Object-Oriented Stable Storage Based on Mirroring

Xavier Caron, Jörg Kienzle, and Alfred Strohmeier

Swiss Federal Institute of Technology in Lausanne
Department of Computer Science
1015 Lausanne EPFL, Switzerland
{xavier.caron, joerg.kienzle, alfred.strohmeier}@epfl.ch

Abstract. Stable storage can be seen as an ideal storage medium that, given a set of failure assumptions, protects user data from corruption or loss. The integrity of the stored data must be guaranteed even in the presence of crash failures. In this paper, we show how to realize stable storage using a technique called mirroring. The main idea is to write the data to two locations instead of one, in a sequential order. If one write operation fails, the technique ensures that the other copy is in a consistent state. It may be the state that was valid before the write operation, or it may already be the new one. Of course, there must be some mechanism to determine which one is correct. The purpose of the paper is therefore to describe the mirroring algorithm, and to present a state automaton covering all possible situations that can occur in the case of crash failures. Finally, an implementation in Ada 95 is presented.

Keywords. Memory Management, Mirroring, Shadowing, Stable Storage, Fault Tolerance, Ada 95.

1 Introduction

The concept of *Stable Storage* has its origins in the realm of transactions and databases. A stable storage unit can be seen as an ideal storage medium that, given a set of failure assumptions, protects user data from corruption or loss. Such a storage unit offers two operations to the user, *Write* and *Read*, which can be used to store and retrieve user data to and from stable storage.

The name of 'stable storage' has been first introduced in [1]. The paper describes how conventional disk storage, that shows imperfections such as bad write operations or decay, can be transformed into an ideal disk with no failures using a technique called mirroring. In this paper, we present this technique and show how to convert any nonvolatile storage into stable storage.

Stable storage guarantees atomicity of the write operation, e.g. either all data is written to the storage unit, or none at all, even in the case of a crash. So a write operation appears as indivisible. As a result, a read operation will always return consistent data. However, the user of a stable storage unit has to design the application in such a way that, after recovering from a crash failure, it can deal with the system either in the old or in the new state without knowing which one holds.

A. Strohmeier and D. Craeynest (Eds.): Ada-Europe 2001, LNCS 2043, pp. 278-289, 2001.

The remainder of this paper is organized as follows: the next section describes mirrored storage and the related algorithms. Section 3 shows how to integrate mirrored storage into a storage hierarchy, how it can be implemented in Ada 95, and how to use it. A conclusion and references complete the paper.

2 The Mirroring Algorithm

In specialized literature, the "*mirroring*" technique, sometimes called "*shadowing*", often refers to duplication of data. For example, the *Ralston Encyclopedia of Computer Science* [2] says:

> *Another recent trend is to duplicate data to enhance reliability. This technique, called mirroring or shadowing, allows systems to continue operation in spite of media, controller, or channel failure. Sophisticated systems also take advantage of the extra I/O path to enhance throughput. On-line reconstruction ("re-mirroring") of a new second copy when one of the original two is lost is also common.*

The main idea is to write data in two locations instead of one, in a sequential order. If one write operation fails, we assume that the other copy is in a consistent state. It may be the state that was valid before the write operation, or it may already be the new one. Of course, there must be a mechanism to determine which one of the two copies contains the valid data. For this purpose, a third storage unit called the *log* is used. It allows us to distinguish between the three possible situations depending on the moment of the crash:

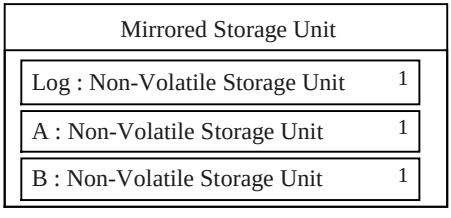

Fig. 1. Components of a Stable Storage Unit based on Mirroring (UML notation)

- The crash happens before or after the write operation, i.e. the log does not indicate any problem,
- The crash happens while writing the first copy, or
- The crash happens while writing the second copy.

The three components used for the algorithm are shown in fig. 1. The data copies are called *A* and *B*.

Mirroring can be used for instance in a transactional system in order to keep uncorrupted a log table mapped on sequential files, as explained in [3].

2.1 Preliminary Assumptions

Before describing the mirroring algorithm, we have to specify our failure assumptions i.e. under which conditions we can guarantee the stability of the storage unit.

If any of the storage units used for storing the log and the data copies does not meet one or more of the following assumptions, then the resulting mirrored storage unit can not be considered stable.

Non-volatile Storage. The storage units used for holding the log and the two data copies must be non-volatile, i.e. they must retain their contents even in the case of a crash failure.

Failure Isolation. Our main assumption is that a crash while executing a write operation on a storage unit can only corrupt the contents of that particular storage unit, and no other data stored on the same device or another device.

Non-destructive Reads. Reading from storage will not corrupt the data, even in the presence of crash failures.

Unbuffered Writes. Our algorithm is composed of *sequential* write operations to the log and data copies. It is essential for the correct working of the mirroring algorithm that when a new write operation begins, the previous one has been completed successfully. This assumption may not be met if the storage device uses buffering or caching that writes physically to the device only when the buffer is full. However, such devices often offer a *flush* operation that forces the buffer to be written out to the device. This operation must therefore be called after every write.

Error-free Reading and Writing. We assume that the storage units holding the two data copies and the log provides error-free read and write operations. For the interested reader, more information on how to construct a higher-level abstraction to handle read and write errors can be found in [1].

2.2 Write and Read Operations

Mirrored Write Operation. The algorithm of the mirrored write operation is summarized in fig. 2. The initial value of the log is *OK*.

1. Set Log to *A* (i.e. *writing to A*)
2. Write the data to A
3. Set Log to *B* (i.e. *writing to B*)
4. Write the data to B
5. Set Log to *OK*

Fig. 2. Mirrored Write Operation

We can illustrate this algorithm with a state automaton (fig. 3). Each state is composed of three values: the value in the log and the two data copies. The log can take the values OK (in short O), A and B. The data copies can be in the states P or N, where P, respectively N, means that the copy is still in the *previous*, respectively al-

ready in the *new* state. The initial state is therefore O | P | P, the final state O | N | N. Each arrow between two states means writing to a storage.

Mirrored Read Operation. The mirrored read operation is just a normal read of any of the two copies, since they are identical after a successful write operation.

2.3 Handling Crashes

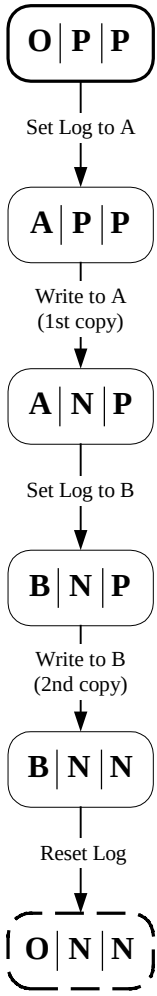

Fig. 3. State Automaton Describing the Mirrored Write Operation

The sequence of the state automaton (fig. 3) can be interrupted at any time by a crash failure. According to our preliminary assumptions, the storage unit that is open in write mode at that moment may get corrupted. Upon restart, before executing any I/O operation, we must check the consistency of the mirrored storage. We *never* check the contents of the data copies itself, only the log. If the log is not set to OK, we have to perform a cleanup operation.

Cleanup Operation. According to our preliminary assumptions only one storage unit can be corrupted by a crash at any given time. The table in fig. 4 summarizes the required cleanup operations based on the information found in the log.

Second Level Crash Failures. We have to pay special attention to new crash failures happening during cleanup, since we always must keep at least one valid data copy. Since we only *read* the valid copy, our failure assumptions guarantee that it will not get corrupted. When a crash occurs there are two possible cases:

- If the new crash happens while repairing a suspected data copy, then the log is unchanged.
- If the new crash happens while resetting the log to OK, then we have successfully repaired the suspected data copy but we cannot know it because the log is corrupted.

In both cases we perform a complete cleanup operation when restarting.

State of Log	Suspected Problem	Cleanup
OK	None.	None.
A (i.e. *writing to A*)	A was not successfully written and might be corrupted.	Copy B to A. Set Log to OK.
B (i.e. *writing to B*)	B was not successfully written and might be corrupted.	Copy A to B. Set Log to OK.
X (i.e. *corrupted*)	Neither A nor B is corrupted, but they might contain different data.	Copy A to B (or B on A). Set Log to OK.

Fig. 4. Cleanup Operation Summary

2.4 State Automaton Describing the Cleanup Algorithm

To verify that our algorithm safely covers all cases, we can represent the complete mirroring algorithm (write and cleanup operation) by a state automaton shown in fig. 5. In addition to being in a normal state, a storage unit can also be corrupted, which is represented by the symbol X. The question mark stands for different possible cases, depending on the damage created by the crash.

We can distinguish four vertical columns in the automaton, each one representing a different kind of system state:

1. *Normal states*: the left vertical column of states represents the normal sequence of write operations in the absence of crash failures.
2. *Crash states*: the states of the second column can only be reached after at least one crash failure. After every new crash, we must restart in one of these states. One or two storage units might be corrupted depending on the number of previous crash failures, but at least one of the data copies is not corrupted.
3. *Re-mirroring states*: the third vertical column is reached when the "re-mirroring" operation, i.e. the copy of one storage unit to the other, is successfully completed.
4. *Final states:* the last column consists of the two possible final states after a successful cleanup operation: either O | P | P or O | N | N, the latter being the normal consistent final state without any crash failure.

The Rubicon Property. There is clearly a separation, shown by a dashed line, between the upper and the lower part of the automaton. When a crash happens in the upper part, the final state after recovery will be the one before calling the mirrored write operation, i.e. O | P | P. Otherwise (writing the first copy to A has been completed successfully), the mirrored write operation will succeed, i.e. the final state will be O | N | N, even if still other crash failures occur.

Unreadable Log and Uncertainty. After a crash, when the log can not be read, i.e. the system is in one of the states X | P | ? or X | N | ?, there is uncertainty about its actual state. However recovery can be performed because the actions to be taken in both states are exactly the same, i.e. copy A to B and set the log to OK.

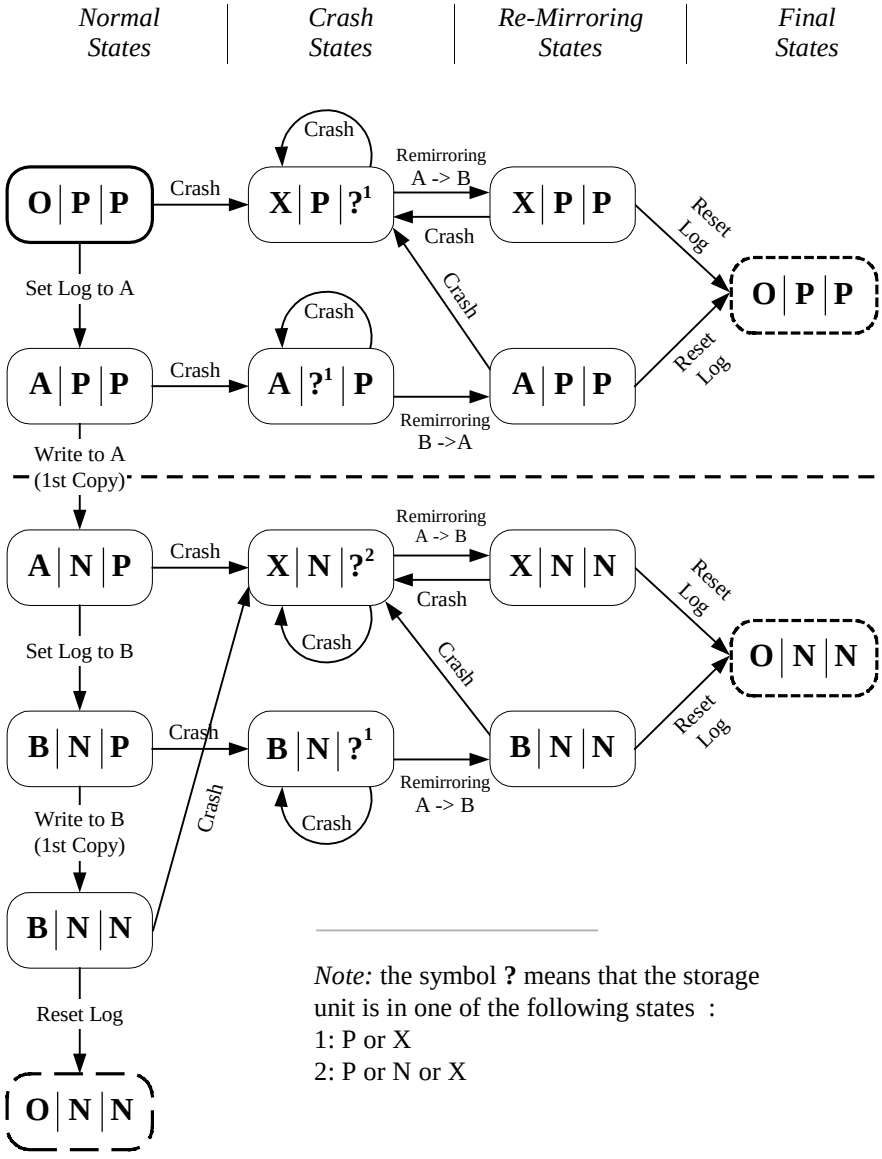

Fig. 5. State Automaton for the Mirroring Algorithm

2.5 Time Needed for the Mirrored Write Operation

Using the automaton of fig. 5 and assigning probabilities to basic operations, we can estimate the time needed to complete a mirrored write.

Time without Crash Failures. We suppose that the two storage units A and B are attached to similar devices. According to our algorithm, the execution time of a mirrored write operation without any crash failure, i.e. the best time, is equal to:

$$t_{normal} = 3t_l + 2t_d \quad (1)$$

where t_l is the time needed to set the log, and t_d the time needed to write the data to A or B.

Time with Crash Failures. To recover from a single crash, it takes the time $t_{restart}$ to restart the application plus the time for one re-mirroring operation, i.e. t_d, plus the time for one log reset, i.e. t_l. However this is only an upper limit, because a crash might occur before the re-mirroring operation is completed. Therefore, if n denotes the total number of crash failures, the time spent in the state automaton is bound by:

$$t_{crash} < t_{normal} + n(t_{restart} + t_l + t_d) \quad (2)$$

Conclusion. If we consider that the time needed to set the log is negligible in comparison with the time needed to write a data copy, we can simplify the equation 1 to:

$$t_{normal} \approx 2t_d \quad (3)$$

It means that mirrored storage needs twice the time of conventional storage. With the high speed of actual storage devices, this should be acceptable in most cases.

2.6 Transition Probabilities

Transition Matrix. Let's denote by p_l the probability that there is a crash while setting the log, and p_d the probability that there is a crash while writing a data copy. We can then assign probabilities to the transitions of the state automaton of fig. 5. The transition matrix is shown in Fig. 6. Rows and columns represent the states of the automaton. The value in a cell is the probability to go from the state represented by the row to the one associated with the column. Note that it is a Markov transition matrix since the sum of the probabilities on a line is always 1, except for the final states.

Probabilities for Final States. As said previously, there are two kinds of final states: either O | P | P, which is the starting state, or O | N | N. We saw that if the automaton succeeds in passing through the dashed line (fig. 5), the final state is O | N | N, and otherwise it is O | P | P. Based on this observation, we can compute the probabilities of the final states.

States	O/P/P	A/P/P	X/P/?	A/?/P	X/P/P	A/P/P	O/P/P	A/N/P	B/N/P	X/N/?	B/N/?	X/N/N	B/N/N	O/N/N	B/N/N	O/N/N
O | P | P	0	$1-p_l$	p_l	0	0	0	0	0	0	0	0	0	0	0	0	0
A | P | P	0	0	0	p_d	0	0	0	$1-p_d$	0	0	0	0	0	0	0	0
X | P | ?	0	0	p_d	0	$1-p_d$	0	0	0	0	0	0	0	0	0	0	0
A | ? | P	0	0	0	p_d	0	$1-p_d$	0	0	0	0	0	0	0	0	0	0
A | P | P	0	0	p_l	0	0	0	$1-p_l$	0	0	0	0	0	0	0	0	0
O | P | P	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A | N | P	0	0	0	0	0	0	0	0	$1-p_l$	p_l	0	0	0	0	0	0
B | N | P	0	0	0	0	0	0	0	0	0	0	p_d	0	0	0	$1-p_d$	0
X | N | ?	0	0	0	0	0	0	0	0	0	p_d	0	$1-p_d$	0	0	0	0
B | N | ?	0	0	0	0	0	0	0	0	0	0	p_d	0	$1-p_d$	0	0	0
X | N | N	0	0	0	0	0	0	0	0	0	p_l	0	0	0	$1-p_l$	0	0
B | N | N	0	0	0	0	0	0	0	0	0	p_l	0	0	0	$1-p_l$	0	0
O | N | N	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
B | N | N	0	0	0	0	0	0	0	0	0	p_l	0	0	0	0	0	$1-p_l$
O | N | N	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 6. Transition Matrix for the Mirroring State Automaton

1. To finish in the state O | N | N, the dashed line must be crossed. The unique possibility is to go directly from O | P | P to A | P | P (with a probability $1\text{-}p_l$) and then directly from A | P | P to A | N | P (with a probability $1\text{-}p_d$):

$$P_{O/N/N} = (1\text{-}p_l) \,.\, (1\text{-}p_d) \tag{4}$$

2. Because the mirroring algorithm guarantees to finish in one of the two kinds of final states, the probability to finish back in the state O | P | P is the complement to 1 of the previous probability:

$$\begin{aligned} & P_{O/P/P} = 1 - P_{O/N/N} \\ \Leftrightarrow \quad & P_{O/P/P} = 1 - (1-p_l) \,.\, (1-p_d) \\ \Leftrightarrow \quad & P_{O/P/P} = 1 - (1 - p_l - p_d + p_l \,.\, p_d) \\ \Leftrightarrow \quad & P_{O/P/P} = p_l + p_d - p_l \,.\, p_d \end{aligned} \tag{5}$$

3 Implementation in Ada 95

We want to be able to write the state of any object to our storage. Ada *streams* can be used for that purpose [4]. A stream is a sequence of elements comprising values of possibly different types. The standard package `Ada.Streams` defines the interface for

streams in Ada 95. It declares an abstract type `Root_Stream_Type`, from which all other stream types must derive.

In Ada 95, the predefined attributes `'Write` and `'Output` are used to write values to a stream by converting them into a flat sequence of stream elements. Reconstructing the values from a stream is performed with the predefined attributes `'Read` and `'Input`. These two attributes make dispatching calls to the `Read` and `Write` operations of the `Root_Stream_Type`. With `'Write` and `'Read`, neither array bounds nor tags of tagged types are written to or read from the stream. `'Output` and `'Input` must be used for that purpose.

[5], [6] construct a framework for providing persistence for Ada objects based on streams. It classifies storage devices in a class hierarchy according to essential properties, like volatility, stability, etc. The abstract root class Storage (fig. 7) defines the common interface for all storage classes, including Read and Write operations. The storage hierarchy is then split into *volatile* and *non-volatile* storage. Data stored in non-volatile storage remain intact even when the program terminates. Among the different types of non-volatile storage, there is then the distinction between *stable* and *non-stable* storage. The mirrored storage finally is a subclass of the stable storage class.

For storing the log and the data copies, we need non-volatile storage units, but the mirroring algorithm is independent of the kind of non-volatile storage actually used.

Using the *Strategy design pattern* [7], we can implement a mirrored storage class, whose instances are supplied with parameters at creation time. These parameters specify the kinds of non-volatile storage the application programmer chooses, depending on the needs of the application. To help him make this choice, a concrete non-volatile storage class must document any applicable constraints and provide information about the performance of its instances. E.g. the log is frequently accessed, but holds only a small piece of information, to the contrary of the data copies.

The structure of the collaboration is shown in fig. 7. The class is an aggregation of three non-volatile storage objects (Log, A and B). The specific kinds of these three objects are chosen when creating an object of the mirrored storage class. It is therefore possible to *reuse* concrete implementations of the non-volatile storage class to create a variety of mirrored storage devices.

Storage Parameters. Because we want to provide persistent storage, there must be some means to uniquely identify storage objects. Storage identification is usually device dependent. Files for instance have file names associated with them, but other storage devices may use different identification means. In order to provide correct identification for each storage type, a parallel hierarchy of storage parameter classes has been introduced. A concrete parameter class contains the necessary identification data for a particular storage device. Each parameter class must also provide operations to convert the parameter to and from a string. This string will be used as a common, device-independent means for identifying storage objects.

3.1 Implementation Details

The mirroring class is defined as follows:

```
type Mirrored_Storage_Type is new Stable_Storage_Type with private;
type Mirrored_Storage_Ref is access all Mirrored_Storage_Type'Class;
... --  Usual operations declarations for non-volatile storage
... --  Read, Write, Get_Current_Size, Open, Close, Delete
```

```
private
   type Mirrored_Members_Type is new Limited_Controlled
      with record
         Log : Non_Volatile_Storage_Ref;
         B : Non_Volatile_Storage_Ref;
         A : Non_Volatile_Storage_Ref;
      end record;

   procedure Finalize (Members : in out Mirrored_Members_Type);

   type Mirrored_Storage_Type is new Stable_Storage_Type
      with record
         Members : Mirrored_Members_Type;
      end record;
```

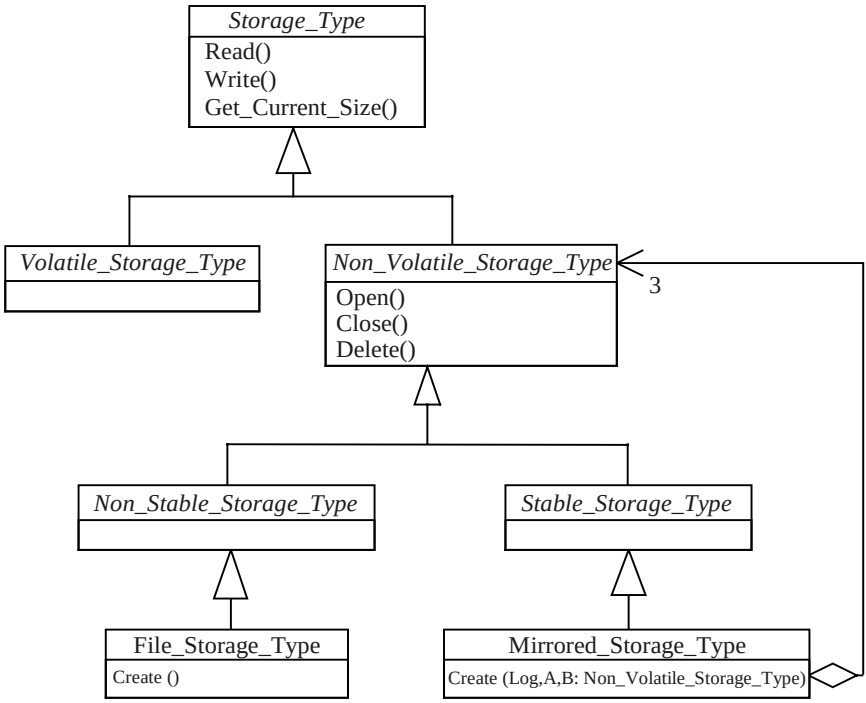

Fig. 7. Mirroring in the Storage Hierarchy

First of all, note how we declare the three components `Log`, `A` and `B` as references to the non-volatile storage class-wide type `Non_Volatile_Storage_Ref`. It means that the implementation does not rely on any specific type of non-volatile storage.

Deriving the members of the mirrored storage from a controlled type allows one to deallocate the three non-volatile storage components in the `Finalize` procedure. In this way, we ensure that the three storage components are released when the mirrored storage is deleted.

3.2 Example of Use

The example shows how to save integers to a mirrored storage unit.

```
with Mirrored_Storage_Params; use Mirrored_Storage_Params;
with Mirrored_Storages; use Mirrored_Storages;
with Streams; use Streams;

procedure Save_Integer is
   My_Params : Mirrored_Storage_Params_Type;
   My_Storage : Mirrored_Storage_Ref;
   My_Stream : Stream_Ref;

begin
   My_Params := String_To_Storage_Params ("AdaEurope");
   My_Storage :=
      Mirrored_Storage_Ref (Create_Storage (My_Params));
   My_Stream := new Stream_Type (My_Storage);
   Integer'Write (My_Stream, 6577);
   Close (My_Storage.all);
end Save_Integer;
```

The package Streams contains the type Stream_Type. A new instance of a stream is associated with a storage object at creation time, as shown in the code. The attribute `'write` makes a dispatching call to the write operation of the associated storage object.

4 Conclusion

We proposed a technique called *mirroring* to convert non-volatile storage into *stable storage*. A stable storage unit can be seen as an ideal storage medium that, given a set of failure assumptions, protects user data from corruption or loss in the presence of crash failures. The technique uses two data storage units instead of one, completed by a log that indicates if data was corrupted during the crash needing cleanup.

The algorithm can be represented by a *state automaton* showing that all cases of even multiple crash failures can be handled. Based on the state automaton we established upper limits for the time behavior. We also showed how to assign probabilities to the transitions and estimate the probability of reaching the final state.

The mirrored storage can easily be integrated in a storage hierarchy, transforming any non-volatile, non-stable storage (e.g. a local file) into stable storage. This design was successfully implemented in Ada 95, but the technique does not rely on any specific feature of the programming language.

References

1. Lampson, B.W., Sturgis, H.E.: "Crash Recovery in a Distributed Data Storage System". *Technical report, XEROX Research*, Palo Alto (June 1979). Much of the material appeared in *Distributed Systems-Architecture and Implementation*, ed. Lampson, Paul, and Siegert, *Lecture Notes in Computer Science*, Vol. 105. Springer Verlag (1981), pp. 246-265 and 357-370.
2. Ralston, A., Reilly, E.D.: *Encyclopedia of Computer Science Third Edition*. Van Nostrand Reinhold, New York (1993).
3. Gray, J., Reuter A.: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, California (1993).
4. ISO: International Standard ISO/IEC 8652:1995(E): *Ada Reference Manual, Lecture Notes in Computer Science*, Vol. 1246. Springer Verlag (1997); ISO 1995.
5. Kienzle, J., Romanovsky, A.: "On Persistent and Reliable Streaming in Ada". In *Reliable Software Technologies - Ada-Europe'2000, Potsdam, Germany, Lecture Notes in Computer Science*, Vol. 1845. Springer Verlag (2000), pp. 82-95.
6. Kienzle, J., Jiménez-Peris, R., Romanovsky, A., Patiño-Martinez, M.: "Transaction Support for Ada". *International Conference on Reliable Software Technologies - Ada-Europe'2001, Leuven, Belgium, May 14 - 18, 2001*, to be published in Lecture Notes in Computer Science, Springer Verlag (2001).
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns*. Addison Wesley, Reading, MA(1995).
8. Kienzle, J., Strohmeier, A.: "Shared Recoverable Objects". In *Reliable Software Technologies - Ada-Europe'99, Santander, Spain, Lecture Notes in Computer Science*, Vol. 1622. Springer Verlag (1999), pp. 387-411.
9. Wolf, T., Strohmeier, A.: "Fault Tolerance by Transparent Replication for Distributed Ada 95". In *Reliable Software Technologies - Ada-Europe'99, Santander, Spain, Lecture Notes in Computer Science*, Vol. 1622. Springer Verlag (1999), pp. 412-424.