of **D3**, basically all direct predecessors' thread identities and states are logged.

When an NE X, which is *not* an NE of message receiving from other APGs, is to be logged, the DRM increments the current GS by 1, and assigns the incremented GS as the LSN of the log record. The log record is appended to the LB of the thread. Meanwhile the TS of the thread is updated to the incremented GS. Let **EDV** (Event Dependency Vector) be a variable length vector of (ThreadID, State) pairs indicating the thread identities and states of all direct predecessors of X. EDV is in the log record of X.

In the following, we use the convention that R appended to a notation indicates order-insensitive accesses (intuitively, Reads) and W appended to a notation indicates order-sensitive accesses (intuitively, Writes). For **D2**, if a synchronization resource is created by an RP, it maintains (ISW, SetofISR), where $ISW=(ThreadIDW, StateW)$ indicates the identity and the state of a thread which issued the most recent order-sensitive access to this synchronization resource; SetofISR is a set of (ThreadIDR, StateR) pairs, indicating the identities and the states of threads which issued order-insensitive accesses after the most recent order-sensitive access.

When a pair is inserted into SetofISR, if there is another pair with the same ThreadIDR in the set, only the pair with the larger StateR will be kept. The operating system has a function to get (ISW, SetofISR) given the synchronization resource's ID:

(ISW,SetofISR) **GetISWR(ObjectID)**

and other functions to update each element of (ISW, SetofISR). Failure-free-version accesses from a thread (ThreadID) to a synchronization resource (ObjectID) are extended as follows (in pseudo-code):

Order-insensitive access

enter a critical section

{ (ISWt,SetofISRt) := GetISWR(ObjectID);

log ISWt; //ISWt ∈ EDV

for the synchronization resource (ObjectID):

add (ThreadID,TS) into SetofISR;

}

do order-insensitive access;

Order-sensitive access

enter a critical section

{ (ISWt,SetofISRt) := GetISWR(ObjectID);

if(SetofISRt = Φ) log ISWt; //ISWt ∈ EDV

else log SetofISRt; //SetofISRt = EDV

for the synchronization resource (ObjectID):

empty SetofISR;

```

    update ISW with (ThreadID,TS);
}
do order-sensitive access;

```

The above **enter a critical section** means that for all accesses to the same synchronization resource, only one application thread can be running inside those critical sections (in curly brackets) at one time.

For the subcase of **D3** for intraAPG messages, before sending, the (ThreadID, TS) of the sender thread is attached to the message. After a message is received, the (ThreadID, TS) of the sender thread will be logged (EDV={ (ThreadID,TS) }). Depending on the specific implementation, when a message is received from a thread queue, the identity of the thread queue (denoted by TQID) may be logged too. After logging, the message (with (ThreadID, TS) detached) is returned to the caller.

4.5.3 Dependency Tracking between APGs

Dependency between APGs comes from the subcase of **D3** for interAPG messages. Before sending, the TDV of the sender thread is attached to the message. When a message is received, if there exists an apgid (x), such that $TDV[x]=(e,gs)$, $MRSH[x,e]=gs'$, $gs' \neq null$ and $gs > gs'$, then this message is an orphan and thus discarded, otherwise, accepted.

After a message is accepted, both the TDV of the sender thread and the message content will be logged. For the logging operation, the DRM increments current GS by 1 and assigns the incremented GS as the LSN of the log record. The log record is appended to the LB of the receiver thread. Meanwhile the TS of the receiver thread is updated to the incremented GS. After that, the GDV of the APG is updated as $\max(\text{current GDV}, \text{TDV in the message})$ and the TDV of the receiver thread is updated as $\max(\text{TDV of the receiver thread}, \text{TDV in the message})$. We define $\max(DV1, DV2)$ as follows. Let $DV3 = \max(DV1, DV2)$, then for any id , $DV3[id] =$

$$\begin{cases} DV2[id] & \text{if } DV1[id] = \text{null} \\ DV1[id] & \text{if } DV2[id] = \text{null} \\ & \text{and } DV1[id] \neq \text{null} \\ \max(DV1[id], DV2[id]) & \text{otherwise} \end{cases}$$

where $\max((E1, S1), (E2, S2)) =$

$$\begin{cases} (E1, S1) & \text{if } E1 > E2 \\ (E2, S2) & \text{if } E1 < E2 \\ (E1, \max(S1, S2)) & \text{if } E1 = E2 \end{cases}$$

After logging, the message (with TDV detached) is returned to the caller.

4.5.4 Log Flushing Rule

When an output message is to be sent by an application thread, or when the LB of an application thread is

short of space, the application thread will flush its current TS. Before this can occur, log records of all predecessors of the current TS must be persistent. Thus before flushing the current TS, for each apgid (different from the APGID of this APG) appearing in the TDV of this thread, if $PSV[apgid]$ is null or less than gs , where $TDV[apgid]=(e,gs)$, the DRM of that APG (apgid) will be requested to flush that APG's state gs .

Right before making a checkpoint, the DRM will flush this APG's current GS. Before that, for each apgid (different from the APGID of this APG) appearing in the GDV of this APG, if $PSV[apgid]$ is null or less than gs , where $GDV[apgid]=(e,gs)$, the DRM of that APG (apgid) will be requested to flush that APG's state gs .

When a DRM receives a request from the DRM of a different APG (APG1), to flush a certain state $S2$ of an APG (APG2), it needs to flush all APG2's log records with their LSNs no greater than $S2$. $S2$ of APG2 may depend on a third APG (APG3)'s state ($S3$), but the DRM of APG3 doesn't need to be contacted by the DRM of APG2 to flush APG3's $S3$. Because of *transitive dependency tracking*, currently APG1 must depend on APG3's certain state Sx , which is no earlier than $S3$, and the DRM of APG3 will be contacted by APG1's DRM to flush APG3's Sx , if Sx is not yet known to be persistent by the DRM of APG1.

When a state S of an APG is requested to be flushed for no matter what reason, all log records of this APG with their LSNs no greater than S must be moved out from LBs following the order of consecutive LSNs and written into a single log stream on disk. However, before flushing a log record R corresponding to receiving an interAPG message, the sender state for R must be persistent. Suppose the TDV recorded in R shows that $TDV[\text{senderAPGID}]= (e,gs)$. Then flushing R is blocked until $PSV[\text{senderAPGID}]$ becomes no less than gs . Before blocking the flush of R , if some earlier log records of this APG have been flushed (but this has not yet been disseminated to other DRMs), this DRM actively notifies other DRMs to update their PSVs. This DRM also notifies other DRMs after S is flushed.

The flushing order of log records reflects all dependency orders among all NEs in the whole system (all APGs), so **all persistent log records are orphan-free**. No disk flushing operations will be wasted. After a checkpoint of an APG is flushed, all persistent log records before that checkpoint can be truncated.

4.6 Dependency Preserving Recovery

After a computer crashes, at restart, the DRM first fetches the MRSH and PSV from the disk, then begins to recover all active APGs on this computer. Each recovered APG will get a new epoch. Upon finishing recovery

of all active APGs, the DRM broadcasts a *recovery message* which contains the maximum recovered state of the previous epoch for each recovered APG. Because of reliability of the channels among DRMs, the recovery message for any previous crash of the computer will reach other DRMs earlier than this one.

Upon receiving a recovery message, a DRM updates its MRSH. Then for each APG (apgid) on the computer, the DRM checks to see if an orphaned NE has happened to it: if there exists an APGID (x) and its most recent epoch e in the recovery message, such that $GDV[x] = (e, gs)$ and gs is greater than $MRSH[x, e]$, then the current state of the APG (apgid) is orphaned, so the APG (apgid) will be rolled back (similar to [8]).

Before rolling back an APG, the active application processes within it are stopped after all active logging operations within them are complete. The log buffers of an APG are located in the DRM. Before rollback, the DRM saves all current content of those log buffers to another location for later recovery.

When an application process of an APG fails without a computer crash, all other application processes of the APG will also be forced to rollback. Although the failure of an application process may cause the most recent log records of some application threads to be lost if those logging operations are still going on when the failure happens, all available log records (persistent + possible volatile) are orphan-free, because (1) the logging operation of an NE begins only after the logging operations of all direct predecessors have finished and (2) all persistent log records are orphan-free.

So the recovery of an APG is triggered by one of three sources: (1) *computer-crash* (the computer on which it is located crashes), (2) *orphan-rollback* (the DRM finds an orphan for it after receiving a recovery message) and (3) *force-rollback* (an application process within it fails). To recover an APG, log records for all its application threads after the latest checkpoint are extracted by a system thread in the DRM from the persistent log stream to the log buffers (It can be done in parallel with the reincarnation from the checkpoint), then any saved volatile log records (only for the rolled back APG) are appended to the corresponding log buffers. After that, all application threads will redo following the log records in the dependency order, not in the order of consecutive LSNs.

During recovery, when an application thread comes across an NE, if the log record of the NE was once persistent, or if the recovery of the current APG results from force-rollback, it is unnecessary to do orphan checking. We use "OrphanCheckNotRequired" to refer to this condition in later pseudocode.

For an application thread, conversion from recovery mode to failure-free mode is either right after reading a log record and finding the NE is an orphan, or, if there

are no orphaned NEs, right before coming to the first NE which needs to write a log record after recovery. The operating system has the functions to get or set the execution mode of an application thread:

ExeMode **GetExeMode**(*ThreadID*) and
Void **SetExeMode**(*ThreadID*, *ExeMode*).

So the recovery-version of a method in a wrapper object begins with:

```
if (there is no log record for current method)
{ SetExeMode(currentThreadID, failure-free);
  execute the failure-free-version and return;
} else ...
```

Later we present only the pseudocode for the case when there is still a log record for the current method. To guarantee that new activities involving wrapper objects don't interfere with activities which are recorded in log records, which should be redone before new activities begin, the function **SetExeMode**(*ThreadID*, failure-free) will set the mode to failure-free immediately, but won't return until all active application threads of the APG have converted to failure-free mode.

4.6.1 Dependency Preserving between APGs

For interAPG messages, both session wrappers and messagebox wrappers have a synchronous receiving method:

Message **ReadNextMsg**().

Suppose LSN' is the LSN of the log record for the current method, TDV' is the message's dependency vector recorded in the log record being read, GDV and GS are those of the current APG, TDV and TS are those of the receiver thread, and $MRSH$ is the current one of the system. The content of a non-orphaned message is acquired from the log record. The recovery-version of **ReadNextMsg**() is extended as follows:

```
if (not OrphanCheckNotRequired)
if (there exists an apgid (x), s.t.  $TDV'[x] = (e, gs)$ ,
   $MRSH[x, e] = gs'$ ,  $gs' \neq null$  and  $gs > gs'$  )
{ //current NE known to be an orphan
  SetExeMode(currentThreadID, failure-free);
  msg := executing failure-free ReadNextMsg();
  return msg;
}
TS :=  $LSN'$ ; GS :=  $\max(GS, LSN')$ ;
GDV :=  $\max(GDV, TDV')$ ; TDV :=  $\max(TDV, TDV')$ ;
return the message (with  $TDV'$  detached);
```

4.6.2 Dependency Preserving within an APG

To follow the dependency within an APG during recovery, the redo of all application threads in the APG satisfies: An NE doesn't happen until all direct predecessor NEs have happened. The operating system has a function to get the current state of an application thread: *TS* **GetTS**(*ThreadID*). If an application thread ends, its ending thread state (TS) will be kept until the end of

the recovery. Suppose LSN' is the LSN of the log record for the current method, EDV (information for all direct predecessors) is from the log record, TS is the thread state of the current thread, and GS is the group state. The recovery-version of a method (*not* message receiving from other APGs) of wrapper objects (RVWO) is extended as follows:

```

while (True)
{ if (every entry (ID,S) in EDV satisfies
    GetTS(ID) ≥ S)
    //direct predecessors have happened
    { TS:=LSN'; GS:=max(GS,LSN');
    do its work and return;
    }
else if (OrphanCheckNotRequired) yield();
else if (there exists an entry (ID,S) in EDV,
    s.t. GetTS(ID) < S and
    GetExeMode(ID) == failure-free)
    { //current NE known to be an orphan
    SetExeMode(currentThreadID, failure-free);
    execute the failure-free-version and return;
    }
else yield();
}

```

Yield in the above algorithm description means that the current thread gives up its execution right, with its status changed from "Running" to "Ready". It's possible for a thread to be scheduled from "Running" to "Ready" several times before it begins to do its work. But it's not expected to happen often.

Now let us examine some special cases of RVWO. For *order-insensitive access*, the EDV contains only ISW (from the log record), and **its work** contains: add (ThreadID,TS) of the current thread into SetofISR of the synchronization resource, then do order-insensitive access. For *order-sensitive access*, if SetofISR (from the log record) is empty, the EDV contains only ISW (from the log record), otherwise, the EDV is SetofISR (from the log record); and **its work** contains: empty SetofISR of the synchronization resource, update ISW of the synchronization resource with (ThreadID,TS) of the current thread, then do order-sensitive access.

A synchronization wrapper can refer to *multiple synchronization resources*. For that, the RVWO algorithm is extended so that: 1) the current thread reports orphans and converts to failure-free execution once an orphan is found through any of the underlying synchronization resources; 2) when no orphan has been found yet, the current thread will yield unless all direct predecessors (caused through all the underlying synchronization resources) have happened.

For *receiving an intraAPG message*, the EDV in the log record contains the (ThreadID, TS) of the sender thread.

Session resources and messagebox resources are shared

resource objects. According to our assumption for Well-Written-Programs, during recovery, both sending and receiving messages through them will follow the same order as failure-free execution. So **its work** for receiving an intraAPG message via a session or messagebox resource contains: read next message via the session or messagebox resource synchronously, then return the message (with (ThreadID, TS) detached).

Sending messages by different application threads to a thread queue is not synchronized and may not follow the same order as failure-free execution. Thus during recovery, the messages may appear in the thread queue in different order from the failure-free execution. But messages sent from a specific application thread to a thread queue are in the same order as during failure-free execution. The operating system has a function get the next message sent from a specific application thread via a specific thread queue synchronously, no matter whether that message is at the head of the queue or not:

Message GetNextMsgByTID(TQID, ThreadID).

So **its work** for receiving a message from a thread queue (TQID) contains: read next message via GetNextMsgByTID(TQID, ThreadID), where ThreadID is the identity of the sender thread, then return the message (with (ThreadID, TS) detached).

4.7 Optimized Dependency Tracking

The dependency within an APG can be tracked without logging the state of the thread with which a direct predecessor NE is associated. Before the recovery of an APG finishes, if an application thread ends, its ending thread state (TS) doesn't need to be kept. So the overhead of dependency tracking within an APG is reduced. This is important because interactions within an APG occur more often than interactions among APGs.

4.7.1 Versioned Synchronization Resources

Each order-sensitive access to a synchronization resource results in a new version. Each order-insensitive access is always upon certain version of the synchronization resource. Each synchronization resource created by an RP maintains a (V,C,IDW,SetofIDR), where V is its current version, C is the number of order-insensitive accesses on the current version, IDW is the ThreadID of the application thread whose order-sensitive access results in the current version, and SetofIDR is a set of ThreadIDs of the application threads which once issued order-insensitive accesses on the current version. When a synchronization resource is created, its version is 0 and IDW is the ThreadID of the creator. The operating system has functions to get (V,C) or (V,C,IDW,SetofIDR) of a synchronization resource given its ID: (V,C) **GetVC(ObjectID)**,

(V,C,IDW,SetofIDR) **GetFullVC(ObjectID)**,

and other methods to update each element of $(V, C, IDW, SetofIDR)$.

Each order-insensitive access to a synchronization resource needs to log the current (V, IDW) . During recovery, an order-insensitive access can't make the actual order-insensitive access (thus will yield, unless the current access is found to be an orphaned NE) until the synchronization resource has reached the expected version (V) . Each order-sensitive access needs to log (V, IDW) if C is 0, otherwise, $\log(V, C, SetofIDR)$. During recovery, an order-sensitive access can't make the actual order-sensitive access (thus will yield, unless the current access is found to be an orphaned NE) until the synchronization resource has reached the expected version (V) and the number of order-insensitive accesses on that version (V) has reached the expected number (C) .

4.7.2 Message Peeking for Message Receiving

For an intraAPG message, only the ThreadID of the sender thread is attached and logged by the receiver. Each session or messagebox resource has a method to check if the next message is available for reading: *Bool NextMsgReady()*. The operating system has a function to check if the next message sent from a specific application thread (ThreadID) via a thread queue (TQID) is available, no matter whether it is at the head of the queue or not: *Bool NextMsgByTIDReady(TQID, ThreadID)*.

When orphan-checking is not required, the recovery-version of the message receiving method just does the actual synchronous message receiving (for receiving from a thread queue, *GetNextMsgByTID(TQID, ThreadID)* is used). When orphan-checking is required, if the next message is not ready yet (for receiving from a thread queue, the message readiness is checked by *NextMsgByTIDReady(TQID, ThreadID)*), the application thread has to yield, unless the current message receiving is found to be an orphaned NE.

The IDW or SetofIDR of the synchronization resource or the ThreadID of the sender thread are logged to enable orphan-checking during recovery. The current NE (access to a synchronization resource or message receiving) is found to be an orphan if some direct predecessors haven't happened (determined by version checking or message readiness checking), and all application threads with which the direct predecessors are associated have converted to failure-free mode or have exited.

4.8 Other Aspects of Explorer

Due to restrictions on the size of the paper, we only outline other aspects of Explorer:

Virtual Session Maintenance To maintain a virtual session, each session wrapper has a buffer (located in the ad-

dress space of application processes) for messages having been sent. The buffered message will be discarded when that message has been delivered at the receiver and its log record has become persistent. It may be resent if it is found to have got lost because of failures at the receiver. A similar idea is presented in [3]. During recovery, messages resent over a virtual session are not sent out to the network, but will be buffered anyway.

Checkpointing A checkpoint of an APG contains all resource objects it created (including buffers used in session or messagebox resources), all its thread queue sets, necessary data in the address space of all its application processes (including buffers used in session wrappers for virtual sessions), and all data structures for recovery maintained for it in the DRM.

A checkpoint must not be made when any current application thread is running in a failure-free-version of a method of a wrapper object. When an APG is being checkpointed, none of its application threads is available for scheduling. They are in special scheduling states, for which the thread scheduler of the operating system is extended. Checkpoints are made periodically after recovery has finished. Upon the finishing of recovery, a checkpoint request is also issued. Flushing a checkpoint can be finished concurrently with the continuous execution of the APG (similar to concurrent checkpointing in [23]).

When reincarnating an APG from its latest checkpoint, for all wrapper objects referencing resource objects in the kernel address space, the handles are updated to refer to the new created ones (for virtualization) after recovery. To address and access all wrapper objects of an application process, support from compilers is required.

Interaction with TRPs Because of the rollbackability of a transaction, among all output messages sent to a TRP, only when "Commit" is to be sent, will the current state of the application thread be flushed. Active transactions when an application process fails or is terminated will be aborted actively by the client side. These operations are done in transaction wrappers.

4.9 Correctness Claim

Under the assumption of Well-Written-Programs, **C_M1** and **C_SD** in section 3.3 are guaranteed by enforcing the dependency among NEs during both failure-free and recovery executions (including flushing the current thread state before sending output messages during failure-free execution).

Dependency across APGs: Transitive dependency among APGs is tracked in transitive dependency vectors. It's a traditional optimistic recovery method and has been shown to be correct [28, 8].

Dependency within an APG: Direct dependency within an APG is tracked. During recovery, all application threads redo following the direct dependency orders: an NE doesn't happen until all predecessors have happened. Whether a predecessor has happened or not is determined by checking if the state of the thread, with which the predecessor is associated, has reached an expected state, or checking if the synchronization resource has reached the expected version (and the expected number of reads have happened), or checking if a message is ready for reading by message peeking. If a predecessor hasn't happened yet, but the thread with which it is associated has converted to failure-free mode or has exited, then that predecessor is known to be an orphan, thus this NE is an orphan.

Based on the assumption of the reliability of message exchanging on the same computer, and on the fact that during recovery, all message exchanging within an APG needs to be redone, exactly-once ordered delivery stated in **C_M2** is reduced to ensuring only the reliability of message exchanging across APGs, which is ensured by virtual session maintenance.

In addition, two special issues are clarified here:

Dependency Vector Calculation during Recovery: During failure-free execution, operations follow the LSN order. While during recovery, application threads don't follow the order of consecutive LSNs, but only the order of dependency. However the GDV of the APG and the TDVs of all application threads will be calculated correctly. For the GDV, after all non-orphaned NEs are executed, every entry except the one for itself is the **maximum** of the corresponding entries in the dependency vectors attached in all messages sent from other APGs to *this APG*. As to the entry for itself, suppose $GDV[\textit{itself}] = (E, S)$, then E is the latest epoch number and S is the **maximum** LSN of all non-orphaned NEs of *this APG*. For the TDV of an application thread, after all non-orphaned NEs are executed, every entry except the one for itself is the **maximum** of the corresponding entries in the dependency vectors attached in all messages sent from other APGs and received by *this thread*. As to the entry for itself, suppose $TDV[\textit{itself}] = (E, S)$, then E is the latest epoch number and S is the **maximum** LSN of all non-orphaned NEs of *this thread*. These are always "maximum" operations. So the order of calculating those dependency vectors doesn't have to be LSN order.

Probability of Finding a Time to Make a Checkpoint: A checkpoint is made only when all application threads of the APG are running outside of the failure-free-version methods of wrapper objects. To get a sense for the chance of making checkpoints for an APG, we do a rough estimation.

Suppose there are **m** application threads in the APG

and each application thread runs in a failure-free-version method of wrapper objects once every **n** time slots. The probability(Pr) of *failure to find a time to make a checkpoint during n consecutive time slots* is

$$\frac{\sum_{(x_1, \dots, x_n)_s} \binom{m}{x_1} \cdot \binom{m-x_1}{x_2} \cdot \dots \cdot \binom{m-x_1-\dots-x_{n-1}}{x_n}}{n^m}$$

where each (x_1, \dots, x_n) satisfies:

$$\begin{cases} x_1 + \dots + x_n = m \\ 1 \leq x_i \leq m - n + 1 (1 \leq i \leq n) \end{cases}$$

When **m** is less than **n**, Pr is 0. When **m** equals to **n**, Pr is $\frac{n!}{n^n}$ and along with increase of **n**, Pr will roughly decrease. When operations involving wrapper objects become frequent (**n** becomes smaller), there is less chance to make a checkpoint.

5 Conclusion

This paper advances a conceptual model to characterize modern commercial platforms and provides a complete recovery framework for that model. This framework requires support from the operating system (resource objects and thread scheduler, etc.), the class library exposing the functions of the operating system (wrapper objects), application programs (Well-Written-Programs assumption), and the compiler (addressing all wrapper objects of an application process when it is reincarnated from a checkpoint, etc.). While designing this recovery framework, we kept in mind the feasibility and efficiency of implementation, and thus we believe our work is a realistic step towards implementing optimistic recovery in current commercial platforms.

Currently only "stop-failures" are considered, but "propagation-failures" (when a failure happens to an application process, the process doesn't stop immediately, but continues to run and may corrupt data) are possible [20]. [30] suggests a way of execution retry to bypass software faults based on checkpointing, rollback, message reordering and replaying, but only for a message passing system. How to design a recovery framework to accommodate automatic retry or other methods of bypassing propagation failures for our conceptual model is one direction for future work.

References

- [1] Lorenzo Alvisi, Bruce Hoppe, and Keith Marzullo. Nonblocking and orphan-free message logging protocols. In *FTCS*, 1993.
- [2] David A.Solomon and Mark E. Russinovich. *Inside Windows 2000 3rd edition*. Microsoft Press, 2000.

- [3] Roger Barga, Dave Lomet, and Gerhard Weikum. Recovery guarantees for general multi-tier applications. In *ICDE Conference*, pages 543–554, March 2002.
- [4] Anita Borg, Jim Baumbach, and Sam Glazer. A message system supporting fault tolerance. In *SOSP-9*, 1983.
- [5] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under unix. In *ACM Transaction on Computer Systems*, pages 1–24, 1989.
- [6] P.Emerald Chung, Woei-Jyh Lee, Y. Huang, D. Liang, and Chung-Yih Wang. Winckp: a transparent checkpointing and rollback recovery tool for windows nt applications. In *FTCS*, June 1999.
- [7] Om P. Damani, Ashish Tarafdar, and Vijay K. Garg. Optimistic recovery in multi-threaded distributed systems. In *SRDS*, pages 234–243, 1999.
- [8] O.P. Damani and V.J.Garg. How to recover efficiently and asynchronously when optimism fail. In *ICDCS-16*, May 1996.
- [9] E.N.(Mootaz) Elnozahy and W. Zwaenepoel. Manetho:transparent rollback-recovery with low overhead, limited rollback and fast output commit. In *IEEE Transactions on Computers*, pages 526–531, May 1992.
- [10] E.N.(Mootaz)Elnozahy, Lorenzo Alvisi, Yi min Wang, and David B.Johnson. A survey of rollback-recovery protocols in message-passing systems. In *ACM Computing Survey*, volume 34, pages 375–408, Sept. 2002.
- [11] Arthur Goldberg, Ajei Gopal, Kong Li, Rob Strom, and David F. Bacon. Transparent recovery of mach applications. In *USENIX-1 Mach Workshop*, 1990.
- [12] Jim Gray. Why do computers stop and what can we do about it. In *ICRDD-6*, June 1987.
- [13] Object Management Group. *Fault Tolerant CORBA Specification*. <http://cgi.omg.org/cgi-bin/doc?ptc/00-04-04>, 2000.
- [14] J.H.Slye and E.N.Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. In *FTCS-26th*, pages 250–259, 1996.
- [15] David B. Johnson. Efficient transparent optimistic rollback recovery for distributed application programs. In *SRDS-12*, October 1993.
- [16] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In *FTCS-7*, 1987.
- [17] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *J. of Algorithms*, 1990.
- [18] David Lomet and Gerhard Weikum. Efficient transparent application recovery in client-server information system. In *SIGMOD Conference*, pages 460–471, June 1998.
- [19] Hong Va Long and Divyakant Agrawal. Using message semantics to reduce rollback in optimistic message logging recovery schemes. In *ICDCS*, 1994.
- [20] David E. Lowell, Subhachandra Chandra, and Peter M. Chen. Exploring failure transparency and the limits of generic recovery. In *OSDI*, 2000.
- [21] Microsoft. *.NET Framework SDK Reference*. <http://msdn.microsoft.com/library/default.asp>, 2003.
- [22] J.Roger Mitchell and V.J.Garg. A non-blocking recovery algorithm for causal message logging. In *SRDS*, 1998.
- [23] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under unix. In *USENIX*, January 1995.
- [24] Jeffrey Richter. *Advanced Windows 3rd edition*. Microsoft Press, 1997.
- [25] A.Prasad Sistla and Jennifer L. Welch. Efficient distributed recovery using message logging. In *SPDC-8*, pages 223–238, August 1989.
- [26] Sean W. Smith and David B. Johnson. Minimizing timestamp size for completely asynchronous optimistic recovery with minimal rollback. In *SRDS*, pages 66–75, 1996.
- [27] Sean W. Smith, David B. Johnson, and J.D.Tygar. Completely asynchronous optimistic recovery with minimal rollbacks. In *FTCS-25*, 1995.
- [28] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. In *ACM Transactions on Computer Systems*, pages 204–226, August 1985.
- [29] Yi-Min Wang and W. Kent Fuchs. Optimistic message logging for independent checkpointing in message-passing systems. In *SRDS-11*, 1992.
- [30] Yi-Min Wang, Yennun Huang, and W. Kent Fuchs. Progressive retry for software error recovery in distributed systems. In *FTCS-23*, 1993.
- [31] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung, and Chandra Kintala. Checkpointing and its applications. In *FTCS-25*, 1995.