A System of Patterns for Fault Tolerance*

Titos Saridakis

NOKIA Research Center
PO Box 407, FIN-00045 NOKIA Group, Finland
Tel: (+358) 7180 37293 Fax: (+358) 7180 36308
titos.saridakis@nokia.com

ABSTRACT

Many fault tolerance techniques that have been devised, applied and improved over the past three decades represent general solutions to recurring problems in the design of fault tolerant computer systems. This document presents some of the best known such techniques, formatted as patterns and organized by a classification scheme into a system of patterns for fault tolerance. This pattern system reveals the relations among the presented patterns for fault tolerance and delineates a number of ways in which these patterns can be used to refine each other. In turn, these refinement relations create design frameworks for the development of fault tolerant systems with different efficiency and complexity characteristics.

Keywords: Design Framework, Fault Tolerance Pattern, Pattern classification.

1 INTRODUCTION

The indissoluble bonds of computers and failures have produced a plurality of fault tolerance techniques that can satisfy, potentially, any requirement regarding the behavior of a computer system in the presence of faults. Consequently, the development of fault tolerant systems does no longer rely on the (re)invention of ways to deal with various faults that may occur; rather, it relies on the selection of the most appropriate one among the well-understood fault tolerance techniques. Each such technique provides a solution to a recurring fault tolerance problem under a set of clearly defined assumptions about the type of the failures it deals with and the constraints about the system behavior it guarantees. Hence, a well-understood fault tolerance technique outlines a pattern that applies to concrete problems in the design of fault tolerant systems in the specific context defined by aforementioned assumptions and constraints. In this document a set of fault tolerance techniques are formatted and presented as patterns following a form similar to the one used in [3].

In general terms, fault tolerance provides techniques to confront faults and their consequences in a system. These techniques describe the detection of errors in a system, and the means that ensure the recovery of a system from errors or the masking of errors in a system. The patterns presented in this document cover all these three constituents of fault tolerance, which are error detection, recovery and masking. The different ways in which the presented patterns can be combined to produce a complete solution for the design of fault tolerant systems are captured in a classification scheme for the fault tolerance patterns. This classification scheme transforms the

presented set of patterns into a system of patterns that provides guidelines on how to refine the design of a system to transform it into its fault tolerant counterpart.

Although the patterns presented in this document provide solutions to fault tolerance problems, the content of this document is addressed to software designers and architects and not to fault tolerance experts. The presented patterns capture widely used fault tolerance techniques from the field of distributed systems [10] and their comprehension does not require profound fault tolerance expertise.

Each pattern in this document presents a solution to a specific problem in detecting, recovering from, or masking an error. Combining these patterns according to the guidelines given by the classification scheme provides complete solutions to fault tolerance problems in the design of a system. Hence, each such combination of patterns forms the basis for a design framework for fault tolerant systems with specific properties (e.g. regarding the failure types they can cope with, the number of simultaneous errors they can tolerate, the time and complexity overhead of the fault tolerant mechanisms, etc).

The remainder of this document is organized in four sections. Section 2 contains a summary of the background information regarding fault tolerance that is necessary for a non-expert to follow the presentation. Section 3 presents a set of patterns that capture well-understood fault tolerance techniques for error detection, recovery and masking. In section 4 these patterns are organized as a pattern system with the help of a classification scheme that reveals their relations (mainly dependency and refinement relations). The same section contains also a discussion on the relation of the presented system of fault tolerance patterns with other well-known pattern systems. The document concludes in section 5 with a brief summary of the presented work, a brief evaluation of the importance of the system of fault tolerance patterns in the software design and some reflections on the future of pattern systems for problems specific to non-functional properties (e.g. security, timeliness, configurability, etc).

2 BACKGROUND

The long term and profound study of failures in computer systems has delivered a clear understanding of the different types of failures that may occur and a variety of techniques for dealing with them. The intent of this section is to provide a summary of the fault tolerance background that would help system designers and architects to probe deep into the patterns presented in the following section. This background is taken from [7] and it includes the system model adopted in this document, the definitions of fault tolerance related terms, a description of the failure types considered in the context of this document, and a brief presentation of essential fault tolerance concepts.

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 passage of time and its internal logic. By “well-defined behavior” we mean 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. It may be possible to decompose a system into constituent (sub)systems. In CBSE terms a system is a component that may consists of the assembly of a number of smaller components. In OO terms a system is a composition of objects, each of which may be itself a composition of smaller objects.

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 its constituent (sub)systems, 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. 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, which in turn makes the application crash which is a **failure**. By crashing, the application leaves blocked the socket ports it used which is a **fault** and the computer on which the application crashed has socket ports which are not used by any process nevertheless not accessible to running applications which is an **error**, and which in turn leads to a **failure** when another application requests these ports.

Based on the above, a fault in a system may propagate to the system's environment. A system is called fault tolerant when it can deal with faults and their consequent errors in such a way that it does not violate its specification, i.e. the environment of a fault tolerant system does not perceive a failure of the system. Hence, a fault tolerant system does not propagate faults to its environment. Fault tolerance techniques are practical methods that describe how to detect an error and confine it within a system. The confinement can be based on the restoration of the subsystem on which the error was detected before that error infects other parts of the system, or it can be based on the masking of the error occurrence (e.g. by isolating the subsystem on which the error was detected and using some form of redundancy to deliver the expected output).

Each fault tolerance technique provides different guarantees regarding the properties associated to the system qualities such as the time or the space overhead introduced to the normal execution of the system, the efficiency of the reaction to a failure, the design complexity added to the system, etc. In general, fault tolerance techniques are based on the following principles:

- Constituents of a fault tolerant system monitor other constituents for failure occurrences. By observing a failure, the monitoring subsystem can detect an error on the monitored subsystem. These monitoring activities are often called *error detection*.
- In order to enable the restoration of a subsystem after an error has been detected on it, appropriate information regarding the subsystem may be saved when certain conditions are met (e.g. at regular time intervals, right after the subsystem delivers some output according to its specification, when the subsystem decides by its own to save the appropriate information, etc). This saving activity is often called *checkpointing.* The appropriate information save in a checkpointing activity may vary from a complete snapshot of the internal subsystem representation (i.e. the state of the subsystem) to selected piece of its internal representation that have changed since the last checkpoint.
- When a monitoring subsystem observes a failure on a monitored subsystem, it may activate a mechanism that will use the last checkpoint of the latter subsystem in order to eliminate the error that led to the observed failure and restore the subsystem to an error-free state. These restoration activities are often called *error recovery*.

- In some cases, when a monitoring subsystem observes a failure on a monitored subsystem, it does not let the erroneous behavior of the latter subsystem affect any other parts of the overall system by using a some form of redundancy (e.g. a duplicate of the failed subsystem) to cover up for the observed failure. These activities are often called *error masking*.

Before proceeding with the design of a fault tolerant system, the designer must determine the following two issues. First, the system designer must determine the type of failures that will be confronted by the fault tolerance mechanism. Different fault tolerance techniques have been developed to deal with different failure types and they may differ in all three means for error detection, recovery and masking. Hence, the failure types that will be confronted by a system play a decisive role in the selection of the fault tolerance techniques that can be applied to render the system fault tolerant. Some representative failure types are (see [10] for more information on failure types):

- *fail-stop* failures where the failed system ceases execution without producing any output and the failure is detectable by its environment,
- *crash* failures where the failed subsystem ceases execution without producing any output but the failure might not be detectable by its environment,
- *omission* failures where a subsystem fails to deliver output to (send omission), or receive input from (receive omission) its environment, and
- *byzantine* failures where the failed subsystem exhibits arbitrary behavior.

The second issue that must be determined is the *unit of failure* in the fault tolerant system. The unit of failure is the minimum part of the system (i.e. the minimum subsystem) where an error will be confined. Given the recursive decomposition of a system into subsystems, any subsystem may potentially be decomposed to smaller constituent systems. By defining the unit of failure, the system designer determines the subsystems that will be monitored for failures. These subsystems may not be fault tolerant themselves, and the fault tolerance mechanism that will be put in place will not provide any guarantees about them experiencing failures. However, their composition will contain error detection, recovery and/or masking activities that will render the resulting system fault tolerant with respect to the faults that may appear inside each unit of failure. For example, in a distributed system consisting of a number of machines interconnected over a network, the unit of failure can be set to be a machine or the set of processes that belong to the same application running on a single machine, or even the individual processes.

Once the failure type and the unit of failure issues are sorted out, the designer has a clear indication about the what fault tolerance mechanisms to choose and where to apply them in the system in order to make it fault tolerant. Still, a number of other factors will influence the final decision of the exact fault tolerance mechanism to be employed and its exact configuration. These factors include the number of simultaneous errors that may occur, the design, space and time complexity of the fault tolerant mechanism and how these align with the requirements about the corresponding system qualities, etc.

3 FAULT TOLERANCE PATTERNS

This section presents in the form of patterns a selection of fault tolerance techniques that deal with error detection, recovery and masking.

3.1 Fail-Stop Processor

Dealing with byzantine failures is extremely difficult and costly because of the arbitrary nature of the error that leads to the failure. For example, a system that exhibits failures of byzantine semantics may deliver erroneous output, or no output at all, or it may duplicate (correct or erroneous) output. Dealing with all different possible errors is costly (e.g. different detection techniques for different types of errors) and bears a big overhead regarding the design, space and time complexity of the system. When developing a system out of constituents which by their very nature may experience byzantine failure, it is desirable to transform these constituents to constituents with equivalent functional specification but with more "designer friendly" failure semantics. The **`Fail-Stop Processor`** pattern [14] describes one way for achieving that.

3.1.1 Context

The **`Fail-Stop Processor`** pattern applies to a system that has the following characteristics:

- The system is *deterministic*, i.e. its output is solely defined by its initial state, the sequence of inputs it has processed so far and the current time (in terms of clock time and/or time elapsed since the system initialization).
- The errors the system may experience are *transient*, i.e. as opposite to permanent errors like those caused by algorithmic faults.
- The errors the system may experience are not due to *errors in the input* it receives.
- The errors the system may experience cause it to exhibit *byzantine* failures.

3.1.2 Problem

In the above context, the **`Fail-Stop Processor`** pattern solves the problem of transforming the byzantine failures to fail-stop failures by balancing the following forces:

- The error is confined within the failed system and does not infect its environment.
- The error is detected by the environment.
- The time overhead on error-free system execution is kept very low.

3.1.3 Solution

The solution to the above problem suggested by the **`Fail-Stop Processor`** pattern is based on the replication of the system and the comparison of the replicas output for unanimity. Each one of the replicas is called a *processor* in the remainder. All processors are identical to the system on which the **`Fail-Stop Processor`** pattern is applied, hence they are deterministic. They are all initialized simultaneously and they receive exactly the same input, hence at any given time and in the absence of errors they must produce the same output. If the output produced by the processors are not exactly the same, then an error has occurred and the ensemble of the processors must be shut down in order to prevent the propagation of the error in the environment.

Note that since the processors are identical, if a software bug exists in one of them then it exists in all and it may cause an error on all processors that will be manifested in the same erroneous output, which cannot be detected by comparing the processors' outputs. This is why the context in which the **`Fail-Stop Processor`** pattern applies assumes only transient errors. Similarly, if the error in the system is a consequence of erroneous input, then both processors will produce the same (erroneous) output and the error will not be detectable by comparison of the processors' outputs. Hence, the context in which this pattern applies does not cover erroneous input.

The number of simultaneous errors that this solution can deal with depends on the number of the processors that have been created. In general, with *N+1* processors this solution guarantees to provide fail-stop failure semantics in the presence of *N* simultaneous errors on the processors. The simplest form of this solution consists of two processors and is capable of transforming a single error of byzantine nature on one of the processors into a fail-stop failure.

3.1.4 Structure

The solution suggested by the **`Fail-Stop Processor`** pattern consists of three entities:

- The *processors*, which are deterministic systems, they are identical to each other and they may experience byzantine failures. To deal with *N* simultaneous errors *N+1* processors are required, each of them mapped to a different unit of failure.
- The *distributor*, which ensures that all the processors receive exactly the same input (in terms of content and delivery order). It must be mapped to a different unit of failure than any of the processors in order not to get affected by the errors that may occur on them.
- The *comparator*, which receives the outputs of all the processors and compares them. If there is not unanimity in the outputs or if not all processors deliver output, the comparator notifies the environment about the error and stops producing any further output. The comparator must be mapped to a different unit of failure that any of the processors in order not to get affected by the errors that may occur on them.

Figure 1a illustrates graphically the structure of a simple fail-stop processor with two processors. All input to the fail-stop processor is received by the distributor, which subsequently sends it to both processors. All output of the two processors is sent to the comparator which decides whether an error has occurred on any of the processors and if so, it shuts down the whole fail-stop processor. Figure 1b gives the activity diagram that describes the fail-stop processor.

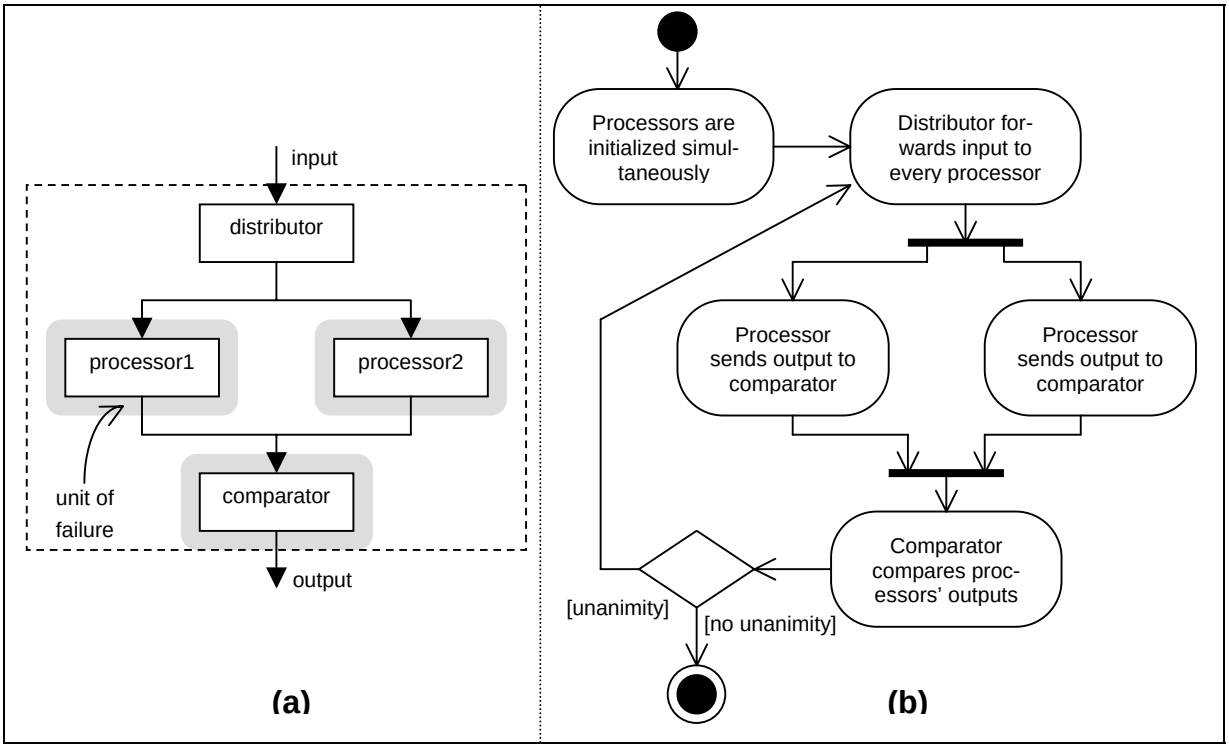

Figure 1. The structure (a) and the activity diagram (b) of the **`Fail-Stop Processor`** pattern.

Besides the above entities involved in the solution suggested by the **`Fail-Stop Processor`** pattern, there is another entity needed which will be responsible for putting in place the fail-stop processor mechanism, i.e. for initializing all processor simultaneously and assuring that they have the same initial state. However, this entity is not specific to the fault tolerance solution suggested by this pattern and hence it is not further discussed here.

3.1.5 Consequences

The **`Fail-Stop Processor`** pattern has the following benefits:

+ It introduces low time overhead since the processors function in parallel. The overhead amounts to the time it takes for the distributor to multicast the input it receives and the time it takes for the comparator to compare and deliver the output.
+ The design complexity introduced by the **`Fail-Stop Processor`** pattern is low, since the distributor and comparator, which are entities specific to this pattern, have quite simple functionality.
+ Since many fault tolerance techniques assume crash failures, the **`Fail-Stop Processor`** pattern is the basic building block for many fault tolerant architectures that have to deal with byzantine failures.
+ The processors are replicas of the original system on which the **`Fail-Stop Processor`** pattern is applied, without any additional functionality. This means that in practice the processors can be replicas of a legacy system, which cannot be subject to any internal changes such as those that are needed if additional functionality would be required by the processors.

The **Fail-Stop Processor** pattern imposes also some liabilities:

- It introduces relatively elevated space overhead that is proportional to the number of simultaneous errors it can deal with.
- Although the processors may have byzantine failure semantics, the distributor and the comparator must not experience byzantine failures. The **Fail-Stop Processor** pattern can be recursively applied to each of these entities but eventually there must be distinct units of failure in the system, which do not experience failures of byzantine type.
- The distributor and the comparator introduce single points of failure in the system. The distributor can be replaced by a atomic broadcast protocol if such is available. Both the distributor and the comparator can be render fault tolerant by applying some of the following fault tolerance patterns. In any case, the resulting design complexity of the **Fail-Stop Processor** pattern is elevated compared to the one graphically presented in Figure 1.
- The **Fail-Stop Processor** pattern does not provide means to tolerate faults in a system. Rather, it provides means detect errors and to render the most demanding type of failures (byzantine failures) into fail-stop failures that are must easier to deal with from a design and an implementation standpoint.

3.1.6 Related Patterns

The **Fail-Stop Processor** pattern mainly aims at transforming errors that lead to byzantine failures into errors that lead to fail-stop failures. The evolution of this pattern that actually masks errors that lead to byzantine failures is the **Active Replication** pattern (see section 3.10).

3.2 Acknowledgment

One way to detect crash failures is to have the subsystem, which receives some input, to acknowledge the reception to the sender. The sender, which is charged with the error detection responsibility, sets a timer right after providing the input to the monitored subsystem and the latter must acknowledge the reception of the input within a given time interval, often called a *timeout*. If no acknowledgement arrives at the sender within the predefined timeout, the sender can deduce that an error has occurred on the monitored subsystem.

3.2.1 Context

The **`Acknowledgment`** pattern applies to a system that has the following characteristics:

- The errors the monitored system may experience cause it to exhibit *omission* or *crash* failures.
- The frequency of interactions between the monitored system and monitoring system may *vary* a lot.
- The time it takes for the monitoring system to contact the monitored system is *bound* and *known*.

3.2.2 Problem

In the above context, the **`Acknowledgment`** pattern solves the problem of detecting an error on a system by balancing the following forces:

- The time overhead introduced by the detection mechanism should be kept to a minimum.
- The information that the monitored system has failed is of importance *only* when that system is in use (i.e. it has received some input and it is expected to produce the corresponding output).
- The communication between the monitored system and the monitoring system must not increase unnecessarily, e.g. there must not be any communication overhead when the monitoring system does not wish to interact with the monitored system.

3.2.3 Solution

The solution to the above problem suggested by the **`Acknowledgment`** pattern is based on acknowledging the reception of input within a given time interval. Once the monitoring system provides input to the monitored system, it set a local timer to a predefined timeout. While the timer is counting down from the timeout, the monitoring system waits to receive an acknowledgment regarding the input reception by the monitored system. If the acknowledgment arrives before the timeout is expired then the monitored system is considered to function correctly, otherwise an error will be detected to have occurred on the monitored system.

Notice that the solution can detect errors that lead to omission or crash failures of the monitored system, but by no means does it guarantee the correctness of the input reception by the monitored system or the correct process of the input by the latter.

Sometimes, especially in distributed systems, the lack of an acknowledgment within the timeout period might be due to transient communication errors. In order to prevent such failures from resulting into the detection of an error on a system that has

not actually experienced one, it is possible to set a number of successive attempts to send the same input and wait for the corresponding acknowledgement before the monitoring system deduces that an error has occurred on the monitored system.

3.2.4 Structure

The solution suggested by the **`Acknowledgment`** pattern consists of the following entities:

- The *sender*, which together with the next entity (timer) constitute the monitoring system. The sender is the system that needs to contact the monitored system.
- The *timer*, which is responsible for counting down from the timeout every time an input is provided to the monitored system. When the timeout period expires for *N* consecutive times without receiving an acknowledgment from the monitored system, the timer detects an error on the monitored system and informs the sender. The timer must be mapped to a different unit of failure than the monitored system (usually it is mapped to the same unit of failure as the sender of the input to the monitored system).
- The *receiver*, which together with the next entity (acknowledger) constitute the monitored system. The receiver is the system to which the sender intends to contact.
- The *acknowledger*, which is responsible to send an acknowledgment to the timer every time the monitored system receives input. The acknowledger must be mapped to the same unit of failure as the receiver.

Figure 2 gives a graphical illustration of the structure and the activity diagram of the **`Acknowledgment`** pattern.

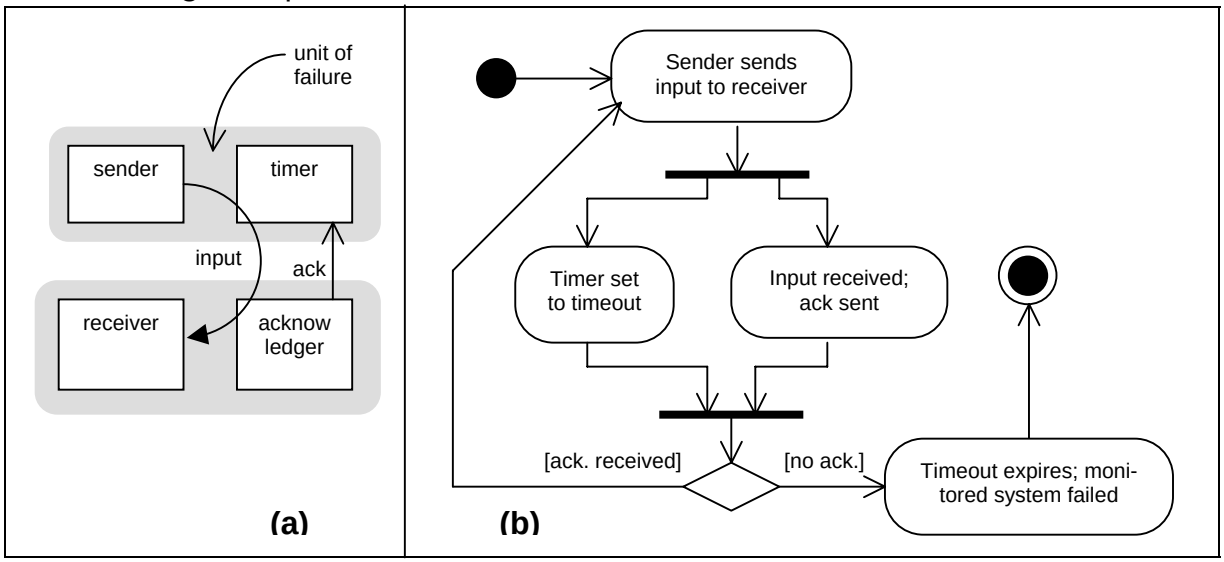

Figure 2. The structure (a) and the activity diagram (b) of the **`Acknowledgment`** pattern.

3.2.5 Consequences

The **`Acknowledgment`** pattern has the following benefits:

+ It introduces low time overhead that amounts to the time needed to set the timer.
+ The design complexity introduced by the **`Acknowledgment`** pattern is very low, as shown in Figure 2.

+ The **Acknowledgment** pattern does not introduce any space overhead. Both the timer and the acknowledger entities do not map to additional architectural or software components; rather they describe some additional functionality associated with the monitoring and the monitored system respectively.
+ In the case of OO systems, where the interaction among subsystems (objects) is based on method invocations, the acknowledgment can be merged with the reply to an invocation, reducing further the design complexity introduced by the **Acknowledgment** pattern.

The **Acknowledgment** pattern imposes also some liabilities:

– The error on the monitored system is detected only after some input has been issued to it. This means that although an error might have already occurred on the system long time ago, it will remain undetected until the moment when some input is sent to the monitored system. Hence, the time overhead for detecting an error can be quite elevated.
– The timeout must be set based on the time it takes for the input to reach the monitored system plus the time it takes for the acknowledge to reach the monitoring system. In many cases it is very difficult to have an exact figure for the aforementioned times, and the timeout is usually the result of estimations based on statistical observations. Hence, configuring an optimum timeout is not a trivial task.
– The timer and the acknowledger represent functionalities that are added to the monitoring and the monitored system respectively. Consequently, the application of this pattern on legacy systems might be very demanding in terms of implementation effort and space overhead (e.g. in case the legacy system needs to be encapsulated in a container which provides the timer or the acknowledger functionalities.

3.2.6 Related Patterns

In order to ensure that the error detection provided by the **Acknowledgment** pattern concerns a real problem (i.e. a crash of the receiver) and not just a transient communication failure, the **Acknowledgment** pattern can be combined with other patterns like the **Riding Over Transients** and the **Leaky Bucket Counter** patterns from [1].

3.3 I Am Alive

Another way to detect crash failures is to receive in regular time intervals notifications from the monitored system as a sign that it is alive. These “I am alive” signals serve as indication of the well-being of the monitored system.

3.3.1 Context

The **`I Am Alive`** pattern applies to a system that has the following characteristics:

- The errors the monitored system may experience cause it to exhibit *crash* failures.
- The frequency of interactions between the monitored system and monitoring system may *vary* a lot.
- The frequency of communication between the monitored system and the monitoring system is clearly *below the saturation limit* of the communication network.
- The time interval between successive outputs from the monitored system is *not bound* or *not known*.
- The time it takes for the monitoring system to contact the monitored system is *bound* and *known*.

3.3.2 Problem

In the above context, the **`I Am Alive`** pattern solves the problem of detecting an error on a system by balancing the following forces:

- The detection of an error on the monitored system must take place as soon as possible, even before the environment needs to communicate with the monitored system.
- The monitoring system must have a regularly updated knowledge regarding the occurrence of errors on the monitored system.

3.3.3 Solution

Given the variation in the interaction frequency between the monitoring and the monitor systems and the unbound time interval between their successive interactions, applying the **`Acknowledgment`** pattern will not allow the monitoring system to have a regularly updated knowledge regarding the occurrence of errors on the monitored system. The solution to the above problem suggested by the **`I Am Alive`** pattern is based on the regular notification of the monitored system well-being to the monitoring system. In order to minimize the time needed to detect an error in the case of long idle communication periods, the monitored system must send in regular time intervals, called again timeouts, a signal of its well-being (e.g. a message saying "I am alive"). On the other hand, the monitoring system sets a timer to the value of the timeout and waits for the "I am alive" signal from the monitored system. If the timeout expires without receiving the expected "I am alive" signal then the monitoring system detects an error on the monitored one.

Notice that the solution can detect errors that lead to crash failures of the monitored system, but by no means does it guarantee the correctness of the monitored system execution.

Similar to the **`Acknowledgment`** pattern, the **`I Am Alive`** pattern can be slightly modified to tolerate transient communication failures and not let them to lead to the detection of an error on a correct system. The technique is the same: instead of a single timeout, the monitoring system waits for *N* consecutive timeout periods for the

"I am alive" signal. Only if the *N*-th timeout expires without having received any "I am alive" signals, the monitoring system detects an error on the monitored one.

3.3.4 Structure

The solution suggested by the **`I Am Alive`** pattern consists of the following entities:

- The *monitor*, which together with the next entity (timer) constitute the monitoring system. The monitor is the system that needs to know about the well-being of the monitored system.
- The *timer*, which is responsible for counting down from the timeout continuously until it receives an "I am alive" signal or until the timeout expires. In the former case it just resets the timeout and starts the countdown anew; in the latter case it detects an error on the monitored system. If the timer is configured to deal with *N-1* transient communication failures, the error will be detected only after the *N*-th consecutive timeout has expired without receiving any "I am alive" signal. The timer must be mapped to a different unit of failure than the monitored system (usually it is mapped to the same unit of failure as the monitor).
- The *subject*, which together with the next entity (beacon) constitute the monitored system. The subject is the system which the monitor needs to know about its well-being.
- The *beacon*, which sends "I am alive" signals in regular time intervals that are smaller or equal to the timeout the timer uses. The beacon must be mapped to the same unit of failure as the monitored system.

Figure 3 gives a graphical illustration of the structure and the activity diagram of the **`I Am Alive`** pattern.

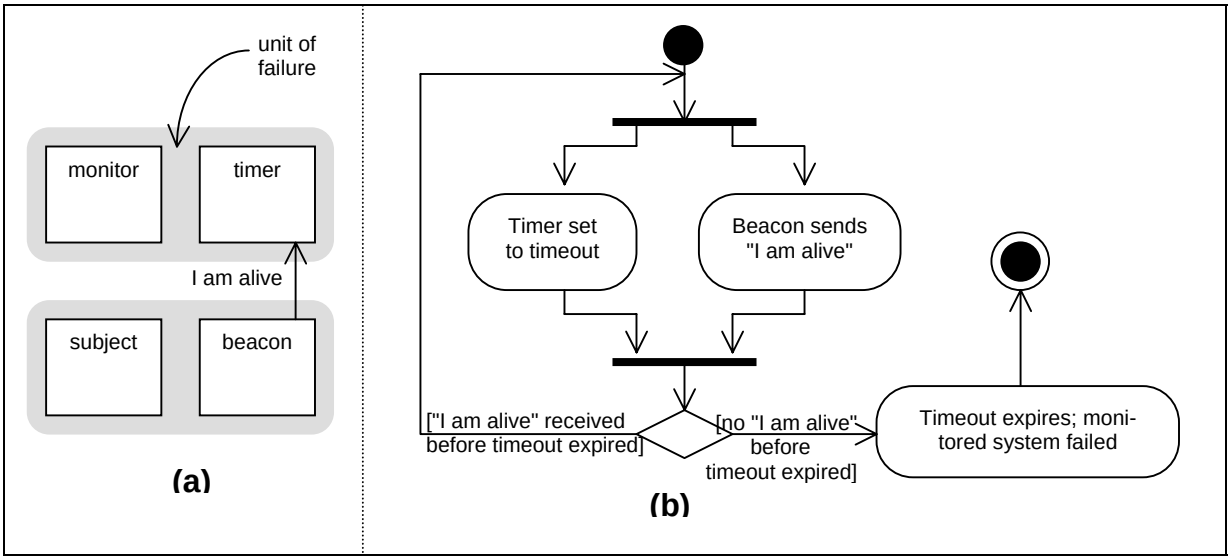

Figure 3. The structure (a) and the activity diagram (b) of the **`I Am Alive`** pattern.

3.3.5 Consequences

The **`I Am Alive`** pattern has the following benefits:

+ It introduces low time overhead (setting the timer at the monitoring system and sending the "I am alive" signal at the monitored system).
+ The design complexity introduced by the **`I Am Alive`** pattern is low, as shown in Figure 3.

+ The **I Am Alive** pattern does not introduce any space overhead. Both the timer and the beacon entities do not map to additional architectural or software components; rather they describe some additional functionality associated with the monitoring and the monitored system respectively.
+ The **I Am Alive** pattern detects error on the monitored system at a regular basis, even during long idle communication periods.

The **I Am Alive** pattern imposes also some liabilities:

– It introduces a communication overhead (due to the "I am alive" signals) even when the monitored system is not active (i.e. it does not receive any input and it is not expected to produce any output other than the "I am alive" signals).
– It introduces computational overhead both to the monitoring and to the monitored system (the timer and the beacon respectively) even in the case of idle communication periods.
– The timeout must be set based on the time it takes for the "I am alive" signal to reach the monitoring system. In many cases it is very difficult to have an exact figure for the aforementioned times, and the timeout is usually the result of estimations based on statistical observations. Hence, configuring an optimum timeout is not a trivial task.

3.3.6 Related Patterns

In order to ensure that the error detection provided by the **I Am Alive** pattern concerns a real problem (i.e. a crash of the subject) and not just a transient communication failure, the **I Am Alive** pattern can be combined with other patterns like the **Riding Over Transients** and the **Leaky Bucket Counter** patterns from [1].

3.4 Are You Alive

The **I Am Alive** pattern introduced communication and computational overhead even in idle communication periods as the price for detecting an error on the monitored system in minimum time. A more flexible solution regarding that overhead is to have the monitoring system probing the monitored system for its well-being. This solution is captured in the **Are You Alive** pattern.

3.4.1 Context

The **Are You Alive** pattern applies to a system that has the following characteristics:

- The errors the monitored system may experience cause it to exhibit *crash* failures.
- The frequency of interactions between the monitored system and monitoring system may *vary* a lot.
- The frequency of communication between the monitored system and the monitoring system is clearly *below the saturation limit* of the communication network.
- The time interval between successive outputs from the monitored system is *not bound* or *not known*.
- The time it takes for the monitoring system to contact the monitored system is *bound* and *known*.

3.4.2 Problem

In the above context, the **Are You Alive** pattern solves the problem of detecting an error on a system by balancing the following forces:

- Errors occurred on the monitored system must be detected in the shortest period that is convenient for the monitoring system.
- The communication overhead introduced must be relatively low.

3.4.3 Solution

In order to avoid the regular communication introduced by the **I Am Alive** pattern and at the same time to be able to detect an error on the monitored system even though no input needs to be provided to it like in the **Acknowledgment** pattern, the monitoring system may probe the monitored one whenever it wishes by sending an "are you alive" signal to it. Upon reception of that signal, the monitored system must send back an acknowledgment to inform the monitoring system about its well-being. This acknowledgment is very similar to the one send in the **Acknowledgment** pattern as a reaction to the communication from the monitoring system. After sending the "are you alive" signal, the monitoring system sets a timer to a predefined timeout and waits for the acknowledgment before that timeout expires. If the acknowledgment does not arrive within the timeout period then the monitoring system detects an error on the monitored one.

Notice that the solution can detect errors that lead to crash failures of the monitored system, but by no means does it guarantee the correctness of the monitored system execution.

Similar to the previous two patterns, the **Are You Alive** pattern can be slightly modified to tolerate transient communication failures and not let them to lead to the detection an error on a correct system. The technique is the same: instead of a single timeout, the monitoring system waits for *N* consecutive timeout periods for the ac-

knowledgment of the "are you alive" signal. Only if the *N*-th timeout expires without having received any acknowledgments, the monitoring system detects an error on the monitored one.

3.4.4 Structure

The solution suggested by the **`Are You Alive`** pattern consists of the following entities:

- The *monitor*, which together with the next entity (timer) constitute the monitoring system. The monitor is the system that needs to know about the well-being of the monitored system.
- The *timer*, which is responsible for sending an "are you alive" signal, setting a timeout and counting down from it while waiting for the acknowledgment to arrive. If no acknowledgment arrives before the timeout expires, the timer detects an error on the monitored system. If the timer is configured to deal with *N-1* transient communication failures, after the timeout expires the timer will send again an "are you alive" signal and repeat the previous process. An error will be detected only after the *N*-th consecutive timeout has expired without receiving any acknowledgment. The timer must be mapped to a different unit of failure than the monitored system (usually it is mapped to the same unit of failure as the monitor).
- The *subject*, which together with the next entity (acknowledger) constitute the monitored system. The subject is the system, which the monitor needs to know about its well-being.
- The *acknowledger*, which is responsible to send an acknowledgment to the timer every time it receives an "are you alive" signal. The acknowledger must be mapped to the same unit of failure as the monitored system.

Figure 4 gives a graphical illustration of the structure and the activity diagram of the **`Are You Alive`** pattern.

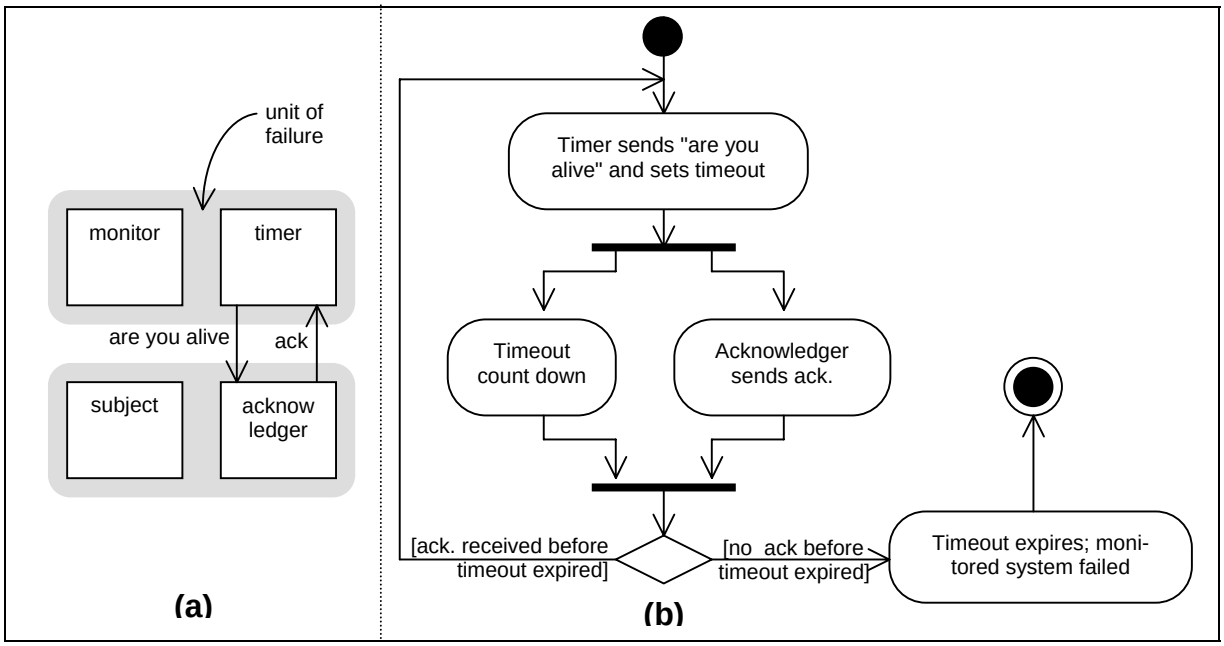

Figure 4. The structure (a) and the activity diagram (b) of the **`Are You Alive`** pattern.

3.4.5 Consequences

The **Are You Alive** pattern has the following benefits:

+ It introduces low time overhead (at the monitoring system the overhead amounts to setting the timer and sending the "are you alive" signal and at the monitored system the overhead amounts to sending the acknowledgment).
+ The design complexity introduced by the **Are You Alive** pattern is low, as shown in Figure 4.
+ This pattern does not introduce any space overhead. Both the timer and the acknowledger entities do not map to additional architectural or software components; rather they describe some additional functionality associated with the monitoring and the monitored system respectively.
+ This pattern detects errors on the monitored system on demand from the monitoring system.

The **Are You Alive** pattern imposes also some liabilities:

– It introduces communication overhead (due to the are-you-alive signals) even when there is no need for communication according to the system specification.
– It introduces computational overhead both to the monitoring and to the monitored system (the timer and the acknowledger respectively) even in the case of idle communication periods.
– Like in the previous two patterns, the setting an optimum timeout is not a trivial task.

3.4.6 Related patterns

In order to ensure that the error detection provided by the **Are You Alive** pattern concerns a real problem (i.e. a crash of the subject) and not just a transient communication failure, the **Are You Alive** pattern can be combined with other patterns like the **Riding Over Transients** and the **Leaky Bucket Counter** patterns from [1].

3.5 Roll Forward

Once an error has been detected on the monitored system, the system must recover from it in order to qualify as fault tolerant. As a reminder, error recovery includes the actions of re-establishing the last correct state that has been saved in some previous checkpoint. One way of achieving error recovery is to have the fault tolerant system consisting of two replicas. One replica will be reacting to every input sent to the fault tolerant system. If no error occurs on that replica and that replica produces successfully the output designated by its specification, then the second replica is rolled forward to the new system state. If an error occurs while the replica is processing the received input, then the failed replica is discarded and the second replica remains unaffected by the error.

3.5.1 Context

The **`Roll Forward`** pattern applies to a system that has the following characteristics:

- The errors the system may experience are *detectable*.
- The errors the system may experience are not due to *errors in the input* it receives.
- Error-free executions of the system are clearly below any time constraints imposed on them.
- The system is capable of *exporting* its current state and *importing* a new state.

3.5.2 Problem

In the above context, the **`Roll Forward`** pattern solves the problem of recovering from an error on a system by balancing the following forces:

- The time to recover from an error must be kept minimum.
- The error-free system execution must not violate any time constraints imposed on it.

3.5.3 Solution

The solution to the above problem suggested by the **`Roll Forward`** pattern is based on the use of two replicas of the system. One replica will process the new input to the system and if no error occurs then the other replica will roll forward to the new state. The operation of rolling forward is based on the first replica exporting the new system state and the second replica importing it. If an error occurs on the first replica during the processing of the input or the delivering of the output or during the export of its new state, then that replica is discarded and the second replica, which is unaffected by the occurred error, takes over the responsibility of providing the functionality expected by the fault tolerant system. If an error occurs on the second replica during the import of the new state, then the second replica is discarded and the first replica, which is unaffected by the occurred error, takes over the responsibility of providing the functionality expected by the fault tolerant system.

Notice that the failure semantics to which the occurred errors may lead are not important for the solution suggested by the **`Roll Forward`** pattern. The only requirement is that the error is detectable, which is one of the characteristics of the context in which this pattern applies.

The above solution can recover from the occurrence of a single error. However, it cannot deal with the occurrence of two or more simultaneous errors (e.g. one error occurring on the first replica and at the same time another error occurring on the second replica). To be able to recover from *2N* simultaneous errors we need to create *2N+1* replicas so at least one will remain unaffected by the occurred errors.

Another limitation of the original solution is that the fault tolerant system loses its fault tolerance capabilities after recovering from the first error occurrence. Even in the revised solution for dealing with *2N* simultaneous errors, the fault tolerant system eventually loses its fault tolerance capabilities (e.g. after the occurrence of *2N* simultaneous errors or after the occurrence of *2N* consecutive errors). To preserve the fault tolerance capabilities of the fault tolerant system, a mechanism that offers dynamic management of the replica group must be put in place. The dynamic replica group management will be responsible to replace with new ones the replicas discarded after the occurrence of errors.

3.5.4 Structure

The solution suggested by the **`Roll Forward`** pattern consists of the following entities:

- The *replicas*, which are copies of the original system identical to each other and which are monitored for errors. Each replica is capable of exporting its state and importing a new state upon request. Each replica must be mapped to a different unit of failure.
- The *manager*, which is responsible for receiving all input meant for the fault tolerant system and forwarding it to the appropriate replica. In the absence of errors, the manager triggers the copy of the new state from the replica that processed the latest input to the other replicas that kept a previously error-free state. The manager also relies on an error detection mechanism to detect errors that may occur on the replicas. When such an error occurs on a replica, the manager is responsible for discarding that replica. The manager must be mapped to a different unit of failure than any of the replicas.

Figure 5 gives a graphical illustration of the structure and the activity diagram of the **`Roll Forward`** pattern. In Figure 5a, block arrows indicate flow of information and the open arrow labeled “copy state” indicates a signal from the manager to the replica which treats the input.

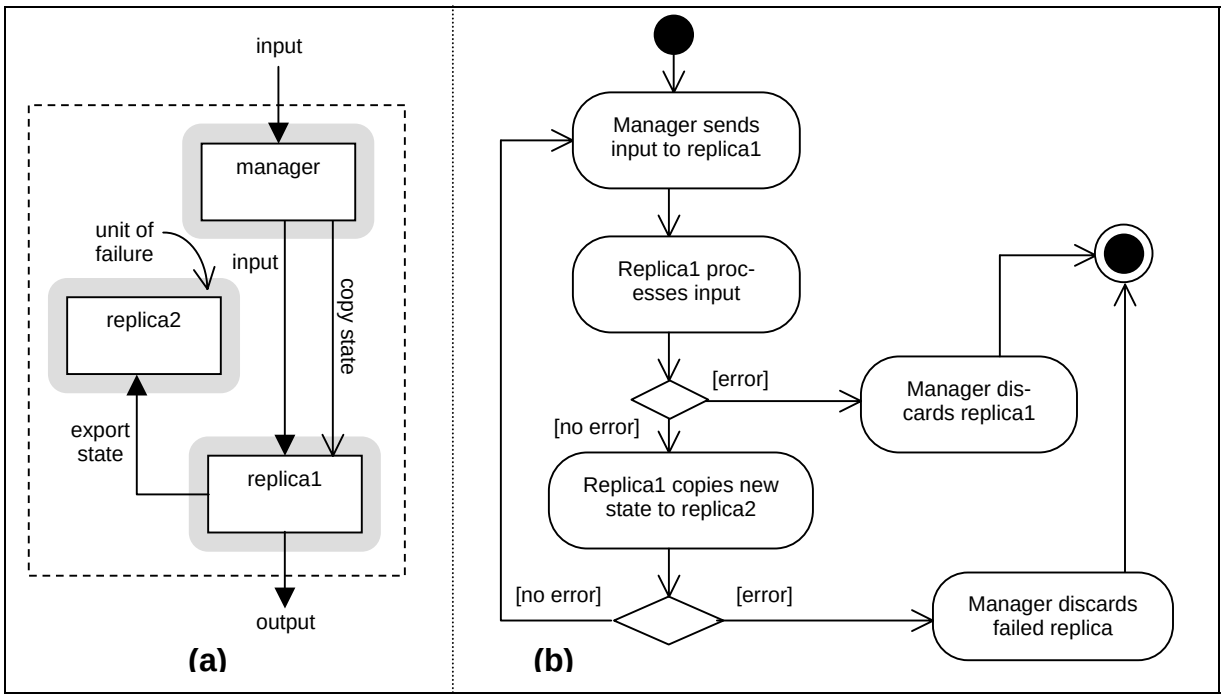

Figure 5. The structure (a) and the activity diagram (b) of the **`Roll Forward`** pattern.

3.5.5 Consequences

The **`Roll Forward`** pattern has the following benefits:

+ The design complexity associated to this pattern is relatively low and it amounts to the introduction of the manager and the roll forward operation (copy the state from one replica to the other).
+ The time overhead imposed by this pattern is low when errors occur: the failed replica is discarded, and the unaffected replica processes the subsequent inputs.
+ Since the manager is responsible for triggering the copy of state from one replica to the other, the frequency of “copy state” operations can be adjusted to meet the desired tradeoff between performance (small number of “copy state” operations) and loss of information in case of an error (number of processing steps since last “copy state” operation).

The **`Roll Forward`** pattern imposes also some liabilities:

– The space overhead introduced by this pattern is relatively elevated; the entire system is replicated.
– The time overhead imposed by this pattern in the absence of errors is high; before the replica is able to receive and process new input, it must copy its new state to the other replica.

3.5.6 Related patterns

The manager entity in the **`Roll Forward`** pattern monitors the replicas for errors. The mechanism for the error detection can be based on one of the **`Acknowledgment`**, **`I Am Alive`** and **`Are You Alive`** patterns presented in sections 3.2, 3.3 and 3.4 respectively.

3.6 Rollback

The **Rollback** pattern presents another way to recover from the occurrence of errors using system replicas. One replica receives and processes the input meant for the fault tolerant system and at certain moments it can checkpoint its state (i.e. export its state to an entity that is not affected by its errors). When an error occurs, the second replica uses the checkpoint to restore some error-free state the former replica had reached before the error occurred.

3.6.1 Context

The **Rollback** pattern applies to a system that has the following characteristics:

- The errors the system may experience are *detectable*.
- The errors the system may experience are not due to *errors in the input* it receives.
- The system is capable of *exporting* its current state and *importing* a new state.

3.6.2 Problem

In the above context, the **Rollback** pattern solves the problem of recovering from an error on the system by balancing the following forces:

- The time overhead for error-free system execution must be kept minimum.
- The error-free state restored after an error occurred is as close as possible to the last error-free state the failed replica had reached before it experienced the error.

3.6.3 Solution

The solution to the above problem suggested by the **Rollback** pattern is based on the use of two replicas of the system and a storage where the checkpoints are kept. The replica, which receives and processes the input meant for the fault tolerant system, exports to the storage at certain moments the state it has reached. If the copying of state from one replica to the other has low time overhead, then the second replica can play the role of the storage. Otherwise, the storage can be a piece of shared memory or a file in a file system common to the two replicas. If an error occurs on the first replica during the processing of the input, that replica is discarded and the second replica imports the last checkpoint and uses it to roll back to the last known error-free state of the fault tolerant system.

Notice that the failure semantics to which the occurred errors may lead are not important for the solution suggested by the **Rollback** pattern. The only requirement is that the error is detectable, which is one of the characteristics of the context in which this pattern applies.

The above solution can recover from the occurrence of a single error. However, it cannot deal with the occurrence of two or more simultaneous errors on the replicas (e.g. one error occurring on the first replica and at the same time another error occurring on the second replica). To be able to recover from *2N* simultaneous errors on the replicas we need to create *2N+1* replicas so at least one will remain unaffected by the occurred errors.

Another limitation of the original solution is that the fault tolerant system loses its fault tolerance capabilities after recovering from the first error occurrence. Even in the revised solution for dealing with *2N* simultaneous errors, the fault tolerant system eventually loses its fault tolerance capabilities (e.g. after the occurrence of *2N* simultane-

ous errors or after the occurrence of *2N* consecutive errors). To preserve the fault tolerance capabilities of the fault tolerant system, a mechanism that offers dynamic management of the replica group must be put in place. The dynamic replica group management will be responsible to replace with new ones the replicas discarded after the occurrence of errors.

If the errors that may occur on the fault tolerant system do not lead to fail-stop or crash failures where the failed system ceases execution, the system replicas are not necessary. The single copy of the system can checkpoint at certain moments its state to the storage and when an error occurs, the failed system which has not crashed can import the last checkpoint and continue its execution from there.

3.6.4 Structure

The solution suggested by the **`Rollback`** pattern consists of the following entities:

- The *replicas*, which are copies of the original system identical to each other and which are monitored for errors. Each replica is capable of exporting its state and importing a new state upon request. Each replica must be mapped to a different unit of failure.
- The *storage*, which is used to store the checkpoints that contain the state that the replica that processes the input exports at certain moments. The replicas may replace the storage in the following way: when the checkpoints are created, each replica exports them to the other replicas and imports form them the checkpoints these replicas have created. However, if the storage is used then it must not be subject to errors, i.e. it must behave like *stable storage* that survives errors.
- The *manager*, which is responsible for receiving all input meant for the fault tolerant system and forwarding it to the appropriate replica. In the absence of errors, the manager triggers the copy of the new state from the replica that processed the latest input to the storage. The manager also relies on an error detection mechanism to detect errors that may occur on the replicas. When such an error occurs on a replica, the manager is responsible for discarding that replica. The manager must be mapped to a different unit of failure than any of the replicas.

Figure 6 gives a graphical illustration of the structure and the activity diagram of the **`Rollback`** pattern. In Figure 6a, block arrows indicate flow of information and open arrows indicate signals from the manager to the storage and to the replica which treats the input.

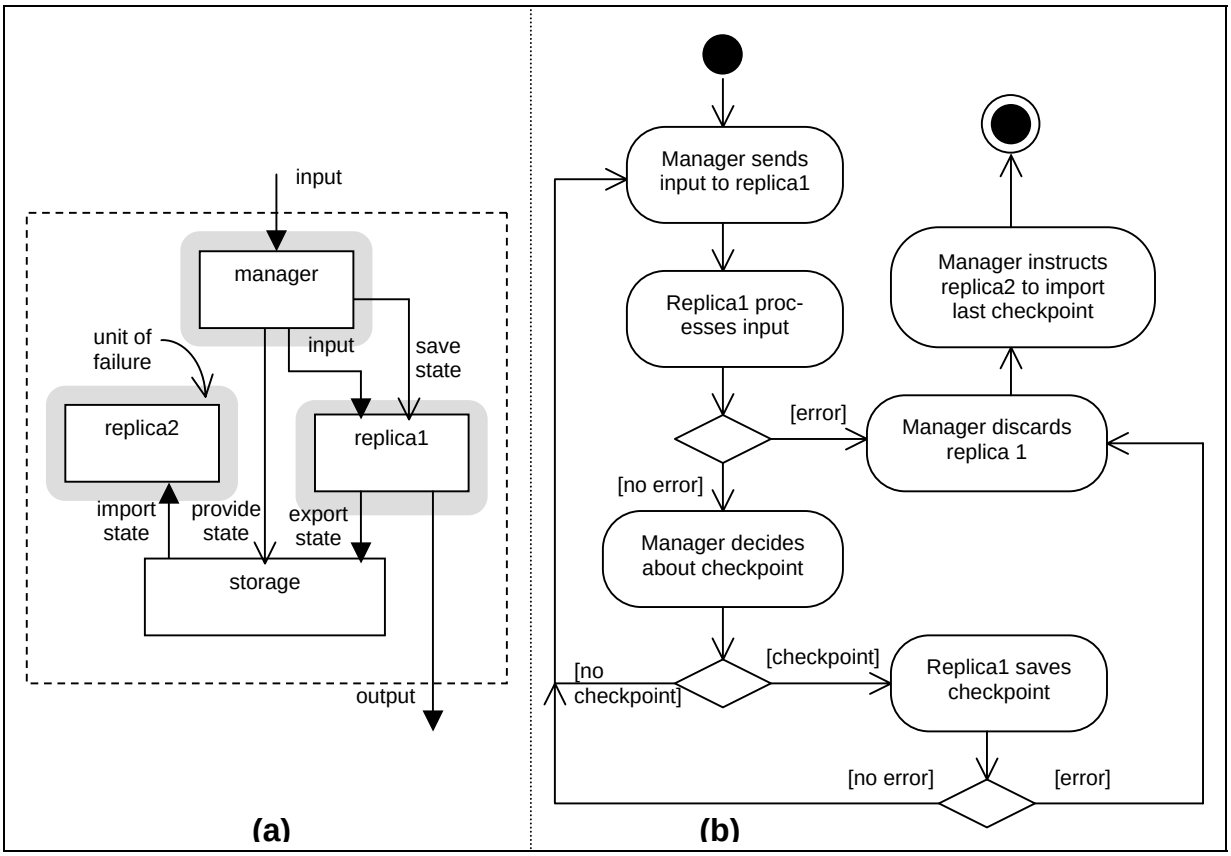

Figure 6. The structure (a) and the activity diagram (b) of the **`Rollback`** pattern.

3.6.5 Consequences

The **`Rollback`** pattern has the following benefits:

+ The design complexity of this pattern is relatively low (comparable to the design complexity of the **`Roll Forward`** pattern) and it amounts to the introduction of the manager and the checkpoint operation.
+ The time overhead introduced by this pattern during error-free system execution amounts to the checkpoint operation when the manager instructs it. Hence the time overhead depends on the frequency in which the manager instructs the checkpoint operation and it can be tuned to take optimal values for a given system.
+ An extreme configuration of the **`Rollback`** pattern is the one where the manager instructs a checkpoint only at the start up of the system. This scheme is often met with the name "system purge" because it rolls the system back to the initial state cleaning it from any possible error effects. The system purge scheme removes the design complexity and the time overhead associated to the checkpoints.

The **`Rollback`** pattern imposes also some liabilities:

– The space overhead introduced by this pattern is relatively elevated, including the replicas and the storage. One way to circumvent this liability is to use only checkpoints. If the fault tolerant system does not experience errors that lead to fail-stop of crash failures then the manager can instruct the failed system to import the last checkpoint and to restore the last error-free state. If the errors experienced by the fault tolerant system do lead to fail-stop or crash failures, then a dynamic replica

management can be used to create a replica just in time to import the last checkpoint. This scheme reduces the space overhead of the **Rollback** pattern but it increases the time overhead in the presence of an error since a new copy of the system must be created and initialized to the last checkpoint.

- The time overhead introduced by this pattern in the presence of an error is relatively elevated and it amounts to the time needed to import the last checkpoint by the replica that takes over the processing of the input to the fault tolerant system.
- Tuning the manager to perform checkpoints in optimal intervals is a difficult task and it requires extensive study of the system implementation in order to create a statistical model that indicates the frequency of errors at different periods of the system operation.
- If the platform where the fault tolerant system will be deployed does not offer any persistent storage or stable storage facilities which could be used to accommodate the storage entity of the **Rollback** pattern, then the designing this entity in a way that does not introduce a single point of failure in the fault tolerant system introduces a significant design overhead.
- In the extreme case of the **Rollback** pattern (the system purge scheme), the time overhead in the presence of an error is elevated since it includes the time necessary to create and initialize a new copy of the system. Moreover, the new copy of the system is found at its initial state and all information associated with the execution of the failed system up until its failure is completely lost. This is often too restrictive to let the system purge scheme be of practical use.

3.6.6 Related patterns

The manager entity in the **Rollback** pattern monitors the replicas for errors. The mechanism for the error detection can be based on one of the **Acknowledgment**, **I Am Alive** and **Are You Alive** patterns presented in sections 3.2, 3.3 and 3.4 respectively.

3.7 Passive Replication

The **Roll Forward** and the **Rollback** patterns provide solutions for recovering from errors that may occur on a system. However, the processing performed by the fault tolerant system while the error occurred is lost in both cases and in the best case the system recovers to the state it had reached right before starting to process the last input it received before the error occurred. In many circumstances this loss of information is not good enough for the fault tolerant system under design. Rather, solutions that provide error masking must be employed. Error masking techniques guarantee that the occurrence of an error will not result in the loss of the results of the processing the system was performing when the error occurred.

One way to mask the occurrence of errors is to employ system replicas, use one of them for handling input and the other replicas as backups in the case an error occurs on the original. This technique is also known in the fault tolerance literature with the name *primary-backup* (see [10]).

3.7.1 Context

The **Passive Replication** pattern applies to a system that has the following characteristics:

- The errors the system may experience are *detectable*.
- The errors the system may experience are not due to *errors in the input* it receives.
- The system is capable of *exporting* its current state and *importing* a new state.

3.7.2 Problem

In the above context, the **Passive Replication** pattern solves the problem of masking an error on the system by balancing the following forces:

- The input received by the system must be processed and the designated output must be delivered independently of whether an error occurs on the system.
- The time overhead for error-free system execution must be kept minimum.
- If an error occurs on the system, the environment is able to tolerate a delayed output from the system which is takes considerably longer than the usual delay occurring in error-free execution.

3.7.3 Solution

The solution to the above problem suggested by the **Passive Replication** pattern is based on the solution suggested by the **Rollback** pattern. Two system replicas are created, one that receives and processes the input meant for the fault tolerant system (called *primary*), and another replica (called *backup*) that remains inactive under error-free execution of the system. Every new input sent to the primary is register to a log facility. A manager decides when the primary must checkpoint to storage its state (e.g. before the registered input is forwarded to the primary). The log forwards to the primary the input and the latter processes it. If an error occurs on the primary while processing the input, the backup is activated, imports the last checkpoint, receives from the log facility the last registered input and starts processing it, replacing the failed primary.

Notice that the failure semantics to which the occurred errors may lead are not important for the solution suggested by the **Passive Replication** pattern. The only re-

quirement is that the error is detectable, which is one of the characteristics of the context in which this pattern applies.

The above solution can mask the occurrence of a single error. However, it cannot deal with the occurrence of two or more simultaneous errors (e.g. one error occurring on the primary and at the same time another error occurring on the backup). To be able to recover from *2N* simultaneous errors *2N+1* replicas are needed so at least one will remain unaffected by the occurred errors.

Another limitation of the original solution is that the fault tolerant system loses its fault tolerance capabilities after masking the first error occurrence. Even in the revised solution for dealing with *2N* simultaneous errors, the fault tolerant system eventually loses its fault tolerance capabilities (e.g. after the occurrence of *2N* simultaneous errors or after the occurrence of *2N* consecutive errors). To preserve the fault tolerance capabilities of the fault tolerant system, a mechanism that offers dynamic management of the replica group must be put in place. The dynamic replica group management will be responsible to replace with new ones the replicas discarded after the occurrence of errors.

In practice, since the backups are inactive prior to the occurrence of an error, the manager may choose not to create the backups at system start up but only when the error occurs. This variance can be practically useful especially in cases where the space constraints are very tight and the creation of a new copy of the system has relatively low time overhead.

If the errors that may occur on the fault tolerant system do not lead to fail-stop or crash failures where the failed system ceases execution, the backup of the fault tolerant system is not necessary. The primary can checkpoint at certain moments its state to the storage and when an error occurs, the failed primary which has not crashed can import the last checkpoint and continue its execution from there.

3.7.4 Structure

The solution suggested by the **`Passive Replication`** pattern consists of the following entities:

- The *primary*, which is a copy of the system that processes input and delivers output in the absence of errors. In addition, the primary exports its state (checkpoint) when the manager instructs so. It must be mapped to a different unit of failure than any of the following entities.
- The *backup*, which is identical to the primary, remains inactive in error-free execution and gets activated when an error on the primary is detected. Upon its activation, the backup is instructed by the manager to import the last checkpoint and then it receives from the log the last input in order to take over primary's role. The backup must be mapped to a different unit of failure than the primary.
- The *log*, which is the facility responsible for registering all input intended for the primary and for replaying the last input when requested by the manager. The log must be mapped to a different unit of failure than any of the primary and the backup.
- The *storage*, which is the facility responsible for storing the state that the primary exports and for providing the last exported state to the backup upon request from the manager. The storage must not be subject to errors, i.e. it must behave like *stable storage* that survives errors.

- The *manager*, which is responsible to activate the backup when an error occurs on the primary, request from the storage to provide the last state saved by the primary, and request from the log to replay the last registered input. The manager relies on an error detection mechanism to detect errors that may occur on the primary. The manager must be mapped to a different unit of failure than any of the primary and the backup.

Figure 7 gives a graphical illustration of the structure and the activity diagram of the **`Passive Replication`** pattern. In Figure 7a, block arrows indicate flow of information and open arrows indicate signals from the manager.

3.7.5 Consequences

The **`Passive Replication`** pattern has the following benefits:

+ The time overhead introduced by this pattern in error-free system execution is low and it amounts to the time needed to register the received input at the log facility and the time needed by the primary to save its state at the storage facility.
+ The space overhead in terms of system replicas introduced by this pattern for error masking is low (2*N* backups + 1 primary for masking 2*N* errors).
+ The communication overhead inside the fault tolerant system in the absence of errors is low and it includes the input registration to the log facility and the saving of primary's state to the storage facility.

The **`Passive Replication`** pattern imposes also some liabilities:

– The design complexity introduced by this pattern is higher that the design complexity of the pattern for error recovery (§3.5 and §3.6) and relatively elevated compared to the other patterns for error masking presented in the remainder.
– In the presence of errors, the time overhead introduced by this pattern is elevated because of the time required to activate the backup, load the last state the primary had saved and replay the last input. In the case where the dynamic replica group management creates the backup when the primary fails (see §3.7.3), the time overhead will be even higher.

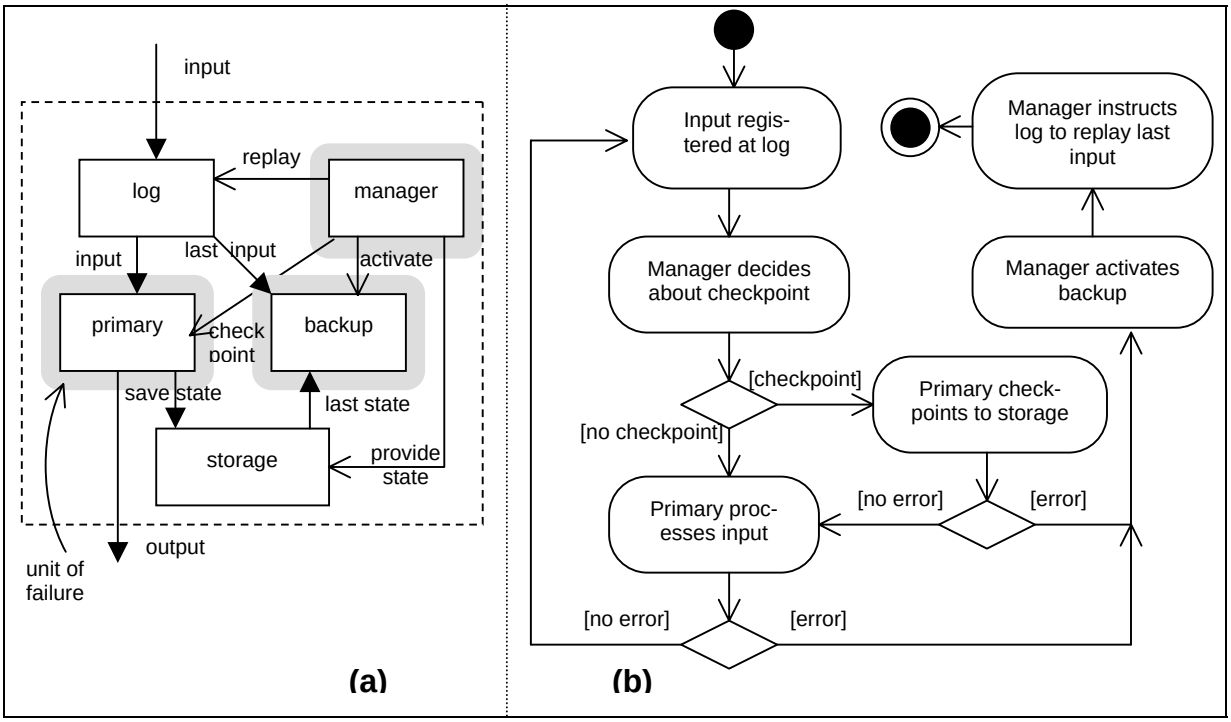

Figure 7. The structure (a) and the activity diagram (b) of the **`Passive Replication`** pattern.

- In addition to the space overhead in terms of system replicas (backups), this pattern introduces a space overhead related to the log and the storage entities.
- If the platform where the fault tolerant system will be deployed does not offer any persistent storage or stable storage facilities which could be used to accommodate the log and the storage entities of the **`Passive Replication`** pattern, then designing these two entities in a way that does not introduce single points of failure in the fault tolerant system introduces a significant design overhead.

3.7.6 Related patterns

The manager entity in the **`Passive Replication`** pattern monitors the primary for errors. The mechanism for the error detection can be based on one of the **`Acknowledgment`**, **`I Am Alive`** and **`Are You Alive`** patterns presented in sections 3.2, 3.3 and 3.4 respectively.

In certain cases where space and cost constraints prohibit the use of a full-fledged system replica as the backup entity, the **`Backup`** pattern (see [16]) can be employed as a lightweight alternative of the **`Passive Replication`** pattern. In the **`Backup`** pattern the backup can be a trimmed down version of the primary, providing only the essential functionality. Usually, when the **`Backup`** pattern is employed, after the occurrence of an error on the primary the fault tolerant system will operate in "emergency mode" using the backup until the primary is repaired.

3.8 Semi-Passive Replication

Another way to mask errors, without having to pay long time penalties when errors occur, is to follow a similar scheme like in the **`Passive Replication`** pattern, only having the primary entity exporting its state directly to the backup entity. This way, the backup is standby and when an error occurs on the primary, the backup needs only the last registered input in order to replace the failed primary. This technique is also known in the fault tolerance literature with the name *cold standby*.

3.8.1 Context

The **`Semi-Passive Replication`** pattern applies to a system that has the following characteristics:

- The errors the system may experience are *detectable*.
- The errors the system may experience are not due to *errors in the input* it receives.
- The system is capable of *exporting* its current state and *importing* a new state.

3.8.2 Problem

In the above context, the **`Semi-Passive Replication`** pattern solves the problem of masking an error on the system by balancing the following forces:

- The input received by the system must be processed and the designated output must be delivered independently of whether an error occurs on the system.
- The error-free execution of the system must suffer minimum time penalties.
- The time penalty introduced by the solution in the presence of errors must be kept low.

3.8.3 Solution

The solution to the above problem suggested by the **`Semi-Passive Replication`** pattern is similar to the solution suggested by the **`Passive Replication`** pattern with the difference that in this case there is no need for the storage entity. Two system replicas are created, one that receives and processes the input meant for the fault tolerant system (called *primary*), and another replica (called *backup*) that imports the state of the primary every time the latter attempts to checkpoint it. Similarly to the **`Passive Replication`** pattern, every new input sent to the primary is register to a log facility. Again, a manager decides when the primary must checkpoint its state (e.g. before the registered input is forwarded to the primary). The log forwards to the primary the input and the latter processes it. If an error occurs on the primary while processing the input, the backup is activated, receives from the log facility the last registered input and starts processing it, replacing the failed primary.

Notice that the failure semantics to which the occurred errors may lead are not important for the solution suggested by the **`Semi-Passive Replication`** pattern. The only requirement is that the error is detectable, which is one of the characteristics of the context in which this pattern applies.

The above solution can mask the occurrence of a single error. However, it cannot deal with the occurrence of two or more simultaneous errors (e.g. one error occurring on the primary and at the same time another error occurring on the backup). To be able to recover from *2N* simultaneous errors *2N+1* replicas are needed so at least one will remain unaffected by the occurred errors.

Another limitation of the original solution is that the fault tolerant system loses its fault tolerance capabilities after masking the first error occurrence. Even in the revised solution for dealing with *2N* simultaneous errors, the fault tolerant system eventually loses its fault tolerance capabilities (e.g. after the occurrence of *2N* simultaneous errors or after the occurrence of *2N* consecutive errors). To preserve the fault tolerance capabilities of the fault tolerant system, a mechanism that offers dynamic management of the replica group must be put in place. The dynamic replica group management will be responsible to replace with new ones the replicas discarded after the occurrence of errors.

If the errors that may occur on the fault tolerant system do not lead to fail-stop or crash failures where the failed system ceases execution, the backup of the fault tolerant system is not necessary. The primary can checkpoint at certain moments its state to the storage and when an error occurs, the failed primary which has not crashed can import the last checkpoint and continue its execution from there.

3.8.4 Structure

The solution suggested by the **`Semi-Passive Replication`** pattern consists of the following entities:

- The *primary*, which is a copy of the system that processes input and delivers output in the absence of errors. In addition, the primary exports its state (checkpoint) when the manager instructs so. It must be mapped to a different unit of failure than any of the following entities.
- The *backup*, which is identical to the primary, received primary's checkpoints in error-free execution and gets activated when an error on the primary is detected. Upon its activation, the backup receives from the log the last input in order to take over primary's role. The backup must be mapped to a different unit of failure than the primary.
- The *log*, which is the facility responsible for registering all input intended for the primary and for replaying the last input when requested by the manager. It must be mapped to a different unit of failure than the primary and the backup.
- The *manager*, which is responsible to activate the backup when an error occurs on the primary and request from the log to replay the last registered input. The manager relies on an error detection mechanism to detect errors that may occur on the primary. The manager must be mapped to a different unit of failure than any of the primary and the backup.

0 gives a graphical illustration of the structure and the activity diagram of the **`Semi-Passive Replication`** pattern. In 0a, block arrows indicate flow of information and open arrows indicate signals from the manager.

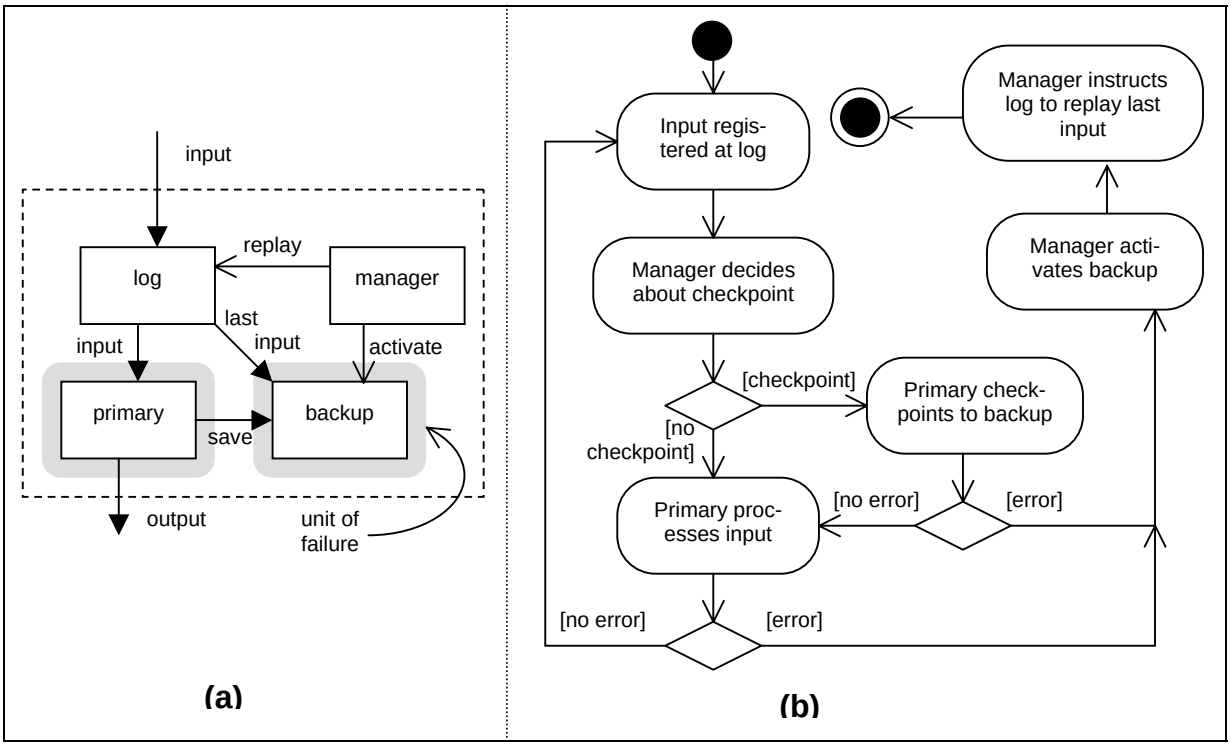

Figure 8. The structure (a) and the activity diagram (b) of the **`Semi-Passive Replication`** pattern.

3.8.5 Consequences

The **`Semi-Passive Replication`** pattern has the following benefits:

+ The design complexity of introduced by this pattern is lower than the one introduced by the **`Passive Replication`** pattern since the storage entity is removed (compare Figure 7a and 0a).
+ The time overhead introduced by this pattern in error-free system execution is low and it amounts to the time needed to register the received input at the log facility and the time needed by the primary to export its state to the backup.
+ The space overhead introduced by this pattern for error masking is low (2*N* backups + 1 primary for dealing with 2*N* errors).
+ The communication overhead inside the fault tolerant system in the absence of errors is low (as low as in the **`Passive Replication`** pattern) and it includes the input registration to the log facility and the checkpoint of primary's state to the backup.

The **`Semi-Passive Replication`** pattern imposes also some liabilities:

– In the case where 2*N* backups are created to deal with 2*N* simultaneous errors, the time and communication overheads in an error-free system execution is high, since it includes the communication from the primary to every one of the backups (for the communication overhead in a group of replicas see [8]). An efficient multicast mechanism can lower significantly this overhead.
– In the presence of errors the time overhead introduced by this pattern is relatively elevated (though, considerably lower than the time overhead introduced by the **`Passive Replication`** pattern in the same case) and it amounts to the time

needed by the log facility to replay the last registered input plus the time to process it and deliver the designated output.

- In addition to the space overhead in terms of system replicas (backups) this pattern introduces a space overhead related to the log entity. Still, this space overhead is lower than the overhead introduced by the **Passive Replication** pattern in the same case since it does not include the storage entity.
- If the platform where the fault tolerant system will be deployed does not offer any persistent storage or stable storage facilities which could be used to accommodate the log entity, then designing the log in a way that does not introduce a single point of failure in the fault tolerant system introduces a significant design overhead.

3.8.6 Related patterns

The manager entity in the **Semi-Passive Replication** pattern monitors the primary for errors. The mechanism for the error detection can be based on one of the **Acknowledgment**, **I Am Alive** and **Are You Alive** patterns presented in sections 3.2, 3.3 and 3.4 respectively.

In certain cases where space and cost constraints prohibit the use of a full-fledged system replica as the backup entity, the **Backup** pattern (see [16]) can be employed as a lightweight alternative of the **Semi-Passive Replication** pattern. In the **Backup** pattern the backup can be a trimmed down version of the primary, providing only the essential functionality. Usually, when the **Backup** pattern is employed, after the occurrence of an error on the primary the fault tolerant system will operate in "emergency mode" using the backup until the primary is repaired.

3.9 Semi-Active Replication

To avoid the complexity involved with exporting and importing the state of a system, the **Semi-Active Replication** pattern provides an alternative way for error masking. Instead of having only one replica actively processing input to the fault tolerant system and the rest of the replicas passively waiting for an error to occur on the primary, in the **Semi-Active Replication** pattern all the replicas are actively processing the input in parallel. If an error occurs on one of the replicas, the others will be able to promptly deliver the designated output. This technique is also known in the fault tolerance literature with the names *hot standby* and *coordinator/cohorts* (see [10]).

3.9.1 Context

The **Semi-Active Replication** pattern applies to a system that has the following characteristics:

- The errors the system may experience are *detectable*.
- The errors the system may experience are not due to *errors in the input* it receives.

3.9.2 Problem

In the above context, the **Semi-Active Replication** pattern solves the problem of masking an error on the system by balancing the following forces:

- The input received by the system must be processed and deliver the designated output independently of whether an error occurs on the system.
- The error-free execution of the system must suffer minimum time penalties.
- The time penalty introduced by the solution in the presence of errors must be kept very low.

3.9.3 Solution

The solution to error masking suggested by the **Semi-Active Replication** pattern is based on a group of two replicas (i.e. a group consisting of identical copies of the system) which process in parallel the input to the fault tolerant system. One of the replicas (called *coordinator*) has the leadership of the group and the other replica (the *cohort*) is monitoring the coordinator for errors. In an error-free execution, the coordinator is the one that delivers the output to the environment, In this case the cohort keeps the output it has produced until it is sure that the coordinator has delivered the output to the environment and then it discards it. If an error is detected on the coordinator then the cohort takes over the responsibility of the coordinator to deliver the output to the environment.

Notice that the failure semantics to which the occurred errors may lead are not important for the solution suggested by the **Semi-Active Replication** pattern. The only requirement is that the error is detectable, which is one of the characteristics of the context in which this pattern applies.

The above solution can mask the occurrence of a single error. However, it cannot deal with the occurrence of two or more simultaneous errors (e.g. one error occurring on the coordinator and at the same time another error occurring on the cohort). To be able to recover from *2N* simultaneous errors *2N+1* replicas are needed so at least one will remain unaffected by the occurred errors.

Another limitation of the original solution is that the fault tolerant system loses its fault tolerance capabilities after masking the first error occurrence. Even in the revised solution for dealing with *2N* simultaneous errors, the fault tolerant system eventually loses its fault tolerance capabilities (e.g. after the occurrence of *2N* simultaneous errors or after the occurrence of *2N* consecutive errors). To preserve the fault tolerance capabilities of the fault tolerant system, a mechanism that offers dynamic management of the replica group must be put in place. The dynamic replica group management will be responsible to replace with new ones the replicas discarded after the occurrence of errors.

In the version of the **Semi-Active Replication** pattern where more that one cohorts exist, there is a need for a protocol which will be used to decide which of the cohorts will take over the coordinator responsibilities once an error is detected on the coordinator. This protocol is often implemented as a replica group management mechanism (e.g. see [8]) based on a variety of schemes like predefined order in the group of replicas, group member voting, etc.

3.9.4 Structure

The solution suggested by the **Semi-Active Replication** pattern consists of the following entities:

- The *coordinator*, which is a copy of the system that processes input and delivers output to the environment of the fault tolerant system. When the coordinator delivers the output to the environment, it notifies the other system replicas (see *cohorts* below) as a synchronization action that enables the latter to start the processing of the following input. The coordinator must be mapped to a different unit of failure than any of the following entities.
- The *cohort*, which is a system replica like the coordinator that receives exactly the same input (same content, same order of delivery) as the coordinator. Besides processing the input, the cohort monitors the coordinator for errors. If the cohort detects an error on the coordinator, then it takes the responsibility to deliver the designated output that should be delivered by the coordinator. In an error-free execution, the outcome of processing the input by the cohort does not produce any output to the environment. The cohort must be mapped to a different unit of failure than the coordinator.
- The *distributor*, which is responsible to ensure that all system replicas (i.e. the coordinator and the cohorts) receive the same input (same content, same order of delivery). It must be mapped to a different unit of failure than the coordinator and the cohorts.

0 gives a graphical illustration of the structure and the activity diagram of the **Semi-Active Replication** pattern. In 0a block arrows indicate flow of information and open arrows indicate signals exchanged among the system replicas.

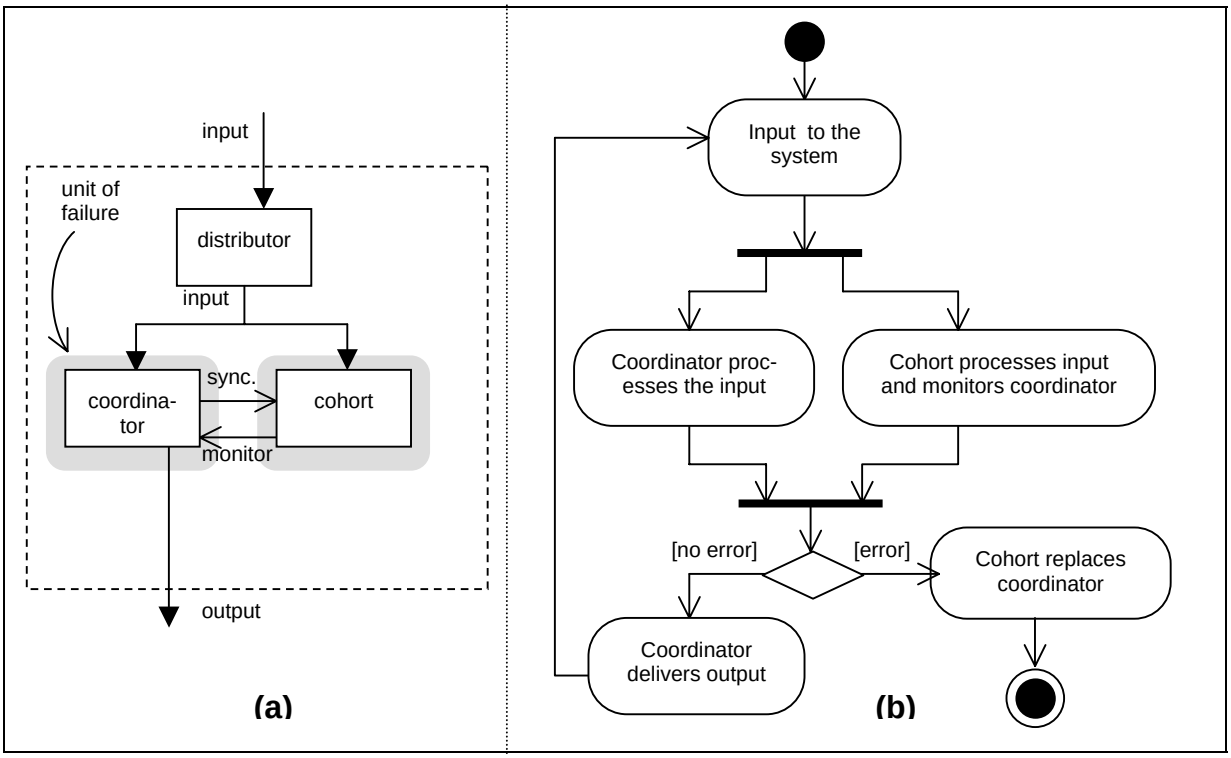

Figure 9. The structure (a) and the activity diagram (b) of the **`Semi-Active Replication`** pattern.

3.9.5 Consequences

The **`Semi-Active Replication`** pattern has the following benefits:

+ The time overhead introduced by this pattern in error-free system execution is very low since the coordinator provides directly the output to the environment and the only penalty is the indirection of the input through the distributor. An efficient atomic multicast mechanism can remove this overhead.
+ In the presence of errors, the time overhead introduced by this pattern is also low and it amounts to the indirection of the input through the distributor and the time needed by the cohorts to detect the error on the coordinator, decide upon the new coordinator and deliver the already produced output to the environment.
+ The space overhead introduced by this pattern for error masking is low (2*N* cohorts + 1 coordinator for dealing with 2*N* errors).
+ The design complexity of the error masking mechanism suggested by the **`Semi-Active Replication`** pattern is lower than the corresponding complexity of the previous two patterns for error masking.

The **`Semi-Active Replication`** pattern imposes also some liabilities:

– This pattern has a high design complexity associated to the management of the group of system replicas including the synchronization, error detection and self-organization (i.e. the election of a new coordinator) capabilities of the group.
– The communication and synchronization overhead of the replica group is elevated. The same relative order of input delivery to the replicas has the overhead of an atomic broadcast protocol. In addition to this the synchronization of the group upon output delivery by the coordinator makes the functioning of the group of replicas significantly communication intensive.

- The stepwise synchronization of the group members with respect to the input they receive, i.e. the coordinator cannot start processing new input before the cohorts have finished their processing and vise versa, forces the group of replicas to perform as slowly as the slowest of its members. This can be a problem if there are significant differences in the performance of the different group members.
- In the case of dynamic management of the replica group (i.e. when a failed member is replaced by a new cohort) in order to sustain the group's fault tolerance capabilities it is necessary for old group members to be able to export their current state and for the new cohort to import that state and initialize itself with it.

3.9.6 Related patterns

The cohorts in the **Semi-Active Replication** pattern monitor the coordinator for errors. The mechanism for the error detection can be based on one of the **Acknowledgment**, **I Am Alive** and **Are You Alive** patterns presented in sections 3.2, 3.3 and 3.4 respectively. Also, the dynamic management and the synchronization of the group of system replicas can be based on the **Object Group** pattern (see [8]).

3.10 Active Replication

The most powerful technique for masking errors is the one suggested by the **Active Replication** pattern. Similarly to the previous pattern, the **Active Replication** pattern uses a group of system replicas where all members of the group actively receive and process every input received by the monitored system. However, opposite to the **Semi-Active Replication** pattern, in this pattern all replicas deliver their output without having to monitor each other or temporarily keeping their output until a selected group member delivers the output to the environment. The error masking technique captured by the **Active Replication** pattern is also known in the fault tolerance literature with the names *server group* and *state machine approach* (e.g. see [15]).

3.10.1 Context

The **Active Replication** pattern applies to a system that has the following characteristics:

- The system is *deterministic*, i.e. its output is solely defined by its initial state, the sequence of inputs it has processed so far and the current time (in terms of clock time and/or time elapsed since the system initialization).
- The errors the system may experience are not due to *errors in the input* it receives.
- The errors the system may experience cause it to exhibit *byzantine* failures.

3.10.2 Problem

How to mask the occurrence of an error by balancing the following forces:

- The input received by the monitored system must be processed and deliver the designated output independently of whether an error occurs on the monitored system.
- The error-free execution of the system must suffer minimum time penalties.
- The time penalty introduced by the solution in the presence of errors must be kept very low.
- The monitored system is deterministic.

3.10.3 Solution

The **Active Replication** pattern is the extension of the **Fail-Stop Processor** pattern (see §3.1) from error detection to error masking. The solution suggested by this pattern is based again on a group of system replicas. Similarly to the **Semi-Active Replication** pattern, all replicas receive the same input (same content, same delivery order) but in this case all the group members deliver their output to the comparator. The latter is an extension of the comparator entity from the **Fail-Stop Processor** pattern, which is responsible to decide (e.g. by majority voting) what is the correct output. In an error-free execution, given that the system replicas are deterministic, the comparator will receive identical outputs from all replicas and it will forward one of these to the environment of the fault tolerant system. If an error occurs on one of the replicas, the output that replica will produce will be different from the outputs produced by the replicas that did not experience an error at the same time. Using three system replicas, the comparator is capable to detect which is the errone-

ous output. Subsequently, the comparator forwards to the environment the correct output and discards the failed replica.

The above solution can mask the occurrence of a single error. However, it cannot deal with the occurrence of two or more simultaneous errors (e.g. two errors occurring one at the same time, each on a different replica). When the comparator uses majority voting to decide on the correct output, in order to be able to recover from *N* simultaneous errors *2N+1* replicas are needed so that the majority of the replicas will be unaffected by the occurred errors and the majority voting will correctly identify the error-free output.

Another limitation of the original solution is that the fault tolerant system loses its fault tolerance capabilities after masking the first error occurrence. Even in the revised solution for dealing with *N* simultaneous errors, the fault tolerant system eventually loses its fault tolerance capabilities (e.g. after the occurrence of *N* simultaneous errors or after the occurrence of *N* consecutive errors). To preserve the fault tolerance capabilities of the fault tolerant system, a mechanism that offers dynamic management of the replica group must be put in place. The dynamic replica group management will be responsible to replace with new ones the replicas discarded after the occurrence of errors.

Like in the case of the **`Fail-Stop Processor`** pattern, the replicas must be deterministic systems (i.e. their output is solely defined by their initial state, the sequence of inputs they have processed so far and the current time) and they must all have been initialized in the same state and they have synchronized clocks.

3.10.4 Structure

The entities introduced by the **`Active Replication`** pattern are:

- The *processors*, which are replicas of the deterministic systems that must be rendered fault tolerant and which may experience byzantine failures. To deal with *N* simultaneous errors *2N+1* processors are required, each of them mapped to a different unit of failure.
- The *distributor*, which ensures that all the processors, receive exactly the same input (in terms of content and delivery order). It must be mapped to a different unit of failure than any of the processors in order not to get affected by the errors that may occur on them.
- The *comparator*, which receives the outputs of the processors and decides (e.g. by majority voting) what will be the output of the fault tolerant system. Once the output is decided, it is delivered to the environment and the processors, which produced a different output (including those that did not produce any output at all), are discarded as failed. The comparator must be mapped to a different unit of failure that any of the replicas.

Figure 10 gives a graphical illustration of the structure and the activity diagram of the **`Active Replication`** pattern.

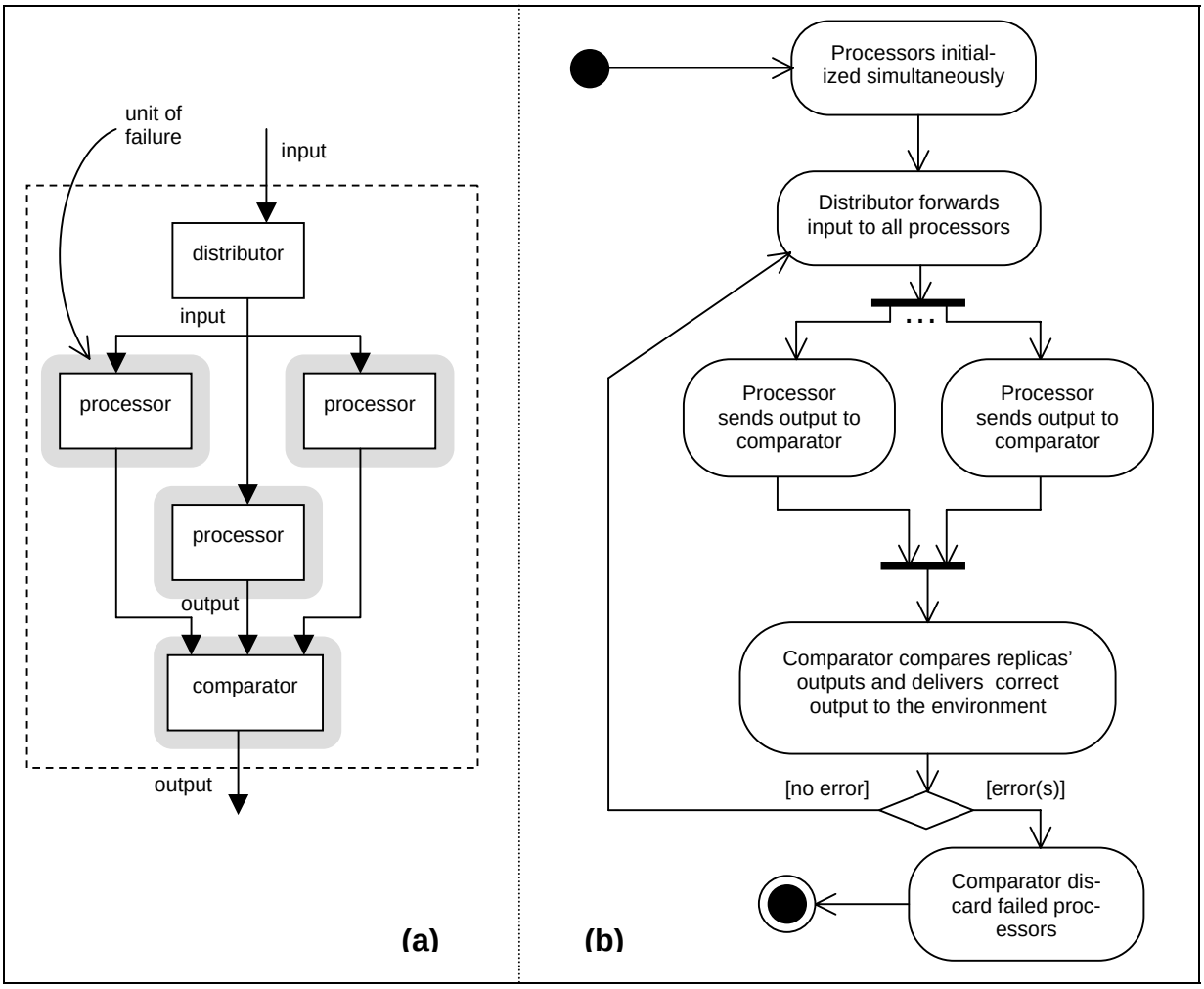

Figure 10. The structure (a) and the activity diagram (b) of the **`Active Replication`** pattern.

3.10.5 Consequences

The **`Active Replication`** pattern has the following benefits:

+ The time overhead introduced by this pattern in error-free system execution is low. The only time penalties introduced by the **`Active Replication`** pattern solution is due to the indirection of the input through the distributor and the time needed by the comparator to decide what is the correct output to be sent to the environment of the fault tolerant system. An efficient atomic multicast mechanism can remove the first time overhead.
+ In the presence of errors, the time overhead introduced by this pattern is also low (lower than the corresponding overhead introduced by the **`Semi-Active Replication`** pattern) and it amounts to the indirection of the input through the distributor and the time needed by the comparator to decide on the output that will be delivered to the environment.
+ The design complexity introduced by this pattern is relatively low (lower than the **`Semi-Active Replication`** pattern and comparable to the **`Fail-Stop Processor`** pattern), since the distributor and comparator, which are entities specific to this pattern, have quite simple functionality.

+ This pattern, opposite to the **Semi-Active Replication** pattern, does not introduce any synchronization overhead stemming from the communication inside the group of replicas.
+ This pattern is one of the very few alternatives for dealing with byzantine failures.

The **Active Replication** pattern imposes also some liabilities:

– The space overhead introduced by this pattern is high, the highest space overhead introduced by the four patterns for error masking presented in this paper: it takes *2N+1* replicas to mask *N* errors. On the other hand, the masked errors may lead to byzantine failures, which is something that the previous three patterns for error masking did not address.
– Although the processors may have byzantine failure semantics, the distributor and the comparator must not experience byzantine failures. The **Fail-Stop Processor** pattern (see §3.1) can be applied to each of these entities in the case where they may experience failures of byzantine type.
– The distributor and the comparator introduce single points of failure in the system. The distributor can be replaced by an atomic broadcast protocol if such is available. Both the distributor and the comparator can be rendered fault tolerant by applying some of the fault tolerance patterns already presented in this paper. In any case, the resulting design complexity of the **Active Replication** pattern is elevated compared to the one graphically presented in Figure 10.

3.10.6 Related patterns

The **Active Replication** pattern is the evolution of the **Fail-Stop Processor** pattern (see section 3.1) for masking errors that lead to byzantine failures. The dynamic management and the synchronization of the group of system replicas can be based on the **Object Group** pattern (see [8]).

4 CLASSIFICATION SCHEME

The fault tolerance patterns presented in the previous section provide solutions to problems of smaller to bigger scale that repeatedly appear in the design of fault tolerant systems (e.g. how to detect, how to recover from, and how to mask an error). To increase the added value of the presented patterns they can be organized in the form of a pattern system. Such a system of fault tolerance patterns would provide substantial help to the system designers and architects in their effort to develop fault tolerant systems. The help that the pattern system would provide is to reveal the relations among the presented patterns and to provide instructions about the combinations of these patterns that produce complete solutions to design problems in the development of fault tolerant systems.

The remainder of this section presents the properties (and hence the added value) of a pattern system, the classification scheme for the fault tolerance patterns and how it is used to organize the presented patterns and transform them to a system of fault tolerance patterns. It also presents the way that the resulting pattern system can be used to create design frameworks for the development of fault tolerant systems, and how it can be combined with other well-known pattern systems (e.g. [3] and [4]).

4.1 Pattern Systems

A pattern system is a collection of patterns together with guidelines for their implementation, combination and practical use in the development of software systems. Pattern systems do not cover completely the solutions that may exists to design problems in the particular domain in which they apply; that would be the definition of a pattern language. Rather, pattern systems describe only certain aspects of the construction of software systems leaving to the designer's skills the completion of the software architecture of the system under construction. In practice pattern systems are easier to build than a pattern language (because they don't have to completely cover their domain of application) and they are more flexible in being applied in different domains of software system design [3].

A pattern system has the following properties:

- It provides a sufficient base of patterns for addressing/resolving design problems in the domain where it is applied.
- It describes all its constituent patterns uniformly.
- It exposes the various relationships among its constituent patterns.
- It provides an organization schema for its constituent patterns.
- It supports the development of software systems with a set of instructions about how to implement and combine its constituent patterns.
- It supports its own evolution, which allows the integration of new patterns and consequently the adaptation of the pattern system to new technology trends.

In the remainder of this section we describe a classification scheme and how it is applied in the organization of the set of fault tolerance pattern presented in section 3 in a way that all the above properties are satisfied.

4.2 Organizing Fault Tolerance Patterns

A straightforward classification scheme for the patterns presented in section 3 is the one stemming from the aspects of fault tolerance (i.e. error detection, recovery, and masking) to which each pattern provides a solution. The two main advantages of this

classification scheme are that it naturally reflects the essential qualities and the application sub-domains of the patterns, and that it creates a set of classification categories with direct dependency relations among them. These dependency relations are founded on the fact that most error masking mechanisms are based on error recovery mechanisms, which are in turn based on error detection mechanisms. This implies that a pattern in the error masking category can be refined by a pattern in the error recovery category in order to elaborate the aspects of the solution regarding the removal of error effects. Subsequently, the error recovery pattern can be refined by some pattern in the error detection category to elaborate the aspects of the solution regarding the detection of an error.

In this classification scheme, the first four patterns (**Fail-Stop Processor, Acknowledgment, I Am Alive,** and **Are You Alive**) belong to the (error) *Detection* category. The next two patterns (**Roll Forward**, and **Rollback**) belong to the (error) *Recovery* category. Finally, the last four patterns (**Passive**, **Semi-Passive**, **Semi-Active**, and **Active Replication**) belong to the (error) *Masking* category. Most of the patterns in the latter two categories rely on an error detection mechanism, which yields a dependency between these patterns and the Detection category. Another dependency relation exists between the **Active Replication** and the **Fail-Stop Processor** patterns where the former is the evolution and the adaptation of the latter in masking errors.

The three aforementioned categories of fault tolerance patterns are not specific to the patterns presented in this document. In fact, existing fault tolerance patterns can be classified under these categories (e.g. the **Backup** pattern [16] can be classified under the Masking category). Moreover, these three categories do not cover the entire domain of fault tolerance. Other categories can be added to this classification scheme in order provide a complete set of categories for fault tolerance patterns. For example, the (error) *Assessment* category can be added to our classification scheme in order to classify patterns like the **Leaky Bucket Counter** and the **Riding Over Transients** patterns (both from [1]) that are used to assess the nature of occurred errors. Figure 11 illustrates graphically the classification scheme and the relations among the different categories as well as the possible extensions of the classification categories and their contents (depicted in shaded shapes). More regarding the extensibility of the classification scheme can be found in subsection 4.4.

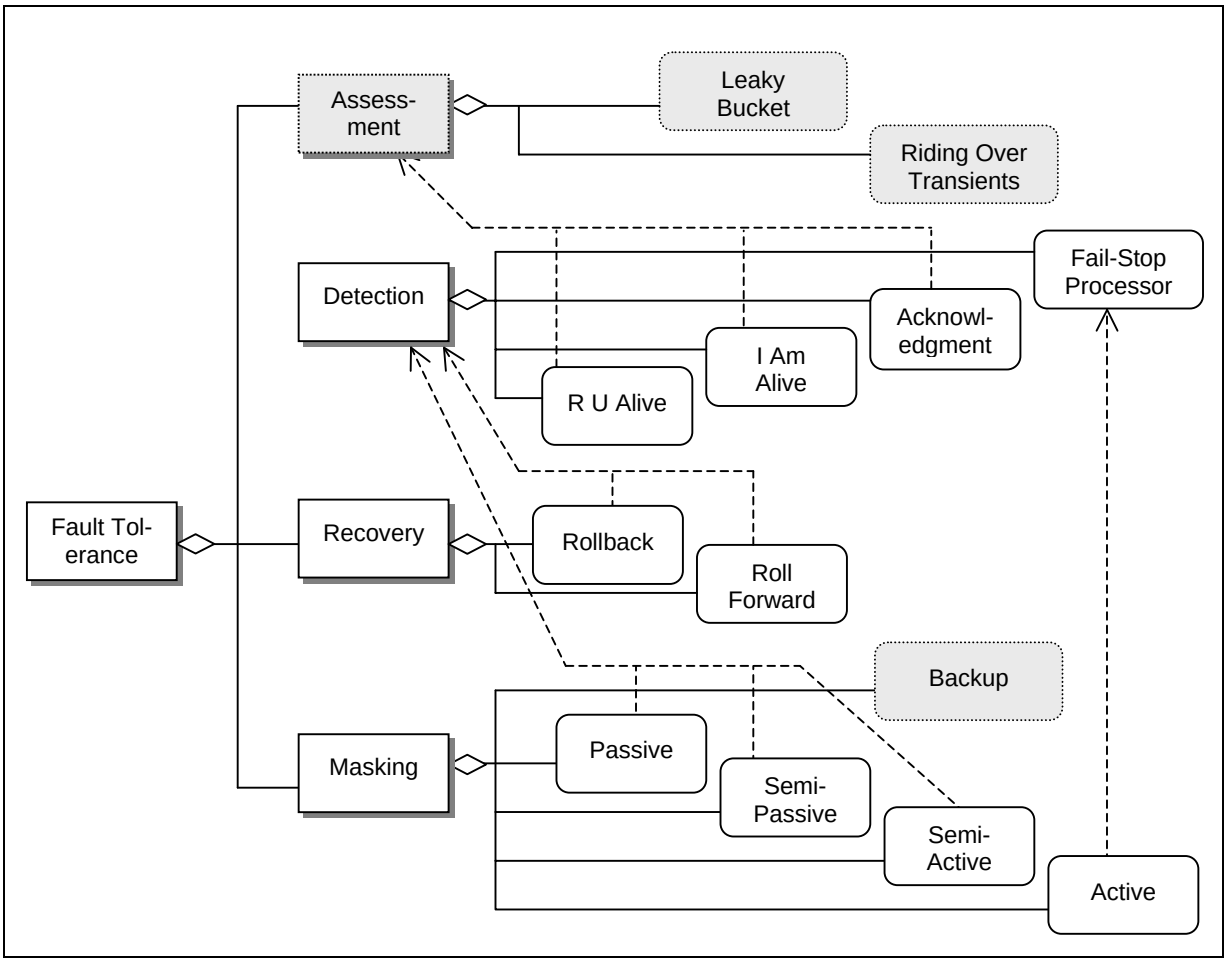

Figure 11. Classification of the presented patterns according to the fault tolerance aspect criterion.

Besides the straightforward classification of the presented patterns based on the fault tolerance aspects to which they are associated, a number of complementary classification schemes can be applied to the patterns presented in section 3. The purpose a classification scheme serves is to help the system designers to find the appropriate pattern for a specific design problem. The classification scheme presented above has its own advantages but it does not give any information about a number of properties associated with the fault tolerance pattern (e.g. complexity, time and/or space overhead, failure types confronted, etc) which can be key elements in the selection of the appropriate pattern for a given fault tolerance problem. Hence, we introduce five additional classification schemes based on the following criteria:

1. the design complexity of a pattern,
2. the space overhead introduced by a pattern in terms of additional entities that need to be created for the mechanism dictated by the fault tolerance pattern,
3. the time overhead introduced by a pattern in the absence of errors,
4. the time overhead introduced by a pattern in the presence of errors, and
5. the failure types to which a pattern applies.

The first four classification schemes regarding complexity, space overhead and time overhead with and without errors, have three categories each: one for low impact, one for medium impact and one for high impact. The classification of the fault toler-

ance patterns to these categories follows the benefits and the liabilities of each pattern as they are described in their consequences. One important note is that for the error detection patterns the distinction between the time overhead in the absence and in the presence of errors is not relevant. However, for reasons of uniform presentation as the rest of the fault tolerance patterns we classify them under these two categories, considering as time overhead the time spent by the system exclusively on error detection activities. For the error recovery patterns the time overhead represents the impact on the time that takes for the system to become operational and accept another invocation. Finally, for the error masking patterns, the time overhead represents the impact on the time it takes to reply to an accepted invocation.

The last classification scheme regarding the failure types supported by patterns has two categories: the crash failure (or failure with stronger semantics like fail-stop failure) and the byzantine failures. Among the presented patterns there are only two, the **`Fail-Stop Processor`** and the **`Active Replication`** patterns, which deal with byzantine failure. The **`Semi-Active Replication`** pattern can also deal with byzantine failures with certain enhancements (see section 3.9), hence it can be placed under the Byzantine category with a note regarding these enhancements. All other seven fault tolerance patterns are placed under the Crash category. Table 1 provides a summary of all six dimensions of the composite classification scheme proposed for the set of fault tolerance patterns presented in section 3.

Pattern	**Complexity**	**Space**	**Time** (no failures)	**Time** (failures)	**Failure Type**	**Fault Tolerance Aspect**
Fail-Stop Processor	M	H	L	L	byzantine	Error Detection
Acknowledgment	L	L	L	L	crash	
I Am Alive	M	L	M	M	crash	
Are You Alive	M	L	M	M	crash	
Roll Forward	M	M	M	L	crash	Error Recovery
Rollback	M	M	L	M	crash	
Passive Replication	M	M	L	H	crash	Error Masking
Semi-Passive Replication	M	H	L	H	crash	
Semi-Active Replication	H	H	L	M	byzantine*	
Active Replication	H	H	L	L	byzantine	

Table 1. The six dimensions of the composite classification scheme for the fault tolerance patterns.

The organization of the fault tolerance patterns outlined by the composite classification scheme depicted in Table 1 and partially in Figure 11, forms the basis for satisfying the first four properties of a pattern system mentioned in subsection 4.1 (i.e. a sufficient base of patterns for the fault tolerance domain which are described uniformly and are organized by a classification scheme which reveals their relationships). The way our system of fault tolerance patterns satisfies the last two properties (i.e. the set of instruction for the combination of the fault tolerance patterns and the support for the evolution of the pattern system) is described in the following subsections.

4.3 Design Frameworks for Fault Tolerant Systems

In the context of software design, a design framework is a partially complete software architecture that defines the structure and the properties for a group of inter-related subsystems, and also the places where adaptations for specific functionality should be made [3]. A design framework can be constructed by applying a pattern (or a set of patterns) in order to create the partially complete design of a software system. Moreover, frameworks include the design decisions about how to combine patterns and how to refine patterns using other patterns. Thus, frameworks alleviate the designer from the process of identifying which combinations of patterns lead to a partially complete solution for a given problem. But the designer has still to decide whether a given framework (i.e. partially complete design of a system) provides a solution that adheres to the system requirements. An example of a framework is the case of the Model-Viewer-Controller (MVC) framework presented in [6]. The **`Model-Viewer-Controller`** architectural pattern [3] is refined by the **`Observer`**, the **`Composite`**, and the **`Strategy`** patterns [4] that elaborate respectively the model-viewer relationship, the relationship between a viewer and its sub-viewers, and the viewer-controller relationship.

The classification scheme illustrated in Figure 11 provides the basis for various design frameworks for the development of fault tolerant systems. The relations among the classification categories and among the patterns provide instructions for combining the fault tolerance patterns in order to produce design frameworks similar to the MVC framework. For example, a design framework for the development of fault tolerant systems that deals with byzantine failures can be created by combining the **`Passive Replication`**, the **`Rollback`** and the **`Fail-Stop Processor`** patterns. The **`Passive Replication`** pattern provides the solution for masking an error by keeping a checkpoint state of the system monitored for errors and an inactive copy of it. The **`Rollback`** pattern is then used to refine the mechanism of activating the copy when an error occurs. Finally, the **`Fail-Stop Processor`** pattern is used to refine the way an error is detected and also to transform the occurrence of a byzantine failure to a fail-stop failure for the fault tolerant mechanism outlined by the combination of the **`Passive Replication`** and the **`Rollback`** patterns. Similar combinations of pattern as indicated by the dependency relations in Figure 11 can produce a variety of small design frameworks for the development of fault tolerant systems with different requirements.

It is important to notice that an individual fault tolerance pattern does not offer a complete solution for fault tolerance in the design of a system; rather it offers a solution to a specific problem (e.g. detection, recovery, or masking of an error) which is part of the complete fault tolerance solution. The different design frameworks that can be created as described above offer complete fault tolerance solutions. Still, the completeness of the fault tolerance solution refers to design problems. In order to provide support for the complete development (from the conception until the coding and the deployment) of a fault tolerant system, the presented system of fault tolerance pattern must be combined with some existing development framework for the construction of fault tolerant systems. Such development frameworks can be environments specifically aimed at the development of fault tolerant systems like fault tolerance support based on the State Machine Approach [15], the ISIS toolkit [2] and the development support for the Object Group pattern [8], or implementations of CORBA 2.6 which provide fault tolerance primitives (fault tolerant CORBA [11]).

4.4 Evolution of the System of Fault Tolerance Patterns

To complete the description of the system of fault tolerance patterns, it remains to show how it satisfies the last property of a pattern system given in subsection 4.1, i.e. the support of the system's own evolution which allows the integration of new patterns. The composite classification scheme summarized in Table 1 can be extended in various ways in order to allow the evolution of the system of fault tolerance patterns. One way to extend the classification is to add new categories reflecting other fault tolerance aspects besides error detection, recovery and masking. An example of how new classification categories can be added to the presented classification scheme is given in subsection 4.2 where the category (error) Assessment was added (see shaded shapes in Figure 11). Other classification categories could also be added which would allow the insertion of patterns that describe solutions for different types of broadcast and order of invocation requests delivery needed in replication-based error masking (e.g. atomic broadcast [9], virtual synchrony [2], etc). Such extensions will result to a number of new relations among the fault tolerance patterns, hence it will increase the number of design frameworks that can be created from the pattern system which will, in turn, result in design support for a wider range of fault tolerance problems, and in more elaborated description of the design solutions.

Another way to extend the composite classification scheme is by adding new (sub)classifications regarding other properties that can be used to qualify the fault tolerance patterns besides complexity, space and time overheads, and failure types. Such new (sub)classifications may include a quantification of the ease of dynamic reconfiguration for the mechanisms described by each fault tolerance pattern, the number of simultaneous errors that each pattern can deal with, etc. These extensions will increase the selection criteria that a software designer may use for selecting the appropriate pattern for a given fault tolerance problem. Hence, the support for design decisions will increase along with the usability of the system of fault tolerance patterns.

Finally, an important way of extending the system of fault tolerance patterns is by investigating the relations of the fault tolerance patterns with patterns in other well-known pattern systems like those presented in [3] and [4]. In some cases, especially with patterns that have an inherited distributed nature (e.g. Master Slave [3], Facade [4], etc), the relation with some fault tolerance patterns (especially those in the error masking category) can be straightforward. In other cases the relations between fault tolerance patterns and patterns from other systems will be more vague or even non-existent. However, independently of how trivial or difficult is the task of merging the system of fault tolerance patterns with other pattern systems the effort is worthwhile. The benefit of such a merge of fault tolerance patterns with other pattern systems would be a basis of partial design solutions that cover both the fault tolerance and the functional aspects of the development of software systems. But most importantly, the benefit would be the set of instructions about how to combine these partial solutions in order to obtain a complete solution for the design of a system that, among other qualities that it will possess, will also be fault tolerant.

5 CONCLUSION

In this document we presented a set of patterns that provide solutions to problems specific to different aspects of fault tolerance including error detection, recovery and masking. Our contribution is not in the conception of these solutions, which actually are well-known solutions that have been applied on fault tolerant computer systems for the past three decades. Rather, our contribution is the formatting of these solutions as patterns for system design and their organization in a pattern system that reveals the dependencies among them. These dependencies outline refinement relations that can be used to support the creation of design frameworks for the development of fault tolerant systems. We have shown that the organization of the fault tolerance patterns by means of a six-dimension classification scheme satisfies the properties of a pattern system, and most importantly those of instructions about how to combine the fault tolerance patterns and support for the evolution of the pattern system. The presented pattern system is by no means complete, neither in terms of the set of patterns it contains nor in terms of the classifications it proposes for the included patterns. It forms however a sufficient basis for a variety of design solutions in the development of fault tolerant systems.

The importance of the presented pattern system in the design of fault tolerant systems is comparable to the importance of other pattern systems in the domain of object-oriented design. Software designers and architects are provided with a set of solutions for fault tolerance specific problems and a set of practical instructions about how to combine these solutions to achieve a complete design of the fault tolerance aspects of a software system. The practical benefits of this system of fault tolerance patterns are twofold. On one hand the designers have a systematic way to investigate the fault tolerance solution that best fits a system under development. On the other hand, experience has shown that the same fault tolerance solutions presented as patterns are assimilated by designers and architects more easily (e.g. see [12]). In particular, personal experience with the presented set of fault tolerance patterns has shown that industrial designers and architects are much more comfortable with patterns than with descriptions of the same solutions when presented in some formal context like the one given in [13].

The full potential of the system of fault tolerance patterns in the support for the design of software systems can be exploited in the merge with a pattern system that supports the development of the functional aspects of software systems (e.g. [3] and [4]). The added value of such a merge is the support it provides for the development of integrated, multi-view software architectures that cover both the functional and the fault tolerance aspects of a software system [5]. The benefits can increase even more when pattern systems providing design solutions for other non-functional aspects of software systems (e.g. security, see [17]) are merged too. However, not all domains of non-functional properties (e.g. configurability, timeliness, usability, etc) are mature enough to produce pattern systems and even in the case of mature domains such as timeliness, related pattern systems are still to appear.

6 REFERENCES

[1] M. Adams, J. Coplien, R. Gamoke, R. Hanmer, F. Keeve, and K. Nicodemus. Fault-Tolerant Telecommunication System Patterns. In J. M. Vlissides, J. O. Coplien, and N. L. Kerth, editors, *Pattern Languages of Program Design - 2*. Addison Wesley, 1996.

[2] K. P. Birman and R. van Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, 1994.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: a System of Patterns, Volume 1*. John Wiley & Sons, July 2001.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, October 1994.

[5] V. Issarny, T. Saridakis, and A. Zarras. Multi-view Description of Software Architectures. In the *Proceedings of the 3rd International Software Architecture Workshop*, pages 81-84, November 1998.

[6] R. Johnson. Patterns and Frameworks. In L. Rising, editor, *The Patterns Handbook: Techniques, Strategies, and Applications*. SIGS Reference Library, Cambridge University Press, 1998.

[7] J. C. Laprie. *Dependability: Basic Concepts and Terminology*. Dependable Computing and Fault-Tolerant Systems, Volume 5, Springer-Verlag, 1992.

[8] S. Maffeis. The Object Group Design Pattern. In the *Proceedings of the USENIX Conference on Object-Oriented Technologies*, pages 294-303, June 1996.

[9] S. Mishra and R. D. Schlichting. Abstractions for Constructing Dependable Distributed Systems. Technical Report TR 92-19, the University of Arizona, August 1992.

[10] S. Mullender (editor). *Distributed Systems, 2nd edition*. ACM Press, 1993.

[11] OMG Document. Fault Tolerant CORBA. Chapter 25 in *the Common Object Request Broker: Architecture and Specification, Revision 2.6*, OMG, December 2001.

[12] R. L. Ramirez. A Design Patterns Experience Report In L. Rising, editor, *The Patterns Handbook: Techniques, Strategies, and Applications*. SIGS Reference Library, Cambridge University Press, 1998.

[13] T. Saridakis and V. Issarny. Developing Dependable Systems Using Software Architecture. In the *Proceedings of the 1st Working IFIP Conference on Software Architecture*, pages 83-104, February 1999.

[14] R. D. Schlichting and F. B. Schneider. Fail-Stop Processors: an Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computing Systems*, 1(3):222-238, August 1983.

[15] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach. ACM Computing Surveys, 22(4):299-319, December 1990.

[16] S. Subramanian and W.-T. Tsai. Backup Pattern: Designing Redundancy in Object-Oriented Software. In J. M. Vlissides, J. O. Coplien, and N. L. Kerth, editors, *Pattern Languages of Program Design - 2*. Addison Wesley, 1996.

[17] J. Yoder and J. Barcalow. Architectural Patterns for Enabling Application Security. In N. Harrison, B. Foote and H. Rohnert, editors, *Pattern Languages of Program Design - 4*. Addison Wesley, 1999.