Design Patterns for Checkpoint-Based Rollback Recovery *

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

Abstract

Checkpoint-based rollback recovery is a very popular category of fault tolerance techniques, which are based on a simple idea: save the system state during error-free portions of the system execution; when an error occurs, use the saved state to rollback the system to a recent consistent state. This way, after an error occurs the system does not have to start its execution from the beginning, which would result in longer execution times or even failure of the system (e.g. when the I/O events that drove the execution of the system are not reproducible). This paper presents three design patterns that capture the most widely used methods for checkpoint-based rollback recovery. The Independent Checkpoint pattern describes the method where constituent components of a system take checkpoints without synchronizing with each other. Synchronization will take place after the occurrence of an error when a consistent system state must be re-established from the partial system states found in the checkpoints. The Coordinated Checkpoint pattern describes the method where constituent components of a system take checkpoints after synchronizing with each other. In this case, no synchronization is required during the re-establishment of a consistent system state after the occurrence of an error. Finally, the Communication-Induced Checkpoint pattern describes the methods where the synchronization of the checkpointing is triggered by communication events. This method combines the benefits of the previous two in terms of time and space overhead incurred to the system execution.

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 that state. In order to be able to do this, 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

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

The general idea behind checkpoint-based rollback recovery techniques is simple. However, elaborating the way this simple general idea can be applied in distributed systems brings forward a number of issues that complicate its application in distributed systems. A witness to the complexity of checkpoint-based rollback recovery techniques is a variety of obscure checkpoint algorithms that have been reported in the related literature. Such algorithms describe ways to synchronize the constituent components of a system when taking checkpoints (e.g. see [19]) and ways to reconstitute a consistent global state from checkpoints that contain partial representations of system states (e.g. see [22] and [13]). Two broad classifications of the checkpoint techniques are those that categorize them according to the software layer which is responsible for taking checkpoints, and those that categorize them according to synchronization of constituent components of a system during the checkpointing.

**Software layer responsible for checkpointing.** Assuming the common separation of the software in OS, middleware and application layers, one issue that checkpoint-based rollback recovery techniques have to address is the layer responsible for taking checkpoints. Taking checkpoints at the OS layer provides transparency of the checkpoint mechanism to the middleware and application layers (e.g. see [3] and [11]). This means that by putting in place the checkpoint mechanism once for a given OS, the middleware and all applications running on that OS can take advantage of it without necessitating and modification. On the other hand, deducing when middleware and applications are in a meaningful state that can checkpointed requires a substantial effort at the OS level. This category of checkpoint techniques is called transparent checkpoints. As an alternative to this approach, the application software can be responsible for taking checkpoints (e.g. see [5] and [20]). Although this solves the problem of identifying meaningful application states that can be checkpoint, it also implies that the checkpoint mechanism is application specific. Thus, all applications that need checkpoint-based rollback recovery have to be modified to integrate the checkpoint mechanism. This category of checkpoint techniques is called explicit checkpoints. Taking checkpoints at the middleware level (e.g. see [1]) attempts to combine the benefits of the previous two approaches while minimizing the compromises; applications are not modified as long as they run on top of the middleware that provides the checkpoint mechanism while, being close to the application layer, it is easier to identify meaningful application states at the middleware level. This category of checkpoint techniques is called implicit checkpoints. This category also includes compiler-assisted checkpoint insertion (e.g. see [16] and [15]).

**Checkpointing synchronization.** In explicit checkpoint, the application has the responsibility to trigger checkpoints. In implicit and transparent checkpoint however, where checkpoints are triggered by middleware or OS, some logic must be implemented to trigger the timing of checkpoints. One category of checkpoint timing techniques are those that coordinates all constituent components of a system when taking checkpoints (e.g. see [4]). The set of checkpoints taken after such coordination is guaranteed to form a consistent global state of the system for the cost of system-wide coordination. This category of checkpoint techniques is called coordinated checkpoints. Another category of checkpoint
timing techniques are those where constituent components of a system take their checkpoints without synchronizing with each other (e.g. [23]). The price to pay for the easiness of checkpointing is the need to find a subset of all taken checkpoints that constitutes a consistent global state of the system. This category of checkpoint techniques is called independent checkpoints. Finally, hybrid checkpoints techniques attempt to use communication events to trigger the checkpoints in order to combine the benefits of the two aforementioned techniques (e.g. see [12]). This category of checkpoint techniques is called communication-induced checkpoints.

In this paper, we describe three design patterns for checkpoint-based rollback recovery, which capture the three categories of checkpointing synchronization mentioned above. In Section 2 we provide some background definitions, including the system model and the necessary fault tolerance concepts. Then, the Independent Checkpoint, Coordinated Checkpoint, and Communication-Induced Checkpoint are presented in Sections 3, 4, and 5 respectively. The paper concludes with a summary of the presented patterns and a discussion on the factors that can lead to the selection of one of them over the others.

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. A state transition is triggered by the system’s logic, or by the reception of input from the environment, or by the delivery of output to the environment. Hence, state transitions model the I/O operations of a system. When a system is decomposed into its constituent components, each component is assigned a portion of the data the system operates on. The system decomposition may produce new state transitions 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
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 [14]. 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).

The term *unit of failure* is used to denote a part of a system that fails independently from other parts of the system. In the system model presented before, the boundaries of units of failures usually coincide with component boundaries. But it is also possible to map more than one components to a single unit of failure. In the latter case, if an error occurs in one of the components that belong to the same unit of failure, all the other components in the same unit of failure will be considered to have failed. In practice, the unit of failure sets the part of a system which is monitored by an error detection mechanism for errors. In practice, the units of failure are mapped to processes, individual computers, or even complete subnetworks, depending on the scale and the purpose of the overall system.

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 [17]. 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. 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 technique rely on taking checkpoints and using them after an error occurred to re-establish a consistent system state. During the recovery period, the checkpoints are used to reconstruct a consistent system state from the partial system states saved in the checkpoints taken by the constituent components. The set of the most recent checkpoints that belong to a consistent global state is called recovery line [20]. Once a consistent state is reconstructed, the system execution is “rolled back” to that state. Hence, rollback recovery techniques consist of two main activities: taking checkpoints and using them to reconstruct a consistent system state after the occurrence of an error. As we will see in the following sections, simplifying one of the two activities complicates the other.
3 Independent Checkpoint

There are certain types of systems that emphasize on high performance and that experience errors very rarely. Examples of such systems are those responsible for the billing of network connections and accesses to services (e.g. in a telecom operator network, in an ISP service network, etc). Such systems need solutions to system recovery that introduce minimum performance penalties during error-free execution, while they can afford rarely occurring recovery periods for restoring a global consistent state during which the system performance will be significantly slower than usual. The Independent Checkpoint pattern captures a solution to system recovery that fits systems with such characteristics.

3.1 Context

The context, in which the Independent Checkpoint 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 system has very tight performance constraints under error-free execution.

- Errors occur rarely and when they do, the system can afford to temporarily slow down or even stop its designated functionality in order to recover from the errors.

- 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 memory resources of the system that are allocated to system recovery are not tightly budgeted, i.e. there is ample memory space where the system recovery mechanism can save recovery data without having to worry about strict garbage collection.

This context describes systems whose components are widely distributed, possibly over distinct administrative domains. In such cases, unnecessary synchronization is not acceptable but when it becomes necessary, e.g. for system recovery purposes, the system components can afford to experience long delays in their executions.

An example of a system that has the above characteristics is the one that offers some web services with fee-based access. The system is composed of identification and authentication components, billing components and service and content access components. These three categories of components are loosely couples in terms of communication among them. For example, once the user is authenticated by the authentication components, the billing components will use the user and session IDs to charge the corresponding customer account with the access fees without having to continue interacting with the authentication components. The same statement is true for the interactions among the billing components and the components that are responsible to process the content before it is returned to the user.

The connection to the web services has very tight performance constraints, otherwise the users will turn to other, faster service providers. Errors are rare and in these rare
cases the service may appear unavailable to its users who will have to attempt to access it anew. Furthermore, the service provider can have file server dedicated to storing recovery data, hence ample storage space is available to the system recovery mechanism. What is important is that, in case of an error, the system restores a consistent state that contains correct billing data, otherwise either the service provider will lose money or users will complain for being billed for content they never accessed.

3.2 Problem

In the above context, the Independent Checkpoint pattern solves the problem of re-establishing a consistent system state after the occurrence of an error by balancing the following forces:

- Components need to save recovery data during error-free execution, in order to be able to recover from an error that may occur.

- Storing of recovery data introduces performance penalties which depend on the access latency of the media used to store the recovery data.

- The more frequently recovery data are saved by all components the more likely is to recover the system to a state closer to the point of the execution where the error occurred.

- The more frequently recovery data are saved the higher the performance penalty on error-free executions becomes.

- For each component, the identification of the recovery data that belong to a consistent global state has a high cost in terms of performance.

3.3 Solution

The Independent Checkpoint pattern suggest the following solution to the above problem. Each individual component in a system to decide on its own when to take checkpoints (i.e. when to save recovery data), without synchronizing with the checkpointing activity of other components. The lack of synchronization of the checkpointing activities of the individual components keeps low the performance penalty during error-free executions. This solution trades the low performance penalty during error-free execution with a high cost of the recovery activity that is launched after an error has occurred. During this activity, the components stop their execution and the recovery mechanism inspects the checkpoints taken by all components in the system in order to identify the recovery line. When the recovery line is identified, the recovery mechanism instructs each component to rollback to its checkpoint present in the identified recovery line. Upon reception of this instruction, each component loads the indicated checkpoint form stable storage and resumes execution from that point on.

In order for the recovery process to be able to identify the checkpoints that belong to the recovery line, each checkpoint must contain some information that relates this checkpoint to checkpoints previously taken by other components. The technique for recording in the independently taken checkpoints the information that relates them to other taken checkpoints is reported in [2] and it is summarized in the following. Each component $C_i$ takes a sequence of checkpoints $k_{i,0}$, $k_{i,1}$, ... during its execution. When a component $C_\alpha$,
after it has taken checkpoint $k_{\alpha,x}$, produces some output that is intended to be delivered as input to component $C_{\beta}$, it piggybacks in the corresponding message the information $(\alpha, x)$, i.e. the identity of the sender and the identity of the last checkpoint the sender has taken before producing the output in question. Component $C_{\beta}$ will deliver the input that corresponds to the aforementioned communication after having taken checkpoint $k_{\beta,y}$. When taking checkpoint $k_{\beta,y+1}$, component $C_{\beta}$ will also record the dependency of the state saved in that checkpoint to the checkpoint $k_{\alpha,x}$.

Based on this dependency information saved with the checkpoints, the recovery mechanism can identify the recovery line following different algorithms (e.g. see [2] and [23]). The basic idea behind these algorithms is to create a dependency graph among individually taken checkpoints and deduce from this graph which checkpoints belong to the recovery line.

**N.B.:** The solution suggested by the **Independent Checkpoint** 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 [21].

### 3.4 Structure

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

- The **recoverable process**, which is the component that checkpoints its state during error-free execution in order to be able to use it and recover from errors that it may experience.

- The **stable storage**, which is the part of the system where checkpoints are saved and which is not subject to errors.

- The **checkpointer**, which is responsible for the checkpoint activity of the **recoverable process**. The checkpointer decides when to take a checkpoint and it is responsible for transferring the recovery data (i.e. the component state to be checkpointed) 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 and issues a request to the system components to suspend their execution. Then, it inspects the checkpoints in stable storage and identifies the recovery line for the system. Finally, it informs each component in the system about which of its checkpoints the component must reload before it resumes its execution.

The **checkpointer** entity is usually, thought not necessarily, mapped to the same unit of failure as the **recoverable process**. On the other hand, both the **error detector** and the **recovery manager** must be mapped to different units of failure than the one where the **recoverable process** is mapped. The reason for this is because they are both used after an error has occurred on the **recoverable process** and if they belong to the same unit of failure as the latter then they will not be able to deliver their designated functionality.
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 checkpoint 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 system components during recovery phases.

Similarly to the stable storage the recovery manager may or may not be a single physical entity (e.g. a new component in the system). If the system developer chooses to distribute it over several physical entities (e.g. one instance of the recovery manager running on each system component), he must consider the subsequent time overhead due to the synchronization of the distributed instances. On the other hand, if the recovery manager is mapped on a single physical entity then the system developer must also deal with error that may occur on the recovery manager itself.

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

Both parts (a) and (b) of Figure 1 are simplified in order to capture the essentials of the Independent Checkpoint pattern without confusing the reader. What is important for the reader to notice in the depicted structure is that the recoverable process and the checkpointer 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. Moreover, a recoverable process corresponds to a distinguishable component in the system. Hence, in a system there will be as many recoverable processes and checkpointers as the number of the constituent system components that are mapped to separate units of failure.

3.5 Implementation

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.

The checkpointer 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. Such approaches allow checkpointing to be transparent to applications but result in large amounts of data with each checkpoint, some of which (often a large volume of them) are not relevant system recovery. Such approaches trade the time and space costs of the checkpointing process for the transparency of the checkpoint process to the application. Alternatively, the checkpointer can be explicitly programmed in the application, in which case the above arguments and the consequent tradeoff are reversed.

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. Since the system can afford slowing down or even halting 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). In many cases, the system developer may not even have to consider the fault tolerance of the recovery manager. In the context where the Independent Checkpoint pattern applies errors are rare. Hence, 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 sim-
ilar distributed implementation approach (e.g. see error detection patterns in [21]). 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 Independent Checkpoint pattern.

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 [21].

3.6 Consequences

The Independent Checkpoint pattern has the following benefits:

+ The time overhead introduced to the error-free system execution is kept very low and equals the time needed to save independently taken checkpoints to stable storage plus the time to piggyback checkpoint information to the output produced by each component. Depending on the implementation choice for the checkpointer and the ratio of the size of state data over the size of the executable image of a component, the performance penalty of the checkpointing activity can be fine-tuned to reach its minimum.

+ The complexity of identifying consistent global states impacts on the system only after an error has actually occurred, i.e. rarely, according to the context in which this pattern applies.

The Independent Checkpoint pattern imposes also some liabilities:

- The time overhead introduced to the system recovery activity is very high (the highest for the three design patterns presented in this paper). This is due to the fact that after the occurrence of an error, the execution of the system components is suspended until the recovery manager calculates the dependency graphs of the checkpoints in stable storage, identifies the recovery line, and informs each component about which checkpoint to load before resuming its execution.

- The lack of synchronization during the checkpoint activity of the components may cause the domino effect [20], i.e. the successive rollback of components from one checkpoint to an older one until all of them reach their initial states. This extreme rollback would cancel the benefits of rollback recovery since it would purge the system back to its initial state while at the same time the recovery process will suffer maximum time penalty.

- The space overhead introduced to the stable storage is very high for two reasons. First, since there is no synchronization when taking checkpoints, a number of checkpoints saved in stable storage may never belong to a recovery line. Although useless, these checkpoints occupy space in the stable storage. Second, and again due to the lack of synchronization during the checkpointing, it is not possible to garbage-collect old checkpoints that no longer belong to the current recovery line. The garbage collection can happen only after the recovery manager has identified the recovery line, which implies that garbage collection can happen only after the occurrence of errors.
- The Independent Checkpoint pattern does not apply to systems that have irreversible communication interactions with their environment (e.g. money dispensed by automatic teller machines, document printed by printers, calls made by phones, etc). In order to ensure that system recovery after the occurrence of errors will not lead to the re-creation of such irreversible communication events, the system components must synchronize their checkpointing activities before such event happen. Such synchronization contradicts the autonomy of components execution, which is a context invariant for the Independent Checkpoint pattern.

### 3.7 Related Patterns

The Independent Checkpoint pattern belongs to the continuation of the work that has been presented in [21]. It is one specialization of the Rollback pattern and it can be combined with the error detection patterns. Also, the Independent Checkpoint pattern has eminent similarities with the other two patterns presented in this paper.
4 Coordinated Checkpoint

A real-time system must meet certain deadlines in its execution. If these deadlines are not met, then the system will experience a failure. A potential reason for missing such deadlines are errors that may occur in the system’s constituent components. If these errors are not treated then they will propagate to other components and they will eventually cause the failure of the system. On the other hand, if these errors are treated, then their treatment must not delay the execution of the whole system, causing it to miss its execution deadlines and hence resulting in a system failure. The Coordinated Checkpoint pattern captures a solution to system recovery that keeps minimum the error-treatment time and allows real-time systems to meet their execution deadlines despite the occurrence of errors in their constituent components.

4.1 Context

The context, in which the Coordinated Checkpoint 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.
- After the occurrence of an error in a component, the time spent in recovering the component must be bounded and minimum.
- 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 memory resources of the system that are allocated to system recovery are strictly bound, i.e. memory recourses can store only a minimum amount of recovery data.

This context describes real-time systems and embedded systems with scarce memory resources. Such systems cannot afford to store excessive recovery data nor can they afford to slow down during recovery phases. Rather, they must store only a limited amount of recovery data, which is bounded to a fixed size known prior to the system execution. They must also complete a recovery phase within bounded time limits, which are too fixed prior to the system execution.

Examples of such systems are those that provide automatic navigation control. These systems consist of a number of sensor components, which return information about the environment where the system calculates the navigation course, plus a number of components that use this information to calculate, among others, trajectory, throttle and breaking values. Clearly, those systems cannot afford recovery phases which slow down the execution of the system, since such delays may lead to severe, even fatal, time failures.

Other examples of systems described by the context above are embedded systems found in smart consumer electronics (e.g. mobile phones and personal digital assistants, set-top boxes and game consoles). In those systems components are tightly coupled together, often serving interactions with humans. Due to the human interaction factor, the focus of these system is to provide consistent time behavior to their environment (i.e. the human)
irrespectively of system errors that should pass unnoticed by the human user. This implies that error-free execution can be moderately penalized by the recovery mechanism, as long as the recovery activity itself does not introduce a noticeable time overhead to the system execution.

4.2 Problem

In the above context, the **Coordinated Checkpoint** pattern solves the problem of re-establishing a consistent system state after the occurrence of an error by balancing the following forces:

- Components need to save recovery data during error-free execution, in order to be able to recover from an error that may occur.
- Storing of recovery data introduces performance penalties which depend on the access latency of the media used to store the recovery data.
- The more frequently recovery data are saved by all components the more likely is to recover the system to a state closer to the point of the execution where the error occurred.
- The more frequently recovery data are saved the higher the performance penalty on error-free executions becomes.
- Saving recovery data that belong to a consistent system state requires some kind of synchronization of all the components in the system.

4.3 Solution

The **Coordinated Checkpoint** pattern suggests the following solution to the above problem. System components synchronize before checkpointing in order to store recovery data that belong to a consistent global state. As a result, the set of the last checkpoints taken by all system components defines the recovery line of the system. The detection of an error in the system execution, which triggers the re-establishment of the system to a consistent global state, is trivial: each component has to load its last checkpoint and continue execution from there. The complexity of the recovery process has been moved from the identification of the recovery line (which as the case for the **Independent Checkpoint** pattern) to the coordination of the checkpointing.

A variety of techniques have been devised for coordinating the checkpointing activity. Conceptually, the simplest of them is the one that employs a two phase commit protocol for every checkpoint. When components must take a checkpoint, the rollback recovery mechanism issues an instruction to all components to take a checkpoint. Upon reception of this instruction, each component takes a tentative checkpoint and acknowledges the fact to the recovery mechanism. When all components have acknowledged their tentative checkpoints, the recovery mechanisms issues another instruction to all components for committing their tentative checkpoints. During the whole process, the regular component execution is suspended.

Blocking the regular system execution during the checkpointing is costly and in some cases introduces more time overhead during error-free execution than the system can afford. The main reason for blocking the system execution during checkpoint is to prevent
the delivery of input, which would result to inconsistent checkpoints. There are various alternatives to blocking system execution that provide solution to the problem of inconsistent checkpoints. If communication channels are FIFO then, before sending a message that corresponds to the first output produced after a checkpoint, a component sends to the same recipient a request forcing the recipient to take immediately a checkpoint (e.g. see [6]). If communication channels are not FIFO, the same effect can be achieved by attaching the request for forced checkpoint to every message that carries output produced after a checkpoint was taken by the component that produces the output. In this case the request for force checkpoint must also contain the index of the checkpoint taken by the component that produced the output.

Another alternative to blocking the system execution during checkpointing is the use of loosely synchronized clocks when they are available. Based on the loosely synchronized clocks and on the maximum deviation between these clocks, checkpointing can be scheduled on regular time intervals. If no error notification has been produced by the end of a time interval then the checkpoint taken during that interval is committed, otherwise the previously committed checkpoint is used to rollback the system (e.g. see [7]).

Finally, another optimization in the synchronization of the checkpointing activity is based on the observation that not all component in the system must synchronize at each checkpoint. Rather, only those that have either directly or indirectly communicated with the component that initiated the checkpoint must synchronize (minimal checkpoint). Identifying these components and performing a two-phase commit checkpointing only among them improves the performance of the Coordinated Checkpoint pattern implementation (e.g. see [13]).

N.B.: The solution suggested by the Coordinated Checkpoint 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 [21].

4.4 Structure

The solution to the problem of system recovery described by the Coordinated Checkpoint pattern outlines the following entities:

- The recoverable process, which is the component that checkpoints its state during error-free execution in order to be able to use it and recover from errors that it may experience.

- The stable storage, which is the part of the system where checkpoints are saved and which is not subject to errors.

- The checkpointer, which is responsible for the checkpoint activity of the recoverable process. This checkpointing activity includes the decision when to take a checkpoint, the transferring the recovery data (i.e. the component state to be checkpointed) to the stable storage, and the coordination with checkpointer instances serving other components in the system.

- 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. Upon reception of an error notification produced by the error detector, this entity issues a request to the system components, which instructs them to suspend their execution, load from the stable storage their latest checkpoint and resume their execution from there.

The same comments for these entities apply here as those that can be found in Subsection 3.4. Figure 2a provides an intuitive illustration of the structure of the Coordinated Checkpoint pattern. Figure 2b contains the activity diagram that describes the functionality of the checkpointing and the consistent state re-establishment activities of the Coordinated Checkpoint pattern.

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

Both parts (a) and (b) of Figure 2 are simplified in order to capture the essentials of the Coordinated Checkpoint pattern without confusing the reader. What is important for the reader to notice in the depicted structure is that the recoverable process and the checkpointer 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. Moreover, a recoverable process corresponds to a distinguishable component in the system. Hence, in a
system there will be as many recoverable processes and checkpointers as the number of the constituent system components that are mapped to separate units of failure.

4.5 Implementation

The implementation issues of the Coordinated Checkpoint pattern are very similar to those of the Independent Checkpoint pattern mentioned in Subsection 3.5. One point where the implementation concerns of those two patterns differ is the recovery manager. Deploying the recovery manager on a single physical entity is acceptable only if the occurrence of errors in it is completely hidden (e.g. by means of failure masking) from its environment. Otherwise, the time overhead to recover the recovery manager from the occurrence of an error may lead to unacceptable delays in the recovery phase of the system, which may result in time failures for the system. Failure masking is a costly means of fault tolerance (e.g. see [21]) both in terms of complexity and in terms of required resources like CPU and memory. These costs can be a prohibitive factor in embedded systems for deploying the recovery manager on a single entity.

On the other hand, having instances of the recovery manager distributed across system components is an appealing alternative. The functionality of the recovery manager is fairly small (suspend, load checkpoint, resume) so replicating it on every system component may be acceptable. The main issue that the developer has to deal with in this case is the synchronization of the distributed instances of the recovery manager, which is necessary at the beginning of the recovery phase.

4.6 Consequences

The Coordinated Checkpoint pattern has the following benefits:

+ The time overhead of the recovery phase (i.e. the re-establishment of a consistent state) is very low and it amounts to the time needed by the recovery manager to instruct all component to suspend their execution, load their last checkpoint, and resume their execution, plus the time that the components need to load their last checkpoint.

+ The Coordinated Checkpoint pattern does not suffer from the domino effect. This means that a system recovery will be completed within bounded time.

+ The communication overhead during the consistent state re-establishment activity of the system recovery is also very low and it amounts to the emission of the instructions from the recovery manager to all components to suspend their execution, load their last checkpoint and resume.

+ The identification of the recovery line is trivial since each set of checkpoints taken by the system components is guaranteed to belong to the recovery line. In the minimal checkpoint optimization, where not all components need to rollback, the identification of the recovery line becomes less trivial, but still remains fairly simple [13].

+ The space overhead introduced by the Coordinated Checkpoint pattern in stable storage is very low. Only the latest checkpoint of each component must be kept in stable storage; previous checkpoints can be garbage collected at any time convenient for the system execution. The minimal checkpoint optimization introduces
a more elevated space overhead in stable storage since more than one checkpoint for some components may need to be stored. Still, this overhead is lower than the one introduced by the Independent Checkpoint pattern, which may even save in stable storage useless checkpoints that can never belong to a recovery line.

The Coordinated Checkpoint pattern imposes also some liabilities:

- It introduces a time overhead in error-free executions of the system, which comes from the synchronization of the components at each checkpoint. Depending on the synchronization scheme used, this overhead may range from the very high overhead imposed by a two-phase commit synchronization that blocks the execution of all components during checkpointing down to moderate overhead of synchronization based on loosely synchronized clocks [7].

- The Coordinated Checkpoint pattern introduces a communication overhead in error-free execution of the system which can also range from high to none. The highest overhead is introduced by the two-phase commit checkpointing where the synchronization activity requires 3N additional messages for a system with N components. The non-blocking versions of the Coordinated Checkpoint pattern require less synchronization messages (or they just piggyback the synchronization requests in regular system messages [9]). Finally, the checkpointing coordination based on loosely synchronized clock does not introduce any additional communication overhead.

4.7 Related Patterns

The Coordinated Checkpoint pattern is another specialization of the Rollback pattern [21], and similar to the Independent Checkpoint pattern, it has a number of common characteristics with the other two patterns presented in this paper.
5 Communication-Induced Checkpoint

The Independent Checkpoint and Coordinated Checkpoint patterns provide two extreme alternatives: either low overhead during checkpointing and high overhead during the consistent state re-establishment at the recovery phase or high overhead during checkpointing and low overhead during the consistent state re-establishment at the recovery phase. The Communication-Induced Checkpoint pattern comes to bridge the gap between those two extremes. It suggests a combination of forced and local checkpoints, which keeps the checkpointing overhead lower than that of the Coordinated Checkpoint pattern while eliminating the domino effect risk of the Independent Checkpoint pattern.

5.1 Context

The context, in which the Communication-Induced Checkpoint 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 system must be able to treat errors that occur in its components within bounded time.

- The system runs on hardware that is capable of performing complex computations (e.g. PC-like hardware with CPU and RAM that allows graph-related computation as opposed to special-purpose hardware dedicated to signal processing).

- The system can afford only minimum synchronization of its components beyond that already compelled by its specification.

- 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 bandwidth of the communication channels in the system is not saturated, i.e. the system can afford to increase the volume of data exchanged among its components.

- The system does not suffer from scarce resources, i.e. the CPU and memory allocated to its components are not very tightly budgeted.

This context describes real-time systems that are not constrained on scarce resources and special-purpose hardware such as PLCs and DSPs. Such systems can afford to increase the communication traffic among their constituent components. They can also afford to perform complex computations in addition to those necessary for their designated function. What these systems cannot afford is to slow down during system recovery or to suffer time penalties during error-free execution such as those caused by the checkpointing activity in the Coordinated Checkpoint pattern.

Examples of such systems can be found in stock-market software. Delays in the system execution cause money loss, which for the given domain is the type of failure with primary importance. The components of those systems are often distributed on a large
scale, making the inter-component communication one of the most costly activities in the system. Increasing the volume of the data contained in each message is not a problem, but increasing the number of inter-component messages in the system is not acceptable. This implies that the system recovery of those systems must rely heavily on local computations at each component and not affect the number of messages exchanged among components.

5.2 Problem

In the above context, the Communication-Induced Checkpoint pattern solves the problem of re-establishing a consistent system state after the occurrence of an error by balancing the following forces:

- Components need to save recovery data during error-free execution, in order to be able to recover from an error that may occur.
- Storing of recovery data introduces performance penalties which depend on the access latency of the media used to store the recovery data.
- The more frequently recovery data are saved by all components the more likely is to recover the system to a state closer to the point of the execution where the error occurred.
- The more frequently recovery data are saved the higher the performance penalty on error-free executions becomes.
- Saving recovery data that belong to a consistent system state requires some kind of synchronization of all the components in the system.

5.3 Solution

The solution suggested by the Communication-Induced Checkpoint pattern is a combination of the solutions suggested by the Independent Checkpoint and the Coordinated Checkpoint patterns. Components in the system take checkpoints both independently (local checkpoints) and after implicit synchronization with other components (forced checkpoints). The former checkpoints are taken without any synchronization with other components, exactly as in the Independent Checkpoint pattern. Hence, relying only on local checkpoints could cause the system recovery to experience a domino effect, which means unbounded time to identify the recovery line and complete the recovery phase. Clearly, this is unacceptable in the context of this pattern. In order to avoid the domino effect, forced checkpoints are taken, ensuring the existence of a recovery line different from the system’s initial state and closer to the error occurrence in the system execution.

The implicit synchronization that leads to forced checkpoints is a combination of the ideas behind the non-blocking version of the Coordinated Checkpoint pattern and the identification of a recovery line in the Independent Checkpoint pattern. Regular communication piggybacks information regarding the checkpoints that the sender has taken. The receiver can then use this information to decide whether or not to take a forced checkpoint. A forced checkpoint helps to avoid rendering useless the checkpoint that the sender of the communication has previously taken. The algorithms used to deduce whether a forced checkpoint must be taken or not are based on the construction of a Z-path and the detection of Z-cycles in it [18].
The algorithms for deciding on force checkpoints are classified into two broad categories: the model-based checkpointing (e.g. see [23]) and the index-based coordination (e.g. see [10]). Algorithms in the first category build a probabilistic model that captures the possibility of the occurrence of Z-cycles (and hence useless checkpoints) and use this information to decide on forced checkpoints. Algorithms in this category result in more forced checkpoints than what is actually needed. Algorithms in the second category place timestamps to checkpoints and piggyback these timestamps in the regular communication among the system components. Timestamps are used to instruct checkpointing in such a way that checkpoints with the same timestamps belong to the same recovery line. Additional information can be included in component communication to minimize the number of forced checkpoints.

**N.B.**: The solution suggested by the Communication-Induced Checkpoint 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 [21].

### 5.4 Structure

The solution to the problem of system recovery described by the Independent Checkpoint pattern outlines the following entities:

- The **recoverable process**, which is the component that checkpoints its state during error-free execution in order to be able to use it and recover from errors that it may experience.

- The **stable storage**, which is the part of the system where checkpoints are saved and which is not subject to errors.

- The **checkpointer**, which is responsible for the checkpoint activity of the recoverable process. The checkpointer decides when to take a checkpoint (local or forced) and it is responsible for transferring the recovery data (i.e. the component state to be checkpointed) 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. Upon reception of an error notification produced by the error detector, this entity issues a request to the system components, which instructs them to suspend their execution, load from the stable storage their latest checkpoint and resume their execution from there.

The same comments for these entities apply here as those that can be found in Subsection 3.4. Figure 3a illustrates graphically the structure of the Communication-Induced Checkpoint pattern. Figure 3b contains the activity diagram that describes the functionality of the checkpointing and the consistent state re-establishment activities of the Communication-Induced Checkpoint pattern.

Both parts (a) and (b) of Figure 3 are simplified in order to capture the essentials of the Coordinated Checkpoint pattern without confusing the reader. What is important for the reader to notice in the depicted structure is that the recoverable process and the checkpointer 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. Moreover, a recoverable process corresponds to a distinguishable component in the system. Hence, in a system there will be as many recoverable processes and checkpointers as the number of the constituent system components that are mapped to separate units of failure.

5.5 Implementation

The implementation issues of the Communication-Induced Checkpoint pattern are very similar to those of the Independent Checkpoint and Coordinated Checkpoint patterns mentioned respectively in Subsections 3.5 and 4.5.

5.6 Consequences

The Communication-Induced Checkpoint pattern has the following benefits:

+ The time overhead is kept relatively low, both for the checkpointing and the con-
sistent state re-establishment activities of the rollback recovery mechanism.

+ The Communication-Induced Checkpoint pattern does not suffer from the domino effect. This means that a system recovery will be completed within bounded time.

+ Communication overhead, in terms of number of exchanged messages, is kept relatively low, since this pattern does not require explicit synchronization messages exchanged among the system components.

+ The space overhead is kept relatively low, since useless checkpoint that can never be part of a recovery line are eliminated by the mechanism that takes forced checkpoints.

The Communication-Induced Checkpoint pattern imposes also some liabilities:

- The volume of the exchanged messages is increased due to the piggybacked timestamps that the checkpointer adds to each exchanged message.

- Under regular functioning, the Communication-Induced Checkpoint pattern introduces more time overhead in error-free executions than the Independent Checkpoint pattern and more space overhead and time overhead during system recovery than the Coordinated Checkpoint pattern. To minimize one of these overheads would mean transforming the Communication-Induced Checkpoint pattern to one of the other two patterns, losing thus the combined benefits. Hence, this pattern cannot perform equally well as the best cases of each of the Independent Checkpoint and Coordinated Checkpoint patterns.

5.7 Related Patterns

The Communication-Induced Checkpoint pattern is also specialization of the Rollback pattern [21], and it has a number of common characteristics with the other two patterns presented in this paper.
6 Epilogue

This paper has presented three design patterns for checkpoint-based rollback recovery, which distill parts of the results on system recovery in distributed systems [8]. This work is a complement to previous work on design pattern for fault tolerance [21] and it can be used to extend the classification of the pattern system presented there. The presented patterns offer alternative solutions to the system recovery problem, each at a different cost in terms of time, space and communication overhead introduced to the system execution.

The Independent Checkpoint pattern is ideal for the development of systems with demanding performance constraints during error-free executions, that do not experience errors often and when they do they can afford to go off service for repairing the failure. The low cost of the checkpointing activity associated to this pattern does not penalize significantly the error-free execution of the system, while the elevated recovery overhead that this pattern introduces can be tolerated when the system goes off service for repair.

The Coordinated Checkpoint pattern is addressed more to the development of systems that have bounded time constraints (yet not high performance ones) on their execution, and they cannot afford long execution delays due to system recovery. In other words, while not reaching very high performance, the system execution characteristics (time and space) need to keep similar values during error-free execution and system recovery periods. The time overhead introduced by the coordinated checkpointing activity can be adjusted (e.g. by using non-blocking checkpointing techniques or loosely synchronized clocks if available) to keep the time overhead during error-free system execution within the acceptable limits. On the other hand, the very low time overhead of the re-establishment of a consistent global state after the occurrence of an error will allow the system recovery activity to impose minimum performance penalties on system execution.

The Communication-Induced Checkpoint pattern is meant for high-performance real-time systems that can perform general purpose computations (as opposed to those system that can perform only special purpose computations such as signal processing). It can also be tuned to fit in most of the cases where the previous two pattern can be employed, except from those extreme cases where the system requires highly optimum performance during error-free executions or highly optimum recovery time after the occurrence of an error. This pattern improves the weaknesses of the previous two pattern, for the price of somewhat lower performance compared to the best performance each of the other two patterns can show.

Acknowledgments

The author would like to thank his shepherd, Lisa DiPippo, for her insightful comments, her time and her patience through the revisions of this paper. Also, the paper owes its final form to the constructive criticism and the suggestions of the participants of the Distributed Real-Time and Embedded Systems focus group in PLoP 2003: Lisa DiPippo, Linda Elkin, Richard Gabriel, Christopher Gill, Robert Hamner, Jerry James, Chang Liu, Toni Marinucci, Douglas Niehaus, Linda Rising, and Paul Rubel.
References


