4
Recovery Techniques

In this chapter we will discuss some ideas and techniques used to recover from failures detected during runtime.

After a failure is detected, it is necessary to take actions in order to:

1. Minimize the overall damage to the system, including damage to data;
2. Minimize loss of work for the users;
3. Minimize the time to resume work;

This means that the failure must be properly handled – a system cannot simply continue to freely execute after detecting a failure, because using damaged components or trusting corrupted data may increase overall damage. A system executing in a corrupted state without any control process to avoid the corrupted data or components cannot be relied on. Anything done after this point becomes suspect.

Minimizing the overall damage to the system, including damage to data can be considered the most important issue to be handled. Working with damaged data can have disastrous consequences depending on the system nature. Properly identifying the failures and the extension of the damages is crucial for recovery oriented software.

The simplest solution to handle a failure is to inform the user that one has been detected, terminate the software and do not allow it to be restarted until the release of a new version with the fault(s) fixed. This complies with minimizing the overall damage to the system, including damage to data, as no further damage can be caused by an already terminated system. But it might mean unacceptable "loss of work" and "time to resume work" for the users.

Another possible solution is to allow restarting of the software, relying on the fact that the failure was caused by a specific state that may not be achieved

again. However, this cannot be done if the failure is due to corrupted data, as working with data in an invalid state may significantly increase damage. A solution would be to allow working only with a new set of data, or with the part of the data that is still reliable (if it is possible to detect which part is still reliable). These solutions also comply with “minimize the time to resume work”, but may mean unacceptable “loss of work”.

Recovery means reestablishing a valid state (of execution, if the software is running, or for the data, if it has been damaged). The issue is that in most cases, useful recovery demands prior preparation of a system. Efforts in architecture, design and coding must be spent for software to have enough redundancy that allows for a proper state restoration.

4.1. Redundancy

Redundancy is a key issue to execute sophisticated recovery processes. It also allows for failure detection. Redundancy may be present in data structures, databases, extra code and hardware. The classical redundancy used in fault tolerant systems consists in replicating software or hardware components. In this work we propose another kind of redundancy, which is not based on replication: data redundancy and code redundancy.

Data redundancy is represented by extra information inserted in data (like in data structures, data sent/received through networks and databases) that is not directly related to fulfilling any of the functional or non-functional requirements of a system. Code redundancy is represented by instruments designed to check the redundant data in order to identify failures and also to trigger recovery [Staa, 2000].

Secure state change, dirty row flag, inverse functions, watchdog and memento patterns discussed in chapter Technologies and tools that support the development of recovery oriented software are examples of code redundancy. Dirty row flag is an example of data redundancy. Assertions can be considered not only code redundancy, but also a third kind of redundancy: the “reasoning redundancy”. When a programmer codes a component that contains assertions, she must care about the functionalities of

the component and also about the executable assertions that will check for the correctness of these functionalities. This imposes a “double reasoning” for the developer, from which she can benefit when writing write code that has a higher chance of being correct by construction.

Redundancy, in most cases, means overhead, which means extra cost in development (architecture, design and coding) and production (more consumption of computational resources). It is necessary to evaluate the cost-benefit for each kind of redundancy that will be developed.

4.2. Recovery and Redundancy

As mentioned before, recovery without redundancy, if not impossible, is very limited. Only simpler strategies can be applied. Defining the level of redundancy required for desired recovery is also an important issue – usually, the best results come from mixing redundancy strategies (code, hardware and data redundancy).

Recovery is usually heavily tied to domain issues, which means that each system will have its own recovery code. Some possible failures may have a common strategy for recovery, so it is important to separate “classes of failures” and “strategies for recovery”. A mapping between them must be created, but as usually the process is failure-driven, thinking about the classes of failures ends up leading to the creation of the recovery strategies, so the mapping usually is established naturally. This mapping must be used to find the best positions (in the code) to put the failure detection and failure handling code (if the use of exceptions is possible, the mapping points the throws and catches positions).

When thinking about recovery code, it is important to take into consideration two kinds of systems:

1. Stateless systems: these systems do not hold any kind of state, internally (volatile memory) or externally (like databases), or its state is not important for the overall execution, or even its state can be reliably restored by any means;
2. Statefull systems: these systems do hold important information that cannot be simply discarded.

Recovery for stateless systems may be achieved by simply restarting it, depending on the requirements for the “time to resume work”. This is usually found on embedded systems, for example, software that controls radio appliances for cars. Often, such software has memory positions where the user may record dial frequencies or sound and level preferences, but losing such information is something acceptable (obviously, if the MTBF is high). For such systems, a watchdog constantly checking for failures, restarting it whenever one is found is usually enough to provide a good level of recovery. Another example would be a system that stores medical profiles externally, in a flash memory for example. Such system may be simply restarted.

Recovery for statefull systems is more complex, because it requires knowledge of previous valid states. However, when this knowledge can be derived or acceptably estimated from sources outside of the system and post-processed, strategies suitable for stateless systems can be used.

For an example of a statefull system for which the recovery strategy can be similar to a stateless one, consider the situation where an odometer sensor is used to determine the positions for a pipeline inspection tool that acquires samples from geometric sensors in order to identify anomalies. This tool works independently: once started, it is placed at the beginning of a pipeline and, some time later (maybe hours or days) it is recovered at the end of the line. The odometer generates a fixed number of pulses per revolution, in such a way that by knowing its diameter, it is possible to estimate the covered space. Consider that the embedded software for this tool, which controls the data acquisition, is sensible to the pulses: after each pulse, it increases an internal counter, acquires a sample and writes this sample into a storage device (like a flash RAM or a hard disk). It can be noticed that, if this counter is lost and starts counting from zero within a very short period of time, it is possible to fix the data externally: it is only a matter of consulting the previous sample and adding its position to all following samples. If a failure is detected during runtime, it is possible to adopt the restart strategy if the reboot time is small enough to guarantee that the range without samples (thus, not covered by the inspection) is acceptable. After the inspection is finished, external software must be run on the data acquired to check for odometer consistency and fix any problems. One may argue that simply adding the previous odometer may lead to lack of precision because the range without samples is not

being taken into consideration. A way to enhance precision is to add a timestamp to each sample given by an external clock that works independently from the embedded software. By using the previous samples to estimate the tool speed, it is possible to estimate the range without samples by counting the amount of time elapsed from the first sample with odometer 0 to the previous last sample with odometer different from zero. This range can then be used to enhance the recovery precision.

Statefull systems in which the state cannot be derived (or acceptably estimated) from external sources need to care about "state tracking" in order to proper recover from failures. However, this state tracking does not necessarily mean "periodically taking a snapshot of the system and saving it". It is possible to have "trial states", where the system is trying to move towards a new state, but keeps the old state available in case a failure happens.

For an example of a statefull system which relies on trial states to recover from failures, consider the example of a pipeline inspection tool that sends data in real time, enabling an online inspection. As it moves, it continuously scans for anomalies. In addition to the embedded software, it requires also software for the operator to use the tool. Consider that:

- The tool does not cover all the pipe, so each scan covers only a specific area of the pipe, and the goal is to cover the whole line;
- Each scan is independent, and has a start point and an end point, and also the timestamp it begun;
- Each scan is stored in a file, for further analysis, as it is being acquired.

A possible failure would be: the software is terminated during a scan, leaving an incomplete scan in the scan-file. The last valid state of a scan-file would require the absence of the incomplete scan, hence this would be a natural state to be rolled back. A way to keep track of this trial state is to mark the scans on the file. For example, in addition to the start timestamp, the end timestamp could also be recorded, but only at the end of the acquisition, when the scan is finished. While the scan is under execution, the end timestamp must be empty. When a scan-file is opened, it must be checked for the most recent scan and, if the scan does not have an end date, it must be deleted. This is exactly what happens with Gipmag and RTScan when operating in real-time data acquisition. Gipmag

and Pipescan also make use of trial states to “measure defects” – measuring a defect is a user-assisted process that involves several steps, and changes a defect data. While in the process, some states for the data of the defect being measured are invalid due to incomplete information (which will be completed as the steps of the process go on). The record for each defect under measurement is marked (using the dirty row flag pattern) and, when a marked record is accessed, it is cleaned and the user is warned about the cleaning operation. All these examples will be discussed in the chapter “Experiments and Results”.