Software Engineering
Laboratory

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

OBJECT-ORIENTED STABLE STORAGE

Diploma Thesis of

Xavier Caron

Computer Science Department

Professor: Alfred Strohmeier

Supervisor: Jörg Kienzle

February 25, 2000, Lausanne, Switzerland

To my goddaughter, Christine

CONTENTS

FIGURES

ACKNOWLEDGMENTS

At this point, I should like to extend my grateful thanks to Professor Alfred Strohmeier of the Software Engineering Lab of the Swiss Federal Institute of Technology (EPFL) for having supervised this diploma thesis and for his detailed interest in this work. I specially wish to express my gratitude to Jörg Kienzle, PhD applicant of the same lab, for his immense help in this work, for all his kindness and angelic patience.

OBJECT-ORIENTED STABLE STORAGE

DIPLOMA THESIS

ABSTRACT

This report is the result of a diploma thesis of Xavier Caron, a computer science student at the Software Engineering Lab (LGL), which is part of the Swiss Federal Institute of Technology (EPFL).

The LGL is currently developing an object-oriented library providing transaction support for Ada programmers. *Transactions* are actions that have the well-known ACID properties (Atomicity, Consistency, Isolation, and Durability). Durability of objects can be achieved by saving their state to so-called *"stable storage"*. The goal of this project is to implement the two following forms of stable storage:

- Stable storage using a technique called *mirroring*. The integrity of the stored data must be guaranteed even in the presence of crash failures.
- Stable storage based on *replication*. Using the distributed programming facilities of Ada 95, a mechanism must be devised that allows to broadcast the information that has to be made durable to a group of collaborating machines for storage. Of course, it should also be possible to retrieve the data in a consistent manner.

This report describes the system design and implementation in Ada 95.

The recent revision of the language Ada standard, Ada 95, introduces a new mechanism called *streams*. This mechanism allows structured data to be flattened. Streams are sequences of elements comprising values from possibly different types (Ada 95 allows programmers to develop their own streams following the standard abstract class interface. For a complete survey the reader is referred to [1 13.3]).

As a small part of his PhD thesis on distributed transactions and tasking, Jörg Kienzle shows in [2] how to use the stream concept for developing new features to provide internal program data saving suitable for *fault tolerance* and *persistence*. For this purpose, he introduces a hierarchy of different *storage types*, presented in chapter 1, useful in different applications domains. At one level of the hierarchy, a *stable storage* class-wide type is defined. It will be used in the transaction system to provide durability for transactional objects.

In this project, we have *to extend* this class-wide type with different concrete stable storage types. Within the time allowed, we have implemented two different kinds of stable storage. The first one is based on a method called *mirroring*. We present it in chapter 2. The second one is based on *replicated, distributed memory*. For that purpose, and after a brief analysis, we decide to design a multi-purpose *remote storage*, as shown in chapter 3. This remote storage has the same communication architecture of the replicated storage but does not need a so elaborated protocol. In chapter 4, the entire replicated distributed storage system is described.

Note that the sections about the implementation of the described concepts are always clearly separated. The reader not interested by the implementation details has every opportunity to skip these, though it could be quite instructive.

User Guarantees on the I/O Stable Operations.

Let us consider an accounting system: assets and liabilities must sum to zero. It is an *invariant* for the system. For any system, we can say that a given state is *consistent* if it satisfies some predicate called the invariant of the system. For a properly used storage system, its invariant must be maintained in spite of crashes and concurrent accesses. The challenge is to provide such facilities.

Let us observe that any computation, which takes a system from one state to another, can be represented by a function *F* (without side effects) such as:

```
state := F(state);
```

The function *F* will change only an *output set* of data, depending only on the values of some other set of data, the *input set*. In the presence of crashes, the *atomicity* of such a computation will be ensured if the assignment itself is atomic, that is, if either all the writes are done, or none of them are. Two such computations *F* and *G* may run concurrently and still are atomic if the input set of *F* is disjoint from the output set of G, and vice-versa.

Thus, our stable storage must guarantee that after recovery from a system crash, for each *write operation*, either all data will have been written, or none will have been. So a write operation appears indivisible. It has the atomicity property. In those conditions, the *read operation* will then always return consistent data. The user has to write his application in such a way that, after recovering from the crash failure, the data can be either in the old state or in the new one.

The *storage hierarchy* presented here is a part of a whole framework (see [3] for complete description) developed by J. Kienzle in his PhD thesis on distributed transactions and tasking. With the intention to provide *persistence* support for Ada 95 programmers i.e. a mechanism that allows the state of an object to persist between different executions of an application, he introduces this hierarchy which lets the programmer choose on what he wants to store the state of the objet.

This storage hierarchy is based on the *design pattern* called *Strategy*, presented in [4]. During creation of an object, the storage can be chosen dynamically. In our diploma thesis, we will present some examples in *Ada 95*, but it does not rely on any kind of special programming language features and therefore can be implemented in any object-oriented language.

1.1 AN ABSTRACT CLASS HIERARCHY

In case of computer crashes or errors caused by software design faults, persistence is used by some software fault tolerance mechanisms to provide state restoration. The objects that we want to make persistent must save their state to some store (a *storage*), so that it can be retrieved again in a later execution. The term *storage* is used in a wider sense here. Sending the state of the object over a network and storing it in the memory of some other computer would for instance also make sense, as long as the data survives program termination.

Nevertheless, all storage types do not have the same properties, and therefore must not all provide the same set of operations. An *abstract* class hierarchy is the most natural way to represent the structure of such storage types. A *concrete* implementation of a storage type must *derive* from one of the storage classes and implement the required operations.

The different storage types are classified in the way presented in figure 1. The `Storage_Type` class represents the interface common to all storage types. The operations `Read` and `Write` represent the operations that allow reading and writing data from and to the storage device. What kind of value types they must support will not be discussed in detail here but we can consider that those operations work with *stream elements*, generated at a higher level by a *serialization mechanism*. Some object-oriented programming languages, for example Ada [1] or Java [5.13], have such a mechanism as a predefined implementation for all value types.

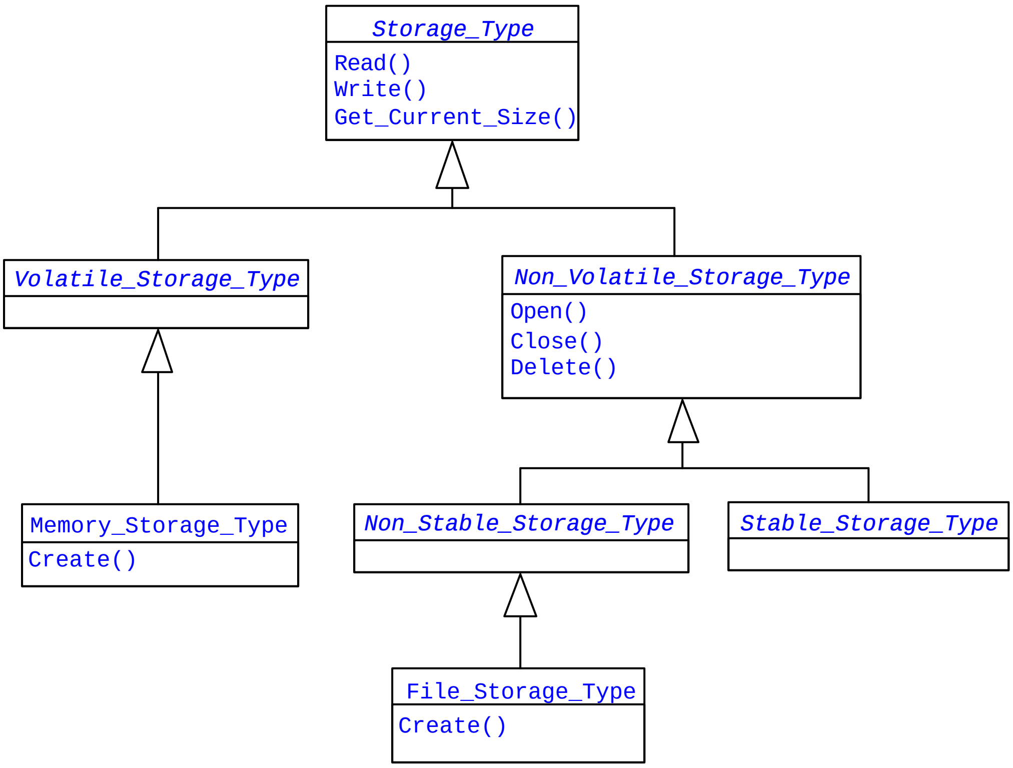

Figure 1 - Storage Class Diagram

Volatile and Non-Volatile Storage

The storage hierarchy is split into *volatile* storage and *non-volatile* storage. Data stored in volatile storage will not survive program termination. An example of a volatile storage is conventional computer memory. Once an application terminates, its memory is usually freed by the operating system, and therefore any data still remaining in it is lost. Data stored in non-volatile storage on the other hand remains on the storage device even when a program terminates. Databases, disk storage, or even remote memory are common examples of non-volatile storage. Since the data will not be lost when the program terminates, additional housekeeping operations are needed to establish connections between the object and the actual storage device, to cut off existing connections, and to delete data that will not be used anymore. Theses operations are `Open`, `Close` and `Delete`.

Stable and Non-Stable Storage

The kind of storage to be used for saving application data depends heavily on the application requirements. Properties such as performance, capacity of the storage media and particularities of usage (for instance write-once devices like CD writers) may affect the choice. Persistence can be implemented in a stronger form to support fault tolerance of different sorts, including tolerating software design faults (bugs), for instance by using the recovery block scheme, or tolerating faults of the underlying hardware, for instance by using checkpoints or recovery points. To apply persistence properly and to choose the suitable storage type, the application programmer has to identify the fault assumptions and to know the reliability of the storage devices that can be used.

This is why among the different non-volatile storage devices, we distinguish *stable* and *non-stable* devices. Data written to non-stable storage may get corrupted, if the system fails in some way, for instance by crashing during the write operation. Stable storage ensures that stored data will never be corrupted, even in the presence of application crashes and other failures.

Different Kinds of Stable Storage

Stable storage has been first introduced in [6]. The paper describes how conventional disk storage that shows imperfections such as bad writes and decay can be transformed into stable storage, an ideal disk storage with no failures, using a technique called mirroring. We present this technique in detail in next chapter. Using this technique, any non-volatile, non-stable storage can be transformed into stable storage.

The mirroring technique is not the only one that can be used to create stable storage. *Database systems* for instance have their own mechanism to guarantee atomic updates of data. Typically structuring updates of data as transactions does this. A transaction can be committed, in which case the updates will be made permanent, or aborted, in which case the system remains unchanged. If any kind of failure occurs during the transaction, the data also remains unchanged. It is possible to write a concrete stable storage class that provides a bridge between the object-oriented programming language and a database. Mirroring is the first main part of our thesis and is described in chapter 2 of this report.

Yet another form of providing stable storage is replication. The state of a persistent object can be *broadcasted* over the network and stored for instance in remote memory. Although memory is usually not seen as non-volatile, from the application point of view it is, since it survives program termination. The group of replicas as a whole can be considered stable, for as long as at least one of the remote machines remains accessible, the data can always be retrieve during a later execution. Replication is the second main part of this thesis and is described in chapter 4 of this report. To achieve replication, we first developed a multi-purpose non-stable *remote storage*, presented in chapter 3.

1.2 A PARALLEL HIERARCHY: THE STORAGE PARAMETERS

When creating an instance of a persistent object, the application programmer must be able to specify on what kind of storage he wants the state of the object to be saved. The object can then create an instance of the corresponding storage class and thus establish a connection to the storage device.

The information needed to create an instance of a concrete storage class is device dependent. To create a new file, a user must typically provide a file name that follows certain conventions, and maybe also a path that specifies in which directory the file should be created. To access remote memory, an IP number or machine name must be provided. To solve this problem, a parallel hierarchy of storage *parameters* has been introduced. It has the same structure as the storage hierarchy (see figure 2). This allows each storage class to define its own storage parameter type containing all the information it needs to uniquely identify data stored on the device.

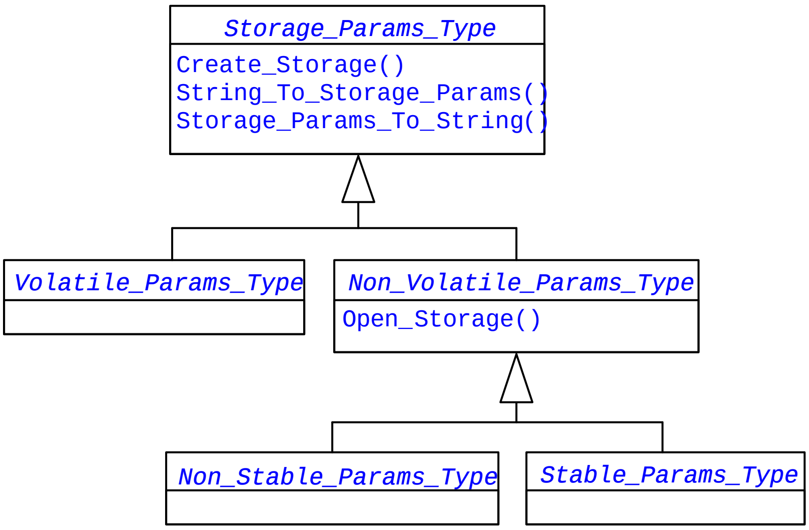

Figure 2 - Storage Parameter Class Diagram

At the same time, the storage parameter class allows a user to create instances of storage classes. An abstract factory operation, `Create_Storage` is declared in the storage parameter class. Each concrete storage parameter class must provide an implementation for this method: the corresponding creator function of the storage class must be called, passing as a parameter the information stored inside the concrete storage parameter instance. Non-volatile storage needs a second creator function, `Open_Storage`, that will instantiate the non-volatile class without creating a new storage on the device. Instead a connection between already existing data and the storage object will be established.

The `Create_Storage` and the `Open_Storage` operations define the connection between the two parallel class hierarchies.

The `String_To_Storage_Params` function is provided to ease the creation of storage parameters. Strings can provide a common way to identify data, regardless of what actual type of storage device the data is stored on. Using the `String_To_Storage_Params` function and its inverse function `Storage_Params_To_String` it is also possible to identify data that moves from one storage device to another one using the same string.

1.3 IMPLEMENTATION EXAMPLE: CREATION OF PERSISTENT OBJECTS

The Storages Package

First, let us have a look at the interface of the hierarchy top-level `Storage_Type`:

```
with Ada.Streams; use Ada.Streams;
with Ada.Finalization; use Ada.Finalization;

package Storages is

   type Storage_Type is abstract tagged limited private;
```

```
      type Storage_Ref is access all Storage_Type'Class;

      procedure Read (Storage : in out Storage_Type;
                      Item    : out Stream_Element_Array;
                      Last    : out Stream_Element_Offset) is abstract;
      --  Read a data item from the storage

      procedure Write (Storage : in out Storage_Type;
                       Item    : in Stream_Element_Array) is abstract;
      --  Append 'Data' to the storage

      function Get_Current_Size (Storage : in Storage_Type)
         return Stream_Element_Count is abstract;
      --  Returns the current size of the storage

      Data_Not_Available : exception;

   private

      type Storage_Type is abstract new Limited_Controlled with null
      record;

   end Storages;
```

`Storage_Type` is privately derived from `Limited_Controlled` in order to allow concrete storage implementations to perform automatic initialization and finalization, if necessary. Disk files for instance should always be closed, network ports should be freed, etc. `Storage_Type` is limited, so it can store, if necessary, other limited data, such as for example file descriptors. Finally, the public view of `Storage_Type` unknown discriminants. That way the user of a storage type is forced to call one of the constructor functions of a concrete storage type; he can not just declare an instance of the type and thereby bypass correct initialization.

The operations provided by `Storage_Type` are `Read`, `Write` and `Get_Current_Size`. The `Read` and `Write` procedures are equivalent to the ones required for the stream type. Actually, the `Read` and `Write` procedures of the stream type are just call-through procedures to the associated storage device. The `Get_Current_Size` function returns the current length of the data associated with the storage in stream elements. This function has been introduced to simplify buffer management.

The Storage Parameters Package

Here is the interface for the top-level class `Storage_Params`:

```
   with Storages; use Storages;
   with Ada.Finalization;

   package Storage_Params is

      type Storage_Params_Type is abstract tagged private;

      function String_To_Storage_Params (S : in String)
        return Storage_Params_Type is abstract;

      function Storage_Params_To_String (Params : in Storage_Params_Type)
        return String is abstract;

      function Create_Storage (P : in Storage_Params_Type)
        return Storage_Ref is abstract;

   private

      type Storage_Params_Type is abstract new Ada.Finalization.Controlled
   with
        null record;

   end Storage_Params;
```

The goal of each function is described in section 1.2. Note that Storage_Params_Type is privately derived from Controlled but it is not limited, because we want to allow copies of the parameters.

Creation of a Package for Persistent Objects Management

Now the user can develop a top-level package which can be used to make any non-limited tagged type persistent. The specification of this package is as follows:

```
with Ada.Streams; use Ada.Streams;
with Streams; use Streams;
with Ada.Finalization; use Ada.Finalization;
with Storages.Non_Volatile; use Storages.Non_Volatile;
with Storage_Params.Non_Volatile; use Storage_Params.Non_Volatile;

generic

   type Base_Type is tagged private;

package Persistent_Object_G is

   type Persistent_Type (<>) is new Base_Type with private;
   type Persistent_Ref is access all Persistent_Type'Class;

   procedure Save_Object (Object : in out Persistent_Type'Class);

   function Restore_Object (Storage_Params : in
                              Non_Volatile_Params_Type'Class)
       return Persistent_Ref;

   function Create_Object (Storage_Params : in
                              Non_Volatile_Params_Type'Class)
       return Persistent_Ref;

private
   …

end Persistent_Object_G;
```

Mix-in inheritance is used to add three new operations to the base type: Create, Restore and Save. Since the persistent object type has unknown discriminants, Create and Restore must be used to declare an instance of a persistent object.

Create_Object will create a new instance from scratch, whereas Restore_Object will try and read the contents of the instance from the storage device identified by the storage parameters, assuming that the object has been previously saved to the device. Save_Object is the operation that must be called to store the contents of the object onto the associated storage.

To create persistent objects, the generic package must be instantiated:

```
with My_Types;
with Persistent_Object_G;

package Persistent_Integer is
   new Persistent_Object_G (My_Types.My_Integer);
```

The following lines of code illustrate how an instance of such a persistent integer type can be saved to a file on a disk :

```
with Storage_Params.Non_Volatile.Non_Stable.File_Storage_Params;
use Storage_Params.Non_Volatile.Non_Stable.File_Storage_Params;
with Persistent_Integer;

declare
   S: Persistent_Integer.Persistent_Ref;
   P: File_Storage_Params_Type := String_To_Storage_Params
                                                     ("filename.pst");
begin
   S := Persistent_Integer.Create_Object (P);
   S.Value := 12345;
   Persistent_Integer.Save_Object (S.all);
end;
```

Let us have a look at the implementation of the `Create_Object` function:

```
function Create_Object (Storage_Params :
                                     in Non_Volatile_Params_Type'Class)
   return Persistent_Ref is
      Result : Persistent_Ref := new Persistent_Type;
   begin
      Result.Data.Storage_Stream := new Stream_Type
           (Non_Volatile_Storage_Ref (Create_Storage (Storage_Params)));
      return Result;
   end Create_Object;
```

We see in the main instruction how a stream is allocated during a call to `Create_Object`. What we want to highlight here is that to create a stream, we need a storage object. To instantiate the storage we call the factory method `Create_Storage`, passing as an argument the given storage parameters. Note that Ada forces us to cast explicitly the returned storage reference to a non-volatile one.

2.1 MIRRORING ALGORITHM

2.1.1 TRADITIONAL MEANING OF THE WORD "MIRRORING"

Literature

In specialized literature, the technique "*mirroring*", sometimes called "*shadowing*" often refers to a duplication of data. For example, in the *Ralston Encyclopedia of Computer Science* [7], we can find the following text:

> *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 failed, it ensures that the other location is in a consistent state. It may be the state that was valid before the write operation, or it may already contain the new data. Of course, there must be some mechanism to retrieve which one is good. We will see that in detail in section 2.1.3.

Mirroring versus Replication

We can consider that mirroring is something different from *replication.* It provides only one duplicate of the data and there is a mechanism to copy one duplicate on the other one, if needed. In replication, there is no communication between the different copies, but their high number will always ensure that one is available and readable.

2.1.2 PRELIMINARY ASSUMPTIONS

Before describing the mirroring algorithm, we have to specify our failure assumptions i.e. in which conditions we can guarantee to the user that it will work. Of course, we can not assure him that his data would survive if the hard disk were mashed by the fall of the roof!

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

Failure Isolation

Our main assumption is that a crash while executing a write operation on a storage can only corrupt the contents of that particular storage, and not other data that might be stored on the same device.

No External Intervention

As we will see in the next section, the stable storage needs a log and two other storages to store information about its state. We suppose that none of these storages is changed, deleted, renamed or moved by an intervention external to the application using the mirrored storage.

Unbuffered Writes

Our algorithm is composed of *sequential* write operations to the different log and data locations. It is essential for the functioning of the mirroring algorithm that when beginning a new writing operation, the previous one was successfully completed. This assumption may not be met if the storage device uses some kind of buffering or caching system that does not really write physically on the device until the buffer is full. However, such devices often offer a `Flush` operation that forces the buffered data to be written to the storage. This operation must therefore be called after every write.

Reliable Reading and Writing

Twenty years ago, a *careful* write operation should have been implemented because of decays i.e. spontaneous events in which some set of pages, all within some one characteristic decay, change from *good* to *bad*. But these days, all is checked and integrated to the hardware and things that go wrong when writing on the disk are usually detected by it. For the read operation, likewise, we do not have to repeat n successive reads within a time T until there is a good answer. Errors are handled directly by the device. For the interested reader, more information on how to construct higher-level abstraction to handle those errors can be found in [4].

2.1.3 DESCRIPTION OF THE ALGORITHM

In this section, we will see how the mirroring algorithm works to convert any storage into an ideal device for recording state, called *stable storage*. This kind of storage guarantees the user to keep consistent data in presence of a crash failure, as shown in the introduction chapter.

Components

For the mirroring algorithm, we need three data locations as showed by figure 3. One is to keep the log information about the state of the mirrored storage: we will call it L. The two others are both copies of the data that we want to store: copy A and copy B. From now on, we will use the term *storage*. Each of the three storages can be of a different kind. Any non-volatile storage of the hierarchy presented in chapter 1 can be used. In theory, even a mirrored storage can be used as a component for itself, so that we obtain a sort of recursiveness, but in practice, it is not useful at all and reduces performance.

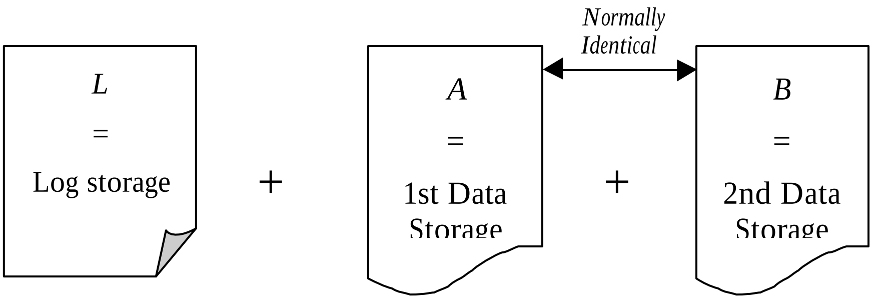

Figure 3 - Components for the Mirroring Algorithm.

Log Storage

In case of a crash failure, we need to know in what state the data files are. We record that information in the log. The log storage can take the following values:

L ::= Ok | Begin_ Writing_A | Writing_A_Finished |?

where ' ? ' means that *L* is *unreadable* (or its content is not comprehensible). The log can only be in the state ' ? ' if a crash occurred during the write of the log.

Mirrored Write Operation

To provide a stable, atomic write operation, the main idea is to write the copy B of the data *only* if we have finished to write the copy A. So we ensure that there is always at least one copy in a coherent state, either old or new. Before and after each write on data copies, we update the log storage L to specify at which stage the process is.

The main algorithm (*L* should be at *Ok* before beginning) is presented in figure 4:

```
1. Put L to Begin_Writing_A.
2. Write on data copy A.
3. Put L to Writing_A_Finished
4. Write on data copy B.
5. Put L to Ok
```

Figure 4 - Mirrored Write Operation.

We can illustrate this algorithm with a state automata (figure 5). For that purpose, we introduce a new notation for the state that the two data copies *A* and *B* can take:

A ::= P | S |?

B ::= P | S|?

where *P* (resp. *S)* means that the data on the copy are the *old* (resp. *new*) one, i.e. *before* (resp. *after*) the mirrored write operation.

' ? ' means that a copy is in an unknown state: it may be consistent or not, readable or not. It is important to note that the mirroring algorithm *never* checks the content of the copies itself. The user has to write his application in such a way that, after recovering from the crash failure, the data can be either in the old state or in the new one.

All proceedings are done with the information provided by the log storage.

In the state automata of the figure 5, each state is composed of three values: the states of *L, A* and *B*. For example *1/S/P* means that we have finished the 2nd stage (writing on *A)* and that we will begin the 3rd stage (updating *L*). After that, the following state will be *2/S/P*.

Each arrow between two states means writing on a single page (*L, A* or *B)* and they are numbered like the stages of the algorithm (figure 4). To simplify the log notation, we introduce the following codes: *Ok* = 0, *Begin_ Writing_A* = 1 and *Writing_A_Finished* = 2.

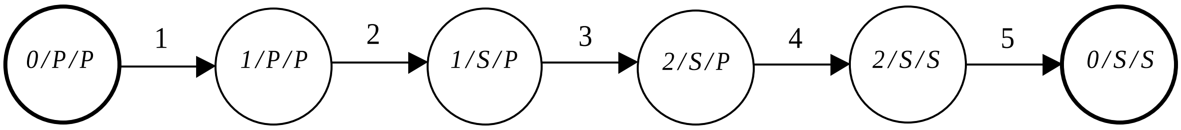

Figure 5 - State Automata Describing the Mirroring Algorithm.

Mirrored Read Operation

The mirrored read operation is just a normal read operation on any of the two copies, as both should normally be exactly identical after a write operation which has succeeded. In our implementation, we decide to always read the copy *A*

Cleanup Operation

Of course, when restarting the application after a crash failure, no I/O operation is allowed until we have checked the consistency of the mirrored storage. If some problem is detected, we have to recover to a consistent state. This is the purpose of a *cleanup* operation. We must check for some problems each time that we open the storage. It is based on the information previously written in the log storage *L*. If *L* contains *Ok* then we are sure the mirrored storage is consistent and there is nothing special to do. But, for example, if *L* is set to *Begin_Writing_A*, we know that there was a crash failure during the writing of *A* , which may be not consistent any more. According to our preliminary assumptions (see section 2.1.2), *A* is the only page affected by the crash, so that, to repair the mirrored storage, we just have to copy *B* on *A*. The figure 6 lists what there is to do for each different log information, when opening the mirrored storage.

State of *L*	Suspected Problem	Solution to repair the storage
Ok (0)	*Nothing*	*Nothing*
Begin_Writing_A (1)	*A* was not successfully written and may be not consistent.	1. Copy *B* on *A*. 2. Set *L* to *Ok*.
Writing_A_Finished (2)	*B* was not successfully written and may be not consistent.	1. Copy *A* on *B*. 2. Set *L* to *Ok*.
? or *unreadable*	If L was damaged during been written, neither A nor B is corrupted but they can be different	1. Copy *A* on *B* (or *B* on *A*). 2. Set *L* to *Ok*.

Figure 6 - Cleanup Operation Summary.

As shown in the fifth row, in the case of a corrupted or unreadable log file, there are two possible solutions: either we copy *A* on *B*, or we copy *B* on *A*. In fact, there is no possibility to know in which state are *A* and *B* i.e. with old data or new one. It depends at which stage the application crashed. The user should take care about that when designing his program. He can call the mirrored write again, if he wants to be sure the storage was updated. In our implementation, we choose to copy *A* on *B* in such a case.

We have to pay special attention to new crash failures happening during recovery of the previous one. It is obvious that the data copy that was intact can not be damaged, because we only read it. However two cases are possible:

- The data copy that was suspected to be corrupt can be corrupted again, if the new crash happens when copying the other one on it. But the log file keeps its state, so that, when restarting again, we just try another time the same cleanup operation.

- A new crash happens when resetting the log file *L* to *Ok*, after having repaired the suspected data copy. So, when restarting again, the rule described in the fifth row of the figure 4 can safely be applied.

2.1.4 STATE AUTOMATA DESCRIBING THE ALGORITHM

To be sure that our algorithm safely covers all cases, we can represent the whole mirroring algorithm (write and cleanup operation) with a *state automata* shown with its legend in figure 7.

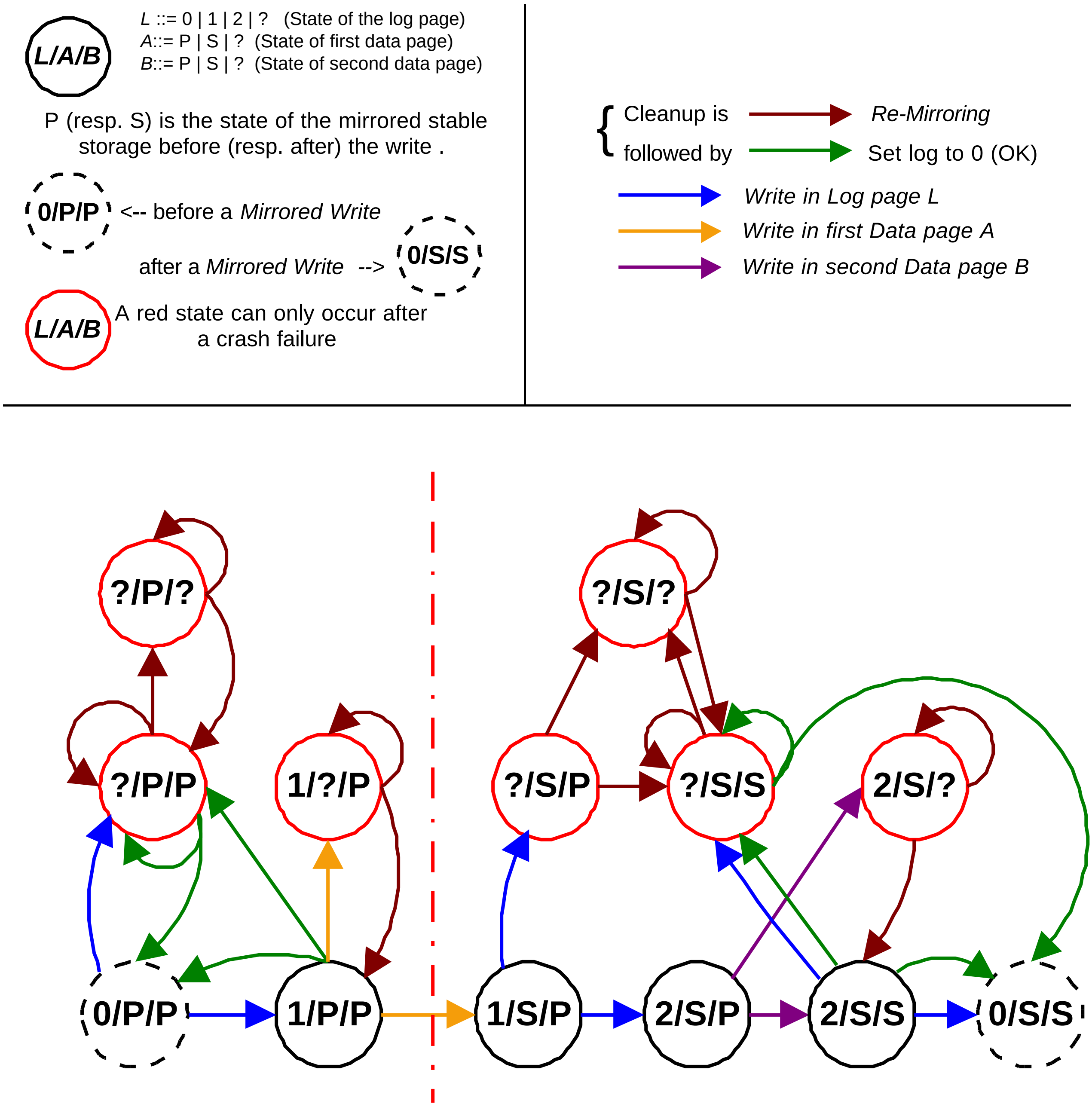
L/A/B
L ::= 0 | 1 | 2 | ? (State of the log page)
A::= P | S | ? (State of first data page)
B::= P | S | ? (State of second data page)
P (resp. S) is the state of the mirrored stable storage before (resp. after) the write .
0/P/P <-- before a Mirrored Write
after a Mirrored Write --> 0/S/S
L/A/B A red state can only occur after a crash failure
Cleanup is followed by
Re-Mirroring
Set log to 0 (OK)
Write in Log page L
Write in first Data page A
Write in second Data page B
?/P/?
?/P/P
1/?/P
0/P/P
1/P/P
?/S/?
?/S/P
?/S/S
2/S/?
1/S/P
2/S/P
2/S/S
0/S/S

Figure 7 - State Automata for the Mirroring Algorithm

We can distinguish *three vertical levels* in the automata :

1. Normal level: the lower line of states, represented as black circles, is the normal sequence of write operations. If there is no crash failure during the stable write operation, we stay at this level and the time taken by the operation is the time for two writes on a data file, completed by the time for three writes on a log file.

2. First crash level: the red states of the middle line are joined after a crash failure. According to the preliminary assumptions (see section 2.1.2), each state has only one storage corrupted, or presumed so. If there is a second (or more) crash failure during the recovery, we can loop or stay at this level until the cleanup succeeds.

3. Second crash level: the two red states of the highest line can only be reached if there is a second crash failure during recovery of a previous one. Note that they are two presumably corrupted storages: the log *L* and the data *B*.

As another important remark, there is clearly a separation, notified by a red dash line, between the left and the right part of the automata. When a crash happens on the left, the final state *after recovery* will be the one before the stable write operation, i.e. *0/P/P*. In the other case (the first data write on *A* is successful), the user can be sure that the stable write operation will be succeed; even if one or more crash failures occur after that, the final state will be *0/S/S*.

2.1.5 TIME NEEDED FOR THE MIRRORING WRITE OPERATION.

Using the automata of the figure 7 and assigning probabilities to some actions, we can estimate the time needed to complete a mirrored write operation in comparison with a normal one. However, this estimation is *purely fictive* as we will consider the whole automata as executing without any interruption. But, in fact, a crash failure stops the application and it is the user responsibility to restart it (maybe there is some hardware to repair, or a network connection to restore, etc…).

We suppose that the log *L* and the two data copies *A* and *B* are on the same location, for example the local disks. According to our algorithm, the execution time of a mirrored write operation *without* any crash failure, i.e. the best time, is equal to:

$$t_{ss} = 3.t_l + 2.t_d$$

where t_l is the time needed to write *L*, and t_d the time needed to write *A* or *B*, because the data copies can be much bigger than the log, which is always equal to a byte.

Transition Matrix

We can do the same distinction when assigning probabilities to crash events: p_l is the probability that there is a crash during the write of *L*, and p_d the equivalent for *A* or *B*. Thus, we can describe a chain for the transitions of the state automata of the figure 7. Note that this is not a Markov transition matrix since the sum of the probabilities on a line can be an integer different of 1. The transition matrix is shown in figure 8. Rows and columns represent the states of the automata. The resulting value at a given row and a given column is the probability to go from the row state to the column state.

	0/P/P	1/P/P	?/P/P	1/?/P	?/P/?	1/S/P	?/S/P	?/S/?	2/S/P	?/S/S	2/S/S	2/S/?	0/S/S
0/P/P	0	$1\text{-}p_l$	p_l	0	0	0	0	0	0	0	0	0	0
1/P/P	$1\text{-}p_l$	0	p_l	p_d	0	$1\text{-}p_d$	0	0	0	0	0	0	0
?/P/P	$1\text{-}p_l$	0	$p_l /1\text{-}p_d$	0	p_d	0	0	0	0	0	0	0	0
1/?/P	0	$1\text{-}p_d$	0	p_d	0	0	0	0	0	0	0	0	0
?/P/?	0	0	$1\text{-}p_d$	0	p_d	0	0	0	0	0	0	0	0
1/S/P	0	0	0	0	0	0	p_l	0	$1\text{-}p_l$	0	0	0	0
?/S/P	0	0	0	0	0	0	0	p_d	0	$1\text{-}p_d$	0	0	0
?/S/?	0	0	0	0	0	0	0	p_d	0	$1\text{-}p_d$	0	0	0
2/S/P	0	0	0	0	0	0	0	0	0	0	$1\text{-}p_d$	p_d	0
?/S/S	0	0	0	0	0	0	0	p_d	0	$p_l /1\text{-}p_d$	0	0	$1\text{-}p_l$
2/S/S	0	0	0	0	0	0	0	0	0	p_l	0	0	$1\text{-}p_l$
2/S/?	0	0	0	0	0	0	0	0	0	0	$1\text{-}p_d$	p_d	0
0/S/S	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8 - Transition Matrix for the Mirroring Automata

Probabilities for Final States

As said previously, we have two final states: either *0/P/P* (which was the unique starting state) or *0/S/S*. We saw that if the automata succeeds in passing through the red dash line (figure 8), we are sure to finish with *0/S/S*. But if there is a crash failure before this stage, we are sure to finish back into *0/P/P*. It is now obvious to say that the probability:

1. To finish in the state *0/S/S* is

$$P_{0/S/S} = (1\text{-}p_l) \,.\, (1\text{-}p_d)$$

because to cross the red dash line, the unique possibility is to go directly from *0/P/P* to *1/P/*P (with a probability $1\text{-}p_l$) and then directly from *1/P/P* to *1/S/P* (with a probability $1\text{-}p_d$).

2. To finish back in the state *0/P/P* is

$$P_{0/P/P} = 1 - P_{0/S/S}$$

$$\Leftrightarrow \quad P_{0/P/P} = 1 - (1 - p_l) \,.\, (1 - p_d)$$

$$\Leftrightarrow \quad P_{0/P/P} = 1 - (1 - p_l - p_d + p_l \,.p_d)$$

$$\Leftrightarrow \quad P_{0/P/P} = p_l + p_d - p_l \,.p_d$$

because as the mirroring algorithm ensure us to finish in one of the two final states, the sum of the two probabilities is *1*.

2.2 MIRRORING CLASS IN THE STORAGE HIERARCHY.

Using this mirroring technique, any non-volatile, non-stable storage can be transformed into stable storage. It is therefore possible to write an implementation of the mirroring algorithm that is independent of the actual non-volatile storage that will effectively be used to store the data. Each of the three components used i.e. the log L and the data copies A and B, can use a different non-volatile storage type, even a stable one. For example, A and B could be local files, and L a memory space on a remote process.

The non-volatile storage class, declares the common interface to all concrete storages. The mirroring class, uses this interface to make calls to a concrete storage implementation. This could be for instance a file storage class that implements storage based on the local file system. The structure of the collaboration is shown in figure 9:

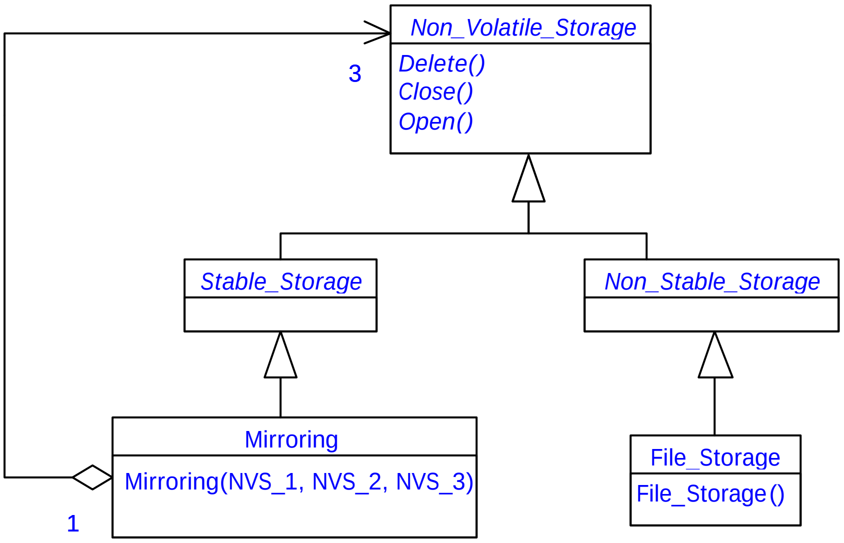

Figure 9 - Mirroring in the Storage Hierarchy

At instantiation time, three non-volatile storage objects (NVS_1, NVS_2, NVS_3) must be passed as a parameter to the constructor of the mirroring class. This is how a variety of mirrored storages can be created by reusing concrete implementations of non-volatile storage. What kind of non-volatile storage the application programmer will choose depends on the needs of the application. To help him make his choice, a concrete non-volatile storage must document the assumptions under which the storage is considered non-volatile and other information that might be useful for the application programmer such as for instance performance.

In this section, we will see how to implement the mirroring algorithm in Ada 95. Remember that our concrete mirroring class is a leaf of the storage hierarchy.

2.3.1 IMPLEMENTATION OF THE LOG STATE: A TRICKY CRASH FAILURE

In a first approach, and following our previous notation, we would have implemented the log state as an integer with the values 0/1/2 equivalent to *Ok/Begin_ Writing_A/Writing_A_Finished*. Any other value would make the log considered as corrupted. However, when thinking in detail about all problems not covered by the preliminary assumptions and that could infirm stability of the mirrored storage, we could consider this one:

Problem Statement

Suppose that during the third stage on the figure 5, when updating the log *L* from 1 (*Begin_Writing_A*) to 2 (*Writing_A_Finished*), a crash happens, which unfortunately set *L* to 0 (*Ok*) instead. This could happen if the value is streamed to the storage bit after bit (this is implementation defined). We will have the following scheme:

Situation	**8 bits**	**Value of L**
Before updating L	**1000 0000**	*L = 1*
Writing 2 in L, crash after updating the first bit.	**0000 0000**	*L = 0 (instead of 2)*

Figure 10 - A tricky case of crash failure

Of course, such a case has a so low probability that we could have neglected it, but for safety-critical software, this would probably be intolerable not to handle this known problem.

Consequence of the Situation

When restarting from the crash, the log file would be read as *Ok* and the mirrored would be considered as perfectly consistent, so that no recovery operation is launched. But in fact, the second data copy *B* was not updated, so that *A* and *B* are each coherent but not both identical. The storage is not stable any more. If reading the storage, this is not a problem, because in our implementation, we always read the copy *A*, which is in the new state. But if there is a crash during the first next write operation, this could affect directly *A*, and loose definitively the updated data. When recovering from this new crash, *B* will be copied on *A*, but *B* was not what it should have been…

Note that a similar failure when updating *L* from 0 (*Ok*) to 1 (*Begin_Writing_A*), or from 2 (*Writing_A_Finished*) to 0 (*Ok*) would not be so unsafe. In the first case, neither A nor B has been changed and both stay identical and coherent. Moreover, it avoids an unnecessary copy of *B* on *A*. In the second case, 0 is anyway the result that we wanted to reach.

Solution Proposed

A *radical* solution would be to systematically copy *A* on *B* when opening a mirrored storage with a log indicating a normal state, i.e. 0 (*Ok*). But not only it is illogical to proceed to a cleanup when the storage seems to be ok, moreover it is very performance-poor.

Another middle would be to use a fourth component: another log file *L'*, which has to be a copy of *L*. In this case, we consider the situation as good, only if *L and L'* are both set to 0 (*Ok*). Within our crash, *L'* would have stayed in the previous state 1 (*Begin_Writing_A*) and at recovery, the error could have been detected and corrected. But this solution introduces more complexity in the algorithm, more writing operations, and finally more failure opportunities.

Note that using a mirrored storage to implement the log file, sort of recursion, will not be a solution at all, as the problem itself would be the same for the stability of *L*! The problem is only reported at a deeper level. Moreover, as said in previous section, it is the user responsibility to choose a concrete storage type for each component, provided it is non volatile.

More than solving the problem after it has happened, our idea is *to prevent* it from happening. By choosing values of 1/2/4 *instead of* 0/1/2, we can avoid one of the states to become another one during a bit writing failure. This is shown in figure 11.

Value of *L*	**Bits**
L = 1 (*Ok*)	0000 000**1**
L = 2 (*Begin_Writing_A*)	0000 00**1**0
L = 4 (*Writing_A_Finished*)	0000 0**1**00

Figure 11 - Byte representation for values 1/2/4

With this choice, if there is a crash failure between writing two bits, either the value of *L* is unchanged or it will have more than one true bit, or it will have none at all. But we let the reader consider that it is impossible to get a crashed value with a single true bit, and *a fortiori* it is impossible that updating *L* from 2 (*Begin_Writing_A*) to 4 (*Writing_A_Finished*) can lead to *L* equals 1 (*Ok*) instead of.

2.3.2 COMPONENTS OF THE MIRRORING CLASS

Our 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 storages

private
   type Mirrored_Members_Type is new Limited_Controlled with record
      Params : Mirrored_Storage_Params_Type;
      MirrorFile1 : Non_Volatile_Storage_Ref;
      MirrorFile2 : Non_Volatile_Storage_Ref;
      LogFile : Non_Volatile_Storage_Ref;
   end record;
   --  Encapsulating the data of the stable storage in a controlled type
   --  gives us the possibility to close the stream's files when they
   --  cease to exist. This is very useful to avoid dangling open files
   --  in case of corrupted streams, where 'Read will raise exceptions.

   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;
```

First of all, look how we declare the three components (`MirrorFile1`, `MirrorFile2` and `LogFile`) as references to the non-volatile storage class-wide type (`Non_Volatile_Storage_Ref`). It means that our implementation does not rely on any specific type of non-volatile storage. Encapsulating the data of a mirrored storage in a controlled type gives us the possibility to deallocate the three internal storages in the `Finalize` procedure. This is done in the body part :

```
procedure Free is new Ada.Unchecked_Deallocation
     (Non_Volatile_Storage_Type'Class, Non_Volatile_Storage_Ref);

procedure Finalize (Members : in out Mirrored_Members_Type) is
begin
   Free (Members.MirrorFile1);
   Free (Members.MirrorFile2);
   Free (Members.LogFile);
end Finalize;
```

2.3.3 VERIFYING THE LOG WHEN OPENING THE STORAGE.

When opening the mirrored storage described by its parameters, we begin by checking the log file.

```
procedure Verify (Storage : in out Mirrored_Storage_Ref)
   is
      Logmessage : Stream_Element_Array (1 .. 2);
      Last : Stream_Element_Offset := 0;
   begin
      begin
         Storage.Members.LogFile :=
         Open_Storage (Get_Log_Params (Storage.Members.Params),
                       In_Mode);
         --  Try to read log location.
         Read (Storage.Members.LogFile.all, Logmessage, Last);
         if Last /= 1 then
            raise Data_Not_Available;
         end if;

         case Logmessage (1) is
            when 1 =>
               --  Everything is OK
               return;
            when 2 =>
               --  Crash during first write (in data1)
               --  Cleanup based upon data2.
               Cleanup (Storage.all, 2);
            when 4 =>
               --  Crash during second write (in data2)
               --  Cleanup based upon data1.
               Cleanup (Storage.all, 1);
            when others =>  --  this should not happen!
               raise Data_Not_Available;
         end case;

      exception
         when Data_Not_Available =>
            --  If log is unreadable, then we choose to copy
            --  the first data storage on the second one.
            Cleanup (Storage.all, 1);
      end;
      --  At this point, Cleanup seems to have succeeded,
      --  open the log again, but in Out_mode, and write OK into it.
      Close (Storage.Members.LogFile.all);
      Open (Storage.Members.LogFile.all, Out_Mode);
      Logmessage (1) := 1;
      Write (Storage.Members.LogFile.all, Logmessage (1 .. 1));

   end Verify;
```

We see how the procedure is written in a simple and natural way. It uses the common operations of the non-volatile storage class: `Read`, `Write`, `Open` and `Close`, without caring about the concrete type of the log and the data storages. The `Cleanup` procedure takes as second parameter either the value 1 or 2, indicating which data storage has to be copied on the other one, and executes the copy using again the `Read` and `Write` operations of the storages.

2.3.4 EXAMPLE OF USE: SAVING AN OBJECT AS PERSISTENT.

To create and access a persistent object on a mirrored storage, we do not have to redefine the persistent object package described in section 1.3. We only need to create mirrored parameters with the appropriated `String_To_Storage_Params` function.

The following lines of code illustrate how an instance of a persistent integer type can be saved to a mirrored storage :

```
with Storage_Params.Non_Volatile.Stable.Mirrored_Storage_Params;
use Storage_Params.Non_Volatile.Stable.Mirrored_Storage_Params;
with Storage_Params.Non_Volatile.Non_Stable.File_Storage_Params;
use Storage_Params.Non_Volatile.Non_Stable.File_Storage_Params;

with Persistent_Integer;

declare
   S: Persistent_Integer.Persistent_Ref;
   P: Mirrored_Storage_Params_Type :=
         String_To_Storage_Params ("mirrored_storage_name",
               File_Storage_Params_Type'
                  (String_To_Storage_Params ("my_name1.data")),
               File_Storage_Params_Type'
                  (String_To_Storage_Params ("my_name2.data")),
               File_Storage_Params_Type'
```

```
                     (String_To_Storage_Params ("my_name.log")));
   begin
      S := Persistent_Integer.Create_Object (P);
      S.Value := 12345;
      Persistent_Integer.Save_Object (S.all);
   end Save_Int;
```

In this example, we create a mirrored storage based on two files as the data storages. The parameters of these two file storages are created by calling the `String_To_Storage_Params` function of the file storage type. We use a third file to store the log. We could have used any other non-volatile storage like, for example, the remote storage type that we will be presented in the third chapter of this report.

After launching the application, the current local directory contains the following files:

```
-rw-r--r--   1 xcaron   students       1 Feb 21 18:31 my_name.log
-rw-r--r--   1 xcaron   students      55 Feb 21 18:31 my_name1.data
-rw-r--r--   1 xcaron   students      55 Feb 21 18:31 my_name2.data
```

We clearly see the log file, named `my_name.log` (in the default case, the extension `.log`, `1.data` and `2.data` are added to the storage name) which has a size of 1 byte, the normal size for a log. The integer value has been streamed in the file storages, corresponding to the identical local files named `my_name1.data` and `my_name2.data`, with the same size of 55 bytes.

The Distributed Systems Annex of the Ada Reference Manual provides a standardized way of programming distributed systems in Ada95. Using this Annex and a compiler that supports it, we can easily program a non-volatile storage that can be located on a remote machine. The global architecture consists of three *processes:* the process using the storage, a server that will be used to locate remote storages and a process running on the remote machine where the physical storage resides. All three processes communicate using the features described in the Distributed Systems Annex of the Ada 95 Reference Manual [1].

3.1 ADA DISTRIBUTED SYSTEMS ANNEX & GLADE

In this section, we will first present a brief overview of the DSA and the implementation of the DSA for the GNAT compiler that we are going to use. Next, we will present the design of the storage.

3.1.1 FEATURES OF THE DSA

The Annex E of the Ada 95 RM defines how a program can be *split* up into a number of *partitions* and how the partitions can *communicate*.

Active and Passive Partitions

Partitions may be *active or passive*. An active partition has its own environment task and copy of the run-time system: it is like a traditional complete program, whereas a passive partition has no threads of control and thus may only include certain kinds of library units; it has no main subprogram and no library level elaboration.

The general idea is that the partitions execute independently other than when communicating. However, unlike a set of quite distinct programs, strong typing is imposed rigorously betweens the partitions. An active partition need not have a main subprogram since its activity could be entirely within library package elaboration.

Pragmas Hierarchy

The Annex defines a hierarchy of four pragmas to categorize the library units:

```
pragma Pure ( … );
pragma Shared_Passive ( … );
pragma Remote_Types ( … );
pragma Remote_Call_Interface ( … );
```

Such a pragma imposes restrictions on the content of the unit where it is placed. **A unit can only depend on (via with clauses) units in the same or higher categories.**

The restrictions for each category are as follows:

- A *pure* unit cannot contain any state; since it has no state a distinct copy can be placed in each partition. This is important since it reduces the need for communication between partitions.

- A *shared passive* unit can have visible state but no tasks or protected objects with entries. It must be preelaborable. There are also restrictions on access types. The general idea is to avoid references to active units. It requires no run-time system.

- A *remote types* unit defines types used for communication between partitions. Its specification must be preelaborable but its body need not. The only visible access types permitted are so-called remote types; these are access to subprogram or access to class wide limited private types.

- A *remote call interface* (RCI) unit cannot have visible state; its main purpose is to define the subprograms to be called remotely from other partitions. Its specification also has to be preelaborable but its body need not.

Only pure and shared passive units can be contained in a passive partition; that way it does not need a copy of the run-time system.

Communication between Partitions

Communication via a *passive* partition is normally through the use of protected objects declared in shared passive units in the partition. But note that an object in a passive partition can be read directly.

Communication between *active* partitions is via remote procedure calls on RCI units. *Stubs* at each end of the communication process such remote calls; parameters and results are passed inside streams. This is all done automatically by the partition communication subsystem (PCS) and need not concern the user.

Library units categorized with the pragma `Remote_Call_Interface` declare subprograms that can be called and executed remotely. A subprogram call that invokes one such subprogram is a classical RPC (Remote Procedure Call) operation; it is a *statically bound* operation, because the compiler can determine the identity of the subprogram being called.

Dynamically bound calls are provided through two mechanisms:

- The dereference of an access-to-subprogram value, i.e. a value whose type is a remote-access-to-subprogram (RAS). We will not use this feature.

- A dispatching call whose controlling argument is an access-to-class-wide operand (remote access on class wide types – RACW). These remote access types can be declared in a RCI package as well.

A remote access type can be viewed as a fat pointer, that is to say a structure with a remote address and a local address (like an URL: `<protocol>://<remote-machine>/<local-directory>`). The remote address must denote the host of the partition on which the entity has been created; the local address describes the local memory address within the host.

Small Example of Remote Call

Thus, suppose you want the Mars Polar Lander spacecraft to return to Earth, you would define a remote call interface package such as:

```
package Mars_Polar_Lander is
   pragma Remote_Call_Interface;
   procedure Come_Back_Please;
end;
```

and configure it on the spacecraft partition. You can call the remote procedure:

```
with Mars_Polar_Lander;
package Nasa_Head_Quarter is
   …
   Mars_Polar_Lander.Come_Back_Please;
   …
end;
```

The call `Come_Back_Please` is executed remotely.

Figure 12 illustrates this distributed application. Partition_Nasa and Partition_SpaceCraft are active partitions. The first one contains the package `Mars_Polar_Lander`. The second one contains the package `Nasa_Head_Quarter`. Both communicate through the network.

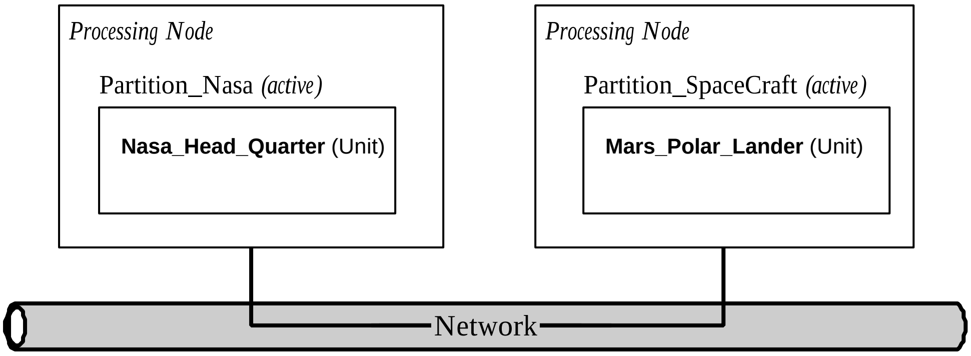

Figure 12 - Example of Distributed Ada95 Application

Note that the means of dividing the system up into partitions is not defined by the language and is done by some appropriate post-compilation tools. For this purpose, we use the ***`gnatdist`*** tool and its configuration language. ***`gnatdist`*** is part of GLADE [8] (GNAT Library for Ada Distributed Environment), that is briefly presented in next section.

3.1.2 DEVELOPMENT OF DISTRIBUTED SYSTEMS WITH GLADE

An important feature of the Distributed Systems Annex (DSA) of Ada is that the user can develop his application the same way whether this application is going to be executed as several programs on a distributed system, or as a single program on a non-distributed system. The design of the DSA is so that the source changes needed to convert an ordinary non-distributed program into a distributed program are minimal. The user does not need any specialised tool to develop his distributed application; his usual software-engineering environment and his usual debugger are enough. Note that a non-distributed program is not to be confused with a distributed application composed of only one program. The later is built by means of the configuration tool and includes the communication library.

With this approach, once the non-distributed version of the program is complete, it has to be configured into separate partitions. This step is simple, compared to that of developing the application itself. The configuration step consists in mapping sets of compilation units into individual partitions, and specifying the mapping between partitions and *nodes* in the computer network. This mapping is specified and managed by means of *GLADE*.

The gnatdist Tool

The tool `gnatdist` and its configuration language allow the user to partition his program and to specify the machines on which the individual partitions are to execute. `gnatdist` reads a configuration file (whose syntax is *Ada-like*) and builds several executables, one for each partition. It also takes care of launching the different partitions (default) with parameters that can be specific to each partition.

How to Configure a Distributed Application

For a beginner user, we could resume the process of configuring a distributed application with GLADE in four points:

1. Writing of a non-distributed Ada application, making changes and using pragmas to specify the packages that can be called remotely.

2. When this non-distributed application is working, writing of a configuration file that maps the user categorized packages onto specific partitions. This concerns particularly remote call interface and remote types packages. Specify the main procedure of the distributed application (responsible for starting the entire distribution).

3. Type the command line '`gnatdist <configuration-file>`'.

4. Start the distributed application by invoking the start-up shell script or default Ada program (depending on some `Starter` option).

Developing a non-distributed application in order to distribute it later is the natural approach for a novice. Of course, it is not always possible to write a distributed application as a non-distributed application. For instance, a client/server application, like our remote storage class, does not belong to this category because several instances of the client can be active at the same time. In the next section, we now present in details the design of our remote storage.

3.2 THE REMOTE STORAGE CLASS IN THE STORAGE HIERARCHY

A remote storage consists of any storage located on a remote machine. It can also be on the same machine as the application using the storage, provided it is mapped to another partition.

What is important is that if the main application stops after a crash failure, the partition containing the storage will not be affected. Thus, memory storage, classified in the storage hierarchy as a volatile storage, transforms into non-volatile storage when used in a remote partition.

A Generic Class

The remote storage can be implemented in a *generic* way using the *Strategy* pattern [7]. During instantiation, a concrete storage will be passed as a parameter. This parameter will determine what kind of storage will be used on the remote machine. The resulting storage will be non-volatile, since it is located on a different machine. It can also be stable, if a stable storage is used to store the data on a remote machine

In the figure 13, we choose to derive the remote storage class from the non-stable, non-volatile class. Figure 14 shows the remote class derived from the stable, non-volatile class.

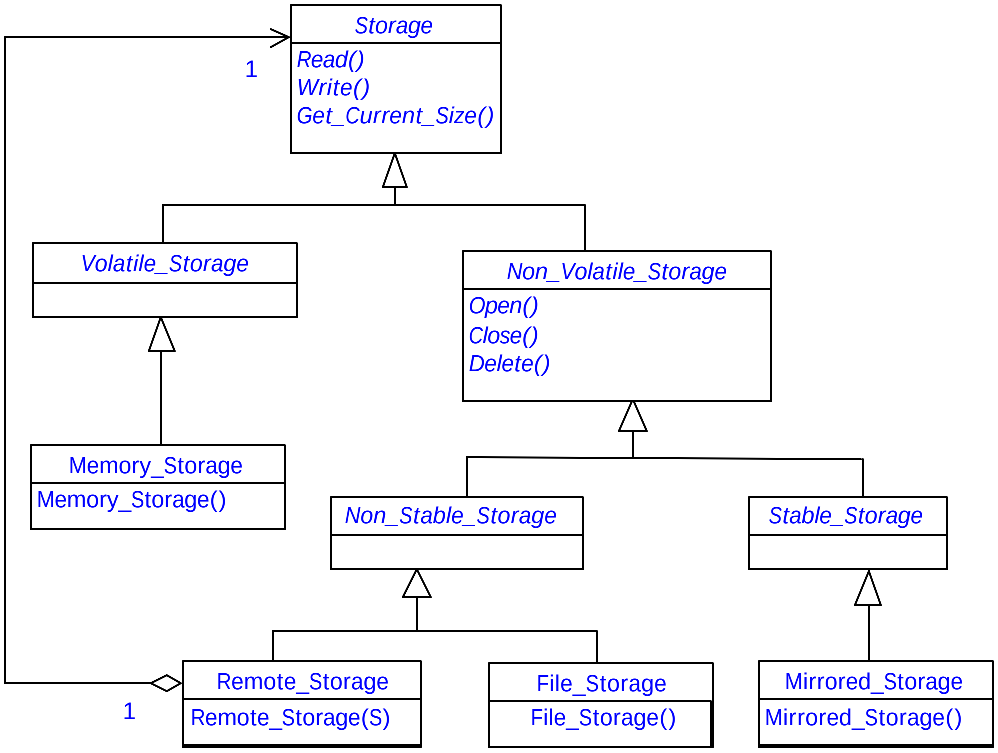

Figure 13 - Remote Storage as Non-Stable Storage in the Hierarchy

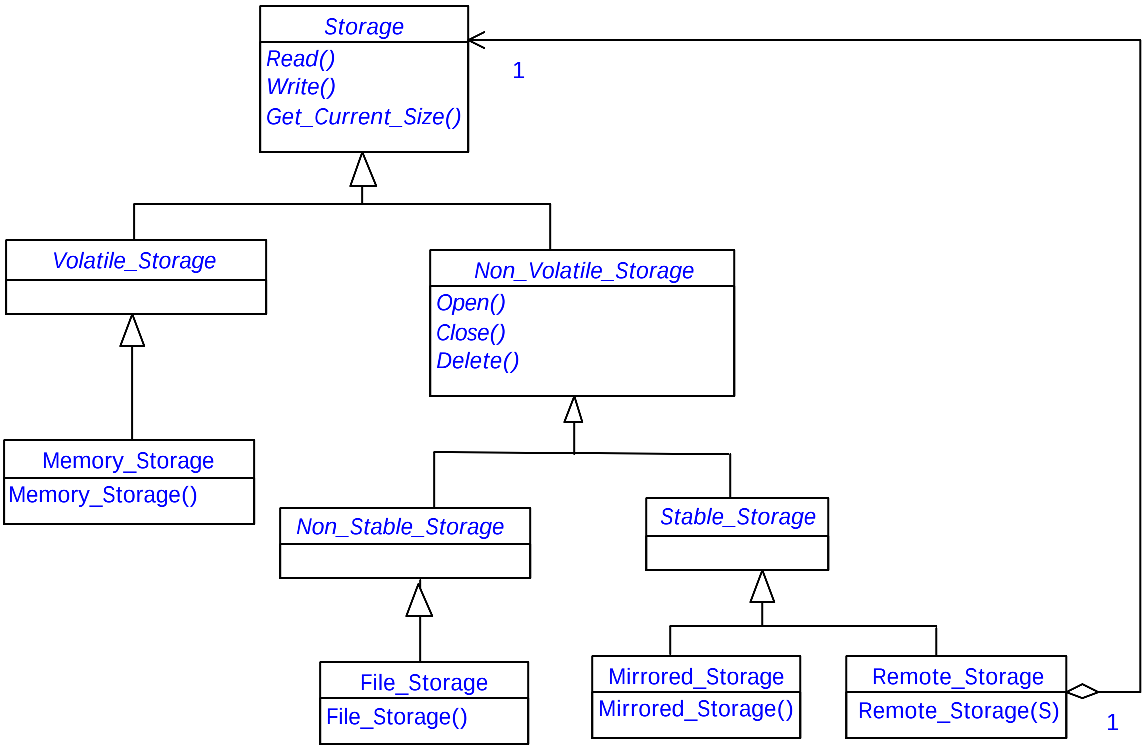

Figure 14 - Remote Storage as Stable Storage in the Hierarchy

The same principle applies to the parameters hierarchy.

3.3 DESIGN OF THE REMOTE STORAGE CLASS

3.3.1 GENERAL SURVEY OF THE DISTRIBUTED SYSTEM

The remote storage class is implemented as a distributed system consisting of three parts:

- The unit mapped on the remote machine, that we will now call *remote relay* since it is the relay between the user partition and the physical storage.

- The *registration server* which will be called by the remote relay to register itself during start-up.

- The *user partition* which asks the registration server for the reference of a registered remote relay and transmits, through the network, operation order, like ‘Open’ or ‘Write’, and so on…

Figure 15 illustrates the interactions between the three partitions:

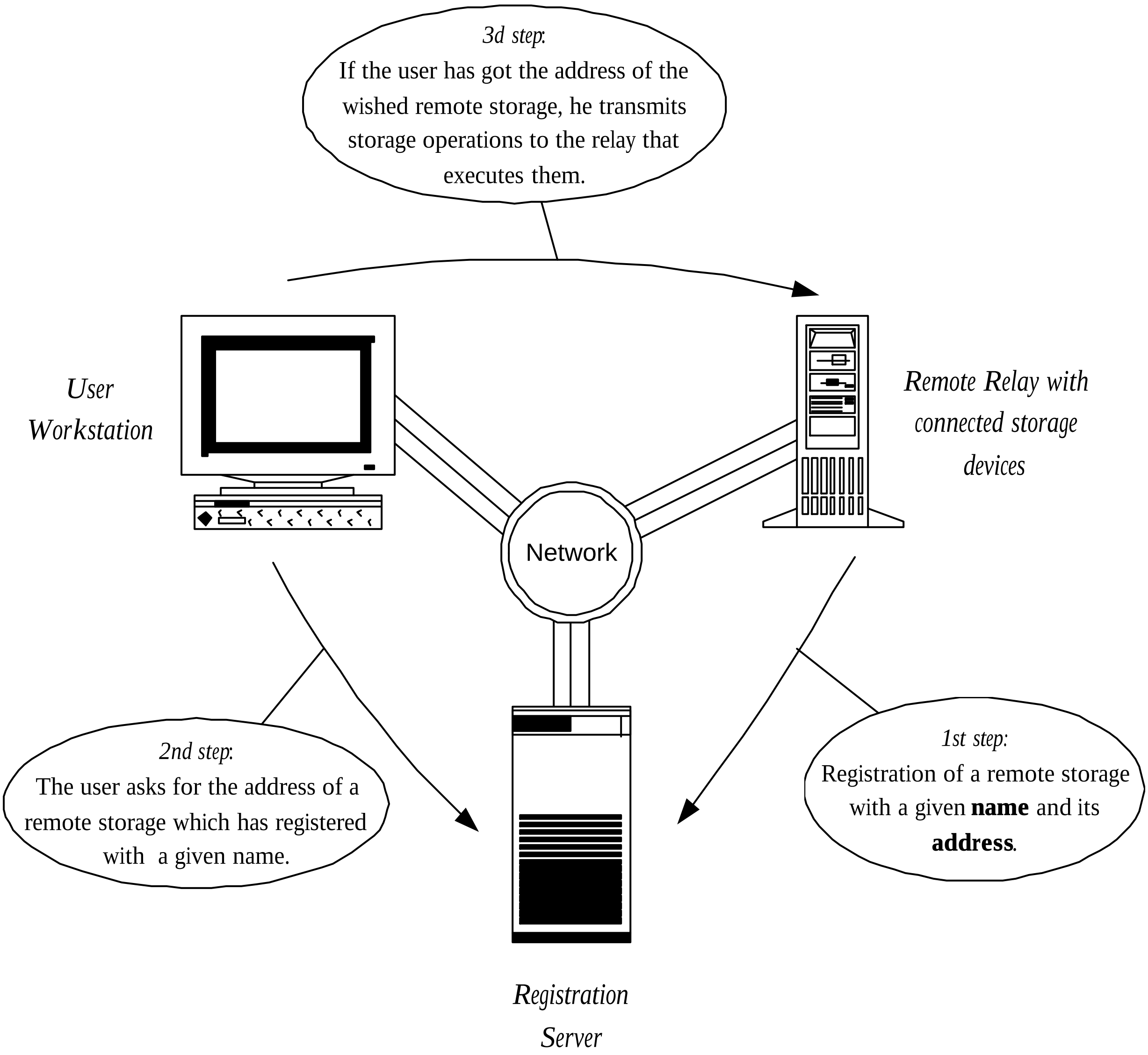
3d step:
If the user has got the address of the wished remote storage, he transmits storage operations to the relay that executes them.
User Workstation
Network
Remote Relay with connected storage devices
2nd step:
The user asks for the address of a remote storage which has registered with a given name.
1st step:
Registration of a remote storage with a given **name** and its **address**.
Registration Server

Figure 15 - Overall Design of the Remote Storage Class

In the next sections, we will see in details the design of each component.

3.3.2 DESIGN OF A REMOTE RELAY

The role of the relay located on the remote machine is to receive operations orders from the user and to transmit these orders to the local storage This storage can be of any kind: volatile memory, local file, mirrored storage composed itself of other storages, and so on…

Registration of the Relay

The first problem to solve is the communication between the user and the relay, since the user cannot immediately know where a relay is located. Therefore, when the relay starts, it has to register itself at the server, waiting on a channel of the network. At registration time, the relay gives two informations to the server:

1. The *name* of the storage (mandatory). The user chooses this name when launching the relay partition on the command line. Note that, at this point, the storage is not created, for the relay does not know what kind of storage to create. The first command that the relay will receive later from the user should be a storage creation order.

2. Its *address*, i.e. a reference to the location where the relay resides. It can be viewed as a fat pointer, that is to say a structure with a remote address and a local address (like an URL: `<protocol>://<remote-machine>/<local-directory>`).

Receiving and Transmitting Storage Operations Commands

Once the relay is registered to the server, a user can ask for its address and send to it, through the network, commands to execute on the storage, like writing, closing, and so on. The first order to be received is the creation or the opening of the storage. As usual, the storage is identified using storage parameters.

At this point, the relay simply acts as a call-through to the storage operations. This is just the reason why we called it a *relay*.

3.3.3 DESIGN OF THE REGISTRATION SERVER

Unlike the remote relay, only one server can exist in our distributed system. The server provides two services:

1. A *registration service* for all remote relays. The restriction is that a relay cannot be registered with a name that is still in use by another relay, so that storage can be clearly identified by the user, without any doubt. All relays are registered, with their name and address, in a list internal to the server.

2. A *search service* for the user who wants to reach a relay with a given storage name. If the searched relay was not registered, the request is blocked until a relay registers itself with the right name. Thus, each time a relay registers itself, the server has to check all the requests until one or none is satisfied. When a search succeeds, the server returns to the user the address of the remote relay. This address is permanent.

To achieve this goal, the server must have the quality to be called remotely from the user partition as well as from the relay partitions. We will see later in section 3.4 which implementation solution is offered by Ada95.

3.3.4 DESIGN OF THE USER PARTITION

The unit that is offered to the user to access a remote storage does not need any complicated design. Each time the user wants to *create* or *open* a storage:

1. He creates parameters for this storage based on the name of the storage.

2. Then, he calls the appropriate procedure for creation or opening with the parameters. This procedure is in charge to ask the server for the address of the remote relay where the storage relies. It waits until an answer is returned.

3. The returned address is permanent. It can therefore be stored locally and used in subsequent operations like *writing*, *reading*, *closing*, *opening*, *deleting*, and so on.

This user unit must provide access to all possible operations on a *non-volatile* storage.

3.4 IMPLEMENTATION OF THE REMOTE STORAGE CLASS

Based on the design described in the previous section, the implementation in Ada95 of our distributed system must take care of how each library unit will be mapped on a partition and how it interacts with the others. Figure 16 illustrates the three processing nodes. Each node contains an active partition and each partition contains the library units that were mapped on it. The purpose on this section is to present the relevant implementation details of each library unit.

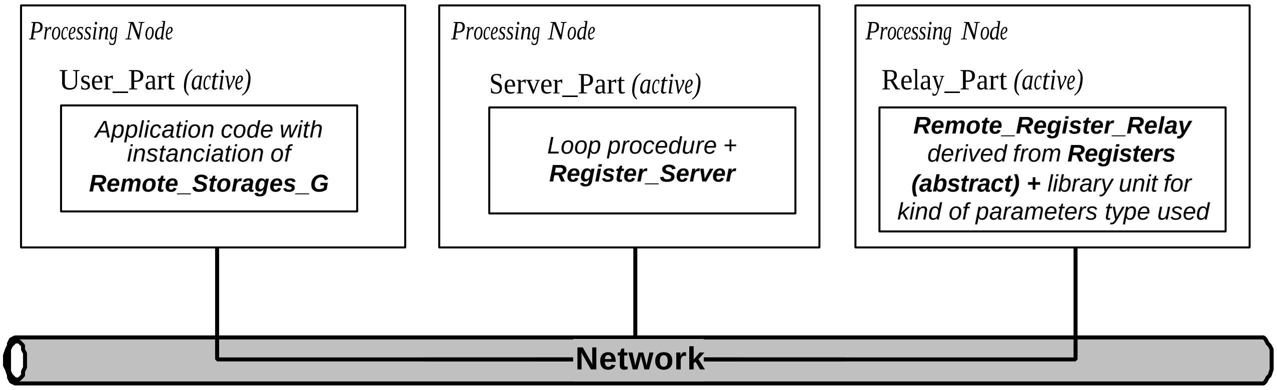
Processing Node
User_Part (active)
Application code with instanciation of Remote_Storages_G
Processing Node
Server_Part (active)
Loop procedure + Register_Server
Processing Node
Relay_Part (active)
Remote_Register_Relay derived from Registers (abstract) + library unit for kind of parameters type used
Network

Figure 16 - Architecture of the Remote Distributed System

3.4.1 REGISTERS: AN ABSTRACT UNIT CATEGORIZED WITH REMOTE_TYPES

We will use remote access to class-wide type (RACW) to communicate with the relay partition. Therefore, we create a new abstract tagged limited type called `Register_Type`, in a unit categorized with the pragma ***`Remote_Types.`*** Library units categorized with this pragma can define distributed objects and remote methods on them.

```
with Ada.Streams; use Ada.Streams;
with Universal_Storage_Params; use Universal_Storage_Params;

package Registers is
   pragma Remote_Types;
```

```
      type Register_Type is abstract tagged limited private;

      procedure Remote_Read (Register : in Register_Type;
                             Item     : out Stream_Element_Array;
                             Last     : out Stream_Element_Offset)
                is abstract;

      procedure Remote_Create (Register : in out Register_Type;
                               Params : in
                                    Universal_Storage_Params_Type'Class)
                is abstract;

      procedure Remote_Open (Register : in out Register_Type;
                             Mode     : in Remote_Mode_Type) is abstract;

      …

      Non_Volatile_Reserved_Action : exception;

   private
      type Register_Type is abstract tagged limited null record;
   end Registers;
```

In the code above, `Register_Type` is the root type of a distributed relays hierarchy, for executing storage operations (creation, writing, and so on) on a remote processing node. In the next sections we will see how to derive and extend `Register_Type`, how to create a distributed object and how to use a reference to it.

All the operations on non-volatile storage must have a corresponding operation in this unit. However, note that a subprogram defined in a `RT` unit is not a remote subprogram.

A very interesting feature for *replication* is that a `RT` unit can be *duplicated on several partitions* in which case all entities are different from each other.

3.4.2 DERIVATION OF REGISTER_TYPE AND CREATION OF REMOTE RELAYS

To implement the interface defined in the `Registers` package, we have to derive from `Register_Type`. Such a derivation is done in our package `Remote_Register_Relay`.

```
   with Ada.Streams; use Ada.Streams;
   with Storages; use Storages;
   with Registers; use Registers;
   with Universal_Storage_Params; use Universal_Storage_Params;

   package Remote_Register_Relay is

      type Remote_Register_Type is new Register_Type with private;

      procedure Remote_Read (Register : in Remote_Register_Type;
                             Item     : out Stream_Element_Array;
                             Last     : out Stream_Element_Offset);

      procedure Remote_Create (Register : in out Remote_Register_Type;
                               Params : in
                                 Universal_Storage_Params_Type'Class);
      procedure Remote_Open (Register : in out Remote_Register_Type;
                             Mode     : in Remote_Mode_Type);

      …

   private

      type Remote_Register_Type is new Register_Type with record
         Storage : Storage_Ref;
      end record;

   end Remote_Register_Relay;
```

The new type `Remote_Register_Type` stores a reference to the physical storage created on the remote machine where the relay partition resides.

Let us examine the body of the package. A remote operation is implemented as a call-through to the operation of the concrete storage pointed to by the reference. For example, to read the storage, we simply have:

```
   procedure Remote_Read (Register : in Remote_Register_Type;
                          Item     : out Stream_Element_Array;
                          Last     : out Stream_Element_Offset)
      is
      begin
         Read (Register.Storage.all, Item, Last);
      end Remote_Read;
```

Any object of type `Remote_Register_Type` becomes a distributed object and any reference to such an object becomes a fat pointer or a reference to a distributed object. A type for such a reference on `Remote_Register_Type` is declared in the registration server, presented in the section 3.4.3

3.4.3 REGISTER_SERVER: A UNIT CATEGORIZED WITH REMOTE_CALL_INTERFACE

The RCI unit `Register_Server` declares subprograms that can be called and executed remotely. Its interface is as follows:

```
   with Registers; use Registers;

   package Register_Server is

      pragma Remote_Call_Interface;

      type Register_Ref is access all Register_Type'Class;

      procedure Register (My_Register : Register_Ref; Name : String);

      function Find_Register (Name : in String)
         return Register_Ref;

   end Register_Server;
```

Register_Ref, a remote access to class-wide type (RACW), is defined. Register_Ref becomes a reference to a distributed object.

The function Find_Register allows another partition to asks for a RACW to a registered relay which a given name. The remote procedure Register allows the registration of a remote relay at its elaboration. That is what we will see now.

Registration of a Remote Relay at the Server

At elaboration of the unit, the partition with the package Remote_Register_Relay has to register itself at the registration server with a name given by the user as an argument in the command line. The procedure Register of the server is called remotely with a RACW and the name string as arguments. Here is the body of the package:

```
package body Remote_Register_Relay is

   …   --  Operations implementation

   My_Remote_Register : aliased Remote_Register_Type;
   Error : Boolean := True;

begin

   for I in 1 .. Argument_Count loop
      if Argument (I) = "--name" then
         if I = Argument_Count then
            exit;
         else
            Register (My_Remote_Register'Access, Argument (I + 1));
            Error := False;
         end if;
      end if;
   end loop;

   if Error then
      Put_Line ("Please specify the name of the storage using --name
                                                          storage_name");
   end if;

   exception
      when Storage_Existant =>
         Put_Line ("Storage existant");
end Remote_Register_Relay;
```

The method of registration is to declare an aliased Remote_Register_Type and to pass a pointer to it as argument when calling the remote procedure Register (we use the attribute 'Access).

The arguments passed by the user in the command line are analyzed to retrieve the name of the storage. If a previous partition has registered itself with the same name, an exception is raised, indicating that this remote storage still exists for the server. This is to avoid confusion.

Use of a Protected Object inside the Registration Server

The server must be task safe, since there might be *simultaneous* registration/find requests. This is why the body of the server package contains a protected object, containing the linked list of all registered partitions.

```
package body Register_Server is

   …  -- Declaration of a linked list type

   protected Protected_List is
         procedure Register (My_Register : Register_Ref; Name : String);

         entry Find_Register (Name : in String;
                              My_Register : out Register_Ref);
      private
         List_Head : Node_Ref := null;
         To_Try : Natural := 0;

         entry Find_Register_Internal (Name : in String;
                                       My_Register : out Register_Ref);
   end Protected_List;

   …  -- Protected body

   function Find_Register (Name : in String) return Register_Ref is
      Result : Register_Ref;
   begin
      Protected_List.Find_Register (Name, Result);
      return Result;
   end Find_Register;

end Register_Server;
```

The procedure `Register` is not an *entry*, because a remote relay can always attempt to register itself, providing the protected object is quiescent.

On the other hand, the function `Find_Register` becomes an *entry* of the protected object, with a *true* barrier. It means that this entry can always be crossed by a task. Then, a relay with the given name is searched in the linked list. If there is one, then its RACW is returned as result, else the task is re-queued to the private *entry* `Find_Register_Internal`. The barrier of this entry can be crossed over when a new relay registered itself. In this case, *all the tasks* waiting are re-queued back to the `Find_Register` entry to search again for the relay with the same name. We can know the number of tasks waiting at the barrier of `Find_Register_Internal` by using the attribute `'Count`

3.4.4 THE GENERIC PACKAGE REMOTE_STORAGES_G

As explained in the design section, the remote storage class will be non-volatile. It can be stable or non-stable, depending on what actual storage is used on the remote machine. This is why the storage has been implemented as a generic package as shown below:

```
with Registers; use Registers;
with Register_Server; use Register_Server;
with …

generic

   type Storage is abstract new Non_Volatile_Storage_Type with private;
   type Params is abstract new Non_Volatile_Params_Type with private;

package Remote_Storages_G is

   type Remote_Storage_Type is new Storage with private;
   type Remote_Storage_Ref is access all Remote_Storage_Type'Class;

   … --  Usual storage operations declaration (Read, Write, Open…)

   type Remote_Params_Type is new Params with private;

   … --  Usual parameters operations declaration
     --  (String_To_Storage_Params, Create_Storage, …)




private

   type Remote_Storage_Type is new Storage with record
      Register : Register_Ref;
   end record;

   ----------------------------------------------------

   subtype Index_Type is Natural range 0 .. 255;

   type Remote_Storage_Name_Type (N : Index_Type := 0) is record
      Name : String (1 .. N);
   end record;

   type Remote_Params_Members_Type is new Controlled with record
      True_Params : Storage_Params_Ref;
   end record;

   procedure My_Write (Stream :
                           access Ada.Streams.Root_Stream_Type'Class;
                      Item : in Remote_Params_Members_Type);
   for Remote_Params_Members_Type'Write use My_Write;

   procedure My_Read (Stream :
                           access Ada.Streams.Root_Stream_Type'Class;
                      Item : out Remote_Params_Members_Type);
   for Remote_Params_Members_Type'Read use My_Read;

   procedure Finalize (P : in out Remote_Params_Members_Type);

   procedure Adjust (P : in out Remote_Params_Members_Type);

   type Remote_Params_Type is new Params with record
      Remote_Storage_Name : Remote_Storage_Name_Type;
      Members : Remote_Params_Members_Type;
   end record;

end Remote_Storages_G;
```

In the public part, we declare the remote storage type and the remote parameters type.

In the private part, it is interesting to see that we store a `Register_Ref`, a remote access to class-wide type (RACW) to the relay object on the remote partition. This reference is obtained by contacting the registration server, during a call to the `Create_Storage` or the `Open_Storage` operation. Using it, it is now possible to transmit to the remote storage what operations are to be executed.

It is more complicated for the parameters, since they need to be passed through the network and they contain a *reference* to a storage parameters object. Thus, we declare a new *controlled* type called `Remote_Params_Members_Type`, and provide our own `Read` and `Write` operations. All non-limited types have default implementation of the stream attributes, but it is possible to replace it for any type via an attribute definition clause (`for Any_Type'Write use Another_Write;`). In order to write a value of a limited type to a stream, such an attribute definition clause is even mandatory. Any procedure having one of the predefined signature shown in [1, 13.13.2] can replace the predefined implementation. In addition we provide finalization and adjustment. It is important to note that the type `Remote_Params_Members_Type` encapsulates any storage parameters (and not only non-volatile parameters). As said previously, the storage declared by these parameters is transformed into a non-volatile one by the fact it is located on a remote machine. The `Remote_Params_Type` is a record composed of this controlled type and of an other type, not controlled, which encapsulates the name of the remote storage, given by the user when calling the procedure `String_To_Storage_Params`.

3.5 A COMPLETE USE EXAMPLE

Instantiation of the Generic Package ***`Remote_Storages_G`***

When a programmer wants to use the remote storage class, he has first to create an instance of the generic package `Remote_Storages_G`. As explained in the section 3.2, the remote storage class can derive of any non-volatile storage, depending on which type the user specified at the instantiation of the `Remote_Storages_G` package. For instance, we could declare our remote storage as a stable storage like that:

```
with Remote_Storages_G;
with Storages.Non_Volatile.Stable;
use Storages.Non_Volatile.Stable;
with Storage_Params.Non_Volatile.Stable;
use Storage_Params.Non_Volatile.Stable;

package Remote_Stable_Storages is new Remote_Storages_G
                      (Stable_Storage_Type,Stable_Params_Type);
```

Thus, this remote storage inherits all the properties of a stable storage. This is the situation illustrated by the figure 14. Note that both the type of the storage and the type of the corresponding parameters is to be specified.

Saving of a Persistent Object in a Remote Mirrored Storage

Of course, the user can then choose which kind of concrete storage has to be created on the remote machine. For example, if he wants to save a persistent integer on a remote mirrored storage, the following code will be appropriate (we use the persistent object package, presented in chapter 1, section 1.3) :

```
with Storage_Params.Non_Volatile.Stable.Mirrored_Storage_Params;
use Storage_Params.Non_Volatile.Stable.Mirrored_Storage_Params;
with Remote_Stable_Storages;
use Remote_Stable_Storages;
with Persistent_Integer;

declare
   S : Persistent_Integer.Persistent_Ref;
   R : Remote_Params_Type;
begin
   R := String_To_Storage_Params ("remote_name",
                 Mirrored_Storage_Params_Type'(
                     String_To_Storage_Params ("mirrored_name")));

   S := Persistent_Integer.Create_Object (R);
   S.Value := 651977;
   Persistent_Integer.Save_Object (S.all);
end;
```

By using the `String_To_Storage_Params` function, we declare our remote storage as a *mirrored storage* on the remote machine. Placing a mirrored storage, which is still stable on a more reliable remote machine, avoids to loose time with local crash failures.

Configuration File for gnatdist

The previous main user program has to be mapped on the user partition of the distributed system. The `gnatdist` tool needs a configuration file, with an *Ada-like syntax*, to specify the mapping. This could be:

```
configuration PersistentSaveConf is

    procedure Persistent_Integer_Save;

    User_Part : Partition := ();
    for User_Part'Main use Persistent_Integer_Save;

    Server_Part : Partition := ();
    procedure Endless_Loop is in Server_Part;

    Relay_Part : Partition := ();

    for User_Part'Termination use Local_Termination;

begin
    User_Part := ();
    Relay_Part := (Remote_Register_Relay,
         Storage_Params.Non_Volatile.Stable.Mirrored_Storage_Params);
    Server_Part := (Register_Server);
end PersistentSaveConf;
```

The following paragraph comments this configuration file line by line:

- Line **1** : Typically, after having saved the configuration file with '`.cfg`' as an extension, the user types: `gnatdist persistentsaveconf.cfg`

  to compile the program and create three executables, one for each partition.

- Line **5, 8, 11** : Declarations of the three partitions composing the distributed system.

- Line **6** : Partition `User_Part` contains no RCI package. However, it will contain the main procedure of the distributed application, called `Persistent_Integer_Save` in this example. Since `Persistent_Integer_Save` uses the instantiation of our generic remote package, the latter is also linked to the user partition.

- Line **9** : The procedure `Bidon` mapped in the server partition loops without end and displays a message like "`Waiting`…" each ten seconds. We need this procedure in order to keep the server running.

- Line **13** : Specify a termination mechanism for `User_Part.` The default is global distributed termination. When `Local_Termination` is specified, a partition terminates a soon as the main task of the partition terminates.

- Line **17-18** : `Relay_part` contains the remote relay unit categorized with the pragma `Remote_types.` Moreover, it must explicitly contain the stable storage parameters unit, in order to allow correct marshalling and handling of the parameters. The remote relay unit only 'with' the storage class-wide type, so the code for each particular storage type used must be configured manually.

- Line **19** : `Server_part` contains the RCI server package.

Launching the Distributed System

Once the code is compiled and the partitions created, we can manually launch each partition:

1. We begin by launching the registration server partition, for example, on the unix station `lglsun1.epfl.ch` by typing:

   ```
   lglsun1->> server_part --nolaunch --boot_server
   tcp://lglsun1.epfl.ch:7234
   ```

2. Then, on the station `lglsun7.epfl.ch,` we launch a remote relay :

   ```
   lglsun7->> relay_part --boot_server tcp://lglsun1.epfl.ch:7234 --name
   remote_name
   ```

   Note that we added the argument with `--name`. At its elaboration, the relay partition registers at the server with this name.

3. Last, we launch the user program on the station `lglsun10.epfl.ch` :

```
lglsun10->> user_part --nolaunch --boot_server
tcp://lglsun1.epfl.ch:7234
```

The `Create_Storage` function (of `Remote_Params_Type`) contacts the registration server and asks for the reference to a remote relay that registered with the name of the remote parameters. If there is none, the calling task is blocked until the registration of such a storage.

In our example, the search succeeds as we used the same name `remote_name` to register our relay (see *Saving of a Persistent Object* at page 44). Then the `Create_Storage` function (of `Remote_Params_Type`) uses the obtained reference to transmit the creation order to the remote relay, with the parameters of the user. On the remote machine, the remote relay calls the `Create_Storage` function of the concrete storage parameters type.

Once our remote storage is created, it keeps the reference to the corresponding remote relay, so that the operations like `Remote_Write, Remote_Open` and so on, do not need to ask the server each time.

4. After the user procedure termination, we can check that the three files composing the mirrored storage were created on the remote station `lglsun7.epfl.ch`:

```
lglsun7->>ls -l
total 10228
(…)
-rw-r--r--   1 xcaron   students       1 Feb 23 14:38 mirrored_name.log
-rw-r--r--   1 xcaron   students      55 Feb 23 14:38 mirrored_name1.data
-rw-r--r--   1 xcaron   students      55 Feb 23 14:38 mirrored_name2.data
```

We clearly see that the log file `mirrored_name.log` was created, and written, thus containing 1 byte of information, as well as the two identical data files containing the 55 bytes of the streamed integer object. The mirrored storage class automatically puts the extensions .log, 1.data and 2.data to the files.

CHAPTER 4 - REPLICATED DISTRIBUTED STABLE STORAGE

In chapter 3, we have shown the design and implementation of a remote storage class. Now we present a stable storage class based on *replication.* The architecture of this distributed system is very similar to the one of the remote system. The difference is that storages with the *same* name are allowed registering at the server; they will form a group, and operations of the replicated storage class will be broadcast to all these remote storages. This change requires a lot of addition to manage dynamically the group of replicas. A protocol must be established.

In the design section, we will focus on the additions and changes made to the remote storage class. Therefore, we *strongly advise* the reader to have a look at the previous chapter before tackling this one.

4.1 GENERAL IDEA

Literature

Traditionally, the main idea of replication is to store copies of a same object on different sites. Here is what we can read about the subject in *McDermid Software Engineer's Reference Book* [9]:

> *Replicated files or other objects are usually provided to ensure resilience to node failures. Replicated data should always appear consistent to the user, even though all the copies may not be identical. Generally, it should be possible to execute any operation on the data at any of the sites holding a copy.*

In our thesis, the "sites" are storages located on processing nodes. The challenge is to manage dynamically the evolution of the group of replicas since some of them can be suddenly unreachable or others can join the group.

With reliability, performance is an important feature. Therefore, operations made on the storages must not be performed sequentially but broadcasting has to be used. Similarly, the user cannot and might not want to wait that all replicas have answered to an operation before performing the next one. A protocol has to be found to solve these problems.

The purpose of this chapter is to go more in details in the design and, for the interested reader, in the implementation of a replicated stable storage.

4.2 PRELIMINARY ASSUMPTION

As for the mirroring algorithm, we have to specify the failure assumptions i.e. in which conditions we can guarantee to the user that it will work. In the replication case, there is only one to consider.

If the following assumption is not verified, then the resulting replicated storage can not be considered stable.

Replica Reliability

We assume that among the replicas, there is always one that can be reached and will execute the operation correctly. Note that this correct replica has not to be the same each time.

To fulfil this requirement, the number of replicas to be used must be chosen depending on the failure probability of one replica and on the reliability of the network.

4.3 DESIGN OF THE REPLICATED STORAGE CLASS

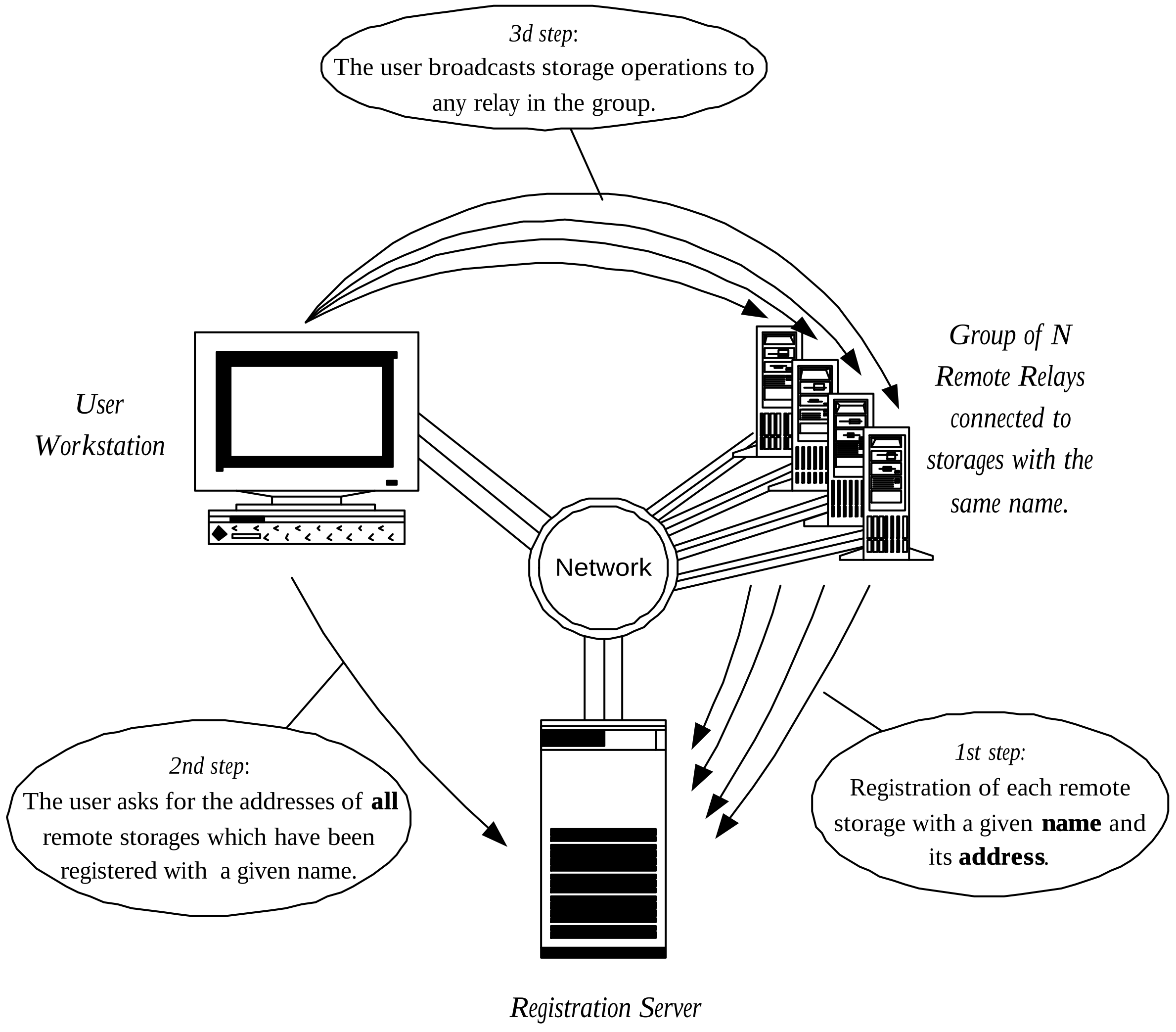
3d step:
The user broadcasts storage operations to any relay in the group.
User Workstation
Group of N Remote Relays connected to storages with the same name.
Network
2nd step:
The user asks for the addresses of **all** remote storages which have been registered with a given name.
1st step:
Registration of each remote storage with a given **name** and its **address**.
Registration Server

Figure 17 - Overall Design of the Remote Storage Class

The design of the replicated storage class is very similar to that of the remote storage class in term of distributed partitions and communication between these partitions. The main difference is that, to be a stable storage, the operations are not send to a single remote storage, but to a group of storages having the same name. Figure 17 shows the global architecture of the replicated distributed system.

Note that the server can manage more than one group. Each time that a relay registers with a name that was unknown before, the server defines a new group with this name. Each relay that registers itself later with this name is automatically joined to the group. That way, if we have N replicas in the group, we can support N-1 failures of these replicas. But according to our preliminary assumption, the last one will always be reachable.

4.3.1 DYNAMIC MANAGEMENT OF AN EVOLUTIONAL GROUP OF REPLICAS

The *worst* thing, in term of performance, would be that the user asks the server for an updated list of active replicas in a group, *each time* that he has an operation to perform on the replicated storage.

A better solution is for the user to have its *own local list* that has to be updated each time a new replica registers at the server. A solution to that problem is to provide *notification* from the server to the user at this occasion. With each group maintained by the server is associated the address of the user of this group. When calling the user to notify the addition of a new replica, the server directly passes the address of this new member.

On the other way, when the user wants to eliminate from the group a replica that is not responding any more, he updates its own list and tells the server to do the same in the global list.

4.3.2 DESIGN OF A REPLICATED RELAY OR REPLICA

The role of a relay located on a remote machine is to receive operation orders from the user and to transmit these orders to the local storage. This storage can be of any kind: volatile memory, local file, mirrored storage composed itself of other storages, and so on…

Registration of the Replica

The first problem to solve is the communication between the user and the replica, since the user cannot immediately know where a relay is located. Therefore, when the replica starts, it has to register itself at the server, waiting on a channel of the network. At registration time, the replica gives two pieces of information to the server:

1. The *name* of the storage (mandatory). The user chooses this name when launching the replica partition on the command line. Note that, at this point, the storage is not created, for the replica does not know what kind of storage to create. The first command that the replica will receive later from the user should be a storage creation order.

2. Its *address*, i.e. a reference to the location where the replica resides. It can be viewed as a fat pointer, that is to say a structure with a remote address and a local address (like an URL: `<protocol>://<remote-machine>/<local-directory>`).

Operation Number

The operations that are send to the replica are numbered. Imagine that, for a moment, there is a communication failure between the user and the replica. The replica cannot know that there was this failure and has probably missed some operations. With a *local counter* compared with the number of each received operation, the replica can know if he missed some operations. If it is the case, it must not answer positively to a read operation; maybe it was missing a previous write operation and its data are not updated. Instead, it has to wait for the next write to be sure of its updating. It can also execute without any restriction deletion orders or closing and opening in write mode since these operations reset the storage. After such an operation, it can continue normally.

4.3.3 DESIGN OF THE REGISTRATION SERVER

Unlike the replicated relay, only one server can exist in our distributed system. The server provides four services:

1. A *registration service* for all replicated relays. A replica must register with a name. If there is already a group with this name, it joins the group, else a new group is created. All relays are registered, with their name and address, in a list internal to the server.

2. A *search service* for the user who wants to reach a group of replicas with a given storage name. If no group exists with this name, or if the number of replicas is to small for the user, the request is blocked until enough new replicas register with the right name. Thus, each time a relay registers itself, the server has to check all the requests to see if one can be satisfied. When a search succeeds, the server returns to the user the list of replica addresses in the group.

3. A *notification* service that informs the user when a new replica registers to the group, and sends it the new address. If the user has not reached the maximum number of partitions that he wants to use, he can add the address of the new replica to its local list.

4. A *removing* service, callable by the user. If the user has communication problems with a replica, he can decide to remove it from the group, i.e. from its local list and from the global list maintained by the server. Note that the criteria to decide to remove a replica can be chosen by the user. Maybe he will attempt three times to communicate or he will wait for a certain time, and so on.

To achieve this goal, the server must have the quality to be called remotely from the user partition as well as from the relay partitions.

4.3.4 DESIGN OF THE USER PARTITION

The unit that is offered to the user to access a replicated storage needs a careful design. Each time the user wants to *create* or *open* a replicated storage:

1. He creates parameters for this storage based on the name of the storage.

2. Then, he calls the appropriate procedure for creation or opening with the parameters. This procedure is in charge to ask the server for the list of addresses of all replicas with the same name. The user gives a minimal and a maximal number of replicas he wishes in the group, depending on his needs. Having a large number of replicas increases reliability of the replicated storage but may decrease performance. The calling procedure waits until an answer is returned. The server keeps the address of the user and associates it with the group of replicas that he uses. If later, a new replica register in this group, the server will notify the user. If the user has not the maximum wished number of replicas, he can include the new one in its local list as an active replica. Else he can be included as a replica in *reserve*.

3. The returned addresses are stored locally and used in subsequent operations like *writing*, *reading*, *closing*, *opening*, *deleting*, and so on. Operations are broadcast to all replicas in the

group, and not performed sequentially since this can result in very poor performance. An incrementing serial number in added to each operation, which allows the replica to know if he missed some operation. If it missed an operation, the replica has no right to answer to a read request with the updated operation number, because it could have missed a write operation. Depending on the kind of operation, the user does not have to wait for acknowledgments of all the replicas. In general, if an operation has only one correct answer, the user does not have to wait for an acknowledgement at all, since our assuption is that always one replica will execute the operation correctly. If, on the other hand, an operation has multiple correct answers, e.g. executing correctly or raising an exception, we must wait until we receive one answer that is satisfying. For example:

- The write, delete and close operations fall in the first category. The user does not have to wait for an acknowledgement.

- For the creation and the opening operation, the user must wait for at least one answer. If one replica has executed the operation correctly without raising an exception, the user can continue. If the first answer although is an exception, such as for instance `Storage_Existant` for the creation operation, then we must wait for the answer of all other replicas. If all other replicas also raise the same exception, then we will raise the exception for the user. But if only one replica executes the operation without raising the exception, we assume that this is the correct replica, and continue normally.

- For the reading operation, the user waits for the first answer of a replica with a correct serial number. The failure assumption ensures that there will be one.

4. The user must deal with communication errors when sending operations to the replicas. According to personal criteria, he can decide to remove a replica from its local list. Then he has to inform the server using the removing service, to do the same in the global list of the server.

This user unit must provide access to all possible operations on a *stable* storage.

4.4 THE REPLICATED STORAGE CLASS IN THE STORAGE HIERARCHY

In section 3.2, we realized that the remote storage class should be implemented in a *generic* way since it can derive from any non-volatile storage.

We do not keep the same method for the replicated storage class, since as it is a stable storage, it can only derive from any *stable* class-wide type. In the actual state of the hierarchy, as stable storage is not divided again in subcategories, therefore using a generic package is not mandatory. Nevertheless, if the hierarchy evolves in the future, we always have the possibility to transform it into a generic package.

In figure 18, we see the place of the replicated storage in the hierarchy. The replication uses a multitude of storages, but at least one.

The replicated storage parameters are also located at the same place in the hierarchy.

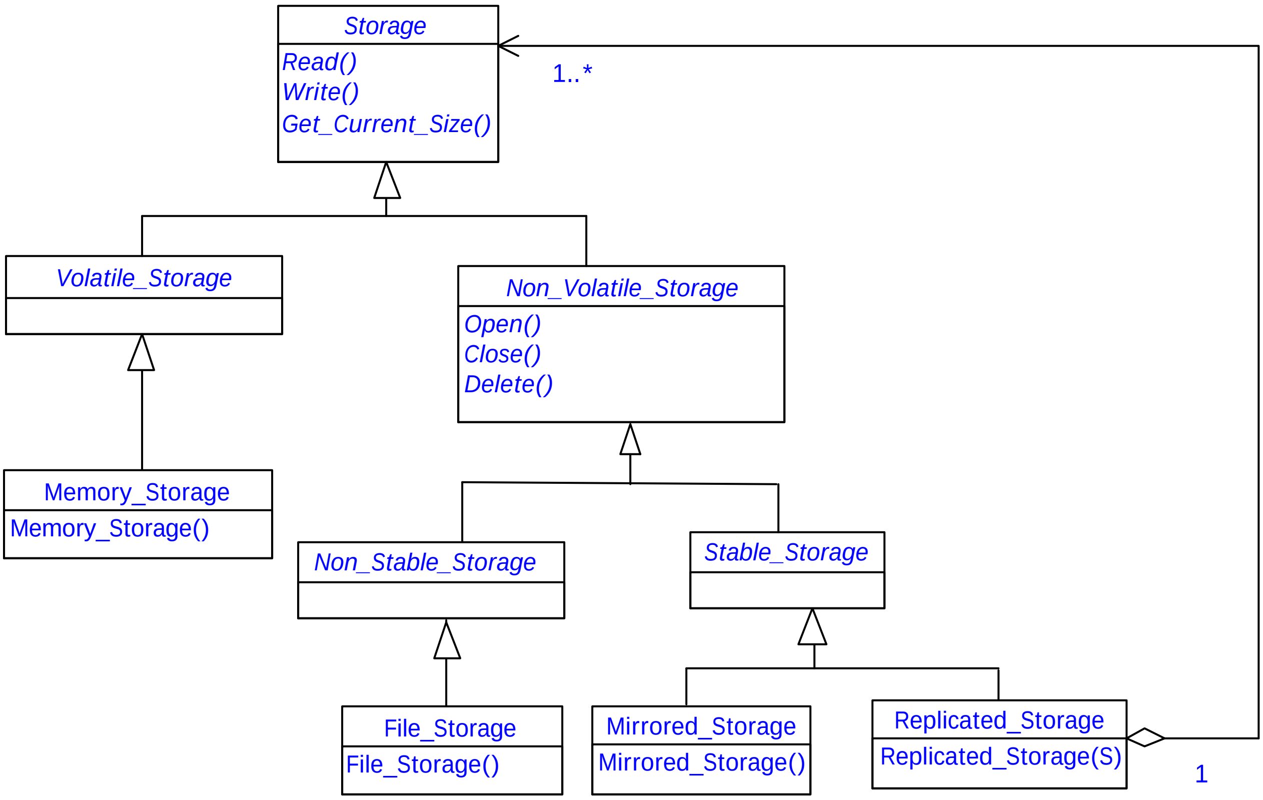

Figure 18 - Replication in the Storage Hierarchy

4.5 IMPLEMENTATION OF THE REPLICATED STORAGE CLASS

The implementation of the replicated storage class is very similar to that of the remote storage class. The distribution of the units in different partitions is the same. There is no additional library unit but all have been more or less changed to integrate the numerous add-ons described in the design section. The most reorganized units are the server unit and the user unit. The latter has to deal with broadcasting and management of dynamics group of replicas. In this section, we will focus on this unit, using protected objects and tasking facilities of Ada95. But before we will have a look at the new design of the server.

4.5.1 A TWO-DIMENSIONAL LINKED LIST IN THE REGISTRATION SERVER

To maintain information on registered replicas, the server needs an evolving dynamic structure. Therefore, we implemented a two-dimensional linked list in order to group the replicas by name. Every group defines a replicated storage. As illustrated in figure 19, a node of the main list represents a group. Each time that a relay registers with an unknown name, a new group, i.e. a new node is added to the main list. Each node contains the name of the group, the number of replicas in the group, and a linked list containing the references to the replicas.

Moreover, each node contains a reference to the user of the group. It is used to *callback* to the user when a new replica is added to the group. The users can then update his local list. In the other direction, the user informs the server when a non-responding replica has to be removed from the server list.

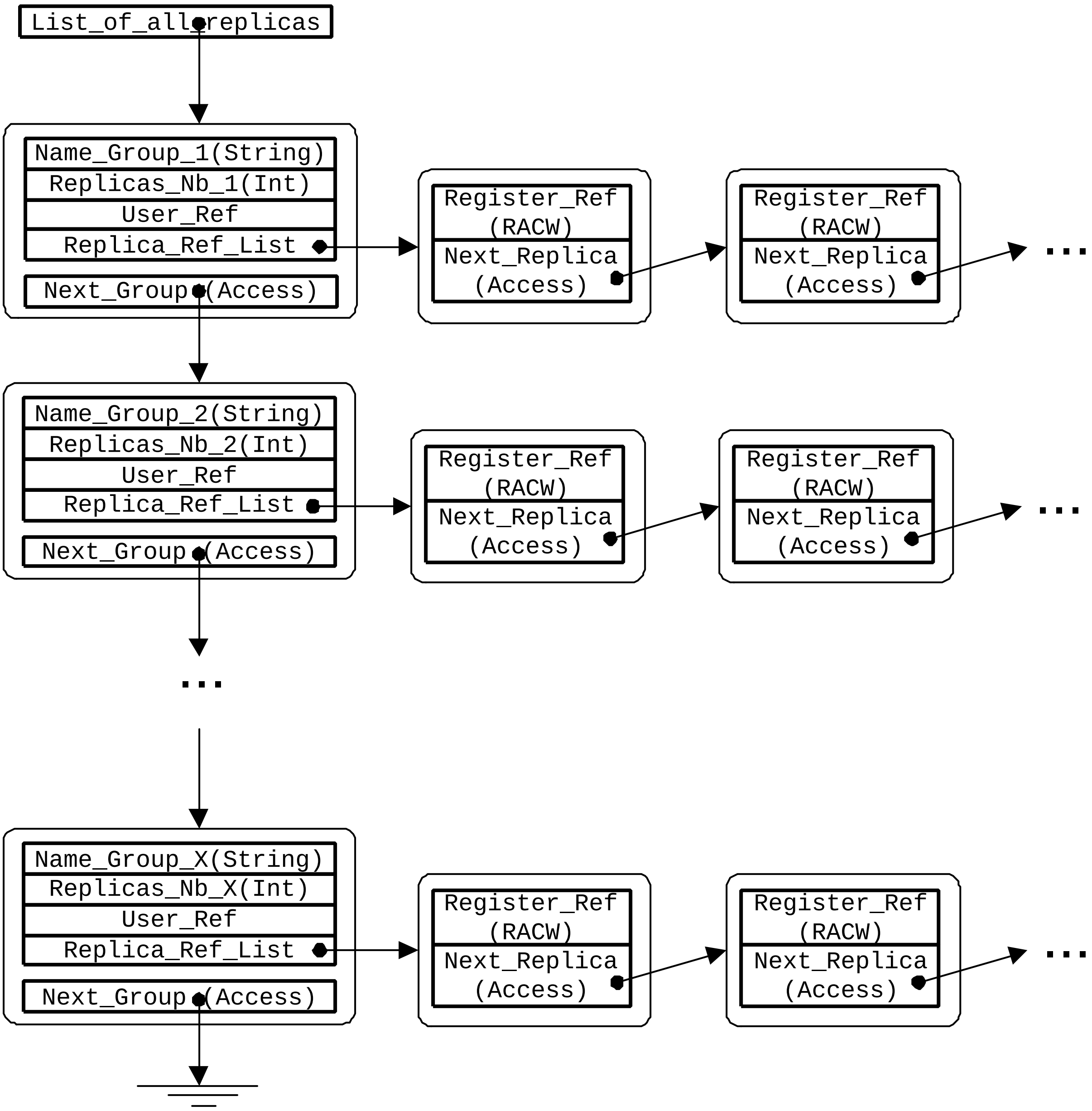

Figure 19 - Two-Dimensional Linked List in the Registration Server

With this structure, the searching service of all replicas with a given name is straightforward, as the server has just to look for the group node and to return the complete list attached to this node.

The code for this list is relatively traditional so we will not show it in this report. Like in the server for the remote class, a protected object is used to guarantee consistent updating of the data in the presence of concurrent calls.

4.5.2 TASKING AND PROTECTED OBJECT IN THE USER PARTITION.

To enhance performance, we use broadcasting of the operations to all replicas of a group as explained in the design section. Therefore, we use the tasking facilities of Ada95.

Avoiding Rendezvous

A first idea could be to create one task for each replica in the group, and to communicate with the tasks using Ada rendezvous.

The task body would contain a big select statement. Each alternative corresponds to a kind of operation to perform on the replica depending on the operation. For example, the type declaration of such a task could be:

```
task type RemoteIO_Task_Type (R : Replicated_Storage_Ref;
                              I : Positive) is
      entry Write_Launch;
      entry Read_Launch;
      entry Create_Launch;
      …
   end RemoteIO_Task_Type;
```

The integer *I* is the index of the replica associated to this task. The body of the task may look like that:

```
task body RemoteIO_Task_Type is
      Item_Ref : Stream_Element_Array_Ref;
      Params_Ref : Storage_Params_Ref;
   begin
      loop
         select
            accept Write_Launch (Item : Stream_Element_Array) do
               Item_Ref := new Stream_Element_Array' (Item);
            end Write_Launch;
            begin
               Replicated_Write (R.Registers (I).all, Item_Ref.all);

            exception
               when  Storage_Not_Existant |
                 Storage_Corrupted =>
                  Put_Line ("task n. " & Positive'Image (I)
                                       & "handles an exceptions");
            end;
            Free (Item_Ref);
         or
            accept Read_Launch do
            …
         end select;
      end loop;
   end RemoteIO_Task_Type;
```

When an operation has to be performed on the replicated storage, one task after the other must be launched. The *critical problem* is that rendezvous are synchronous, i.e. the caller and the callee are blocked until the rendezvous is finished. A task that is still executing a previous operation can not accept a new one, and will therefore block all other tasks. All the broadcasting may be *slowed down* by the fault of a *single* replica.

Therefore, Ada rendezvous are not adapted to our problem and we will not use them.

We have therefore chosen to use asynchronous communication between the main task and the tasks responsible for the communication with the replicas through a protected object.

Main Idea

A replicated storage is a record of many things, but the most important are:

- The parameters of the storage.
- The list of the references (RACW) to the replicas to be used for this storage. The replicas have all the same name.
- Another list of the same size, containing accesses to tasks. One task is associated with one replica in the previous list.
- A protected object called an "operation executor".

The main idea is that all tasks run in parallel, each one working for its replica. When the task has no work to do, it asks the operation executor for a new operation to perform on its replica. When it has finished to perform the operation, it delivers a report to the executor and asks for a new operation. That way, each task works *at its speed*, without slowing down the others.

A Type Representing a Storage Operation

The solution that we have used is the following: We create an abstract class describing a storage operation and for each kind of operation, we derive a concrete type with adding fields useful for this operation.

```
type Operation_Type is abstract tagged record
      Storage : Replicated_Register_Ref;
      Answer_Node : Operation_Node_Ref;
      Mission_Number : Positive;
      Error_Id : Exception_Id := Null_Id;
end record;

type Operation_Ref is access Operation_Type'Class;

procedure Execute (Operation : in out Operation_Type) is abstract;
function Is_Valid (Operation : Operation_Type;
                   Remaining : Integer) return Boolean is abstract;
…
```

The procedure `Execute` can be called by a task to perform the operation. The function `Is_Valid` checks if the answer ot the operation execution is correct or not. If an answer is correct, it must be copied into the component Answer_Node.

A concrete operation will derive from this abstract type, and add additional elements to store its in and out parameters. For instance, the read operation would be represented by :

```
type Read_Operation_Type (Bottom, Top : Stream_Element_Offset)
      is new Operation_Type with record
      Item : Stream_Element_Array (Bottom .. Top);
      Last : Stream_Element_Offset;
end record;
```

The Operation Executor

The operation executor holds a linked list of operations to be performed on replicas as shown in the private part of its interface:

```
protected type Operation_Executor_Type is

      entry Execute_Creation_Operation
               (Storage : Replicated_Storage_Ref;
                Operation : in out Operation_Type'Class);

      entry Execute_Operation
               (Storage : Replicated_Storage_Ref;
                  Operation : in out Operation_Type'Class);

      entry Get_Next_Mission (Number : in Integer;
                              Operation : out Operation_Ref);

      entry Mission_Accomplished (Number : in Integer;
                                  Operation : in Operation_Ref);

      entry All_Done;

   private

      …

      First_Operation : Operation_Node_Ref;
      Last_Operation : Operation_Node_Ref;

      procedure Get_Next_Mission_Number (Number : out Positive);

      procedure Insert_New_Operation (Operation : Operation_Type'Class);

      entry Get_Next_Mission_Internal (Number : in Integer;
                                       Operation : out Operation_Ref);

      entry Wait_For_Termination (Storage : Replicated_Storage_Ref;
                                  Operation : in out
                                               Operation_Type'Class);

   end Operation_Executor_Type;
```

The entries `Execute_Creation_Operation` and `Execute_Operation` are called by the operations of the replicated storage class in order to launch broadcasting.

At creation of the replicated storage, which reference is given as parameter to `Execute_Creation_Operation`, its structures are initialized:

1. Call to the registration server to obtain a list of replicas with minimal and maximal numbers.

2. Creation of corresponding tasks.

3. Creation of an operation executor.

The entries `Get_Next_Mission` and `Mission_Accomplished` are for the tasks in order to ask for a new operation to perform and to report about an accomplished operation. The executor has to analyze the results that are stored in the operation type itself. If there is no new mission for the task, it is requeued to the private `Get_Next_Mission_Internal` entry where it waits for new operations to arrive.

The entry `All_Done` is provided to allow correct termination of the tasks.

The Life of a Task

The life of a task is very simple and a very repetitive, probably boring life. As shown in the following lines of code, it takes a mission or waits for one, executes it, and notifies the executor that the job was done:

```
task body RemoteIO_Task_Type is
      Operation : Operation_Ref := null;
   begin
      loop
         Storage.Operation_Executor.Get_Next_Mission (My_Number,
                                                     Operation);
         Execute (Operation.all);
         Storage.Operation_Executor.Mission_Accomplished
                                                (My_Number, Operation);
         Free (Operation);
      end loop;
end RemoteIO_Task_Type;
```

The actual remote procedure call is done in the `Execute` operation.

Example of Remote Operation: Writing

With the powerful executor, the implementation of a replicated operation is very simple. For example, to do a write, it is enough to create a corresponding `Write_Operation` initialized with the item to be written, and to launch broadcasting by calling the `Execute_Operation` entry. Here is the code:

```
procedure Write (Storage : in out Replicated_Storage_Type;
                 Item    : in Stream_Element_Array)
   is
begin
   declare
      Write_Operation : Write_Operation_Type;
   begin
      Write_Operation.Item_Ref := new Stream_Element_Array' (Item);
      Storage.Operation_Executor.Execute_Operation
        (null, Write_Operation);
      if Write_Operation.Error_Id /= Null_Id then
         Raise_Exception (Write_Operation.Error_Id);
      end if;
   end;
end Write;
```

An elegant and powerful solution

With this implementation, we used the highly efficient features of Ada for concurrency. The result is an elegant and fully object-oriented replicated storage class. Mapped on partitions with GLADE, the system works very well for our example of saving and restoring persistent objects. We will not show here this example, as this is the same that was described in section 3.5 with a remote storage class. The difference is that we can now launch more relay partitions with the same name and that the stable property is achieved.

CONCLUSION

In this diploma thesis, we achieve durability of objects, one of the four properties of transactions, by saving their state to so-called *"stable storage"*. Two forms of stable storage are designed and implemented. The first one is based on the *mirroring* technique, the second on *replication* and the distributed programming facilities of Ada 95. During the design of the replicated storage, it was possible to separate the communication mechanism from the actual replica management. This resulted in the additional development of a non-volatile, non-stable *remote* storage that can be used independently from replication.

The approach heavily relies on the peculiarities of object-oriented programming: our different stable storages are seamlessly integrated in the storage class hierarchy. They can be used to achieve fault tolerance and data persistence. We have demonstrated the great possibilities of reuse: the mirrored storage class can be instantiated with any non-volatile storage, the remote and the replicated storage can be used with any kind of storage. It is thus possible to create a multitude of different storages, depending on the needs of the application programmer.

The example of a generic package providing object persistence demonstrates the flexibility and the elegance of this interface. Any combination of storages can be used in a very simple way, providing very personalized solutions to the user needs. Performance, physical constraints, and so on, are criteria to consider before choosing a kind of storage. With our work, for instance, a user located in Geneva can store all his data with safety on a remote mirrored storage in New York.

During the four months of this diploma thesis, we gain a large experience in object-oriented programming concepts and in advanced features of Ada95 like concurrency handling and the Annex E on distributed systems.

February 25, 2000, Lausanne.   Xavier Caron

REFERENCES

[1] ISO: *International Standard ISO/IEC 8652:1995(E): Ada Reference Manual*, Lecture Notes in Computer Science 1246, Springer Verlag, 1997; ISO, 1995.

[2] Kienzle, J., Romanovsky, A.: "On Persistent and Reliable Streaming in Ada", To be published in *Proceedings of Ada Europe*, 2000.

[3] Kienzle, J., Romanovsky, A.: "A Framework Based on Design Patterns for Providing Persistence in Object-Oriented Programming Languages", submitted to *Joint Modular Languages Conference,* 2000.

[4] Gamma, E., Helm, R., Johnson, R.: *Design Patterns*, Addison Wesley, Reading, MA, 1995.

[5] Atkinson, M. P., Jordan, M. J., Daynès, L., Spence, S.: "Design Issues for Persistent Java: a Type-Safe, Object-Oriented, Orthogonally Persistent System". In *Proc. of the 6th Int. Workshop on Persistent Object Systems,* Cape May NJ (USA), May 1996.

[6] 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* **105**, pp. 246-265 and pp. 357-370, Springer Verlag, 1981.

[7] Ralston, A., Reilly, E. D.: *Encyclopedia of Computer Science Third Edition,* p. 677, Van Nostrand Reinhold, New York, 1993.

[8] Pautet, L., Tardieu, S.: "Inside the Distributed Systems Annex". In *Reliable Software Technologies-Ada-Europe'98,* volume 1411 of *Lecture Notes in Computer Science,* pp. 65-77, 1998.

[9] McDermid, J.: *Software Engineers's Reference Book*, ch. 53.5.7, Butterworth Heinemann, Oxford, 1994.

PERSONAL NOTES ✍