

# Design Patterns for Log-Based Rollback Recovery \*

Titos Saridakis

NOKIA Research Center  
PO Box 407, FIN-00045, Finland  
`titos.saridakis@nokia.com`

## Abstract

Log-based rollback recovery builds on the ideas of checkpoint-based rollback recovery and improves the characteristics of the recovery process. The basic idea capture by the log-based rollback recovery techniques is an extension of the checkpoint idea: in addition to checkpoints, the system also keeps logs about the non-deterministic events that happened after a checkpoint was taken. If an error occurs, the system rolls back to the last checkpoint and then replays the events saved in the logs to move its execution as close as possible to the occurrence of the error. This paper presents four design patterns that capture the most widely used methods for log-based rollback recovery. The first pattern captures the general idea behind log-based rollback recovery and the following three patterns describe specific solutions about *when*, *where* and *how* to keep logs.

## 1 Introduction

Rollback recovery has been one of the most widely used means for system recovery in the occurrence of errors. The basic idea behind it is to model the system execution as a succession of system states and, when an error occurs while the system is reaching some state, to roll the system back to a previously reached state and resume execution from there.

Commonly used techniques for rollback recovery are based on checkpoints: the system saves in stable storage some of the states it reaches during its execution. The saved states are called *checkpoints* and the action is called *checkpointing* or *taking a checkpoint*. The system recovery techniques that are based on taking checkpoints and, after an error is detected, restoring a legitimate system state from the checkpoints previously taken, are qualified as *checkpoint-based rollback recovery* [10].

System recovery techniques qualified as *log-based rollback recovery* improve the characteristics provided by checkpoint-based rollback recovery. In addition to checkpoints, these techniques also log the communication events that happen after a checkpoint. After an error, the system restores its state to a previously taken checkpoint and then replays all the logged events in order to move its execution as close as possible to the occurrence of the error.

---

\*Copyright 2003 © by NOKIA. All rights reserved. Permission is granted to copy for VikingPLoP 2003.

This paper presents four design patterns that capture the most widely used methods for log-based rollback recovery. First, the **Log Keeping** pattern captures the general log-based rollback recovery idea. The **Optimistic Logging** pattern describes the method that does not block the execution of a system while saving the logs, at the expense of a longer recovery process in case of an error. The **Pessimistic Logging** pattern describes the opposite logging method: system execution is blocked until the logs are saved, for the benefit of shorter recovery process in case of an error. Finally, the **Causal Logging** pattern captures a hybrid method in an attempt to combine the benefits of the above two methods in terms of costs incurring to the system executions with and without errors.

## 2 Background

### 2.1 System Model

A *system* is an entity with a well-defined behavior in terms of output it produces and which is a function of the input it receives, the system logic and the passage of time as observed by the system's internal clock. By "well-defined behavior" that the output produced by the system is previously agreed upon and unambiguously distinguishable from output that does not qualify as well-defined behavior. The well-defined behavior of a system is called the system *specification*. A system interacts with its environment by receiving input from it and delivering output to it. A system can be decomposed into constituent (sub)systems, often called system *components*, each component being a system of its own. As such, it interacts with its environment (i.e. other components of the bigger system) by receiving input and delivering output to it, and it can be further decomposed into its constituent (sub)systems.

A system is modeled as a state machine, where states contain the data that the system holds and operates on. State transitions are classified in two categories: those which are triggered by the internal system logic and are not immediately observable by the environment, and those which are triggered by events that are directly observable by the environment of the system. Examples of state transitions in the first category are manipulations the data that a system possesses, e.g. transferring data from memory to registers and back, re-arranging buffers and data-lists in the memory and modifying their content. Examples of state transitions in the second category are all kinds of interactions of a system with its environment, e.g. the reception of a message or the emission of a message and anything else that can be modeled as an I/O operation of the system. Hence, the generation of the content of a message is the result of one or more state transitions in the first category, but the emission of the message to the system's environment is a state transition in the second category.

The state transitions in the second category capture *interaction events*, or simply *events*, that can be perceived by an observer external to the system. Obviously, events model the emission and reception of messages or signals, the delivery and the interception of data and content, and any other kind of interaction between a system and its environment (e.g. moving a robot's arm, flashing an indicator light, buzzing a sound alarm, printing a page in a printer, etc). Besides this, events also model the lack of communication to or from a system with the passage of time. An example of such events is the absence of a message emission from a system within a certain time interval, which is measured by timers that are common to the system and to its environment.

When a system is decomposed into its constituent components, each component is assigned a portion of the data the system operates on. As a result, state transitions that belonged to the first category for the entire system are transformed or broken down to a sequence of events among the system's constituent components. As an example, consider a system that has a data-list with three elements and after decomposing the system into its constituent components the first two elements of the data-list belong to one component and the last element of the data-list to the other. The operation of adding the content of the first two elements of the data-list and storing the result into the third element corresponds to one or more state transitions that belong to the first category, i.e. not directly observable by the system's environment. After the decomposition however, the transferring of the addition result from one constituent component to another corresponds to two state transitions, the emission and the reception of the message containing the addition result by the first and the second constituent component respectively. These state transitions belong to the second category, i.e. directly observable by the environment of each of the constituent components. Hence, the system decomposition may produce new events that capture the I/O operations between the constituent components.

The reception of input is a non-deterministic event for a given system, i.e. a system cannot know beforehand when input will be delivered and what the contents of the input will be. However, for a given input and a given state in which the system receives this input, the execution of the system (i.e. states and state transition including these capturing output delivery) until the reception of the next input is deterministic. Hence, the system execution can be modeled as a sequence of deterministic state intervals, each starting by a non-deterministic event. We also assume that the system has the capacity to detect and capture sufficient information related to non-deterministic events e.g. in order to regenerate/replay them. Thus, the system execution follows the *piecewise deterministic (PWD)* assumption [12].

When decomposing a system into its constituent components, the communication between the components is not instantaneous, i.e. an output produced by one component may take a measurable time before it is delivered to the component that is supposed to receive it. A communication event, after the output is produced by a component  $A$  and before the corresponding input is delivered to the intended recipient component  $B$ , is part of the state of the communication channel between the components  $A$  and  $B$ .

## 2.2 Basic Fault Tolerance Concepts

A *failure* is said to occur in a system when the system's environment observes an output from the system that does not conform to its specification. An *error* is the part of the system, e.g. one of the system components, which is liable to lead to a failure. A *fault* is the adjudged cause of an error and may itself be the result of a failure. Hence, a fault causes an error that produces a failure, which subsequently may result to a fault, and so on [6]. Let us consider the following example:

A software bug in an application is a **fault** that leads to an **error** when the application execution reaches the point affected by the bug. This causes the application crash, which is a **failure**. By crashing, the application leaves blocked the socket ports it used, which is a **fault**. The computer on which the application crashed has socket ports that are not used by any process but still not accessible to running applications, which is an **error**. This, in turn, leads to a **failure** when another application requests these ports.

Faults may occur either in the state of a component or in the state of a communication channel. The resulting errors can be arbitrary modification of state data, loss of state data, and loss or delays of the events in the communication channels. The consequences of these errors can be a variety of failures ranging from byzantine failures (arbitrary or malicious deviation from the system specification), to send- and receive-omission failures (losses of communication events), and crash failures (components or communication channels cease executing).

To take corrective actions and prevent system failures, errors must be detected first and then an error masking, fault repair or system recovery mechanism can be employed to prevent the system from experiencing a failure [7]. The design patterns presented in this paper describe fault tolerance techniques that fall in the category of system recovery. Error detection is not addressed in these patterns; rather, the described system recovery techniques assume that adequate error detection is in place and notifies the system recovery mechanism when errors are detected.

### 2.3 Rollback Recovery Concepts

A *global state* of a system is the aggregation of the states of its constituent components plus the states of the communication channels among its components. A global state is *consistent* if the following two conditions hold true.

1. For every component  $B$  whose state reflects the delivery of an input with content produced by component  $A$ , the state of component  $A$  reflects the production of the corresponding output.
2. For every component  $A$  whose state reflects the production of output intended for component  $B$  and the state of component  $B$  does not reflect the delivery of the corresponding input, the state of the communication channel between  $A$  and  $B$  contains a trace (event plus content) of the intended communication (also called *message in transit*).

The fundamental goal of rollback recovery techniques is to re-establish a consistent global state after an occurred error has caused inconsistencies in the global system state [3]. The consistent state re-established by a rollback recovery technique does not have to be one that has occurred in the system execution prior to the occurrence of the error. It is sufficient that the re-established consistent state could have occurred in the system execution before the error occurred. To accomplish their goal, rollback recovery techniques rely either on checkpoints or on logs or on both. Checkpoint-based solutions being presented elsewhere [10], this paper focuses on the log-based solutions to rollback recovery.

Log-based rollback recovery is based on the PWD assumption and the fact that for all non-deterministic events that cause state transitions to system components the information necessary to replay them (called the event's *determinant* [1]) can be identified and saved as recovery data. Saving these determinants in log records allows a recovery mechanism to use them when recovering a component from an error in order to replay these events locally at the recovering component, without involving the rest of the system in the recovery activity. During the recovery period, some previously reached error-free state of the component where the error occurred needs to be available (e.g. the initial state of the component or some checkpointed state), which can be reloaded after the error occurrence. Then, the logged determinants of all the non-deterministic events occurred

at the failed component must be replayed in order to that component to rollback to a state that belongs to a consistent system state.

A component  $C$  is said to become an *orphan process* [1] when it does not fail but its state depends on a determinant that has not been safely logged and the component that has produced the corresponding event has failed and recovered to a state prior to the production of that event. Clearly, the state of an orphan process is not part of a consistent system state that can be reached after recovery.

## 2.4 Checkpoint-based Patterns

In a complimentary work to this one [10] we have presented three design patterns for checkpoint-based rollback recovery. Here we provide a quick summary of them in order to provide to the reader an intuitive understanding of the commonalities these patterns have with the log-based pattern presented in this paper.

The **Independent Checkpoint** pattern aims at preserving the independence of the system components during error-free executions. The solution suggested by this pattern allows each component in the system to take checkpoints without any synchronization with the other components. As a result, the recovery activity has the responsibility of identifying a consistent system state in the set of the checkpoints independently taken by each component in the system. The low distraction of the error-free execution comes with the price of high performance penalty during recovery periods and the lurking danger of the domino effect [8].

The **Coordinated Checkpoint** pattern represents the other extreme in checkpointing techniques. It focuses on the optimization of the recovery activity and provides a solution where the latest checkpoint by each system component is part of a consistent system state. Hence, the recovery activity is simple and time-bounded. The price to pay for this optimization of the recovery activity is that system components must synchronize when taking checkpoints, increasing this way the performance penalty during error-free executions.

The **Communication-Induced Checkpoint** pattern comes to bridge the above two extreme checkpointing techniques. Instead of using special purpose synchronization, the communication events in the system are used to synchronize the checkpointing activity. This technique guarantees that a consistent system state can be constructed from the checkpoints taken by the system components and hence it is free from the domino effect. The resulting checkpoint scheme behaves better than **Independent Checkpoint** during recovery periods and better than **Coordinated Checkpoint** in error-free executions. But it performs worse than **Independent Checkpoint** in error-free executions and worse than **Coordinated Checkpoint** during recovery periods.

## 2.5 Log-based Patterns

The general solution to log-based rollback recovery is captured by the **Log Keeping** pattern. This pattern describes the log keeping and the recovery activities along with the entities that compose a mechanism that provides log-based rollback recovery. Then, the details on known solutions that deal with how to keep determinant logs and how to eliminate orphan processes are presented in three other patterns.

The **Optimistic Logging** pattern makes the optimistic assumption that error will not occur very often and at any time in the system execution. So, logs are saved following a

lazy scheme that causes a small penalty to the error-free system execution but it may result to orphan processes during system recovery. The **Optimistic Logging** pattern eliminates these orphan processes based on an elaborated dependency tracking mechanism applied on the logged determinants.

The **Pessimistic Logging** pattern makes the pessimistic assumption that errors occur often and at any time in the system execution. Determinants logs are safely saved when they are created and before the corresponding component continue their execution. Hence, the recovery activity does not lead to orphan processes but the penalty introduced to the error-free execution of the system is significantly higher than the corresponding **Optimistic Logging** case.

The **Causal Logging** pattern provides the golden mean between the pros and the cons of the two logging patterns above. It combines the low performance penalty during error-free execution with the lack of orphan processes during the recovery activity. To achieve this, the recovery activity of a component may be assisted by other, error-free components who can provide the determinants of events for which the recovering component did not have the time to keep logs before the error occurred.

## 3 Log Keeping

Certain types of systems, when recovering from errors, need to re-establish an error-free state that is as close as possible to the point in their execution where the error occurred. Such systems are those whose state transitions model irreproducible events, in the sense that reproducing these events would compromise the consistency of the system. Examples of such events are the delivery of cash by an ATM after request from the customer (dispensing twice the money the customer requested breaks the consistency of the banking system), the billing of a call (charging the caller twice for the same call breaks the consistency of the operator's billing system), and the activation of a craft's steering devices (lowering an aircraft twice as much as intended by the pilot breaks the consistency of the steering system). The **Log Keeping** pattern captures a solution to system recovery that fits such system. This solution is based on saving the determinants of the non-deterministic events that cause state transitions in the system components (e.g. received messages, signals and timer notifications) and, when recovering from an error, replay the corresponding events locally at each component in order to roll it back as little as possible from the point in its execution where the error occurred.

### 3.1 Context

The context, in which the **Log Keeping** pattern can be applied to provide system recovery, is defined in term of the following invariants:

- The system is composed from distinguishable components each of which can fail independently from the others.
- The non-deterministic events (or at least the majority of them) in the system components are irreproducible, i.e. it is not acceptable to reproduce during system recovery interactions among components that have already happened during error-free system execution.
- There are memory resources in the system that remain unaffected by the errors that the components may experience (e.g. disk space if only process crashes are considered or replicated memory segments over the components' volatile memories if the crash of entire computers is considered).
- The faults that cause the errors that the system may experience are accidental and transient, i.e. they are not design flaws or software bugs or a result of broken hardware. Hence, the system can re-attempt to follow the same execution steps with negligible likelihood that the same error will occur again.

The second context invariant restricts the applicability of the patterns about checkpoint-based rollback recovery presented in [10]. If non-deterministic events that cause state transitions in components are irreproducible, then checkpoints can solve the system recovery problem only if they are taken after the occurrence of such non-deterministic events. The resulting frequent checkpointing activity quickly becomes unaffordable even for modest density of non-deterministic events and modest size of checkpoint data. This tradeoff is captured in the forces of the problem expressed below.

## 3.2 Problem

In the above context, a problem that rises is the following: *How to recover a system in a consistent state, after the occurrence of an error?* The **Log Keeping** pattern solves this problem by balancing the following forces:

- Taking checkpoints is a costly activity, both in terms of execution time needed to take them and memory space needed to store them (it may require synchronization of the entire system, extraction of each constituent component state, transfer of large amount of data to the storage, etc).
- Saving only the determinants of the non-deterministic events is cheaper than taking checkpoints, both in terms of execution time and memory space (no synchronization among system components is needed, the amount of data corresponding to a determinant is small, etc).
- There is a threshold in the number of determinants saved, after which the benefits in execution time and memory space over checkpointing are lost.
- The saved determinants can be used to replay the corresponding events *locally* to the component that is recovering from an error. In contrast, checkpoint-based recovery affects the *entire* system.
- Recovering a system from its last checkpoint leads to the loss of the system state (as a union of its constituent component states) that was produced as a result of the non-deterministic events that happened since that checkpoint was taken. Recovering a component by replaying the saved determinants leads only to the loss of the recovering component state that was produced as a result of the non-deterministic events for which determinants were not saved.

## 3.3 Solution

*For every non-deterministic event that happens at a component, create a log that contains enough information about the event's determinant and save it in a place that can survive errors that may occur to the given component. After an error occurs, use these logs to replay the events that the failed component had experienced before the error occurrence.*

Each component of the system has a known error-free state safely saved on memory resources that remain unaffected by errors that may occur on that component. This error-free state can be the initial state of the component or a checkpoint. For every non-deterministic event that occurs to a component after it has reached the aforementioned error-free state, the component keeps a log that describes the determinant of this event in such a way that the event can be replayed from that log. If an error is detected to have occurred on a component and a recovery activity is launched, then the recovering component acts as follows.

First, it discards its erroneous state and it reloads its known error-free state. Then, it replays from the logs it kept the events (e.g. received messages) that have occurred since that error-free state. During this period, it does not create any events that can be perceived as non-deterministic events causing state transitions by other components in the system (e.g. it does not send any messages to other components). Rather, all activities that would lead to such events is filtered by the recovery mechanism (e.g. created messages

are silently discarded by the recovery mechanism before sent to their intended recipients). After the last event from the logs is replayed, the component continues its execution normally.

**N.B.:** The solution suggested by the **Log Keeping** pattern to the system recovery problem does not address error detection issues. Design patterns that address the error detection and notification problem are studied elsewhere [9].

### 3.4 Structure

The solution to the problem of system recovery described by the **Log Keeping** pattern outlines the following entities:

- The *recoverable process*, which is the component that logs its non-deterministic events during error-free execution in order to be able to replay them during system recovery.
- The *stable storage*, which is the part of the system where the logs and the error-free states are saved and which is not subject to errors.
- The *logger*, which is responsible for the logging activity of the *recoverable process*. The *logger* creates appropriate logs for each non-deterministic event and transfers them to the *stable storage*.
- The *error detector*, which is responsible for detecting errors that may occur to the *recoverable process* and, when errors are detected, to notify the *recovery manager* (see below) about them.
- The *recovery manager*, which controls the recovery activity. This entity receives the error notification produced by the *error detector* entity, issues a request to the component where the error occurred to discard its current state and reload its known error-free state from *stable storage*. Finally, it supervises the replay of logged events and filters the interaction events the recovering component produce during this period.

When applying the **Log Keeping** pattern on a system, there must be as many *recoverable processes* as the number of the components that must be able to recover from errors. Moreover, the *logger* is mapped on the same unit of failure as the *recoverable process*, usually being a piece of code linked with the component for which it logs its non-deterministic events. This is necessary in order to guarantee that when either of those two entities fails, the recovery activity will be triggered. Otherwise, if the *logger* fails first and no recovery steps are taken, then the system will not be able to recover the associated component should an error occur to it. Clearly, the *error detector*, *stable storage* and *recovery manager* entities must be mapped to a different unit of failure than the *recoverable process*. Otherwise, an error occurring on the *recoverable process* would affect the three aforementioned entities and would render the recovery mechanism void.

The **Log Keeping** pattern allows to localize the recovery process on the component where the error occurred. However, if the rest of the components in the system continue to operate normally during the recovery of a given component then a number of delicate situation may raise. Discarding the erroneous state, reloading the error-free state and replaying the logged events happens in parallel with the normal operation of other

components. Hence, interaction events may be destined to the recovering component (e.g. other components may send messages to the recovering component). In order for the recovering activity to work properly, the *recovery manager* must buffer those incoming interaction events destined to the recovering component until the end of the recovery activity. However, this may not be sufficient since some of those interaction events may expect a response from the recovering component. If this response is delayed beyond certain timeouts then the originating components may wrongfully detect an error on the recovering component. To deal with these cases, the *recovery manager* may have to adjust the timeouts related to the recovering component when the recovery activity starts and re-adjust them back to their normal values when the recovering activity ends. Furthermore, it may be necessary to inform all components about the recovering activity on a given component so they can slow down, or temporarily suspend, their interactions with the recovering component.

The *stable storage* is a logical entity that does not have to map to a single component necessarily. Replicated, distributed memory can serve as *stable storage*. However, the system developer has to consider the potential time overhead of finding the distributed memory segment where a given log is stored. On the other hand, when *stable storage* is mapped to a single physical entity (e.g. hard disk or flash memory), the system developer has to consider the potential time overhead of the simultaneous access of the physical media by all components when they attempt to save or recover logs and application-specific data.

Figure 1a provides an intuitive illustration of the structure of the Log Keeping pattern. Figure 1b contains the activity diagram that describes the functionality of the checkpointing and the consistent state reconstruction activities of the Log Keeping pattern.

Both parts (a) and (b) of Figure 1 are simplified in order to capture the essentials of the Log Keeping pattern without confusing the reader. The simplification is that the application of the Log Keeping structure is elaborated for one out of the many possible constituent components of a system. What is important for the reader to notice in the depicted structure is that the *recoverable process* and the *logger* entities are placed inside the same unit of failure. The remaining three entities are outside that unit of failure, implying that they will not be affected by the occurrence of an error in either of the two aforementioned entities.

Another thing that is worth noticing is that the *recoverable process* corresponds to a distinguishable component in the system. Hence, in a system there will be as many *recoverable processes* and *loggers* as the number of the constituent system components that are mapped to separate units of failure. Finally, there are many *recoverable processes*, *loggers* and *error detectors*, as many as the components of the system that can be rolled back. On the other hand, the *recovery manager* and the *stable storage* are logically unique entities in the system (although in the implementation they can be physically distributed among the system components).

### 3.5 Implementation

The implementation of the Log Keeping pattern consists roughly of the following steps:

- Identify the constituent components of the system that will be mapped to *recoverable processes*.

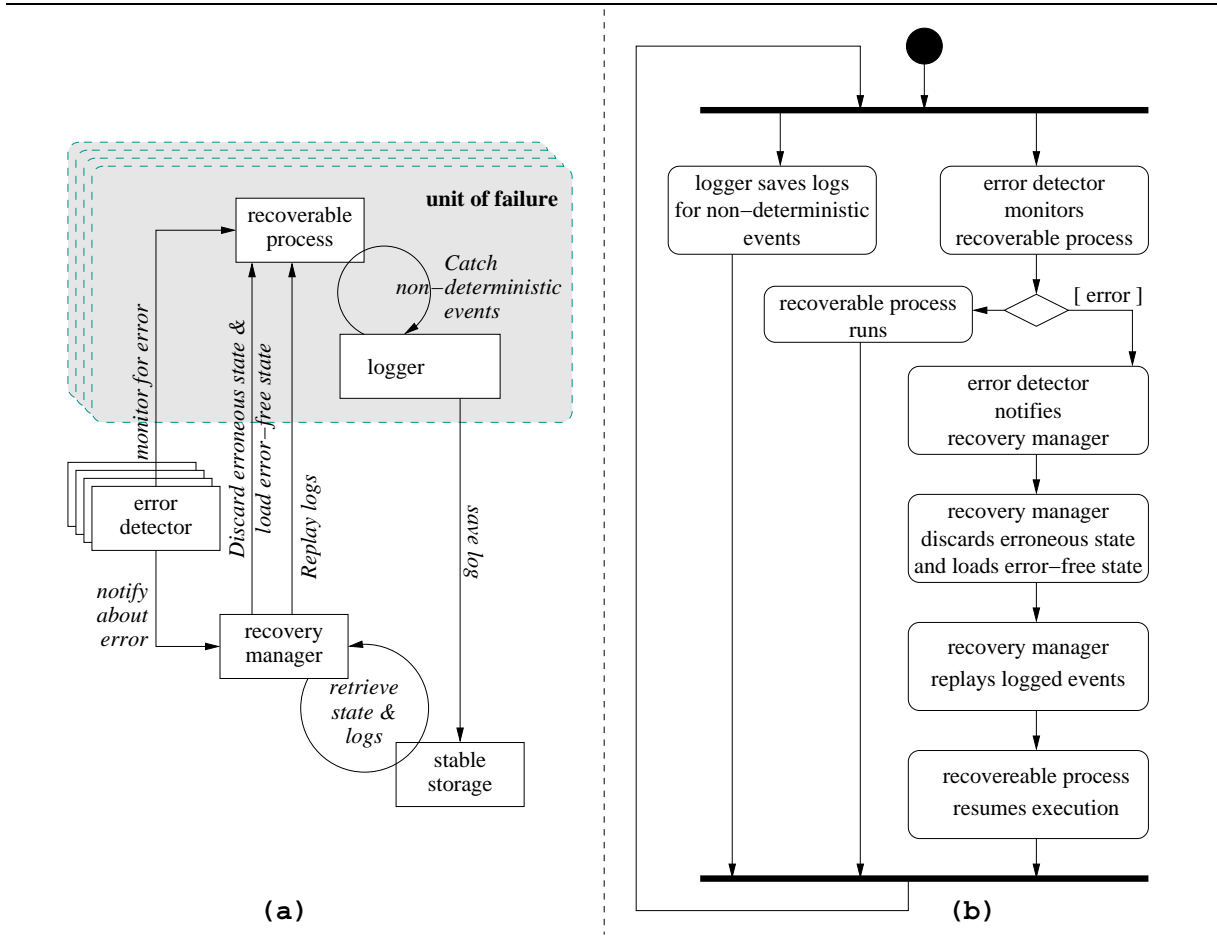


Figure 1: The structure (a) and the activity diagram (b) of the Log Keeping pattern.

- Decide what mechanism will provide the *stable storage* and, if it is not available in the system, implement it (see below for implementation alternatives).
- Decide on the implementation of the *error detector*, i.e. what types of errors can be detected and what mechanism will detect them. Patterns regarding error detection can be found in our previous work [9].
- Implement the *logger*, which will be tightly coupled with each *recoverable process* inside the same unit of failure. The different options regarding the logic that the *logger* can implement (i.e. *HOW* exactly to save the logs) are elaborated in the three patterns that follow in this paper.
- Implement the *recovery manager* (see below for implementation considerations regarding this entity).
- Study the characteristics of the system and calculate the threshold after which the logging process becomes more costly than checkpointing. Combine the logging activity with a checkpointing mechanism (e.g. see [10]) in order to fine-tune the performance of the system during the checkpointing and logging activities and during the log-based recovery after the occurrence of an error.

The implementation of the *stable storage* entity can take several forms. For a 1-tolerant system (i.e. a system that can deal with a single fault in one recovery cycle) the stable storage of one component can be the memory of another component. For a N-tolerant system, the stable storage can be implemented as N-replicated storage in a distributed shared memory deployed across the memories of the system components. Alternatively, RAID-based file system can serve the purpose of stable storage for a N-tolerant system. If the failure probability of storage media such as hard disk or flash memory is significantly lower (e.g. two orders of magnitude) than the failure probability of the *recoverable entities*, then those media can be conventionally employed as stable storage.

The implementation of the *error detector* depends on the error detection strategy that it implements. In practice, the *error detector* may not be a single entity but rather a set of entities which cooperate to provide error detection and notification, as discussed in the error detection pattern presented in [9].

The *logger* implementation can also take several forms. It can be the responsibility of the OS, the middleware or the compiler, causing little distraction to the application development. Alternatively, the *logger* can be explicitly programmed in the application.

The implementation of the *recovery manager* is more demanding. Mapping this entity to a single (possibly new) component in the system may simplify the design of the system but introduces a well-known dilemma in fault tolerance: “*who shall guard the guards*”. Putting in place another fault tolerance mechanism for ensuring the correct functioning of the recovery manager complicates the design of the system and renders void the benefit of design simplicity resulted from mapping this entity to a single component.

On the other hand however, the *recovery manager* is engaged only in the recovery phase and not during error-free system execution. If the system can afford to slow down or even to halt during recovery phases, the mechanism that guarantees the fault tolerance of the *recovery manager* can be a relatively simple one (e.g. if the *recovery manager* fails, stop it and restart it). If errors occur rarely in the system, the developer may not even have to consider the fault tolerance of the *recovery manager*, since the probability of error occurrence in during the system recovery (i.e. right after another error has already occurred) is likely to be negligible.

Alternatively, the *recovery manager* can be distributed across the system components. This approach is particularly appealing when the error detection mechanism follows a similar distributed implementation approach (e.g. see error detection patterns in [9]). However, such an approach implies that every component in the system can potentially play the role of the recovery manager at a given moment. Hence the recovery manager functionality must be replicated across all system components resulting in an elevated space overhead introduced by the implementation of the **Log Keeping** pattern.

An important consideration is the combination of the logging mechanism with a checkpointing mechanism. The extreme case of the **Log Keeping** pattern, where the initial state of a component is reloaded and every logged event that has happened since the component start-up is replayed, is not practically applicable to any but the very short-lived systems. For long-lived systems, the amount of log data plus the time needed to recover a component if only its initial state is known make a pure log-based recovery impractical. In these cases, the combination of logging with checkpointing is required.

Checkpoints provide clear points in the execution of a system where garbage collection of logs can take place (logs regarding events that have happened before the checkpoint can be safely erased since they will never be used in the future for rollback recovery). Hence, checkpoints offer a means for controlling the memory space impact of the logs. Moreover,

checkpoints contribute also in reducing the recovery time; loading the initial state of a component and replaying all the events that have happened since its start-up takes more time than re-loading a recent state and replaying only those events that happened after that state was reached. It is the task of the system designer to combine the logging and the checkpointing activities and to tune the tradeoff in costs incurring during error-free system execution and the benefits in memory space and recovery time.

Finally, an implementation issue that the system developer must address and resolve is the functioning of the environment of a recovering component. During recovery, the recovering component cannot interact with its environment with the same agility as under its error-free execution. Hence, messages sent to a recovering component must be delayed or buffered until the recovery is completed and the component in question is able to receive and process these messages. For the same reason, any timers set on the events that the environment expects to be produced by a component must be suspended or properly adjusted during the recovery of the component in question. That way, error detection mechanisms will not misidentify as an error the lower agility or longer response times of a recovering component.

### 3.6 Consequences

The **Log Keeping** pattern has the following benefits:

- + Keeping a log introduces less time overhead to the error-free execution of a system than taking a checkpoint. Hence, logging can be done more frequently than checkpointing.
- + Recovering from logs localizes the recovery activity to the component on which the error has occurred; only the failed component must be recovered and only the logs regarding events related to that component need to be retrieved and replayed. This is in contrast to checkpoint-based solution for rollback recovery where all the system constituent components are involved in the recovery activity and the cross-examination of the checkpoints taken by all components may be required in order to identify a consistent system state.
- + Recovering from logs minimizes the loss of system state due to errors, since it allows the recovering component to rollback until the last event it was able to log. This is in contrast to checkpoint-based recovery where the system state, which was produced after the checkpoints used to recover the system were taken, is lost.

The **Log Keeping** pattern imposes also some liabilities:

- Although keeping logs is less costly than taking checkpoints, it still introduces a time overhead to the error-free system execution that penalizes the system performance.
- Logging requires more memory space than certain checkpointing techniques (e.g. those described by the **Coordinated Checkpoint** and **Communication-Induced Checkpoint** patterns in [10]). This is because, in addition to the checkpoint (or initial state) of a component, a number of logs are also saved in *stable storage*.
- Recovering from logs takes more time than recovering from checkpoints, for certain checkpointing techniques (e.g. those described by the **Coordinated Checkpoint**

and **Communication-Induced Checkpoint** patterns in [10]). This is because, in addition to loading the checkpoint (or initial state), a number of events are replayed locally at the recovering component.

### 3.7 Related Patterns

The **Log Keeping** pattern is one specialization of the **Rollback** pattern and it can be combined with the error detection patterns that has been presented in our previous work [9]. Also, the **Log Keeping** pattern is tightly related to the other patterns presented in this paper, which elaborate on the way the *logger* entity works.

## 4 Optimistic Logging

The activities of the `Log Keeping` pattern that interfere with the error-free system execution is the monitoring for error and the logging of the determinants. From those two, the logging activity is the one that introduces the bigger run-time overhead, which amount to capturing the non-deterministic events, creating the log record for the corresponding determinant, and storing it to stable storage. For certain types of systems this performance overhead during error-free executions must be kept as low as possible, while those system can afford to significantly slow down or even temporarily halt during recovery periods. The `Optimistic Logging` pattern describes a solution to log keeping that fits systems with such characteristics.

### 4.1 Context

The context, in which the `Optimistic Logging` pattern can be applied to provide system recovery, is defined in term of the following invariants:

- For every non-deterministic event that occurs at a component, a log containing its determinant is kept according to the `Log Keeping` pattern (see Section 3).
- Errors do not occur very often and/or there are phases in the component's execution where the probability of errors is quite low.
- Saving logs in stable storage is the most costly aspect of the logging activity.
- Each component has spare volatile memory space where logs can be temporarily stored before saved on stable storage.

This context allows the optimistic assumption that there are periods of a component's execution during which errors will not occur, or at least the likelihood of an error occurrence is very low.

### 4.2 Problem

In the above context, a problem that rises is the following: *How to log the determinants of the non-deterministic events that occur in a system?* The `Optimistic Logging` pattern solves this problem by balancing the following forces:

- Saving logs to *stable storage* accounts for most of the time overhead introduced by the logging activity to the error-free system execution.
- Saving the logs in blocks of many to *stable storage* introduces lower performance overhead than saving each individual log to stable storage separately.
- If a component fails, the logs that have not been saved to *stable storage* are lost. This may result in some other component(s) in the system becoming orphan process(es).
- Orphan processes must be rolled back to a state that is part of the consistent system state, which the rest of the system components (which are not orphan processes) have reached.

### 4.3 Solution

*Buffer the logs to the component's volatile memory and save them to stable storage in blocks.*

The *logger* entity from the **Log Keeping** pattern has access to a segment of volatile memory where it can temporarily buffer the logs it creates. When the buffer is full or when the execution of the *recoverable process* reaches a certain predefined point, the *logger* saves the buffered logs to the *stable storage*.

If a component fails, the logs in the volatile memory of its *logger* are lost. The recovering activity can use the logs saved in *stable storage* to rollback the failed component. However, the state to which the component can be rolled back may cause other components in the system (e.g. those that produced the events whose determinants were in the *logger's* lost memory) to become orphan processes. The *recovery manager* identifies these components and rolls them back too to a state that brings the system to a consistent global state.

For the *recovery manager* to be able to rollback orphan processes in case they appear during the recovery activity, the log records contain also dependency tracking information in addition to the event determinants. This information is piggybacked in the messages sent among the components and allows the identification of the determinants on which the current state of the component depends. If the logs of these determinants are lost after the occurrence of an error, then the components whose state depends on them are orphan processes and have to be rolled back. The recovery activity that deals with the orphan processes can be either synchronous or asynchronous.

In synchronous recovery (e.g. see [11]), the *recovery manager* checks all the logs in *stable storage* and in the volatile memory of the survived *loggers*. This can be done either by interrogating each *logger* separately or by instructing the *loggers* to flush their buffered logs to *stable storage* and get all the logs from there. Then, the *recovery manager* calculates which components must be rolled back based on the dependency tracking information found in the logs, and coordinates the rollback activity. During the synchronous recovery, the system halts its normal execution.

In asynchronous recovery (e.g. see [12]), the *recovery manager* rolls back the failed component first and calculates which other components become orphan processes because of this rollback. Then, each of those components is also rolled back and the *recovery manager* checks again which other components may become orphan processes due to the last rollback, and so on. During asynchronous recovery, the system does not halt its normal execution. Only the components which are rolled back at each time halt their normal execution.

### 4.4 Structure

The **Optimistic Logging** pattern does not introduce any new entities in the system where the **Log Keeping** pattern has already been applied. It only defines one new connection among the *logger* and *recovery manager* entities of the **Log Keeping** pattern and elaborates on their functionality. Figure 2a updates Figure 1a to include the connection between the *logger* and *recovery manager*. Figure 2b contains the activity diagram that elaborates on the functionality of the *logger* and *recovery manager* according to the solution described by the **Optimistic Logging** pattern.

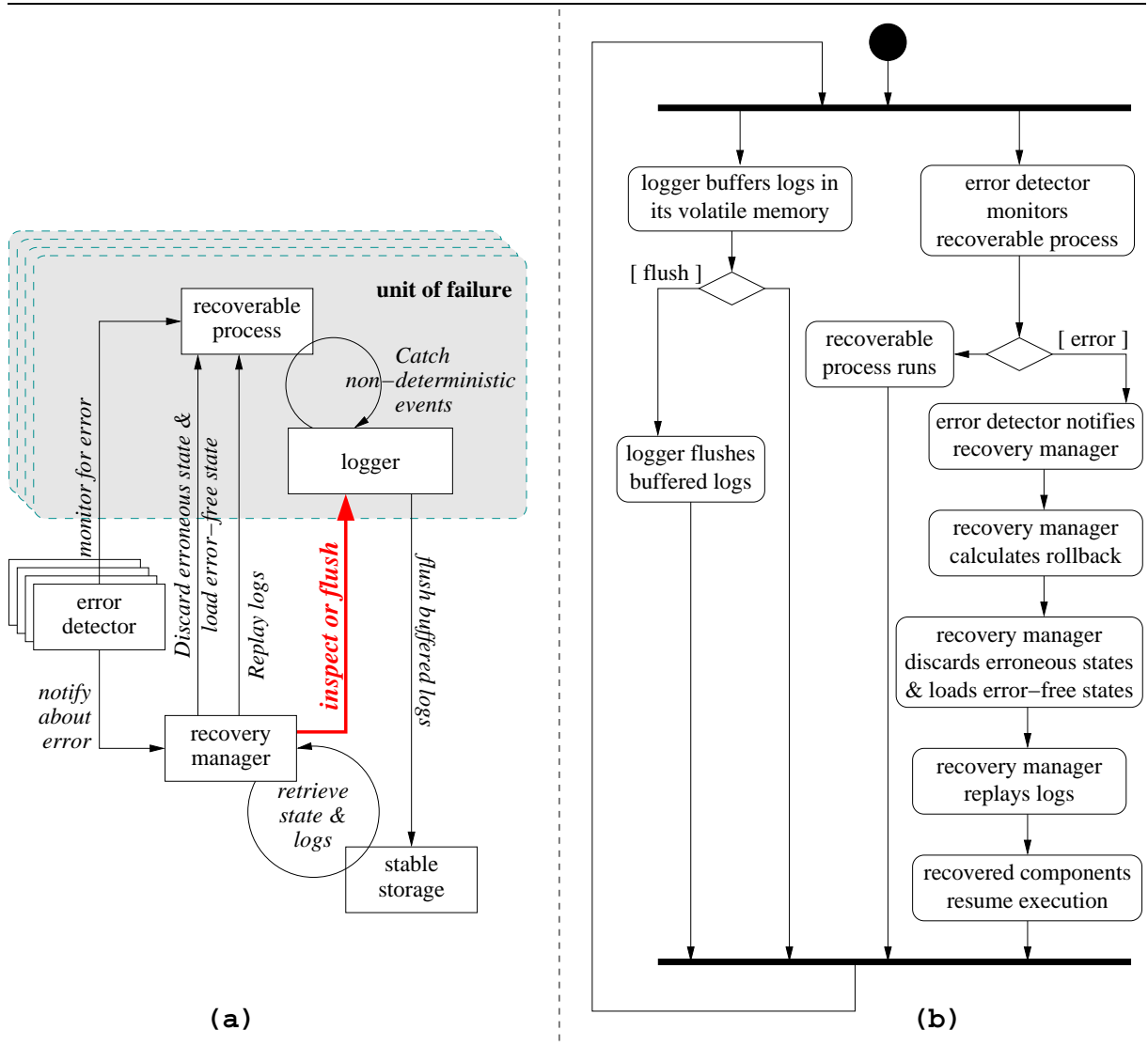


Figure 2: The structure (a) and the activity diagram (b) of the Optimistic Logging pattern.

## 4.5 Implementation

In theory, the implementation of the **Optimistic Logging** pattern can be a literal mapping of the entities depicted in Figure 2a to software components. In such case the *recovery manager* would be a single component entity in the system, which supervises the recovery activity. However, the practices reported in the related literature (e.g. [12] and [11]) describe a distributed approach where the *recovery manager* and the *logger* are united in a single piece of software that is linked with every system component (i.e. *recoverable process*). The distributed instances of the *recovery manager* communicate using the optimistic logging protocol, which is nothing more than the sequence of steps that must be taken upon the occurrence of an error to identify the components that must be rolled back. In the reported implementation of the optimistic logging protocol, the *stable storage* is mapped to files stored on hard disks.

Another issue that is worth mentioning is that the optimistic logging implementations

reported in the literature are built on top of a checkpoint protocol. The similarities in the structure of the **Log Keeping** pattern with the checkpoint patterns [10] provides for the integration of the two in a comprehensive rollback recovery solution. When combining the **Optimistic Logging** pattern with the **Coordinated Checkpoint** pattern, the checkpoints trigger the garbage collection of the logs saved in *stable storage* that have been saved before the checkpoint. This provides a very cost-effective way for triggering garbage collection, which is one of the weak points of the **Optimistic Logging** pattern (see the liabilities of the **Optimistic Logging** pattern in the Consequence section below).

## 4.6 Consequences

The **Optimistic Logging** pattern has the following benefits:

- + The time overhead introduced to the error-free system execution is kept low, especially when the access to the stable storage is very costly. The time overhead in error-free execution mainly amounts to the time needed by the *recoverable process* to piggyback dependency information in the messages it exchanges with other *recoverable processes* in the system, plus the time needed by the logger to create and buffer the determinants' logs and the time needed to flush the buffered logs to *stable storage*.
- + Since storing the logs in *stable storage* happens asynchronously and non atomically with respect to the communication events they refer to, the mapping of *stable storage* to a single component in the system (e.g. the file server) does not introduce a potential performance bottleneck to the system.
- + Combining the **Optimistic Logging** pattern with checkpoint patterns (e.g. see [10]) allows the system designer to fine-tune the system performance both during error-free executions and recovery periods by adjusting the frequency of checkpoints and buffered logs flushes to *stable storage*.

The **Optimistic Logging** pattern imposes also some liabilities:

- The loss of logs stored in the volatile memory of the component where the error occurred may lead to the rollback of other components that did not fail but they became orphan processes due to the rollback of the failed component. This increases the time overhead of the recovery activity.
- During the recovery activity, the *recovery manager* must examine the dependency data saved in the logs, in order to identify any orphan processes that might have been created. To do this, the logs that are buffered in the volatile memory of the *loggers* of the components that have not failed must be either flushed to *stable storage*. Alternatively, the *recovery manager* can retrieve by querying those *loggers* directly. Both cases increase the time overhead of the recovery activity.
- Garbage collection of logs saved in *stable storage* that are not needed for rollback recovery after a certain point in the system execution is a difficult task. This is due to the fact that identifying useless logs would require to run the dependency tracking process, which is used during the recovery activity. However, running this process during error-free executions would penalize the performance of the system

and decrease the low time overhead benefits that this pattern offers to error-free executions.

- In the case where the asynchronous variant of the **Optimistic Logging** pattern is applied, the one-by-one recovery of the failed component and the components that become orphan processes may cause the phenomenon of exponential rollbacks [11]. This results in additional time overhead of the recovery activity.
- When the **Optimistic Logging** pattern is combined with the **Independent Checkpoint** pattern to provide a comprehensive rollback recovery solution, the recovery activity is potentially subject to a combination of the exponential rollbacks phenomenon [11] and the domino effect [8]. As an extreme result of the correlation of these two phenomena, the entire system may end up rolling back to its initial state after having performed in vain every possible rollback for every single component in the system.

## 4.7 Related Patterns

The **Optimistic Logging** pattern is directly related to the **Log Keeping** pattern presented in Section 3, for which it provides an elaborated solution to the way the *logger* and *recovery manager* entities function. In the same sense, it also related to the other two patterns presented in the remainder of this paper, which provide alternative approaches on how the two aforementioned entities of the **Log Keeping** pattern function.

## 5 Pessimistic Logging

There are cases where the solution suggested by the `Optimistic Logging` pattern to the system recovery problem is not acceptable. For example, for hard real time systems where bound execution times are of outmost importance, the time overhead of the recovery process might not be acceptable. In such cases another solution is needed which can keep the time cost of system recovery very low and the times of error-free executions and executions with error comparable. The `Pessimistic Logging` pattern suggests such a solution.

### 5.1 Context

The context, in which the `Pessimistic Logging` pattern can be applied to provide system recovery, is defined in term of the following invariants:

- For every non-deterministic event that occurs at a component, a log containing its determinant is kept according to the `Log Keeping` pattern (see Section 3).
- Errors may occur at any time and recovering from them must cause minimum impact on the system performance.
- It is more important to keep the system execution time under strict bounds than to optimize the performance for certain periods only (e.g. during error-free parts of the system execution).
- Accessing the *stable storage* introduces a bounded time penalty to the system performance.
- The recovery activity must be confined to the component where the error occurred.

These invariants imply that the recovery from errors must have a hardly noticeable impact to the system execution and force the pessimistic assumption that the system must be ready to perform the minimum cost recovery activity at any time.

### 5.2 Problem

In the above context, a problem that rises is the following: *How to log the determinants of the non-deterministic events that occur in a system?* The `Pessimistic Logging` pattern solves this problem by balancing the following forces:

- Accessing the *stable storage* has a high (yet bounded) cost.
- Orphan processes cause recovery activity to span beyond the boundaries of the component where an error occurred and, thus, increase the cost of the system recovery.
- Blocking a component's execution until the log of the event, which causes the state transition that is about to happen, is saved in *stable storage* ensures that the log will survive subsequent errors in the given component.
- Minimum recovery time implies minimum buffering requirements for the communication intended to the recovering component.

### 5.3 Solution

The **Pessimistic Logging** pattern presents an alternative approach to log keeping than the one described by the **Optimistic Logging** pattern. Instead of buffering logs in volatile memory, the logs are immediately saved in stable storage in a synchronous way with respect to the execution of the component concerned by the corresponding events. In other words, the state transition that is triggered by the occurrence of a non-deterministic event at a given component does not take place before the log record containing the determinant of that event is saved in *stable storage*. Since no logs are buffered in volatile memory, errors do not cause the loss of determinants and, consequently, there is no risk of orphan processes as a result of a recovery activity. On the other hand, the error-free execution of the system is penalized with frequent costly accesses to *stable storage*.

The tasks of the *logger* are straightforward. Capture the non-deterministic events that trigger state transition in its associated *recoverable process*, block the execution of the latter until it creates the corresponding log and saves it in *stable storage* and release the *recoverable process* to allow it continue its execution. The most important of these tasks is the saving of each log to *stable storage*. There are two variants of synchronous log saving reported in the literature. In one case special hardware has been employed in order to ensure the atomicity of logs saving in *stable storage* [2]. In another case [5], the atomicity of saving logs in *stable storage* is relaxed for non-deterministic events that correspond to the emission of a message, provided that the corresponding determinant is also saved at the sender (a.k.a. *sender-based message logging or SBML*).

The tasks of the *recovery manager* are also straightforward. When an error is reported to have occurred on a *recoverable process*, instruct that component to discard its erroneous state and load its last known error-free state from *stable storage*. Then, the *recovery manager* replays locally to the recovering component all the non-deterministic events logged by its *logger* and buffers the non-deterministic events that occur due to the normal execution of the rest of the system and that concern the recovering component. Finally, once the component has recovered, the *recovery manager* delivers the buffered events and lets the recovered component (and its logger) continue their normal execution.

### 5.4 Structure

The **Pessimistic Logging** pattern does not introduce any new entities in the system where the **Log Keeping** pattern has already been applied. It only elaborates on the functionality of the *logger* and *recovery manager* entities. Figure 3a is identical to Figure 1a. Figure 3b contains the activity diagram that elaborates on the functionality of the *logger* and *recovery manager* according to the solution described by the **Pessimistic Logging** pattern.

### 5.5 Implementation

The cases of pessimistic logging that have been reported in the literature (e.g. see [2] and [5]) do not map the *recovery manager* on a separate component unique in the system. Rather, the *recovery manager* is implemented as a distributed entity whose instances are deployed on the system components. These distributed instances communicate using the pessimistic logging protocol, which describes a sequence of interaction steps that allow the *recovery manager* instances on components that did not experience an error to initialize

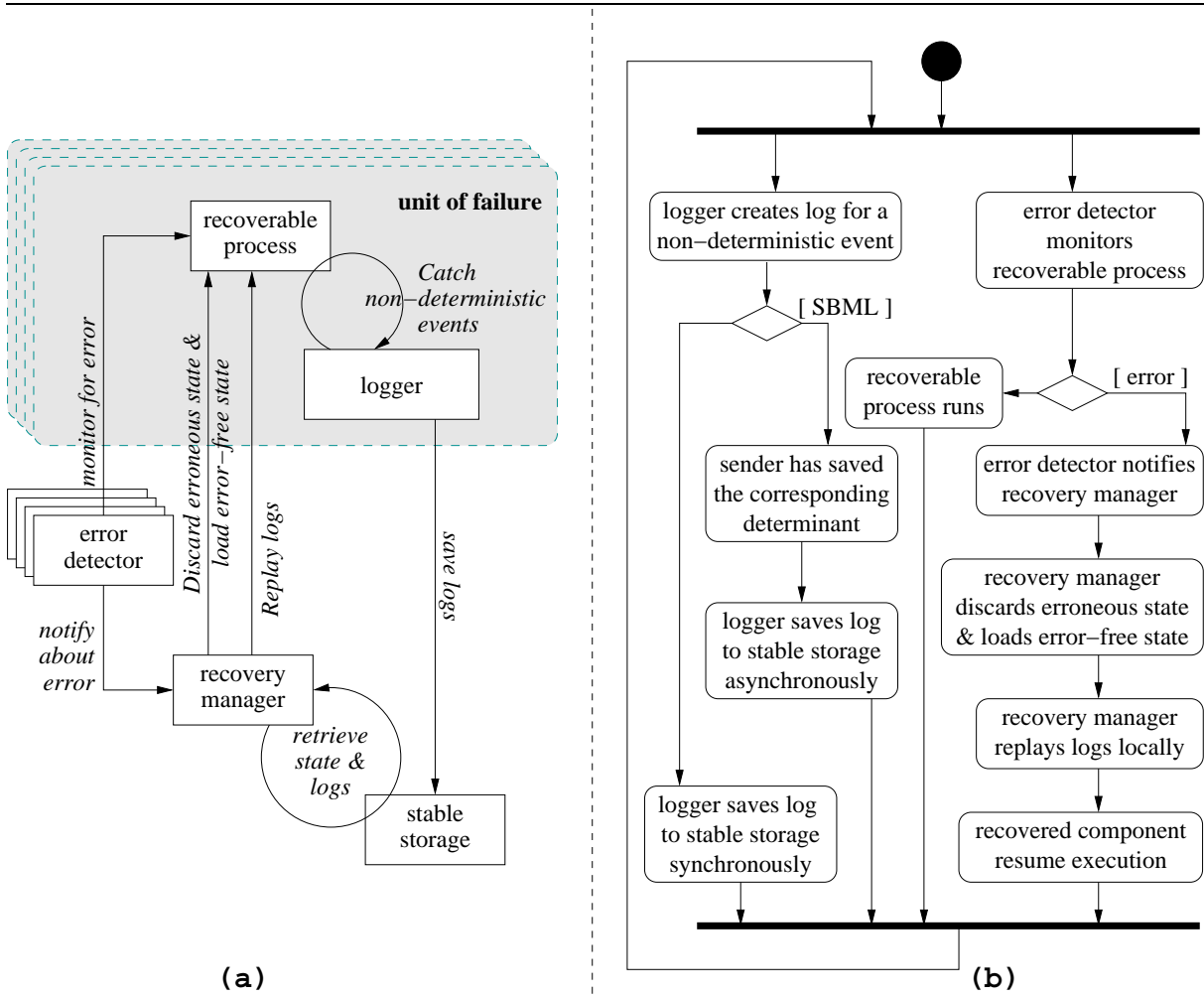


Figure 3: The structure (a) and the activity diagram (b) of the Pessimistic Logging pattern.

the *recovery manager* instance on the components where the error occurred. Then, the latter instance performs the recovery steps outlined in Figure 3b.

Similarly to the *Optimistic Logging* pattern, the pessimistic logging implementations reported in the literature are combined with a checkpoint protocol. This does not come as a surprise, since the memory space occupied by logs can rapidly increase beyond affordable limits for any system. On the other hand, taking checkpoints helps in bounding the memory space requirements for the logs, since the logs that refer to events that occurred before a checkpoint can be garbage collected once the checkpoint is taken. The similarities in the structure of the *Log Keeping* pattern with the checkpoint patterns [10] provides for the integration of the two in a comprehensive rollback recovery solution. The combination of the *Pessimistic Logging* pattern with the *Independent Checkpoint* pattern provides a very appealing solution to rollback recovery since it combines cheap checkpointing activity and bounded recovery activity localized on the component where the error occurred. Despite the theoretic benefits of this combination, the author is not aware of any implementations reported in the related literature.

## 5.6 Consequences

The **Pessimistic Logging** pattern has the following benefits:

- + The time overhead introduced to the error-free system execution is bounded and it amounts to the time needed to capture non-deterministic events, create the corresponding logs and store them in *stable storage*. This allows real-time systems to correctly schedule their tasks, taking into account the logging activity of the system recovery mechanism.
- + The time overhead introduced by the recovery activity is kept bounded, minimum and localized at the component where the error has occurred.
- + The **Pessimistic Logging** pattern does not result in orphan processes during the recovery activity.
- + The design complexity of the *recovery manager* and the *logger* is low (e.g. as compared to the **Optimistic Logging** pattern) since there is no need for dependency tracking and identification of other components that need to rollback besides the one where the error occurred.
- + Combining the **Pessimistic Logging** pattern with checkpoint patterns (e.g. see [10]) allows the system designer to fine-tune the system performance both during error-free executions and recovery periods by adjusting the frequency of checkpoints and buffered logs flushes to *stable storage*.

The **Pessimistic Logging** pattern imposes also some liabilities:

- The performance penalty introduced by the logging activity is high (e.g. as compared to the **Optimistic Logging** pattern) because it requires a two-phase commit in the worst case or double logging (at the receiver and at the sender of a communication event). This cost can be amortized in case special purpose hardware can be employed to facilitate the message logging.
- The synchronous nature of the logging activity makes the mapping of the *stable storage* entity onto a single component in the system (e.g. file server) a potential performance bottleneck.

## 5.7 Related Patterns

The **Pessimistic Logging** pattern is directly related to the **Log Keeping** pattern presented in Section 3, for which it provides an elaborated solution to the way the *logger* and *recovery manager* entities function. In the same sense, it is also related to the **Optimistic Logging** pattern presented in Section 4 and the **Causal Logging** pattern presented in the following Section.

## 6 Causal Logging

The **Optimistic Logging** and **Pessimistic Logging** patterns provide two extreme alternatives: either low overhead logging, which cause low performance penalty during error-free system execution but high overhead during the recovery periods, or high overhead logging, which cause high performance overhead during error-free system execution but low overhead during recovery periods. The **Causal Logging** pattern presented in this section comes to bridge the gap between those two extremes for the price of less than minimum performance penalties both in error-free system execution and during recovery periods.

### 6.1 Context

The context, in which the **Causal Logging** pattern can be applied to provide system recovery, is defined in term of the following invariants:

- The system cannot afford high performance penalties during error-free execution.
- The system cannot afford high performance penalties during recovery periods.
- The recovery activity must be confined to the component where the error occurred.

### 6.2 Problem

In the above context, a problem that rises is the following: *How to log the determinants of the non-deterministic events that occur in a system?* The **Causal Logging** pattern solves this problem by balancing the following forces:

- Synchronous logging introduces a high performance penalty during error-free execution.
- Orphan processes cause recovery activity to span beyond the boundaries of the component where an error occurred and, thus, increase the cost of the system recovery.
- If a component fails, the determinants that were in its volatile memory are lost.
- If some determinants cannot be retrieved (e.g. because they have been lost due to an error that occurred on the single component in whose volatile memory they were stored) orphan processes may appear during the recovery activity.
- Storing a determinant in the volatile memory of more than one component ensures that the determinant will survive a single error.

### 6.3 Solution

The solution suggested by the **Causal Logging** pattern is a combination of the solutions suggested by the previous two patterns. Like in the case of the **Optimistic Logging** pattern, logs may reside in the volatile memory of the components. Similar to the case of the **Pessimistic Logging** pattern, the logs that correspond to events that affect the state of a component cannot be lost due to errors that may occur in any other component. This is achieved by having each component to hold in its own volatile memory those

determinants whose logs are not saved in *stable storage* and which affect its own state. Let's assume that an error occurs on a component  $C$  and after recovering from it another component  $C'$  becomes an orphan process because there are no logs in *stable storage* to roll the recovering component  $C$  to a state that would allow the current state of  $C'$  to be part of a consistent system state. Then, the component  $C'$  has in its own volatile memory the determinants of the events that affect this state. Hence, it can provide them to the *recovery manager* who can subsequently replay them locally to component  $C$  to roll it to a state that does not render  $C'$  an orphan process.

The way to keep the determinants that are not saved in stable storage in the volatile memory of the component whose state is affected by the corresponding events is to piggyback those determinants with the communication of events. Let's take for example the emission of a message  $m$  from component  $C$  to  $C'$ . This emission is the result of a number of events that are causally related to the sent message. Hence, the determinants of these events are piggybacked in the message and sent along with it to component  $C'$ . Piggybacking the entire graph of causally related determinants would soon render the communication overhead unaffordable for the majority of real-world systems. Hence, in practice [4], each message from  $C$  to  $C'$  contains only the delta of the graph since the last message from  $C$  to  $C'$ , instead of the entire graph.

This solution has implications to the tasks performed by the *logger* and the *recovery manager*. The *logger* must create and keep in memory the antecedence graph [4] (a graph that connects the determinants of the causally related events) of the events that it generates (e.g. message and signal emissions, timer events, alarms, etc). It must also keep the track of which part of the antecedence graph it has sent to each component with which it communicates in order to calculate the delta from the current graph and send it in every new communication of its associated component.

The tasks of the *recovery manager* are similar to the homonym entity from the **Optimistic Logging** pattern. When recovering a component from an error, the *recovery manager* must also monitor the rest of the system components in order to identify those that are about to become orphan processes due to the performed recovery. Once these components are identified, the *recovery manager* requests from the corresponding *loggers* the determinants that are lost due to the error. After obtaining them, the *recovery manager* replays them locally at the recovering component, which subsequently rolls to a state that does not cause any other component in the system to become an orphan process.

## 6.4 Structure

The **Causal Logging** pattern does not introduce any new entities in the system where the **Log Keeping** pattern has already been applied. It only defines one new connection among the *logger* and *recovery manager* entities of the **Log Keeping** pattern and elaborates on their functionality. Figure 4a updates Figure 1a. Figure 4b contains the activity diagram that elaborates on the functionality of the *logger* and *recovery manager* according to the solution described by the **Causal Logging** pattern.

Note in Figure 4b that the added link between the *recovery manager* and the *loggers* is not active at the same time as the error notification link between the *error detector* and the *recovery manager*. In fact, a communication on the latter link regarding an error detected on component  $C$  may trigger communications over the links between the *recovery manager* and the *loggers* associated to all other system components except that of  $C$ .

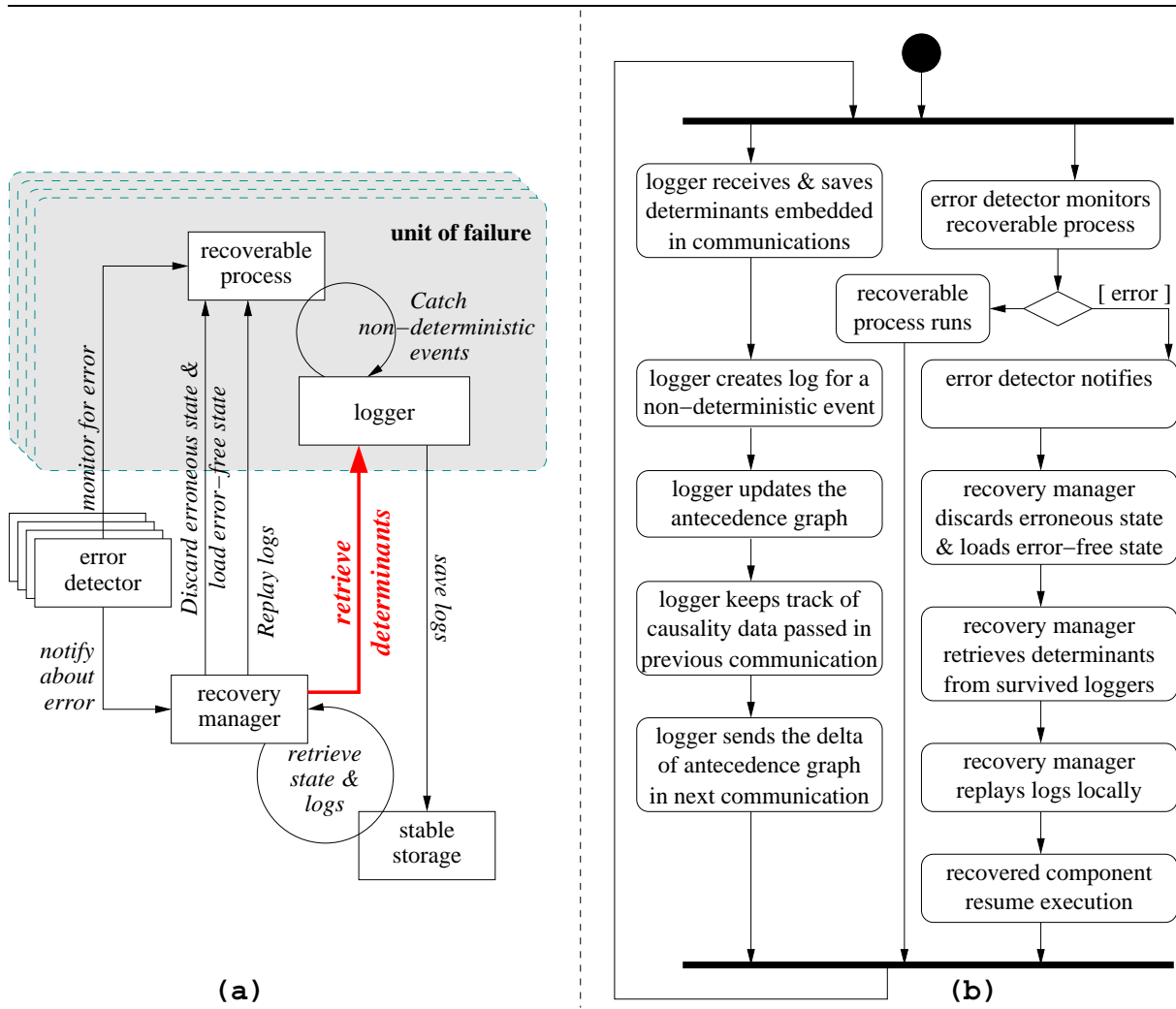


Figure 4: The structure (a) and the activity diagram (b) of the Causal Logging pattern.

## 6.5 Implementation

The implementation of the Causal Logging pattern (e.g. see [4]) shares many similarities with the implementations of the previous two logging patterns. The *recovery manager* and the *logger* entities are built in the same software, which is linked to every recoverable component in the system. The resulting distributed instances of the *recovery manager* communicate using the causal logging protocol, which describes the following steps taken during the recovery activity:

- Retrieve the logs from the *stable storage* and replay them locally to the recovering component.
- Identify the survived components that become orphan processes and retrieve from them the determinants that are causally related to their current state.
- Replay the corresponding events locally to the recovering component.
- Resume normal execution of the recovered component.

Also, similarly to the previous two logging patterns, the implementation of the **Causal Logging** pattern can be combined with the implementation of a checkpoint pattern. Such combination restricts the amount of memory space needed to store logs and it facilitates garbage collection and trimming of the antecedence graph. A checkpoint identifies the backmost point in the execution of a component where the component can be rolled back. Hence, all logs and parts of the antecedence graph that refer to events which have happened before the checkpoint can be safely garbage collected.

## 6.6 Consequences

The **Causal Logging** pattern has the following benefits:

- + The **Causal Logging** pattern performs better than the **Pessimistic Logging** pattern in error-free executions and better than the **Optimistic Logging** pattern during recovery periods.
- + The recovery activity is localized at the component where the error has occurred.
- + The **Causal Logging** pattern eliminates orphan processes that may temporarily occur during the recovery activity.
- + Despite the potential temporary creation of orphan processes during recovery periods, the **Causal Logging** pattern is free of the exponential rollbacks phenomenon [11].
- + Combining the **Causal Logging** pattern with checkpoint patterns (e.g. see [10]) allows the system designer to fine-tune the system performance both during error-free executions and recovery periods by adjusting the frequency of checkpoints and of logs saved to *stable storage*.

The **Causal Logging** pattern imposes also some liabilities:

- The **Causal Logging** pattern performs worse than the **Pessimistic Logging** pattern during recovery periods and worse than the **Optimistic Logging** pattern in error-free executions.
- The design complexity of the *recovery manager* and the *logger* is high (e.g. as compared to the **Optimistic Logging** and **Pessimistic Logging** patterns) due to the need for causality tracking, frequent antecedence graph updates, and identification of the components that become orphan processes during recovery periods.
- Garbage collection is more complex and more time consuming than for the **Pessimistic Logging** pattern.

## 6.7 Related Patterns

The **Causal Logging** pattern is directly related to the **Log Keeping** pattern presented in Section 3, for which it provides an elaborated solution to the way the *logger* and *recovery manager* entities function. In the same sense, it also related to the other two logging patterns that have been previously presented in this paper.

## References

- [1] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal, and Optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, February 1998.
- [2] A. Borg, J. Baumbach, and S. Glazer. A Message Passing System Supporting Fault Tolerance. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 90–99, October 1983.
- [3] E.N. Elnozahy, L. Avisi, Y.-M. Wang, and D.B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.
- [4] E.N. Elnozahy and W. Zwaenepoel. Manetho: Transparent Rollback Recovery with Low Overhead, Limited Rollback and Fast Output Commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.
- [5] D.B. Johnson and W. Zwaenepoel. Sender-Based Message Logging. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, pages 14–19, July 1987.
- [6] J. C. Laprie, editor. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, 1992.
- [7] V.P. Nelson. Fault-Tolerant Computing: Fundamental Concepts. *IEEE Computer*, 23(7):19–25, July 1990.
- [8] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, June 1975.
- [9] T. Saridakis. A System of Patterns for Fault Tolerance. In *Proceedings of the 7th European Conference on Pattern Languages of Programs (EuroPLoP)*, pages 535–582, June 2002.
- [10] T. Saridakis. Design Patterns for Checkpoint-Based Rollback Recovery. In *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP)*, September 2003.
- [11] A.P. Sistla and J.L. Welch. Efficient Distributed Recovery Using Message Logging. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 223–238, August 1989.
- [12] R. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.