Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

Disaster recovery in heterogeneous environments

Rouyre, Frédéric

Award date:
2000

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 02. May. 2026

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique

Disaster Recovery in heterogeneous environments

Frédéric ROUYRE

Mémoire réalisé en vue de l'obtention
du titre de Maître en Informatique

Année Académique 1999-2000

Résumé

Dans l'optique des procédures de "Disaster Recovery", les procédures de backups occupent encore une place de choix. Dans ce mémoire, nous nous efforcerons de voir pourquoi les procédures de backup ne sont plus adaptées aux configurations informatiques actuelles, et comment étendre la notion de backup afin de mieux répondre aux nouvelles contraintes imposées par ces nouvelles configurations. Et cela dans des milieux hétérogènes. Nous ferons également un bref exposé des techniques de Disaster Recovery employées pour la protection de serveurs.

Abstract

In the perspective of the Disaster Recovery procedures, the backup procedures still occupy a central place. In this paper, we will try to see why the backup procedures are not anymore suitable for the current computer configurations, and how to enhance the backup notion in order to better answer the new constraints imposed by those new configurations, and that in heterogeneous environments. We will also briefly describe the Disaster Recovery technics employed for the servers protection.

Contents

Acknowledgements 1

Introduction 3

1 Disaster Recovery Technics: Presentation 9
- 1.1 Redundant Array of Devices . . . 9
  - 1.1.1 Principles . . . 9
  - 1.1.2 RAID Industry Standard . . . 10
- 1.2 Process Migration and the Like . . . 15
  - 1.2.1 Computers Cluster . . . 15
  - 1.2.2 Crash Detection . . . 16
  - 1.2.3 Process Migration . . . 16
  - 1.2.4 Practical Example: Distributed Web Servers . . . 17
  - 1.2.5 Conclusion . . . 17
- 1.3 Human and Building Resources Management . . . 18
- 1.4 Conclusion . . . 18

2 Heterogeneous Environment Implications 21
- 2.1 Computers & Operating Systems . . . 21
  - 2.1.1 Computers . . . 21
  - 2.1.2 Operating Systems . . . 22
- 2.2 Network . . . 22
- 2.3 Software Architecture . . . 22
  - 2.3.1 Preliminary Idea . . . 23
  - 2.3.2 Application Design . . . 23

3 Backup Software: HSMS & HSMS-CL 31
- 3.1 HSMS: Presentation . . . 31
  - 3.1.1 HSMS Archives . . . 32
- 3.2 HSMS-SV . . . 32
- 3.3 HSMS-CL . . . 33

3.4 Operation Mode 33
3.4.1 Centralized Operation 33
3.4.2 Decentralized Backup 34
3.5 HSMS-CL and Application Specific Modules 34
3.6 HSMS-CL and Computer Recovery 35

4 Computer Recovery **37**
4.1 Platform Study: The Intel i386 Case 37
4.1.1 Foreword 37
4.1.2 Computer Bootstrapping 38
4.2 Virtual Platform Approach 42
4.2.1 Components Identification 42
4.2.2 Evaluation 44
4.3 Operating System Study: The Linux Case 44
4.3.1 The Secondary Loader 45
4.3.2 The Kernel Image 45
4.3.3 The Kernel Boot Process 45
4.3.4 The Kernel Is Up and Running 46
4.4 Virtual Operating System 46
4.4.1 The File-System 47
4.4.2 Evaluation 47
4.5 Computer Recovery: The KISS Approach 47
4.5.1 Keep it Simple... 47
4.5.2 ... and Stupid 48
4.5.3 First Solution: Locking 48
4.5.4 Second Solution: Intelligent Buffering 49
4.5.5 File-System Type 49
4.5.6 The KISS Evaluation 50
4.6 Computer Recovery: The Components Approach 50
4.6.1 Manual Operations 50
4.6.2 Automatic Operations 53
4.6.3 Operating System & Applications Specific New Components 56
4.6.4 Conclusion 58

5 Implementation: Computer recovery (bsrecov) **59**
5.1 Introduction 59
5.2 The Disaster Recovery Commands 60
5.3 Disaster Recovery Database Management 60
5.3.1 Design and Description 60
5.3.2 Implementation 61

5.4 Modules Management 68
5.4.1 Design and Description 68
5.4.2 Implementation 70
5.5 Remote Objects Database Backup/Restore 75
5.6 Application Configuration File 75
5.6.1 Motivation 75
5.6.2 Description 76
5.7 BsRecov for Linux/i386: Practical Example 77
5.7.1 Presentation 77
5.8 Conclusion 78

6 Conclusion **79**
6.1 Feelings 80

Glossary **81**

A BsRecov/Linux/i386 sources **85**
A.1 C Sources 85
A.1.1 main-wrapper.c 85
A.1.2 main.c 86
A.1.3 object.c 98
A.1.4 api.c 102
A.1.5 config.c 110
A.1.6 bserror.c 115
A.2 Include files 116
A.2.1 main.h 116
A.2.2 object.h 116
A.2.3 api.h 116
A.2.4 modules.h 117
A.2.5 config.h 117
A.2.6 bserror.h 118
A.2.7 version.h 118
A.2.8 bsapi.h 118
A.3 Makefile 119
A.4 Dynamic Shared Objects Code 120
A.4.1 libdpt.so 120
A.4.2 libdev.so 130
A.4.3 libfs.so 134
A.5 Linux disk set 141
A.5.1 Bsrecov /Linux/i386 init source 141

B BsRecov /Linux/i386 Manual **155**
B.1 Introduction 155
B.2 Usage 155
B.2.1 Bsrecov 155
B.2.2 Configuration File: bsrecovrc 156
B.2.3 Standard Modules 156
B.2.4 Recovery Disks set 157
B.3 Architecture 158

List of Figures

1 The Backup Problematic 4
2 Disaster Recovery Operation Time Line 5

1.1 RAID Level 0: Striping 11
1.2 RAID Level 1: Mirroring and Duplexing 12
1.3 RAID Level 2: Hamming code ECC 13
1.4 RAID Level 3: Parallel transfer with parity 13
1.5 RAID Level 4: Independent data disks with shared parity disk 14
1.6 RAID Level 5: Independent data disks with distributed parity 15

2.1 Compilation and Linking (Static) 28

3.1 Backup & Migration with HSMS 32
3.2 Centralized Backup 34
3.3 Decentralized Backup 34

4.1 Hard Disk: MBR Format 39
4.2 Floppy Disk: Boot Sector Format 39
4.3 Partition Table Entry 40
4.4 Boot Process Components Summary (Example) 51
4.5 Computer Object Hierarchy 54

B.1 Bsrecov General Architecture 158

Table of Abbreviations

API Application Programming Interface

CPU Central Processing Unit

DNS Domain Name Server

DR Disaster Recovery

ECC Error-Correcting Code

HSMS Hierarchical Storage Management Software

HSMS-CL HSMS Client

HSMS-SV HSMS Server

KISS Keep It Simple and Stupid

LINUX Linux Is Not UniX

OS Operating System

RAID Redundant Array of Independant Disks

Acknowledgements

First, I'd like to thank Mariève, my future wife, for her constant encouragements and for giving us Nicolas, a wonderful baby. Although having a little child was so tiring, this is also one of the most beautiful expericence. For that, I want to thank them once again.

I'd also like to thank all the members of the team where I made my training period at Siemens Software S.A.. Even if they asked me not to do so, a special thanks goes to Olivier (for the review of this paper, even when he was ill), Henry (for the time we passed on my version of HSMS-CL), Antonio (for the constructive discussion we had in front of numerous cups of coffee), Marc (for his objective view on Linux), Fabian (for helping me to start this work and his wise advises), Nadine (for borrowing me a so useful cup) and the rest of the team for being so hospitable. Once again, thank you all.

Thank to Marc Charlier for reading and correcting this work. I hope he was not too disappointed by my english and that this little sentence (that he did not read) does not contain too much mistakes.

Thank to Siemens Software S.A. for providing a wonderful coffee-O-matic machine and for accepting me during my training period.

Thank to Oce Software Laboratories for giving me my first job. I hope our future collaboration will be very fruitful.

A final thank has to go to my promoter, Pr. Jean Ramaekers, who inspired that work and with who I had valuable discussions. I had to thank him once again for giving me the courses that I enjoyed the most during my studies: Operating Systems and Security.

Introduction

In our so insecure society, what is more normal than protecting our data. Actually, computers are everywhere in our life of today. And they act as more and more important tools. Some parts of our society cannot do anything without them. Data are then like gold and they must be protected against any kind of disaster. Furthermore, data have to be recoverable if we fail to protect them. That is, in short, what disaster recovery deals with.

Of course, like in other technics, there are multiple areas where disaster recovery technics comes to aid. In fact, there are disaster recovery technics for almost all the industry components. More than protecting our data, disaster recovery deals with the life of an entity. And disaster recovery, in its globality, allows strategic, economic and technical levels.

In this work, we will voluntary limit our scope to some technical levels. More practically, we will concentrate our effort on computer sciences means. And still more practically, in heterogeneous environments. We will study what technics are available to recover data in such environments and we will try to do some research on the development of a disaster recovery tool.

The starting point of this disaster recovery study is that computers are generally not so well protected against disasters. Although disaster recovery is a fashionable subject, it seems that the data are so important that they get all the focus. Of course, data are the most important part of a computer center, because they are difficult to replace[1]. But, the sole protection of the data cannot permit the recovery of a computer center. And if we look at the most used data protection scheme, i.e. backups, it is not enough to allow the recovery of computers. Just because backups handle "operational" data and poorly the computer data.

But we are forced to admit that in heterogenous environments, backups are often the sole, affordable, way of protecting the valuable data. Of course, if someone could afford to duplicate three or four times any computer of his center on far different locations, and in real time, he could be sure that he risks nothing (or very little). But, although data (and therefore the equipment that processes them) are so important, the data protection offers no profitability. Therefore, investment costs in disaster recovery means are heavy and difficult to write off.

[1] And computers can be easily replaced

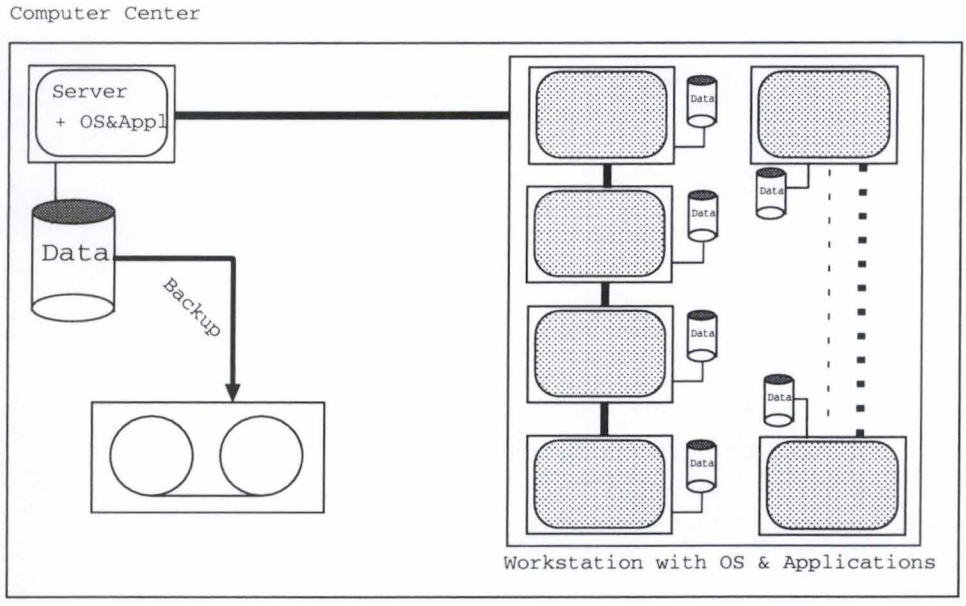

Figure 1: The Backup Problematic

The Backup Problematic

One aspect of the backup problematic, and certainly the most important, is the change of the computer center architecture. With the appearance of powerful workstation, terminals connected to central servers were replaced by those new kinds of workstation. The computational logic is then not longer located on one central server but on a multitude of workstations. What the best architecture between centralized computational power and the distributed one is, a question that we will not answer. But, it is certain that backup procedures, devised for centralized servers are not valid anymore, as we can see it on figure 1.

Furthermore, each workstation tends to have its own operating system and applications (database server,...). Therefore, we are in a situation where data are located on central servers[2], just as before. But operational data tends to be also located on those *"intelligent"* workstations.

The problem is that a backup of the central server and workstations data just saves the data. When centralized computing was the mainword, losing a workstation (i.e. terminal) was not a problem. Just because it could be easily replaced by another one without the need of serious human intervention. It really was the first implementation of *plug'n'play.* Losing the central server was not too serious a problem. Only one server needed to be set up, data being restored hence after[3].

Now, in heterogeneous environments with different kind of "workstations"(which act

[2]Mainframes are still in use!

[3]Administrators being capable of doing that rapidly. We could also mention that some specialized societies could provide a freshly set up computer still faster.

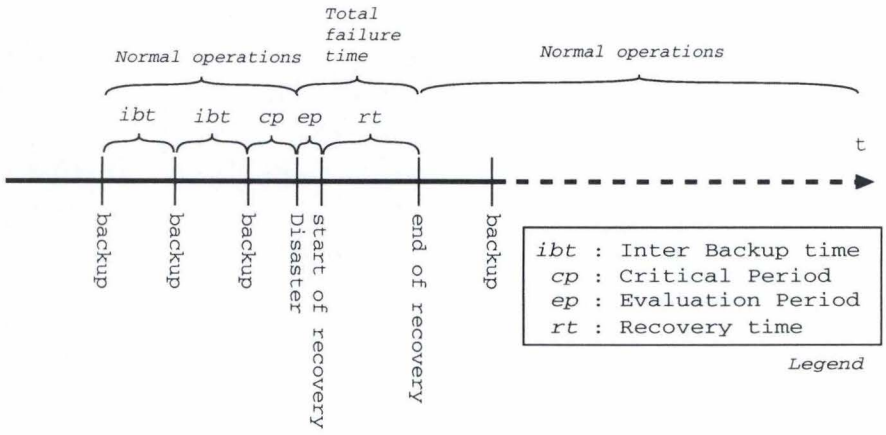

Figure 2: Disaster Recovery Operation Time Line

sometimes like real servers), if we are faced with a great disaster, a lot of administrators will be needed to handle the recovery operation of the equipment[4] and data. And that is a problem, because data have no value by themselves! They are valuable only because human factors can process them, at least one time. And to process them, human factors need equipments, which nowadays, tends to be difficult to recover.

In fact, in today's architecture, the more we have equipment, the more the recovery time increases in case of a large disaster[5]. As far as the the data are concerned, there is just no problems. Because the recovery time increases proportionally with the amount of data to back up. But it does not really increase with the number of installed computers[6].

The Disaster Recovery Problematic

Now, if we look at the figure 2, we see that the most important constraint in disaster recovery operation is to limit the global recovery time ($ep + rt$), and to recover a state similar to the one before the crash. Another achievement of a disaster recovery policy is then also to limit the critical period. I.e. the period during which the data produced cannot be recovered. Of course, the critical period largely depends on the type of data held on the considered computers. This is why disaster recovery technics employed really depends on the usage of the considered computer.

If we consider database servers holding important commercial transactions, backups are not enough to protect the data. Because losing one transaction can be dangerous (valuable). So, backups are not the solution[7] for this kind of computer.

[4] It takes a long time

[5] And even the recovery time of a simple workstation is heavy when we take into account its relative importance.

[6] We just have to add the transaction opening and closing time. But it can be done in parallel, so the time increase can be greatly reduced.

[7] Despite they should allways be done, who knows what can arise?

As far as the workstations are concerned, the critical period can be greater as, normaly, no workstations should contain vital data. The length of the critical period permitted is then inversely proportional to the value of the data held on the computer. In short, the most valuable things for servers are the data and for the workstations, the equipment itself.

In fact, we should not speak about equipment. Because the only way to "recover"a device is to repair it or to buy a new one. When we speak about recovering the equipment, we actually mean: "Recover the non-operational data that make the computer itself work". I.e. data that have no value for any operational task except to make the computer able to process operational data.

Our Scope

I did my training period at Siemens Software S.A.. And during this period, I could see how backup strategies where implemented for large computer centers. I also studied a backup product, HSMS-CL[8], well suited for backups of a large amount of data.

HSMS-CL has great properties. Actually, it works on a large variety of platforms, and is then well suited for heterogeneous environments. Furthermore, it can back up the data of various database engines and even some operating systems data[9], in certain cases. It is then also well suited to be integrated in disaster recovery policy. In fact, HSMS-CL is an advanced backup product, capable of achieving disaster recovery tasks. But it lacks some features to pretend to be a disaster recovery tool[10].

I also saw how big servers were protected, using a mix of hardware and software technics (also described in chapter 1). And I then realized that a lot of systems and administrators (or even societies) were available to recover those systems in case of a large disaster[11].

But, astonishingly, the workstations were only protected by backups, regularly. In fact, this is not so suprising as all computer centers are protected like that. We will then concentrate our work on extending the backup possibilities available for the workstations in heterogeneous environments. And that, in order to solve the backup and the disaster recovery problematic described above.

Plan

The first chapter presents some expensive disaster recovery technics usually implemented on valuable servers. The second chapter will present the problems that one could encounter in heterogeneous environments when thinking about disaster recovery. It will present issues related to operating systems, hardware platforms, network protocols and ap-

[8]Which will be described by a chapter in this work.

[9]The registry, for example, in Window NT systems

[10]And it has not that pretention

[11]Or even in order not to suffer of a disaster at all (in operational terms, of course).

plication development. This chapter takes some advance as it already contains key concepts of the application we will develop.

The third chapter is a brief description of the backup product used where I made my training period. In fact, more than the backup product will be presented. Actually, HSMS is at first a hierarchical space management software used for the backup (and restoration) of workstations.

The fourth chapter will deal with an in-depth study of computer recovery in heterogeneous environments. This chapter will describe what a computer needs to boot and to operate. It will also present some methods that could be used to perform computer recovery operations.

The fifth chapter will detail the implementation of a computer recovery tool, well suited for heterogeneous environment.

Notes

As disaster recovery is a broad subject, I will sometimes speak about computer recovery. This term is more precise than the too generic disaster recovery. But it lacks the disaster part, that is important. Actually, we will speak about computer recovery when a disaster has occured, either a fault of the system or an external event (fire, flood,...). We must then keep in mind that a disaster is an unexpected event. So, what we describe in this work could be unsuitable for predicted events (like the upgrade of a computer).

A second note could be made on the vocabulary employed. I will often use the terms "crash" and "destroy". Those two terms where chosen because they imply the occurence of a disaster (either a fault or an external event). But they lack the notion of unexpected event. The reader should add that notion to these two terms to precisely understand this work.

Chapter 1

Disaster Recovery Technics: Presentation

Trying to do some researches on disaster recovery in heterogeneous environments would be helpless without describing what already exists. We will develop what could be used to ensure the recoverability of data in today's computer center.

But, we must keep in mind that those recovery technics have computer servers as target. So, those technics are not so usefull for the purpose we described in the introduction, because those technics are so expensive that no one can afford to implement them on every computer present in his center. Perhaps we could take some concepts from those technics, and use them in our researches.

1.1 Redundant Array of Devices

One of the fears of every computer user is to have his hard disks crashed. Besides, they are the most sensible device. It is where every data are stored. The idea behind the redundant array of devices is to have the informations duplicated. Of course, they are duplicated on separate devices to ensure optimal security.

1.1.1 Principles

Architecture

In normal mode, when data are about to be written on disk, a command is sent along with the data to the disk controller, which is in charge of really storing them. With redundant devices, the controller receives the same command as in normal mode but effectively writes the data on two or more disks. Because this system is driven by the controller, it is very independent of the hardware and software platform used. Only the attaching part of the controller and the driver has to be adapted to the platform in use.

Synchronizing & Splitting

Synchronizing[1] is the operation by which two or more disks will be mirrored (made identical). The result is a unique logical disk. When a writing operation on the logical disk is performed, the controller dispatches the data to all the disks forming the logical unit. The information are then duplicated on one or more devices.

Although the logical disk approach can give a good idea of the operational mode of these type of devices, it isn't actually true. The logical disk is in fact a real disk called the "Master". The synchronization process attaches "Slave"disks (which are also real disks) to the master disk. Then, any attempt to write on the master disk will make the data duplicated on all the slave disks.

The duplication of the data can be achieved in two ways:

serialized The controller waits for the data to be written on all disks before returning from the write call (more secure but slow, unsuited to remote solutions).

unserialized The controller writes the data on the master, puts the data destinated to the slave devices in a command queue and returns from the call. The actual writing of the redundant data is made later by processing the command queue (by a separate controller for example) (less secure, adapted to remote solutions).

Splitting[2] is the operation by which slave disks are detached from the master disk. Suppose for example that the master disk has crashed. The data are then unavailable. To recover (or rather recuperate) the data, an operator will have to split the master and slave disks and use a slave disk in place of the master disk. The crashed disk can now be replaced by a new one and the synchronization can now take place again. Note that the switched disk can now be a slave disk.

Note: the fact that a disk is a master or a slave is not a physical property, either of the disk itself or of the controller. It must be configured by the operator.

1.1.2 RAID Industry Standard

RAID stands for Redundant Array of Independent Disks[3]. It's presently one of the most commonly chosen methods to ensure data availability, integrity and recoverability. Furthermore, RAID can also be used to speed up disks access but this is off topic and we will not insist on that fact. We will also notice that the basic principle described above is not strictly followed for each level of operations but that the primary goal still applies.

---

[1] This term is taken from the vocabulary employed by EMC[2] (the company that produces boxes of redundant devices) because they are well-suited, but the definitions will be more general to avoid too specific descriptions.

[2] ibidem.

[3] In the original paper, it standed for Redundant Array of Inexpensive Disks.

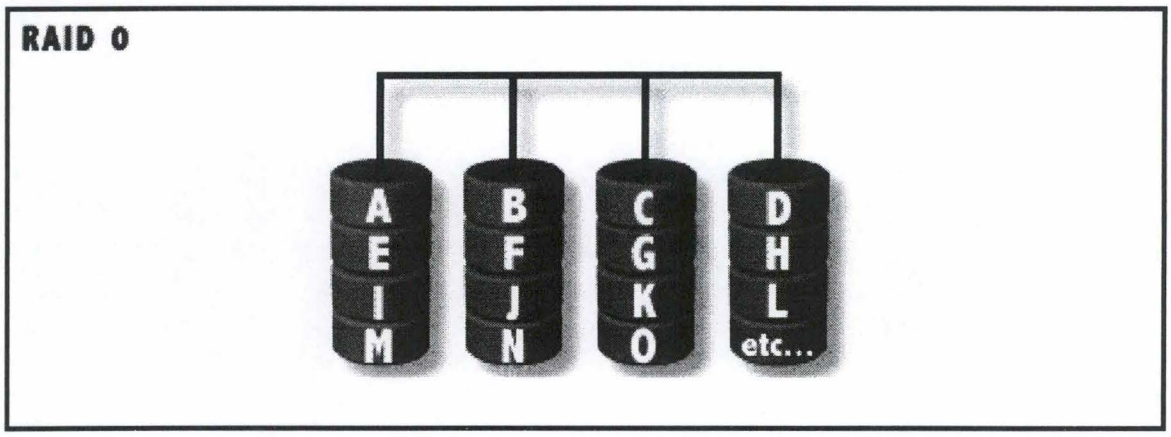

Figure 1.1: RAID Level 0: Striping

RAID standard defines 7 levels of operations, ranging from 0 to 6. We will develop them all in the order defined by standard. RAID was described for the first time in [18]. Note that RAID 0 was not a part of the original paper.

RAID Level 0: Striping

Level 0 is illustrated at figure 1.1. Data are striped across disks (i.e. consecutive sectors are distributed on 2 or more disks). For example, on a 4 disks system, sector A is put on disk 1, sector B on disk 2, sector C on disk 3, sector D on disk 4, sector E on disk 1,... This system is not fault tolerant because it doesn't permit any data redundancy. Its main preoccupation is to maximize the I/O performance as the I/O load is spread on multiple SCSI channels and drives.

However, as the load is distributed on more than one physical drive, the MTBF[4] can be enhanced. Data security is therefore better on a RAID Level 0 system that on standard SCSI system. But it can't be guaranteed as a disk failure will lead to data loss.

RAID Level 1: Mirroring and Duplexing

Level 1 is illustrated on figure 1.2. Data are accessed on mirrored pair. Each block of data is therefore redundantly stored on two disks instead of one, allowing the replacement of a trashed disk immediately after a crash. Furthermore, two concurrent reading accesses can be carried out with this level as read commands can be queued on two drives. RAID level 1 is highly similar to the principle described above. Also note that because writing operations on the two devices are processed in parallel, the system does not suffer any performance losses.

[4]Mean Time Before Failure

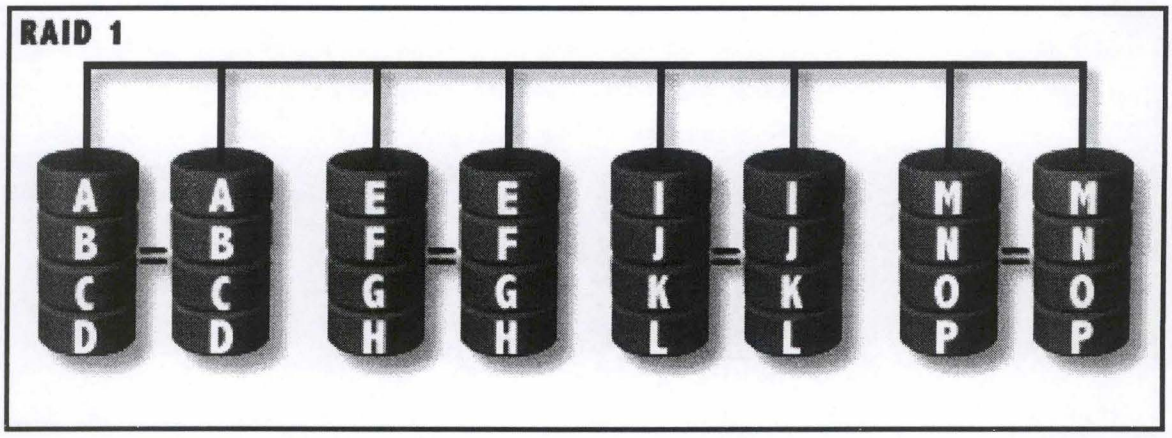

Figure 1.2: RAID Level 1: Mirroring and Duplexing

The redundancy of data also means that no rebuilding of data is needed in case of failure. This is the simplest redundant storage system but it has some drawbacks as:

- RAID Level 1 is often realized through software implementation and that leads to speed decrease[5] and no hot swapping[6] possibility.
- A bad storage space allocation due to a true redundancy of data.

RAID Level 2: Hamming Code ECC

RAID level 2[7] improves RAID level one in the way it uses storage space. Instead of creating a backup of each data block, RAID level 2 uses the Hamming ECC[8]. The ECC can correct simple error but requires multiple disks. The RAID level 2 configuration is composed of data disks and ECC disks. Each bit of data word is stored on the data disks and the Hamming code of this data word is stored, as illustrated at figure 1.3, on on an ECC disks. RAID level 2 uses the striping technic to distribute the data words on multiple disks. Losing one disks is then not a problem as the data of the others disks (the rest of the data word) and the ECC can be used to recover the lost data.

As we can see, space allocation is better here than in RAID level 1 but is still not the best, because data disks with small word size require a lot of ECC disks[9]One ECC per data word). Of course, because it can only correct simple errors, if all the disks are crashed, the lost data cannot be recovered.

[5]Due to the workstation main CPU utilization
[6]The capacity of a disk controller to allow disk to be replaced while the system is running
[7]Not currently implemented due to high costs of the additional disks
[8]Error Correcting Code
[9](

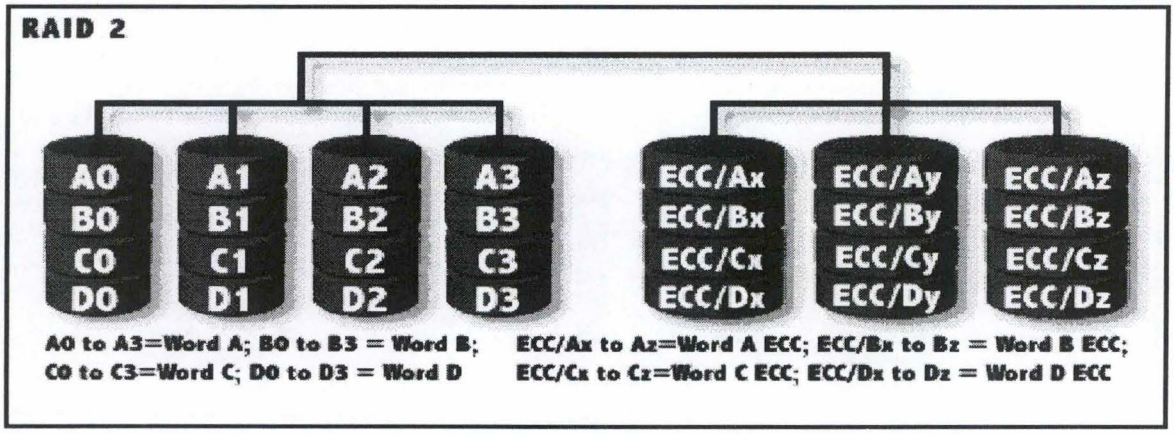

Figure 1.3: RAID Level 2: Hamming code ECC

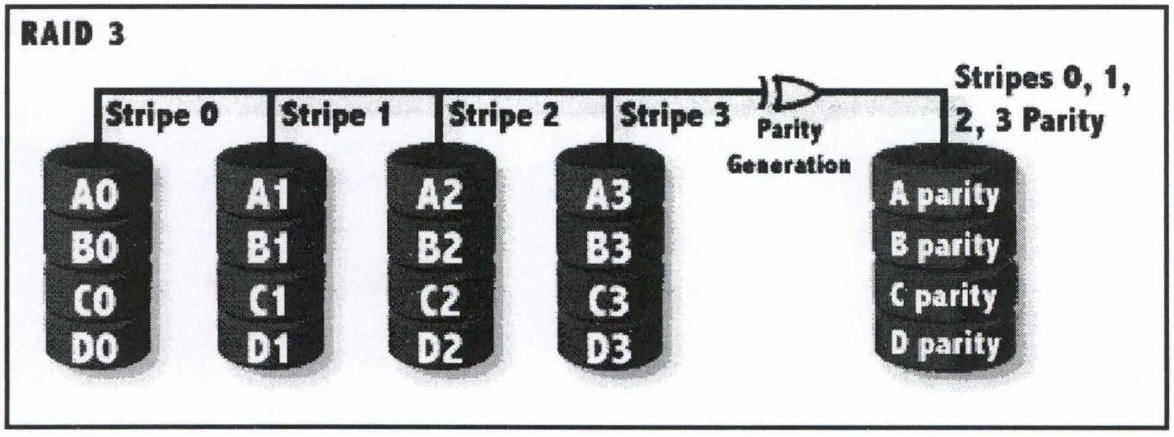

Figure 1.4: RAID Level 3: Parallel transfer with parity

RAID Level 3: Parallel Transfer With Parity

RAID level 3 employs striped data[10] and a parity disk calculated on each stripe, as illustrated in figure 1.4. The parity information is calculated on each write and checked on each read. Reconstruction of a failed disk is also possible here but only if one disk fails. As RAID level 3 uses each disk on a read or write access, no parallel access can then be done. The performances depend then of the larger of the data accessed. RAID level 3 is therefore adapted when large portion of data are to be accessed.

[10] a single block of data is distributed on multiple disks

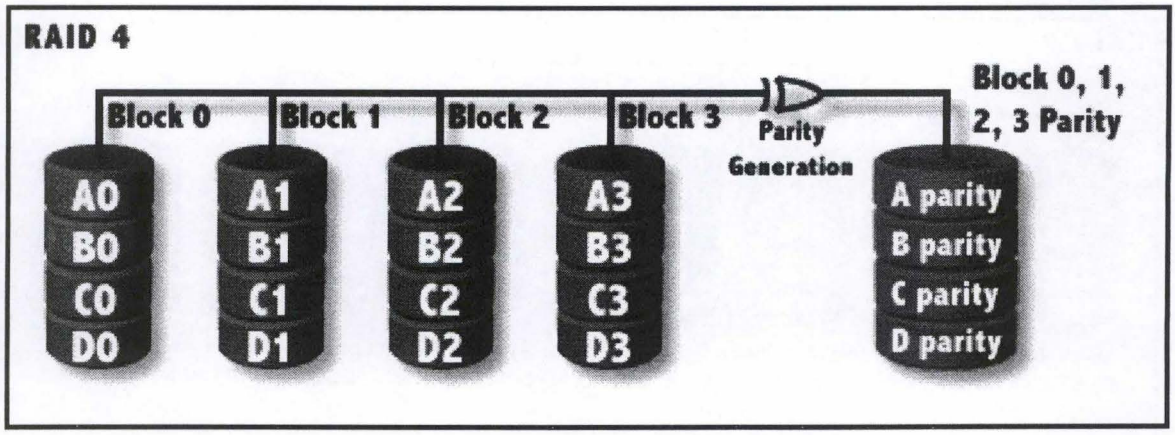

Figure 1.5: RAID Level 4: Independent data disks with shared parity disk

RAID Level 4: Independent Data Disks With Shared Parity Disk

This RAID level is similar to level 3 except that the size of the stripe is larger, in order to enhance data access performance for small reads but it leads to a much more complex parity encoding scheme. RAID level 4 is illustrated in figure 1.5. Due to the complexity of the parity encoding scheme[11], this level is not commonly implemented.

RAID Level 5: Independent Data Disks With Distributed Parity

RAID level 5 improves level 4 by distributing the parity on the data disks, spreading the workload as depicted on figure 1.6. Due to the versatility[12] of this level, this is the most commonly implemented. The drawback of this versatility is its huge difficulty to recover a failed disk. Note also that RAID level 5 is better suited for read operations than write operations.

RAID Conclusion

As we can notice, RAID systems are well versed for system performance improvements and less for data recovery. Actually, the only real redundant solution is RAID level 1. It provides true data redundancy, better read performances and no write performance losses. For our purpose, this is the RAID solution to implement[13]

Of course, RAID level 1, implemented in a single box, would be inappropriate in case of a box crash. But there are now commercial solutions, where data are mirrored through a fiber optic link. That provides remote mirror storage, efficient even in case of a local disaster.

[11] And then the bottleneck of the parity disk

[12] Transaction performance is the best of all RAID levels and the parity space allocation is also the best

[13] Provided it is implemented in hardware and hot swapping is available.

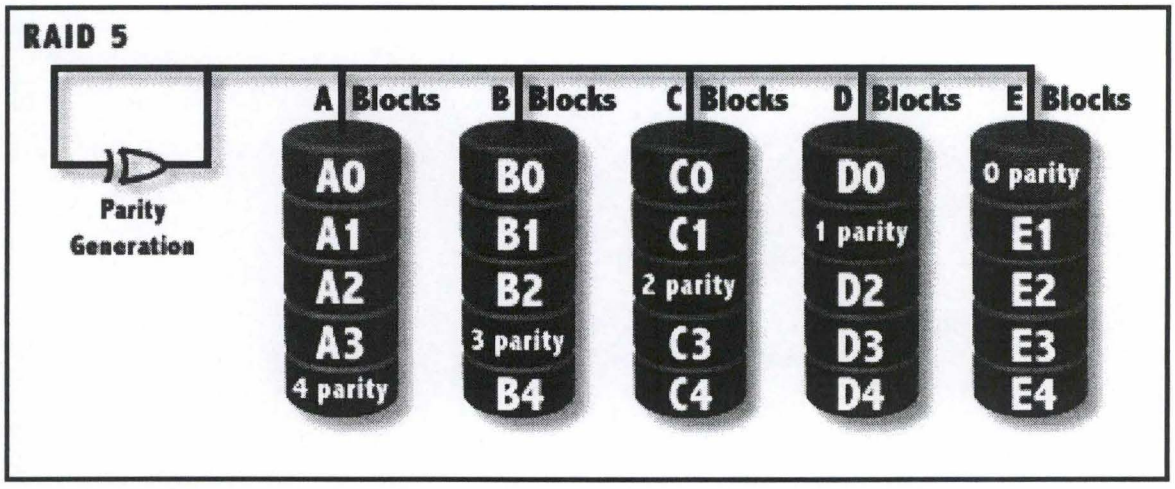

Figure 1.6: RAID Level 5: Independent data disks with distributed parity

But we have to be honest, and although RAID systems are often used, they cannot provide a 100% recoverability of data. If, for example, the file system is corrupted, the OS will not be able to access any files, even if the destroyed disk is replaced and recovered. This is an important point that we already mentioned in the introduction. Actually, RAID protects the data on disks but cannot guarantee that the data are those we want. If a computer or an application behaves badly and it begins to corrupt data, RAID only ensures that the corrupted data will be available, just in case of a disk malfunction.

1.2 Process Migration and the Like

The preceding method is secure in the case of a disk crash but the failure could occur everywhere, in any parts of the computer. So it can only be considered as a "data securing process". To achieve almost 24h/24h operational functioning, a system known as process migration would be useful. The bases of this system are a computer cluster, a crash detection system and a process migration system.

1.2.1 Computers Cluster

A computer cluster is a group of networked computers that put their resources in common to achieve one or more operational tasks. Setting up a cluster is a complicated task as it requires specialized softwares and a really good network architecture (essentially to achieve good performances).

To permit the development of parallel process, the specialized softwares provide the necessary API to perform I/O operations and process synchronization (shared memory,

message queues and semaphores for example) between remote nodes[14].

Despite this system was developed to achieve high performance in complicated and highly parallel computation tasks, it can also be used for process migration [15]. This is what we will tackle shortly.

1.2.2 Crash Detection

Crash detection is a main functionality of the system. Actually, the migration of one process cannot be scheduled but has to be decided in real time, and in our case, when a failure occurs.

One problem is that certain events, such as CPU failure, could prevent the detection routine to run. This is why the state of a machine is often controlled by another node in the cluster (a control node) and it is that node that decides when a process migration occurs.

The presence of a control node imposes the presence of a supplementary computer in the cluster. And because this node has to monitor other nodes, the number of exchanged control packets can occupy a lot of the network bandwidth. The way the monitor controls the cluster is then a critical issue that must be well implemented[16].

Furthermore, the crash of the control node can affect the operations of the cluster and suppress all process migration possibilities. This is why control node could be duplicated too.

1.2.3 Process Migration

Another problem is that a running process works on data, and has its own environment. So if the control node decides that a process must migrate, all the environment and the data have to be transfered too. This is a difficult task and there is no simple solution. But if a system crash occurs and if one or more systems can take back all the tasks of the crashed computers, the cluster provides uninterrupted operations.

In fact, today's clusters, when used for the purpose of uninterrupted operations[17] don't migrate processes but allocate dynamically one computer for each task[18]. The monitor is then really a process allocator. The main difference between process allocation and process migration is that when a failure occurs, the task currently running isn't preserved. The task is aborted[19]. One should note that the task could be resumed on another node (from

[14] A computer member of the cluster

[15] We will see that process migration is very theoretical

[16] In such a way that exchanged messages don't perturb the cluster primary functions

[17] Called fail-over cluster

[18] A task is an operation that must be processed by the cluster

[19] In case of process migration, the task could be theoretically handled by another computer in the cluster but nothing says that the process could really be migrated on another node (due to a critical failure of the computer running the task)

the beginning). But it is far different from a process migration where the task should be resumed exactly where it was interrupted[20].

1.2.4 Practical Example: Distributed Web Servers

As e-commerce applications appear, web servers have to be more and more robust against various incident. Furthermore, they have to handle more and more requests as the success of the web is a fact nowadays.

In the beginning, when the need of distributed servers[21] appeared, a simple solution to distribute the connections amongst the different nodes was to review the way DNS[22] worked. And that to implement what is now called a RR[23] DNS. With a RR DNS, when an address resolution request arrives, the DNS server responds with the IP address of one node in the cluster. When another request comes, the server responds with the IP address of the next node in the cluster. In fact, the server has for a single FQDN[24] a pool of m IP addresses corresponding to each of the m node of the cluster. At each request[25], the server cycles in the pool of IP addresses to deliver another one.

Of course, this system can "allocate"a task to a failed node, but the system is, in some measure, still usable. We will not discuss the performance of a RR DNS (this is not our scope) but we see that the fact that a request can be allocated to a wrong node is a real problem. A possible improvement of this system could be that the RR DNS monitors all the nodes and responds only with IP addresses of working nodes[26], avoiding nodes that are unavailable at that moment.

Of course, if a node fails when serving a request, that request will be aborted, but the same request allocated on a different node will succeed[27].

The reader interested in Distributed Web Servers or fail-over cluster can consult [9, 10, 22] for informations and pratical implementation.

1.2.5 Conclusion

Fail-over clusters and process migration are certainly interesting approaches. But the price to pay to implement such configurations is heavy. The main target of those solutions is the server market. The workstations are therefore not protected by those solutions. The workstations are however primordial in computer centers[28].

[20] Or nearly when check-points are used in the application

[21] Or a cluster of servers

[22] Domain Name Server

[23] Round-Robin

[24] Fully Qualified Domain Name

[25] Or all the N-requests to accomodate with the power of each node

[26] In our case, nodes able to serve a request

[27] Provided that the monitor knows that the node is unavailable. And that is the problem of every fail-over cluster.

[28] What about a call-center with fully functional servers but without any workstations running?

Furthermore, in heterogeneous environments, those systems rapidly become complicated. And the day-to-day administrative tasks would become so important that it is almost impossible to implement in pratice.

1.3 Human and Building Resources Management

Why are those human and building resources so important when we should speak about disaster recovery? Simply because that's an important part of a disaster recovery procedure. Actually, a disaster can be *more* than a computer crash, it could be the total destruction of a site.

Without any resources, a company cannot be operational with solely backups or whatever computer recovery systems. That's why any company head should put management strategies in place to cope with disasters.

The questions that a manager has to cope with are for example:

- Who do we have to contact?
- What site will be used during the crisis (cold backup, hot backup site)?
- Which suppliers can provide all what I need?
- Who must come to work and who must not?
- ...

It is now important that the reader keeps in mind that a disaster can be more than a disk crash. A disaster can touch just as easily the smallest parts of a computer as the biggest operational site.

Examples of such planning[29] can be found in [15] and [31]

1.4 Conclusion

As we can see, the securing process against data loss or corruption of computers can imply a lot of different technics[30]. From technical one to strategic one. And the price to pay, in technical and human resources is a major issue of every disaster recovery procedure.

Furthermore, we have seen that some recovery systems could fail. It is why a disaster recovery procedure has to consider the good old backup solutions. They are cheap and they have proved their efficiency. But for some recovery operations, they are inadequate as they only manage to recover files and, sometimes, application specific data. The other

[29] Called disaster recovery plan or business continuity plan

[30] All systems weren't covered here

great advantage of "simple" backup systems is that they are available for a large majority of the computers used nowadays.

Actually, this work deals with disaster recovery in heterogeneous environments. So, I will develop a disaster recovery application prototype that extends the possibilities of known backup solutions to achieve disaster recovery for heterogeneous computer environments (and we will call that computer recovery). For that purpose, the chapter 4 will describe what we should backup and restore on a computer and the chapter 5 will describe the implementation of the prototype.

Chapter 2

Heterogeneous Environment Implications

Before proceeding to our computer recovery researches, we are going to study the different implications of an heterogeneous environment. And that, to provide a strong analysis of computer recovery in heterogeneous environments. Actually, focusing on a particular platform could be very risky and could provide false results.

Furthermore, we will also concentrate ourself on application developpement in heterogeneous environments. And that, because the porting job can be a major and costly issue. So if we want to achieve a cheap solution suitable for a large variety of hardware and software platforms, we have to consider that question too.

2.1 Computers & Operating Systems

2.1.1 Computers

Computers and computer parts can be very different while providing almost always the same services. The Intel i386 platform is a very common one and is the most present platform in the home computer market. They can be used to solve complex problems as a Sparc can do it. But they are incompatible.

Although they use very different architectures, they read their data on disks, they can be connected to other computers across multiple networks, they display their data on similar screen, they pick up user's input from keyboard, etc.

In conclusion, they are just functionally equivalent (although one can be more convenient than the other for certain tasks) and that is an important concept. Disaster recovery operation have to brought back a computer or a system to its functional state. So, a first step while studying DR procedure is to know how a computer can be functional. What are the different parts of a computer, and that in a very general manner.

It does not induce that we can just pick a platform and be sure that the approach resulting of that platform will be applicable to another one. Just because we risk missing

some important parts. For example, Apple computers boot a part of their operating system from ROM while i386 computers boot their operating systems entirely from disk. Taking the Apple as a reference could lead to think that only specific parts of the operating system have to be restored in case of a total crash when it is not true for other platforms.

To cope with that, we have to define a generic computer, a virtual one that has all the components of every-day life computers without really existing. While it is not guaranteed that this approach will be error free, it is the best solution in order to be as complete as possible.

2.1.2 Operating Systems

Most principles that applied just above apply to operating systems too. In functional terms, they are all equivalent (except for some very specialized one such as embedded operating systems).

An operating system provides I/O API (or system calls), i.e. a way to communicate uniformly with various hardware extension and users. Between OS, the APIs are all different but the base of each one is the same (It is especially true in the case of UNIX systems where almost all system calls are the same)

A virtual approach will be also envisaged here. We will describe a generic operating system that includes all the needed functionalities. We should then be able to study real disaster recovery functions.

2.2 Network

No one could imagine a disaster recovery software running on a stand-alone machine. At least, backups of important data are made across a network and they are stocked on one or more remote servers. Although it is sure that a part of a disaster recovery software should be running on each protected computer, it is important to decentralize the backup of the data, at least to protect it from a local disaster (localized fire for example).

This is is certainly the most simple issue because an heterogeneous communication channel has already been implemented. Almost all operating systems implement a TCP/IP stack, so virtually all networked computers can communicate with each other.

No virtual approach has to be studied here, we will simply base our research on a real universal protocol: TCP/IP.

2.3 Software Architecture

Disaster recovery software can be implemented in various ways but to achieve a comprehensive architecture suitable for heterogeneous environments, we actually have to define a strong and efficient architecture. We will try to directly address this problem here and

that architecture will be implemented in chapters 9 and 10. Note that this is not a detailed architecture and we will only determine a very global architecture.

2.3.1 Preliminary Idea

Before we begin, we will ask us the following questions:

1. **Where are the DR data collected or recovered?** This question is easy. The data are collected or recovered locally. Trying to access computer components remotely is a very difficult issue because it would require the implementation of a distributed I/O API on every different node, it would be slow and the network would induce more failure points. The "Client Part"of the DR software has to run on the client. We'll see that this particular software can be very well suited for heterogeneous platforms.

2. **How are the data saved?** When data are collected or accessed for recovery, they must be easily accessible and the way they are stocked must be portable. The aim of the chosen solution is to put the complexity of the heterogeneous environment in the "Data collector[1]". The client software has then a standard fashion of storing and accessing its data.

3. **Where are the DR operation launched from?** DR operations are of course local to a computer because they concern that particular node. Furthermore, the administrator or the operator of the node is more informed of the change occurred on that machine. But when critical operations must be accomplished, a centralized processing is always a good idea and allows a consistent DR policy. Without being too pessimistic, a computer security policy cannot rely only on the goodwill of hundreds of local administrators/operators (although most of them did a good job). A DR software must then permit local and remote operations. But as we previously saw, remote recovery is not a good idea. So the software must at least provide remote processing of the DR data collect.

4. **Where are the DR data stocked?** We have already spoken about the way they were stored but we must also know where they should be stored. Remotely is more secure because a local disaster will not affect them. But if each component of the data must directly be saved remotely, that can be very slow and increase the probability of a crash. So the best solution can be to create a "data container"on the local computer and to store it on a remote server just after it is filled. The creation of the container is then very fast and its remote backup secure.

2.3.2 Application Design

Without being too accurate, DR operations involve data savings or data backups. One feature of chapter 6 will be to cover which data will be collected, saved and recovered, and

[1]Here, the backup solution

how that will be achieved.

Here, we will not describe the data handling but how the application should be constructed. As we must provide an application that can run in an heterogeneous environment, we must choose a method where it will be easy to construct the executable for various platform. That's we are going to analyze now.

Platform Specific Static Executable

This is one of the most used approaches. Imagine an application that should run on various (incompatible) platforms. The choice of the programming language is of course an important issue. An assembler version will be very expensive by nature because assembly is very specific to a particular platform and this is a bad choice. A C (C++) version would be more portable as this language was defined with portability in mind. But the portability factor depends on more than the language, it also depends on the operating system.

While porting a program like:

```
#include <stdlib.h>
#include <stdio.h>

int main() {
    printf("Hello Universe!");
    return 0;
}
```

would be obvious, the following would be more problematic:

```
#include<stdio.h>
#include<stdlib.h>

/* Not portable */
#include<linux/hdreg.h>
#include<sys/ioctl.h>
/* --- */

int main() {
    struct hd_geometry geo;
    int descr;

/* Portable but what is the meaning ??? */
    descr = open("/dev/hda", O_RDONLY)
/* --- */

/* Not portable */
```

```
    ioctl(descr, HDIO_GETGEO, &geo);
/* --- */
    close(descr)
    return 0;
}
```

As we can see, getting the geometry of a hard disk is a very specific issue and even seemingly portable code can have some particularities.

While the open() call is common in standard C libraries, opening a "/dev/hda" file has no meaning on a NT computer; but on a linux box, it means that we want read/write/special access on the first IDE hard disk.

A second issue is the ioctl() system call. Although it exists on all Unix platforms, this one is specific to linux and isn't portable. This program should then be completely rewritten in order to run reliably on other platforms.

A solution to this problem can be either to use commonly used pre-processor functions like "#define, #ifdef, #endif"which tend to make the source code unreadable or to make different source trees for each platforms, which is cost effective.

Another programming technique would be to virtualize the hardware by designing virtual functions (not to confuse with C++ virtual function) that call specific modules dependent of the platform being used. We could design the second application like this:

```
Get_geo function specification:

int Get_geo(int n, struct my_geometry geo);
-> Get the geometry of disk number n and put it in our own defined structure
-> geo.


Module: geo_linux.c
Export: Get_geo function

Module: geo_nt.c
Export: Get_geo function

...

main.c:

#include <stdio.h>
#include <stdlib.h>
#include <mydef.h> /* Include the definition of our my_geometry structure */

int main() {
```

```
    struct my_geometry geo;
    Get_geo(1, &geo)
    return 0;
}
```

By defining a good project file (a file that describes how a project should be built), the correct object module will be linked with the main object module, resulting in a specific executable for the desired platform. This is certainly a good choice for static software with low variation of functionalities, but it is not true for DR softwares. Every day, new hardware appears on the market and the operating systems have to adapt to that new hardware. This is why DR products should be very adaptable to new OS/Hardware features. Furthermore, clients who buy DR solutions expect them to work for more than 3 or 6 months.

Dynamic Linking Executable

Be warned that there is no 'magic' in that solution. The written code has still to be compiled for each platform that has to be supported.

Description The concept pending in this solution is to differentiate the portable and non-portable part of a software while achieving a high degree of adaptability.

The solution is the so-called "shared-objects "(which are also called Dynamic Linking Library or DLL in short by some Redmond's designers). This approach is used more and more nowadays to adapt software rapidly. One example is Netscape, that uses that technology to support their plug-ins.

Another advantage of that solution is that when a well-documented interface with the main application is provided, individuals can also add functionalities adapted to their needs. Just imagine that each surfer on the Internet should upgrade they Netscape browser anytime a new functionality appears, it would make a lot of traffic on the network and Netscape should certainly consider buying new servers.

Portable Part: Main Application Note: The reader should read dynamic module when she/he reads module.

As mentionned, the portable part must provide a well documented interface (or API) to the shared-objects in order to be efficient. This is the task of the portable part of the software.

In our case, what do we have to furnish to the specific modules?

1. **A method to allow module registration:** when loading the module, the main application has to call an init function specific to that module. The module has then to call a registration function of the API to inform the main application of which

functions have to be called in order to accomplish specific operations (recovery, DR data collect, informations,...).

2. **A uniform and portable method to store and access data:** as the main application has no knowledge of what type of data should be stored or accessed and that the final processing of the data is handled by the main application, it has to furnish a general API that modules will call to store or access their data.

3. **A uniform and portable method to process module specific configuration options:** as specific modules are parts of the main application, they should not use specific configuration switches or files. Options must therefore be passed by the main module and processed by the involved modules.

Furthermore, the main application has to handle where and how to store and recall the DR data repository, and which modules have to be called and when.

Non-Portable Part: Specific Modules The specific modules handle the real DR data collect. Although we are covering heterogeneous environment, we will develop a concrete example based on the Linux operating system in appendix A. Thanks to the API furnished by the main application, the only modules have to cope with the private structure they will handle and the functions they will use to access their DR data.

This way, the developer will enjoy all the advantages of the dynamic linking executable and shared-objects.

Dynamic Linking: How it works During the compilation, the source code is transformed to a so-called object file, containing the platform specific binary code, symbols information and various data.

For each function or variable that will be used by the program, a symbolic name is assigned and that symbol contains a reference to the code or the data it represents. When an application is statically linked, the linker puts all the object files together and resolves the undefined symbols in the different objects. The application can then be started (possibly after the loader has done some relocations). An illustration of the compilation-linking process is depicted in figure 2.1 : the main.c object define the main() function that calls the functions fun1 and fun2 declared and defined respectively in object1.c and object2.c.

In the case of a dynamically linked executable, the two objects (object1 and object2) will be compiled as shared objects. The main difference is that the shared objects are compiled and linked in such a way that the code can be executed from anywhere in memory (PIC or Position Independent Code) and that the symbol table can be exported.

The compilation of the main module occurs as usual but when the linking is performed, a small code is added and the entry point of the application points to it. This code is responsible for loading the symbol table from the shared object and updating the local (application) symbol table. After some relocations, the application can be executed.

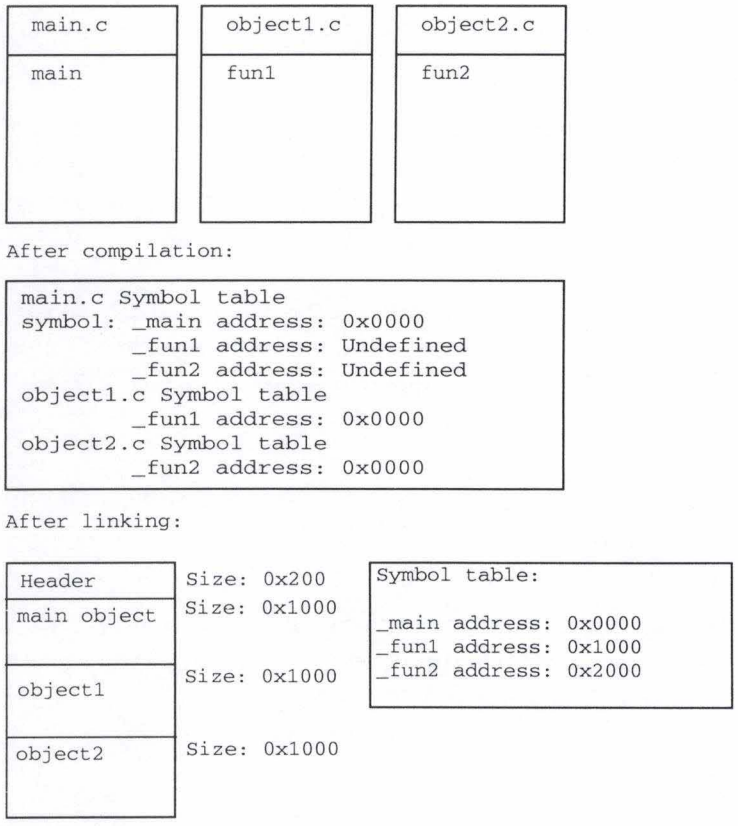

Figure 2.1: Compilation and Linking (Static)

Although the base of the dynamic linking is explained, this is not actually what we want because the main module has to be recompiled each time we want to add new functionalities in the shared objects and each time we want to add new shared objects. The next point will explain On-demand dynamic linking.

On-Demand Dynamic Linking Actually, we want a system where we just act as the little init code we "saw" in the above explanation. As we are lucky, most modern operating systems furnish the necessary API needed to accomplish this.

In order to make use of the library, the system API furnishes a dl_open() function that act as a library (and symbol table) loader. Then, we have to use another system call, dl_sym() – that uses the symbol table –, to obtain the address (a pointer) pointing to the desired shared binary code.

To illustrate this, we will analyze a simple example:

```
Shared module 1 (module1.c -> module1.so) :

#include <stdio.h>
#include <stdlib.h>
```

```
static void entry_code() {
    printf("Hello world!\n");
}
```

Shared Module 2 (module2.c -> module2.so) :

```
#include <stdio.h>
#include <stdlib.h>

static void entry_code() {
    printf("Hello universe!\n");
}
```

Main application object (example.c -> example) :

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int main(int argc, char **argv) {

    void *shared_lib;
    void (*module_function)();

    argc--;
    if (argc) {
/*
    Open the module and resolve all undefined reference
    now (RTLD_NOW)
*/
shared_lib=dlopen("./module1.so", RTLD_NOW);
    } else {
/*
    Open the module and resolve all undefined reference
    now (RTLD_NOW)
*/
shared_lib=dlopen("./module2.so", RTLD_NOW);
    }
    /*
Get the pointer to the entry_point function in the opened
module.
    */
```

```
    module_function=dlsym(shared_lib, "entry_code");
    (*module_function)();
    dlclose(shared_lib);
}

On linux, the three module can be compiled with the following command:

gcc --shared -fPIC -o module1.so module1.c
gcc --shared -fPIC -o module2.so module2.c
gcc -o example example.c -ldl
```

The example application, when launched without argument will print "Hello universe!"and when executed with at least on argument : "Hello world!".

If we want to change what is to be printed on the screen, changing the module would be sufficient, and no recompilations of the main object have to be done.

We have now a good template on how our DR software will be build. The main application will provide the API that modules will use to access and store their data. And modules will perform real DR operations.

Chapter 3

Backup Software: HSMS & HSMS-CL

In this chapter, we will present a backup software available in heterogeneous environments. It'll be used by our software to achieve computer recovery operations. HSMS & HSMS-CL are products of Siemens AG.

3.1 HSMS: Presentation

Running on top of Siemens BS2000 mainframes, HSMS is a Hierarchical Storage Management Software that permits backup, restoration, migration and data transfer of files and job variables.

Those tasks are facilitated by the fact that HSMS uses a three level hierarchy to achieve its tasks:

At the first level (which is the processing level - S0), we found high-speed disks with short access times. Those disks are managed by DMS, the basic BS2000 data handling system. It is also at that level that data are processed by HSMS.

At the second level (S1), we found high capacity disks that have longer access time and lower throughput (cheaper disks). The data on those disks are managed by HSMS (But HSMS files are still managed by DMS). The S1-level is used for data migration and backup.

The last level (S2) consists of magnetic tapes, magnetic tape cartridges and optical disks which are cheaper than disks. The access time of this level is not an important issue. This level can be used for migration, backup, archival and data transfer between BS2000.

Using those three levels, HSMS is able to manage data to maximize the processing power and to minimize the storage costs of BS2000 mainframes (As seen on figure 3.1). One should note that backups are not restricted to files but concern also the catalog entries. Either backups concern only the catalog entries either they concern the catalog entries and the data.

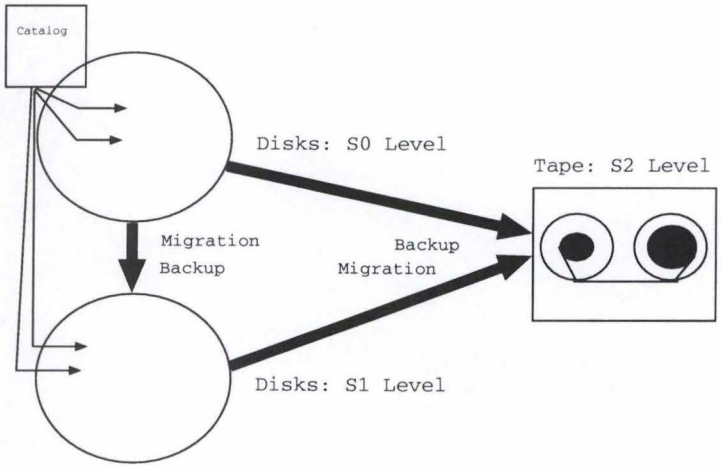

Figure 3.1: Backup & Migration with HSMS

Another remark concerns the backup of migrated data. Once they are recovered, they are put on S0 volumes[1] wiping out the catalog entry pointing to that file on S1 or S2 volumes. This is due to the fact that the catalog entries of migrated files are kept in the catalog of the S0 volumes (To allow transparent access to migrated files).

3.1.1 HSMS Archives

Perhaps this is a bit strange in regard to the backup and archival theory but HSMS handles *archives* for archival and backup. An archive is the basic management unit of HSMS. Each one consists of:

- The definition of its attributes.
- The archive directory used for managing files, job variables and volumes saved in this archive.
- The volumes and save files containing the saved data.

One particularity of HSMS is that it uses separate archive for backup, archival and migration. Because HSMS can also backup and archive nodes data, there are also two supplementary archives for node backup and archival.

3.2 HSMS-SV

HSMS can work automatically if the administrator has configured it properly but HSMS operations can also be launched through HSMS statement (command).

[1] S0 volumes are used for the processing level, so S0 volumes are usually disks

The problem of the node backup or archival, is that those operations would require each user to log on a BS2000 to backup or archive their data. To simplify this work, a client-server was developed to permit the execution of HSMS operations from remote hosts.

HSMS-SV is the server part of this architecture and is responsible for launching HSMS statement. HSMS-SV is in fact a sort of translator between the HSMS protocol and the HSMS statements and is not really a big piece of code.

3.3 HSMS-CL

Historically, HSMS used to (and can still) access remote hosts data through NFS. As it was slow, a protocol was defined and HSMS-CL is now used to handle backup and archival of remote nodes in connection with HSMS (through HSMS-SV). The term used to describe a node on which HSMS-CL is installed and running is active client. On the opposite, we would speak about passive client (a client where NFS is installed).

HSMS-CL is composed of:

A daemon which is responsible for communicating with HSMS to parse the node file system and to send or receive the (compressed) data.

Client applications which are used to execute HSMS operations by a node operator.

3.4 Operation Mode

3.4.1 Centralized Operation

In this mode, the administrator of the BS2000 has to launch remote nodes backup (by the mean of HSMS statement or scheduled batches).

An example is depicted in figure 3.2 (Backup). After the administrator has launched the HSMS statement, HSMS "forks"an HSMS subtask that contacts the daemon part of HSMS-CL on the remote node and sends it a request with the operation parameters.

The HSMS-CL daemon then spawns a process that parses the local file system to find files matching the HSMS request. Paths to matching files are sent to HSMS that "forks"a subtask.

That subtask is the real operational task. It is responsible for retrieving (sending) files that have to be backed up or to archived (restored). The exchange of data is handled by a new operation specific process of the HSMS-CL daemon and the BS2000 HSMS subtask.

For the sake of performance, parsing and processing are done in parallel when a lot of files are being processed.

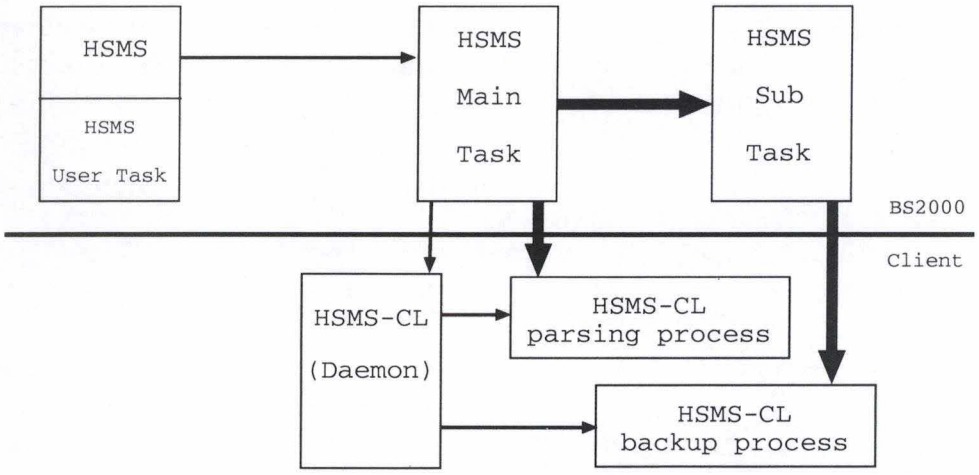

Figure 3.2: Centralized Backup

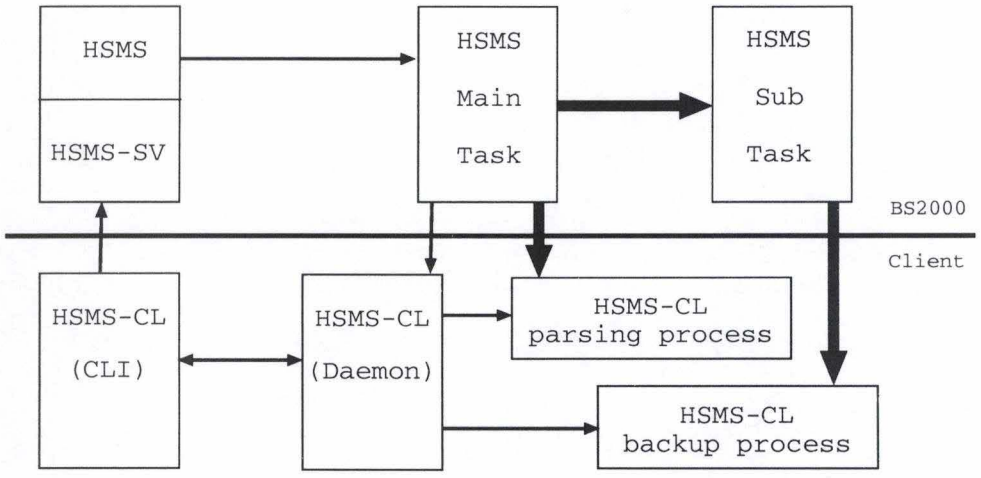

Figure 3.3: Decentralized Backup

3.4.2 Decentralized Backup

If we take a glance at figure 3.3, we see that the decentralized backup is just as a centralized backup except that it is a HSMS-CL client that contacts HSMS (through HSMS-SV) to query the execution of HSMS operations.

3.5 HSMS-CL and Application Specific Modules

Nowadays, it is often mandatory to backup or archive more than files. Users are working on databases and to maintain their consistency, there is more to do than a simple backup

of the database files. Some operations[2] have to be done on the databases prior to the wanted operation.

That is why the developers of HSMS introduced the *ASM*. When the backup, archival or restoration of an *ASM protected directory* has to be done, the control is passed to the configured module that will handle the parsing for that directory (backup and archival) and the operation (backup and archival) to accomplish. For restoration purpose, the call to the appropriate module is done through a file type definition (*Packet type*) furnished by HSMS and not through an analyze of the full path of the restored file.

3.6 HSMS-CL and Computer Recovery

As far as files or databases (data) are concerned, HSMS-CL can cope with disaster recovery. It means that destroyed data can be recovered, but only when the system is operational.

It means that the computer must be running to restore the deleted or corrupted files or databases. But when the operating system (or a database application) is corrupted, HSMS does not come to help and an administrator has to recover the computer manually. Furthermore, as HSMS-CL can be integrated in a highly heterogeneous environment, procedures for a lot of different platforms could be very difficult and cost effective to establish.

Although some tests have been carried out, no general procedures have been validated. Real disaster recovery is then not supported and HSMS-CL stays what it is: a backup/archival software. The question was to know how to add disaster recovery possibilities into (or aside) HSMS-CL without breaking the *heterogeneous* characteristic of HSMS-CL.

Due to those constraints, it is evident that an in-depth coverage of disaster recovery and heterogeneous environment had to be realized. As heterogeneous environment had already been studied, we will study (in the next chapter) what disaster recovery implies and how to cope with it.

The final aim of the work is to have a real disaster recovery software able to integrate itself in an heterogenous environment and to operate in or with HSMS-CL and HSMS.

[2]Because the files can be backed up directly, some locking and consistency problems have to be resolved, first.

Chapter 4

Computer Recovery

As we said, disaster recovery implies that a defective computer is brought back to its functional state. So, we have to know how a computer bring itself to that state. I.e. how it passes from the switched off state to the fully functional state and that without even considering any disaster, at first.

Before considering the virtual computer and the virtual OS, we will see how a computer really bootstraps, because a virtual approach has to base itself on something real. The virtual computer (and OS) description should then avoid any "particularities" of the real platform (operating system) studied.

The choice of the computer and OS studied is of course a difficult subject as there are a lot of different platforms. The possible best approach is to study more than one computer and operating system.

That study is particularly difficult for computers as they are often the results of private companies researches. On the other hand, a lot of the operating system theories were developed in the universities (BSD, for example, was a UNIX operating system developed at the University of Berkeley) and so, those theories are largely available.

As private companies are somewhat egoistical to provide free information to the public audience (other companies can afford to pay to obtain the right information) on the inner working of their products, studying a hardware platform is difficult. Luckily, there is a well-defined and well-documented platform, the i386 PC compatible computer. I know this is not the "best" computer right out there but it can be very easily studied as it was (and still is) largely documented in books.

4.1 Platform Study: The Intel i386 Case

4.1.1 Foreword

Before describing how it works, we will introduce the memory model used on that platform. In fact, we must distinguish two operational modes as described by Intel:

- The real mode (no multi-tasking possibilities)

- The protected mode (multi-tasking possibilities)

In real mode, the memory is accessed through a segment:offset pair which are both 16 bits unsigned integers. Each segment is then 65536 bytes long. We could then theoretically access 4GB of memory but it isn't actually true due to the historic development of those platforms.

At the beginning the processor could only access 1MB of memory, using a 20 bits linear address. The segment:offset address pair was converted into a linear address by shifting the segment value of 4 bits to the left, obtaining a 20 bits unsigned integer and by adding the offset value to the resulting 20 bits "segment" (In fact, we could see the segment as a 20 bit value where the four low-order bits are hidden to the programmer and are always set to null).

This is why the real mode of the Intel i386 platform can only address 1MB of memory and why the FFFF:0000 address is equivalent to FFF0:00F0. With that in mind, we see that the top address of the i386 in real mode is FFFF:000F.

In protected mode, this is fundamentally different as the system programmer has a full control over the memory model used: segmented, paginated or linear. But for compatibility purpose, the memory is still accessed through a segment:offset pair. But now, the offset is a 32 bits unsigned integer value. One segment can then be as large as 4GB (but this is not mandatory as the segment size, in protected mode, is not fixed).

For a full description of the i386 protected mode memory models, the reader can consult [11, 12, 13]

4.1.2 Computer Bootstrapping

When an i386 is powered up, the processor is put into real mode and begin to execute code (which is often an unconditional jump to another location in memory) at the FFFF:0000 address. This code comes from a non-volatile memory (ROM or NVRAM in computer acronyms) mapped to RAM and is called the BIOS (Basic Input Output System).

This code is responsible for initializing the computer to be able to boot operating systems (i.e. initializing the video card, checking the system memory, initializing the disk controller, setting interrupt-handler and basic services to access hard disks and floppies,...) from floppies, CD-ROM or, more generally, hard disks.

When all this init phase is terminated, the BIOS loads the first sector (512 bytes) of the floppy disk (if present) or of the first hard disk at the 0000:7C00 address and passes the execution control to that address by means of an unconditional jump (it also checks that the executable marker is correct, see below).

Offset	Nature	Size
000h	Executable Code	<1BEh
1BEh	1st partition table entry	16 Bytes
1CEh	2nd partition table entry	16 Bytes
1DEh	3rd partition table entry	16 Bytes
1EEh	4th partition table entry	16 Bytes
1FEh	Executable marker (AA55h)	2 bytes

Figure 4.1: Hard Disk: MBR Format

Offset	Nature	Size
000h	Executable Code (and OS specific data)	<1FEh
1FEh	Executable marker (AA55h)	2 bytes

Figure 4.2: Floppy Disk: Boot Sector Format

MBR, Partitions and Boot Sectors

As the BIOS accesses disks through a head/cylinder/sector triple (although this is not always the way disk controllers-OS interactions work, SCSI and IDE for example) the first sector is situated on the head 0, cylinder 0, sector 1 (sectors begin at 1).

This first sector is called the MBR for hard disks and the boot sector for floppy disks. The structure of this sector is well defined (although it is not absolutely mandatory to follow that structure) and is depicted on figures 4.1.2 (hard disk) and 4.2 (floppy disk).

The hard disk MBR contains four partition table entries. A partition is in fact a logical disk and it permits a logical organization of large physical disks and protection of logical entities against file-system corruption (for example).

As we can see on figure 4.2, a floppy disk boot sector does not contain partition tables entries by default but this is a standard and nothing prevents someone to create partition on floppy disks although no current operating system would implement means of recognition of those theoretical partitions (and, considering the floppy disks capacity, it would be foolish to divide a floppy into partitions).

Each partition table entry is formated as explained on figure 4.3. The only tricky part of this structure is what is called the cylinder-sector encoding: the high-order 8 bits of the structure represent the low-order 8 bits of the cylinder, the low-order 6 bits, the 6 bits of the sector and the two resulting bits (bits 7 and 6), the high-order 2 bits of the cylinder.

In CHS (cylinder/head/sector) addressing, the cylinder is therefore represented by a 10 bits unsigned integer value and the sector by a 6 bits unsigned integer value, what limits a disk to:

Offset	Nature	Size
00h	Partition state (00h non-active, 80h active)	1 byte
01h	Begin of partition: Head	1 byte
02h	Begin of partition: cylinder-sector	1 word (2 bytes)
04h	Type of partition	1 byte
05h	End of partition: Head	1 byte
06h	End of partition: cylinder-sector	1 word (2 bytes)
08h	Number of sectors between the MBR and the 1st sector of the partition	4 bytes
0Ch	Number of sectors in the partition	4 bytes

Figure 4.3: Partition Table Entry

- 256 heads
- 1024 cylinders
- 64 sectors

As each sector is generally (physically) 512 bytes long, the largest disk that an i386 could access is 8.5GB in size. Quite odd as modern operating system are able to manage disks larger than this limit.

In fact, CHS addressing is only used by the BIOS (and still now for compatibility reasons) and not by the operating system (except some old ones like MS-DOS, but it suffered of a lower limit due to BIOS system call limits). The way it accesses sectors is known as linear addressing. I.e. the operating system sees each sector on the disks as if they were consecutive and the CHS addressing scheme does not matter at all.

Concerning the partition table entry, there is no problem seeing that the information is stored redundantly with two 32 bits unsigned integers (the last two double words of the partition table entry). This is what is really used by modern operating systems. The CHS information is only stored for compatibility purpose with older operating systems like MS-DOS or Novell NetWare.

A further note could be made on the number of allowed partitions. If we look at the MBR structure, we see only 4 partition entries but we can define more partitions than that. The solution is to define a so-called *extended* partition.

An extended partition is in fact a block of space of the disk that will be seen as a virtual disk. On the first sector of this block, we will find another MBR, where additional partitions can be defined. Please note that all references in those additional partitions are made relative to the start of the virtual disk and not of the physical disk.

The extended partitions are identified by a partition type value of 05h, 0Fh or 85h and the way they are handled in chain has been defined as follow:

- The MBR can only contain one extended partition.
- The first partition table entry of an extended partition table is a primary partition.
- The second partition table entry of an extended partition table is an extended partition or is empty.
- The third and fourth partition table entries of an extended partition table are empty.

MBR Boot Code

We have seen how disks can be structured and how the BIOS loads the MBR from a disk or from the boot sector of a floppy disk. We will see now how the MBR boot code passes control to operating systems.

Once the MBR code has control, it searches an active partition into the partition chain (partition state value is equal to 80h) and once it has found one, it loads the first sector of that partition somewhere in memory (system dependent) and branches into the beginning of the memory area where the sector was stored.

It is finally up to this boot sector to load the operating system and to give it control. This final stage is highly system dependent and not standardized at all. Please note that in certain cases, it is the MBR code that directly load the operating system without searching for an active partition. This is of course less standard but that often used in multi-boot configurations (configurations where several different operating systems are installed).

In the case of a floppy disk boot, the boot sector loaded directly fetches the operating system and no search for an active partition is performed (as it would not make sense).

Lets finish on this boot topic with a note on the MBR boot code size. Despite those piece of code are generally written in a low-level language (assembler), they are generally short, but for a boot manager that can load more than one operating system (such as LILO), a maximal size of 01BEh bytes is really short. This is why there are often a few sectors reserved between the file system (partition) beginning and the boot sector (MBR). The MBR or the boot sector acts then as a loader for a more complicated and bigger software stored on these reserved sectors.

A way of reserving sectors after the MBR is to align the beginning of partition on cylinder boundaries. But, as we have seen before, the number of sectors per cylinder is now usually a fake information and it can only be used as a hint and not to place the partition beginning on a physical cylinder boundary. Reserving sectors between the boot sector and the file-system beginning often works the same way, but is highly dependent on the operating- and file system used.

Evaluation

It seems that a lot of work has to be done by the computer before an operating system runs. In fact, our description stopped just before it begins to operate. And if we realize that this is just true for one platform, it could be scary.

Well, the main topic of this detailed description was to have a template of what happened just before the operating system starts to do its job. And while it is not a universal description as it is sure other platforms do not operate **exactly** like this, it is sure that some concepts are also used. Our further work is then to identify the key concepts.

4.2 Virtual Platform Approach

In order to complete with the heterogenous implication of this paper, we are about to describe a virtual computer that has the fundamental concept of almost every computer types (avoiding esoteric ones). Virtual is a common term nowadays but here, it only means that it does not exist. We will not try to introduce some beliefs.

4.2.1 Components Identification

So, let's start the "building"of our virtual computer. It is obvious that a sort of BIOS must exist but what is so particular in the i386 case is the despotism used to boot the computer. In that particular case, the boot process is always the same (I admit there are some parameterizations possible) and no user interactions are required or possible before the boot process comes to the MBR or boot sector loading.

It would mean that every i386 computer does not need disaster recovery procedure (or almost none) before the MBR or the boot sector is loaded. Meanwhile, some platforms like Alpha computers have complex "BIOS"that permits full control on what is booted and how it is booted. In fact, the i386 computer manufacturer have decided to leave that operation to vendor specific operating systems, and we have then no standard for that on that platform.

We do not have to discuss what the best approach is (although I personally found that a common and standardized BIOS version is more adapted as administrators have only to concentrate on the platform used and not on the operating systems to accomplish early administration tasks) but we have to consider both. So the first component that we can identify on our platform is a user controlled BIOS with its own data. The data contained in that BIOS will describe how, where and which operating system will be booted.

We will not use the term BIOS for our purpose but the more generic term of *primary loader*. The primary loader will (or should) have the following tasks:

- Checking and initializing the installed memory
- Checking and initializing the installed hardware (I/O, ...)
- Finding the various peripherals and initializing them
- Allowing administrator to configure the boot process
- Passing the hand to the secondary loader

We must also be cautious when we speak about multi-boot platform where multiple operating systems can be loaded, as some platforms are specifically designed to boot only one operating system. And so, the primary loader is highly tied up to that operating system. Because the i386 is able to operate with various operating systems, it seems it was not such a bad choice to describe that platform.

The Secondary Loader

The primary loader introduced the secondary loader, and our main goal is to define it in an heterogeneous flavour. As the secondary loader is loaded by the primary loader, that generally has no strong knowledge of the operating system loader (despite some exceptions), it has to be situated on a fixed position on a bootable media. Its role is to actually load the operating system.

To illustrate what the primary and the secondary loader are, we could take back our i386 platform description. It is obvious that the BIOS is a part of the primary loader but what about the MBR? Although it is situated on a fixed location (and so complying with the property of the secondary loader), we have to define it as a part of the primary loader. Indeed, the MBR normally knows nothing or little about the OS it has to launch. The secondary loader would then be (in the i386 case) the boot sector of each partition.

The OS Kernel

Actually, when we said that the secondary loader has to load the operating system, we miss up something really important because we have to make a distinction between the operating system and the kernel:

The kernel is the core of an operating system, it provides all the functionalities that the operating system will use to accomplish its works. We could see the kernel as the layer between the hardware and the operating system. It is its role to finish loading the rest of the operating system.

The operating system is all the environment (kernel included) in front of which the user is when the system has finished to boot, excluding all the so-called third-party applications.

To give an example, word processors are not part of the operating system but shells (or command interpreters) or basic file copy command are. But of course, the frontier is sometimes difficult to draw between the operating system and third party applications (this is why, perhaps, some people consider the operating system as being the kernel but this is not our goal to discuss that).

So, to be more specific, the secondary loader loads the kernel of the operating system which is then in charge of loading the operating system, but that is an OS issue and not a computer one.

Disks Organization

We also have to define how the disks will be organized on our virtual computer, and because partitioning is very common practice, introduce that concept too. Of course, having a partition scheme as complicated as the i386 is not required and in our case, not necessary.

In order to avoid the problem of having the CHS disk addressing scheme, we will simply use a linear disk addressing one. We will also suppose that sectors are of fixed size and they should be accessed as if they were consecutive on the disks.

A typical question is to know if we have to put a MBR-like sector on the disk. In fact, we will reserve the first sector of each disk to store the disk structure, i.e. the partition table. A partition table entry can be defined by a triple:

- the partition start offset in sector (relative to the beginning of the disk, starting at 1 because the sector 0 is reserved for the table).
- the partition length in sector.
- a bootable flag

Not having chained partitions is not a problem as it's just a particularity that one platform can have but it is not our purpose.

Having no executable code in the first sector means that the primary loader contains no code on the disk. It is therefore up to the primary loader in ROM to handle the partition table structure and to load the secondary loader. This one has to be located somewhere on the partition and for facility, we will put it on the first sector of each partition.

Also note that a non bootable partition should have its first sector reserved as the bootable flag for that partition could be changed.

4.2.2 Evaluation

We have not described our virtual platform as we had described the i386, we don't explain which memory model it should use, or even the form of each instruction that the processor can understand. So we don't really build a computer but we now have a strong template of a modern, typical, computer bootstrap. And that, in a very heterogenous way as we avoided platform particularities.

4.3 Operating System Study: The Linux Case

Of course, we will not describe the operating system at this would be a huge and unnecessary task but we will explain how the kernel is loaded by the secondary loader and how it achieves its boot process to the login prompt. Once again, we were lucky as the secondary loader (which can act as a part of the primary loader, we will see this later) called

LILO is a well documented free software and its source code publicly available. Enough to satisfy our need for knowledge. A second warning has to be issued here because we will speak about the i386 version of Linux.

4.3.1 The Secondary Loader

To stay consistent, the Linux secondary loader puts itself on the boot sector of a partition (although it can be installed on the MBR but this is less standard. The purpose of such installation is to have a practical boot manager loaded first). If we do not consider LILO as a boot manager, but as a Linux loader only, we have a normal Linux secondary loader, except for one point:

In the literature, the *monitor* is defined as a part of the primary loader and its task is to determine how the operating system is booted. On UNIX platforms, for example, the single user mode used for critical administration tasks is enabled through the use of the monitor.

As LILO is used to enable the parameterization of the Linux boot process, it is also a part of the primary loader. But this feature is a particularity of the i386 Linux version and is an esoteric capability furnished to alleviate the problem of the i386 platform that has, by default, no good primary loader. We will just forget that feature here and simplify a bit the Linux boot process.

4.3.2 The Kernel Image

On Linux, as on all UNIX systems, the kernel is put into a file, on the root partition file-system. It means that the secondary loader would have to know the structure of the file-system, seeing that it has to access a real file. But in fact, this is not true. To avoid any file-system particularities, the software that installs the secondary loader asks the kernel how the file is stored on disk and LILO uses this information to load the kernel into memory, it has then no need to know anything about the file-system used.

Note: That is not always true as some operating systems like MS-DOS, use the file content table to fetch the location of the kernel files (which are IO.SYS and MSDOS.SYS in the MS-DOS case). Those secondary loaders have therefore a knowledge of the file-system used.

4.3.3 The Kernel Boot Process

Just after LILO gets the hand (from the standard MBR boot code), it moves itself to 09A0:0000 in memory and once there, load the rest of its code (called the secondary loader in LILO terminology) and branch to it. Once this secondary loader (LILO terminology) is activated, it loads the descriptor sector that contains informations on where and how to load the kernel image. Once it knows the location of the kernel image on disk, it loads it and transfers the control to it.

What is really particular in the Linux case is that the the kernel image is composed of three objects:

- a boot sector
- a setup code
- the kernel code itself

When LILO is used to load the kernel image, the boot sector is not used and the control is actually transfered to the setup code which initializes various things and then loads the kernel code (the kernel image boot sector is only used to boot the kernel from floppy disks). As we can see, this is a complicated process and far from being as simple as we could imagine. But again, this is a particularity of the system. And our three steps concept (primary loader ->secondary loader ->kernel image) is still right. The only complicated thing is the number of processes involved in each part of the boot process.

The only consideration that we must retain is the fact that the kernel image is actually a file on a file-system and that has great implications for us.

4.3.4 The Kernel Is Up and Running

Once the kernel executes itself, we have the core of the operating system but it is still unusable! The loading of the remaining part of the system is as follows:

1. The kernel mounts the root file-system (/)
2. The kernel spawns its first process which is an executable located on the file-system (generally /sbin/init or /init)
3. This init process initializes the console (which is a sort of primary terminal) and launches various configuration scripts (in order to check the file-systems, to setup swap spaces, to set the system clock, to mount other devices, to launch various daemons, etc...)
4. The init process spawns then terminal control processes which are in charge of setting up the terminals and of presenting the login prompt on those terminals.

4.4 Virtual Operating System

To begin our approach, we'll ask ourselves the following question: What are the components of a modern operating system?

To answer that question, remember the virtual platform study and the primary loader. This one was used to choose which operating system has to be booted and how (through

the use of the monitor). Therefore this is not a part of the operating system (but it can in some cases interact with it).

The secondary loader is used to boot the operating system and has a good knowledge of it, but except for booting, it furnishes no services to the user. It is therefore a sort of layer between the primary loader and the operating system. But because it has operating system specific function and that it is closely tied with it, we will consider here that the secondary loader is a part of the operating system.

4.4.1 The File-System

We have seen that the kernel image is loaded by the secondary loader and this one is used because we pass from direct access through the disk sectors to a file access in a structured hierarchy: the file-system.

We can therefore identify a frontier during the boot process: before the secondary loader, all the disk accesses are performed through sectors loading and after the kernel is loaded, all the accesses are achieved through the file-system (sometimes, file-system accesss are achieved by the secondary loader, therefore we have a small possible sliding of the frontier described). This frontier has some implications for computer recovery as we will see later in this paper.

4.4.2 Evaluation

Therefore we have identified three components of the operating system:

- Outside the file-system
  - The secondary loader
- Inside the file-system
  - The kernel image (or core system file(s))
  - The operating system files

One particularity of our approach is the secondary loader which is actually a part of the operating system while it must be situated on a fixed location on disk, and is then closely linked with the platform definition.

4.5 Computer Recovery: The KISS Approach

4.5.1 Keep it Simple...

We made our way through the boot process description to understand how we should handle computer recovery operations but we forgot to notice one thing: all the data stored

on a computer are put on disk sectors. So, a keep it simple and stupid idea would be to say: if we ever want to backup all the data of a computer in a very heterogeneous fashion, we would simply have to backup all the sectors of each disk, without even bothering about the operating system installed. This nice action is often called the *mirroring* of disks.

As we would save all the data on the disks, a crash would not disappoint us as we would have saved all the disks, almost physically. We would only have to restore the data sector-by-sector on a fresh new disk or on the old disk (if still usable).

4.5.2 ... and Stupid

Yes, that method is stupid because it forgets a few things:

1. A running computer is a dynamic environment where data are read and written on disks very often.
2. Concurrent accesses sometimes (often?) lead to unsubstantiality.

To understand that, let us take a simple example. Suppose we are updating a database of millions of records and we want to backup our computer for disaster recovery purpose. While we are reading sectors, the database application is writing other sectors. We therefore end up with an inconsistent backup where the updates of the database are inconsistent; just because some sectors on the disks where backed up before being updated and some after being updated.

If we restored that backup, we would have detrimental problem in our database. This solution has then to be rejected.

4.5.3 First Solution: Locking

A simple solution to that problem would be to stop all the processes that could write data on the disks, or to lock all the disks for writing. While this could be a solution, there are also big problems as:

- We have to lock the disk when it is in a consistent state.
- Do we have a way to know when the disk is in such a state?
- Can we afford to lock a server for writing during the backup time?

As we can see, this solution also has some important drawbacks and can therefore not comply with our needs as we should discover a way to backup the data with interrupting the processing time of the computers as shortly as possible (and that is clear that backing up a huge amount of data would take some time).

In fact, the big problem of the solution is that we want to bypass the operating system and the applications to backup the data for disaster recovery.

4.5.4 Second Solution: Intelligent Buffering

Let us imagine an operating system feature where all modifications made to a file are put in a huge buffer. On opening a file, the operating system allocates a buffer[1] where all the modification will be put for later writing.

We could describe this feature with the following (hypothetic) system calls:

il_open(file) Opens a file with intelligent buffering enabled: A buffer is allocated somewhere. If the file has already been opened by a il_open() system call, this instance must fail.

read(file, sector) Reads a file owned sector. If this sector is present in the buffer and intelligent buffering is enabled, return the data contained in the buffer, otherwise (sector not in buffer or file opened through a normal open() system call) reads it from disk.

write(file, sector) Writes the sector into the buffer.

il_close(file) Closes the file and puts the sectors contained in the buffer on disk. But only if no backup operations are in progress. If this is the case, holds on the request.

backup_open(disk) Requests the disk backup mode (begins backup operations). (must wait that no il_close(file) system calls are in progress).

backup_read(disk) Reads the disk sectors.

backup_write(disk) Writes backed up sectors on disk.

backup_close(disk) End up the disk backup mode.

If we set the conditions that all files opened for writing are opened through the il_open() system call, we have an operating system on which the files on disks are always in a consistent state, and then it is possible to implement our KISS solution without interrupting the computer functions. But actually, we rely on a feature of a specific operating system and we have to work in an heterogeneous environment, so this solution has to be rejected (while being interesting for operating system development).

4.5.5 File-System Type

The KISS solution also has another drawback: even if the operating system provides system calls suitable for keeping data files in a consistent state, what about the consistency of the file system?

Actually, we can consider the file-system as the main data file for the operating system. If we wanted to use KISS, the operating system should provide us with a way to determine

[1] Possibly on disk.

when the file-system is in a consistent state and to keep that state during the backup of all sectors (what can be achieved through what we called intelligent buffering). But once again, such implementation would require a particular operating system and is out of topic for this work.

The reader might be interested in some development in file-system technology and can consult [21]

4.5.6 The KISS Evaluation

While simple, this solution was too stupid to be suitable for our needs. However, it can be used when the computer is off-line. I.e., if we have a way of accessing disks while the usual operating system is not loaded (through the use of a light operating system on floppy disks for example), we could use that solution (which is used for *generating* standard setup in some enterprises: a typical computer is set up, and then the disks are mirrored through the use of a specialized software. Once an administrator wants to setup another similar computer, he/she only has to make a sector-by-sector copy of the mirrored disks on the target computer).

4.6 Computer Recovery: The Components Approach

We have to work in an heterogeneous environment, and we repeat it very often. Therefore we have to design a generic way of *seeing* disaster recovery operations. As the KISS method failed because of too specific operating system needs we have to try a perhaps more difficult, but universal method of doing what we want to achieve.

We will call it the components approach because we will use our virtual platform and our virtual operating system to try to discover what we actually have to save and to recover to handle computer recovery operations.

4.6.1 Manual Operations

In order to develop a complete computer recovery procedure[2], we will first see how a disaster recovery operator would handle the full recovery of a computer.

Computer External Environment

The first task of the operator in charge of disaster recovery is to identify the computer type (brand & model) and the computer equipments (main memory size, number and type of disks, hardware components installed,...). With that information, he/she will be able to know on which platform the recovery procedure will work in case of machine total destruction. It is obvious that after a certain time, he/she could of course not find the

[2]An example of a computer recovery procedure can be found in [14]

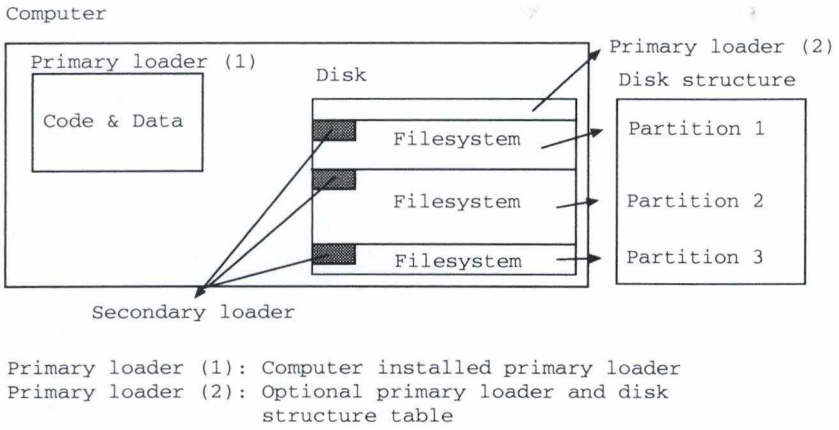

Figure 4.4: Boot Process Components Summary (Example)

exact configuration he/she had on the computer market, but computers generally tend to be backward compatible. So having a more powerful machine (in term of speed or memory capacity) is often not a problem. This process will be called the *hardware inventory* process.

The second task would be to note the operating system type and version and all the installed applications – types and versions – (database applications, office suite,...). This process is obviously called the *software inventory*.

These two processes can be summarized by the inventory process.

Computer Internal Environment

Once the operator has noted all the external environment of the computer, it has to backup the computer internal environment. I.e. the way the hardware and software environment were configured. How the hardware was configured was certainly very important a few years ago, when all configurations where achieved through dip switches and jumpers directly on the cards but this way of doing tends to disappear nowadays. The software configuration of the hardware is in favor now and it will certainly facilitate the computer recovery process.

The way of backing up the internal environment will be guided by our virtual platform and operating system description. For this, we will remember the boot process description that we made earlier in this chapter. The various components of the boot process are depicted on figure 4.4 (The kernel image, operating system and application files are not shown for readability but are included in the file-system).

The problem here is the operator cannot write down all the files content and the binary information coded on the disks and in the primary loader. He/she has to write down the logical configuration of the internal environment. When we said that backup solutions where closely connected with disaster recovery, we underlined an important point. Actually,

all the files are to be backed up by such a software solution and that is often the only "disaster recovery"procedure implemented in most computer centers. As those products exist, we will not speak about them. We will only cover the unsupported parts of those products.

The Primary Loader (Platform Part) We are here in a very particular part of the disaster recovery operations because the configuration data of the platform specific primary loader is very platform specific.

The only way to save this information is to go through all the monitor features and to note all the customizable options. The platform manual is handy in that case as it will allow the operator to know how to access all the features.

The Primary Loader (Disk Part) and Disk Structure Once the operating system has been loaded, the operator often has the necessary tools to access the disk structure informations (fdisk for Linux, prtvtoc for IRIX, Logical Volume Manager for HP-UX,...). A screen copy of the output of those utilities is handy. The operator has to understand the information provided in order to restore it later.

Our virtual OS could, for example, have a tool that reads the first physical sector of the disks and displays the information contained on it in a readable fashion:

```
vos# disk_struct_print
Disk 0:
    Partition 1:
Start: 32 (sectors relative to sector 0)
Size : 5000 (sectors)
    Partition 2:
Start: 5032 (sectors relative to sector 0)
Size : 10000 (sectors)
Disk 1:
    Partition 1:
Start: 32 (sectors relative to sector 0)
Size : 10000 (sectors)

vos#
```

As our virtual operating system has no primary loader code located on the disk, the operator has nothing to do to save it. But we must not forget that some platforms can have a primary loader part on the disk. The problem then is to save the code of this primary loader and there is no easy way to do that.

That is often why a manual disaster recovery operation is difficult. Binary code is almost impossible for a human to write down. Sometimes, operating systems provide tools to write a suitable primary loader there, sometimes they do not. Disaster recovery

operations then end up in a dead end (but it is a pessimistic view of the problem and there are often solutions). The problem is that the operator loses the control of the operations at that moment if he/she doesn't have handy software available.

The Secondary Loader The operator has no way of accessing the secondary loader, but here it is not a problem. As it is an operating system part, it will generally be restored by a new operating system installation. However, in very specific or particular installations, the computer operator cannot be sure that the original secondary loader will be restored.

Partitions File-System Type Once again, the operator has to note the file-system type of each partition on each disk. As the operating system provides tools to discover that, this is not a difficult task. We will not say much about it.

The Handy Backup Software As we said, backup tools are very important as they permit the backup and restoration of files while the system is running. However, we will see that it sometimes has some limitations such as its inability to save some particular file types.

Manual Restoration

The previous point explains how to know almost everything about the computer configuration and therefore is seemingly helpful in computer restoration. But if we try to discover how to put this information back on the computer, there is no easy way and the only thing of which we are almost sure is that we should certainly use the installing tools of the considered operating system. But how we should do it is another question.

Actually, each operating system has its own installation method and the operator has to be well in touch with the considered operating system to know how to restore the computer. More over, he/she must have a very good knowledge of his/her platform and operating system to be able to restore a computer after a crash, even if he/she has noted all the described information.

A "good"summary of manual restoration would be : "Get the same computer and the operating system (and applications) installation media. Re-install the operating system (and applications). Re-install your backup solution and restore all the files that can be restored. You should now have a working computer."

It is perhaps a good summary but it is certainly not a good computer recovery model as we cannot be sure of anything. Sadly, this is often the only computer recovery solution that backup softwares can provide.

4.6.2 Automatic Operations

The figure 4.4 showed us the different parts of an operational computer, i.e. the parts that permit the computer to boot. We also saw that it was almost impossible for a human

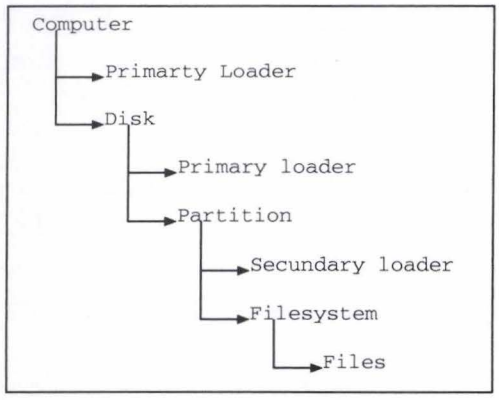

Figure 4.5: Computer Object Hierarchy

being to save this information and still more difficult to restore it. And that, because no suitable tools were provided by the operating systems.

It is obvious that full automatic restoration is also impossible but it's certainly possible to help the disaster recovery operator in his/her task by designing a tool that can handle all the process of a manual operation.

To understand such a tool, let us represent the different computer components as a hierarchically structured collection of objects (see figure 4.5).

Now that we have a strong knowledge of what objects have to be saved in order to restore them, the "only"thing we have to design is the disaster recovery tool.

Saving Operations

The computer is running and we have access to all the operating system functions. The disaster recovery tool has to use those functions in order to save all the objects we need to recover the computer.

This information has to be stored somewhere and as we studied HSMS-CL, we will use it in our computer recovery study. Of course, there are design implications of using HSMS-CL but we will forget them for now. We will just assume that we will use HSMS-CL to store this information remotely on one (or more) central backup BS2000 server.

If we try to devise a generic algorithm, it would look like this (and that for all platforms):

```
init_db_store()

get_computer_identity()
store_computer_identity()

get_computer_primary_loader_data()
```

```
store_computer_primary_loader_data()

foreach installed_disk in &disk do
    get_disk_primary_loader(&disk)
    store_disk_primary_loader()

    get_disk_structure(&disk)
    store_disk_structure()

    foreach partition(&disk) in &part do
        get_partition_secondary_loader(&part)
        store_partition_secondary_loader()

        get_partition_filesystem_type(&disk)
        store_partition_filesystem_type()
    hcaerof
hcaerof

get_os_specific_info()
store_os_specific_info()

close_db_store()
backup_db_store()

launch_backup_software()
```

In fact, this algorithm is a very simple one when we have studied the different components of a computer. A software designer would have to know how to use the operating system facilities furnished to be able to save this information. So, this theoretical approach shows us how to identify which functions we have to find in a particular operating system to devise a computer recovery tool.

Restoring Operations

It's a broader subject as we cannot base ourselves on the possibilities furnished by the running computer. We have to start from nothing. Therefore, a computer recovery tool must provide a light and easy to use operating system in order to boot the freshly repaired or bought computer.

As no generic operating system can handle all the particularities of various ones, this subset of the operating system will be of the same brand as the previously installed one. Developing a suitable installation bootable on various computers is not an easy task and requires a good knowledge of the inner- working of the operating system and this is certainly one of the most complicated tasks in our approach of computer recovery.

Furthermore, on top of this subset installation, we have to find a computer recovery tool and a backup/restore software. But those softwares take room. So it is almost impossible to put an operating system, a computer recovery tool and a backup software on a single floppy disk (except for not so common media). It means that we have to use a CD-ROM media or a remote copy of the operating system (via NFS for example). This has two implications:

- the OS has to get access to a CD-ROM drive
- the OS has to get access to the network

The second implication is still stronger. Because backup softwares generally access their data on a remote server.

The operating system must then be able to access a network card and CD-ROM drive. Due to the variety of those two devices, developing a generic enough operating system installation is very difficult. Either the boot process of the light OS has to include a driver selection or has be parametrized in advance for each concerned computer. The two solutions have their drawbacks: the first one requires more work from the disaster recovery operator and the other one multiplies the need of media when various computer configurations are present (despite homogeneous in term of computer platform). The best solution is to provide the possibility for the administrator to choose his preferred solution.

Once those problems are overcome, the operator can launch the computer recovery operation using the disaster recovery tool. The first step of this process is to get the recovery information database back (through the use of the backup software for example). Once the recovery information is recovered, the tool will use the operating system functions to recover it on the computer (after verifying that the recovery database matches the type of computer we are about to recover. It does not need to be a perfect match as a greater disk than the crashed one, for example, shouldn't be a problem.). Once the structure of the computer has been restored, the backup software can do its job and restores the files it has saved.

We should now have a running computer. Note however that some further processes could be necessary with some specialized software to restore databases (which require running database servers).

4.6.3 Operating System & Applications Specific New Components

The reader might have noticed that we only spoke about operating systems and hardware platforms but an heterogeneous environment can be defined by a group of similar computer running under a similar operating system but running different database engines, for instance.

This case cannot be well modelized by our components approach because we have stopped the development of the components approach very early after the recovery of the file systems, assuming that backup softwares would handle further recovery for us.

But it can be true as more often, applications have a special way of installing themselves. And it can't then be possible to restore the data they handle without further processing. In fact, our components approach lack a few components.

Actually, what we called "files"are more than simple files and this term is a bit too much generic to allow a further development of our components approach. In fact, we can make a distinction between multiple types of files. UNIX, for example, defines the following type of files (following the stat (2) man page):

- Regular files
- Symbolic links
- Directories
- Character devices
- Block devices
- Socket devices

But, that is UNIX point of view about files and we do not want a particularity of a platform or operating system to come to light in our approach. Therefore, we will forget all about symlinks, regular files and the likes and we will take a more logical approach to differentiate two types of "files":

1. The files that contain executable code, that can be executed by the platform CPU.
2. The files that contain arbitrary data.

A further distinction can be made between the files belonging to the operating system and the ones belonging to installed applications. Of course, the frontier between the files belonging to the operating system and the other ones is very difficult to draw. Furthermore, there is a dependence between operating system files (executables and data) and application files. And that because operating system files are required by applications to perform their tasks. But this is also true for applications that need other application files to work correctly (but without being part of the needed application, though).

We have consequently four new components:

- Operating system executable files
- Operating system data files
- Application executable files
- Application data files

We have a new component hierarchy, starting from the files node in figure 4.5, composed of operating system files, which start new branches pointing to application files, themselves starting new branches pointing to new applications files and so on. Therefore we still have a structured logical hierarchy of various components. But we are faced with a real difficulty. We said that the files backup/recovery process was handled by a backup software. But this software has no idea of our components approach and has no knowledge of our hierarchy. To solve that problem, the only solution is to drive the backup software when recovering (backuping?) the files, and to put only the needed information to drive the backup software in its own database. Sometimes, it would even be necessary for the recovery tool to restore some files itself (in case the backup software does not handle certain types of files).

4.6.4 Conclusion

We now have a detailed description of the different logical components of a computer, in almost all its available forms. We should then be able to mix the studies we made about heterogeneous environments implications and this generic description to develop a disaster recovery tool that would easlily be updatable in order to meet the requirements of various platforms.

The great advantage of the logical approach is to remove all the constraints of a to technical approach. Indeed, we saw that if we go to deeper in the inner-working of a computer (like with the KISS approach), we are faced with a lot of consistency problems. The fact is that those problems exists. But the components approach leaves those issues to the developper and permit a greater flexibility on how to resolve them.

Chapter 5

Implementation: Computer recovery (bsrecov)

5.1 Introduction

We have already spoken of some design aspects of a disaster recovery tool in the chapter 2 and 4. In order to avoid any confusion, we will not recall all the notions that we described in those chapters. We will just develop the application (called bsrecov, as specified by the chapter title) step by step, in this order:

- The disaster recovery commands
- Disaster Recovery Database Management (I)
- Modules Management (I)
- Options Management (I)
- Remote Object Database Backup/Restore
- Application configuration file
- Modules Developpment

For each *category*, we will describe the way it should work. Then will follow the implementation in C (If marked with an (I) in the list). Please note that the full implementation (in compilable form) will be put in the Appendix.

Because these are closely connected subjects, options management will be covered inside the modules management section.

5.2 The Disaster Recovery Commands

Before developing the design, we will describe the basic commands that our tool will perform. It will perform:

1. The backup of the disaster recovery objects database.
2. The restoration of the disaster recovery objects database.
3. The backup of disaster recovery objects in the local database.
4. The recovery of disaster recovery objects from the local database.
5. The restoration of the previously backed up computer files[1]
6. The display of information concerning the installed modules.

That is all what bsrecov can do when someone executes it. In fact, bsrecov has to be as simple as possible, at least when faced with a recovery operation. If it was unusable due to the number of configuration flags, it wouldn't be helpful for the operator in charge of the recovery operation. And that is its primary goal!

In appendix B, you will find a version of the bsrecov manual. The reader might find valuable information there to understand more precisely what bsrecov exactly is. And what it does, more precisely.

5.3 Disaster Recovery Database Management

5.3.1 Design and Description

In order to backup the disaster recovery objects, we will put them in a file called the disaster recovery database. As we want the database to be simple and easily accessible, we will use a library called *GDBM*. Quoting the authors, *GDBM (GNU dbm) "is a library of routines that manages data files that contain key/data pairs. The access provided is that of storing, retrieval, and deletion by key and a non-sorted traversal of all keys. A process is allowed to use multiple data files at the same time."* Consult [17] for a full reference.

Of course, if we remember the way we described the computer objects, we had a hierarchically organized collection of various objects. In its stock implementation, GDBM does not permit the management of such objects, so we have to implement a way of putting such objects in a *GDBM* file. We will implement that using a hierarchy table that will be put in the *GDBM* file using a fixed key. This table will describe the hierarchy in terms of parent/child pair. As the key of the hierarchy table is fixed and known, it is easy to load and to save the table in the database. The parent/child pairs are 2 unique keys that identify an object in the database. It means that putting an object into the database implies

[1] The objects processed by the used backup tool. Here, HSMS-CL.

the generation of a unique key (or object identifier). Another property of each object in the database is that each object (except perhaps the root objects) has a parent object. So, to load an object from the database, we only have to supply a valid key. But to save an object in the database, we have to supply a valid key[2] and the object identifier of a parent object.

With the following description, we are now able to identify the following functionalities:

- Unique Key Generation
- Hierarchy table initialization
- Hierarchy table loading
- Hierarchy table saving
- Objects loading
- Objects saving
- Hierarchy traversal functions

To achieve a better organized database file, we will define a layout where each object is stored in the database through the use of a "Generic Object Pointer". This generic object pointer is a structure that hold the application private object type and a private key (hidden to the application), referencing the real object data in the database. We can then provide the application (or modules) with a simple way of marking their objects (or to assign them limited properties). We might also note that this type of layout imposes that the database management unit furnishes the object identifier[3].

5.3.2 Implementation

Data Types and Variables Definition

Here are described the different types and variables needed by the database management functions:

```
/* The Generic Object Pointer Type */
typedef struct {
    int type;      /* object propreties field */
    long ref;      /* real object key identifier */
} GObject;

/*
```

[2]As this key has to be unique in the database, it would be easier if the database management unit furnished that key.

[3]And it is not a problem! This is what we wanted!

```
   a variable that contains the next current unique object
   identifier. We start at 1 because the h_table has a key id of 0
*/
long personal_id=1;

/*
   The hierarchy table as a pointer to an array of long long integer
   (64 bits on an i386). The 32 high order bits represent the parent
   key and the 32 low order bits represent the child key
*/
static long long *h_table;

/* various hierarchy table properties */
static long long hierarchy_size;
static long long hierarchy_top;
static long long hierarchy_pos;

/* The pointer to the GDBM_FILE handle */
GDBM_FILE file
```

Functions Description and Code

```
/*
:Function:       generate_key
:Description:    Generates a unique key in the application execution context (or
                 the database context).
:Arguments:      none
:Return Value:   unique key (long)
:Pre-condition:  none
:Post-condition: the generated key is unique
*/
long generate_key() {

     personal_id++;
     return (personal_id-1);
}

/*
:Function:       init_hierarchy
:Description:    Initializes the hierarchy table
:Arguments:      none
:Return Value:   none
:Pre-condition:  none
:Post-condition: the hierarchy table h_table is initialized
*/
```

```
void init_hierarchy() {

    hierarchy_size = 0;
    hierarchy_top = 10;
    h_table = (long long *)
        malloc(hierarchy_top*sizeof(long long));
    if (!h_table) bserror(ERR_MALLOC,"init_hierarchy()");
}


/*
:Function:       add_hierarchy
:Description:    adds a parent/child pair in the hierarchy table.
:Arguments:      - parent (long) : parent key identifier
                 - child (long) : child key identifier
:Return Value:   none
:Pre-condition:  The hierarchy table (h_table) must be initialized.
:Post-condition: the new parent/child pair is in the hierarchy table
*/
void add_hierarchy(long parent, long child) {

    hierarchy_size++;
    if (hierarchy_size == hierarchy_top) {
        hierarchy_top += 10;
        h_table = (long long *)
          realloc((void *) h_table,
          hierarchy_top*sizeof(long long));
        if (!h_table) bserror(ERR_MALLOC,"add_hierarchy()");
        }

    h_table[hierarchy_size-1] = parent;
    h_table[hierarchy_size-1] = (h_table[hierarchy_size-1] << 32)
        | child;
}

/*
:Function:       save_hierarchy
:Description:    Saves the hierarchy table in the GDBM database file.
:Arguments:      none
:Return Value:   none
:Pre-condition:  - The hierarchy table (h_table) must be initialized
                 - The GDBM database (file) is opened
:Post-condition: The hierarchy table is saved under the key 0 in the GDBM
                 database (file).
*/
```

```
void save_hierarchy() {

    datum key, content;
    long keyid;

    keyid=0;
    key.dptr = (void *) &keyid;
    key.dsize = sizeof(long);

    content.dptr = (void *) h_table;
    content.dsize =  hierarchy_size*sizeof(long long);

    if (gdbm_store(file, key, content, 0))
        bserror(ERR_WRITEDB,"save_hierarchy()");
}

/*
:Function:       load_hierarchy
:Description:    Initializes the hierarchy table with the one found in the
                 GDBM database file.
:Arguments:      none
:Return Value:   none
:Pre-condition:  GDBM database (file) is opened.
:Post-condition: The hierarchy table is initialized with the hierarchy
                 table found in the GDBM database (file).
*/
void load_hierarchy() {

    datum key, content;
    long keyid;

    keyid = 0;
    key.dptr = (void *) &keyid;
    key.dsize = sizeof(long);

    content = gdbm_fetch(file, key);

    if (!content.dptr) bserror(ERR_OBJNF,"load_hierarchy()");
    h_table=(long long *) content.dptr;
    hierarchy_size = (content.dsize / 8);
    hierarchy_pos = 0;
    hierarchy_top = (content.dsize /8)+1;
}

/*
```

```
:Function:        findnext_hierarchy
:Description:     Finds the next child of the given parent, if it exists,
                  in the hierarchy table.
:Arguments:       - parent (long) : parent key identifier
                  - where (long long *) : private variable
:Return Value:    a child key identifier (long)
:Pre-condition:   none
:Post-condition:  the returned value is a child of parent. If the
                  parent doesn't exist of has no more child, the
                  return value is 0.
*/
long findnext_hierarchy(long parent, long long *where) {

    int ok;
    long value;
    long child, par;

    ok = 1;
    value= 0;

    hierarchy_pos = *where;
    while ((ok) && (hierarchy_pos < hierarchy_size)) {
        child = h_table[hierarchy_pos] & 0xffffffff;
        par = h_table[hierarchy_pos] >> 32;

        if (par == parent) {
            ok = 0;
            value = child;
        }
        hierarchy_pos++;
    }
    *where=hierarchy_pos;
    return value;
}

/*
:Function:        findfirst_hierarchy
:Description:     Finds the first child of the given parent, if it exists,
                  in the hierarchy table.
:Arguments:       - parent (long) : parent key identifier
                  - where (long long *) : private variable
:Return Value:    the first child key identifier of the parent (long)
:Pre-condition:   none
:Post-condition:  the returned value is the first child of parent. If
                  the parent doesn't exist of has no childs, the
```

```
                return value is 0.
*/
long findfirst_hierarchy(long parent, long long *where) {

    hierarchy_pos = 0;
    *where = 0;
    return findnext_hierarchy(parent, where);
}

/*
:Function:       api_load_object
:Description:    Loads the object (and its properies) referenced by the given
                 key from the GDBM database file.
:Arguments:      - ref (long): an object identifier
                 - type (int *)
                 - d_size (int *)
:Return Value:   (void *): a pointer to the object with ref as key
:Pre-condition:  GDBM database (file) is opened.
:Post-condition: - if the object with key identifier ref exists in the
                   database, *type contains the properties of the object,
                   *d_size contains the size of the object in bytes and
                   the returned pointer points to the object itself.
                 - if the object doesn't exist, the returned pointer is NULL
                   and the content of *type and *d_size is undetermined.
*/
void *api_load_object(long ref, int *type, int *d_size) {

    datum key, content;
    GObject temp;
    void *retval;

    retval = NULL;

    key.dptr=(void *) &ref;
    key.dsize=sizeof(long);

    content = gdbm_fetch(file, key);
    if (!content.dptr) goto err;

    *type = ((GObject *) content.dptr)->type;

    key.dptr = (void *) &(((GObject *) content.dptr)->ref);
    key.dsize = sizeof(long);

    content = gdbm_fetch(file, key);
```

```
    if (!content.dptr) bserror(ERR_INCON,"api_load_object()");
    *d_size=content.dsize;

    retval = content.dptr;

err:
    return retval;
}

/*
:Function:       api_save_object
:Description:    Saves the given object in the GDBM database file and gives
                 it parent as parent in the hierarchy table. The function
                 assigns an object identifier to that object.
:Arguments:      - data (void *): a pointer to an object
                 - d_size (int): the size of the object in bytes
                 - type (int): the properties of the object
                 - parent (long): the key identifier of the parent object
:Return Value:   (long): the key identifier of the saved object.
:Pre-condition:  GDBM database (file) is opened.
:Post-condition: - if the return value is > 0, the object is saved in the
                   GDBM database and its key identifier is equal to the
                   return value. The new parent/child pair (parent/retval)
                   is put in the hierarchy table.
                 - if the return value is equal to -1, the operation has failed.
*/
long api_save_object(void *data, int d_size, int type, long parent) {

      datum key, content;
      GObject temp;
      long key_id;
      int retval;

      retval = -1;
      key_id = generate_key();
      temp.type=type;
      temp.ref=generate_key();

      key.dptr=(void *) &key_id;
      key.dsize = sizeof(long);

      content.dptr=(void *) &temp;
      content.dsize=sizeof(GObject);
```

```
        if (gdbm_store(file,key,content,0)) goto err;

        key.dptr=(void *) &temp.ref;
        key.dsize=sizeof(long);

        content.dptr=data;
        content.dsize=d_size;

        if (gdbm_store(file,key,content,0)) {
            key.dptr=(void *) &key_id;
            key.dsize=sizeof(long);
            if (gdbm_delete(file,key)) {
                bserror(ERR_INCON,"api_save_object()");
            }
            goto err;
        }

        add_hierarchy(parent, key_id);
        retval = key_id;
err:
        return retval;
}
```

5.4 Modules Management

5.4.1 Design and Description

To avoid an application that would be too particular, we said we would implement a modularized one where all platform particularities would be implemented through dynamic shared objects (also called modules)[4]. Here, we will describe what a module should contain and how we will manage its use in the main application.

Module Skeleton

In order to be usable, a module should export[5] a function called init_module with the following prototype: *int init_module()*.

This function we will be called at a certain stage of the application execution and will be responsible for registering[6] the module (and its options) in the main application through API calls that we will describe shortly.

[4]Modules linked during the execution phase.

[5]Exporting a symbol is a building issue and the reader should consult appendix A to see how the application and modules are really built.

[6]We will describe this notion later.

Of course, the module has to be able to execute disaster recovery operations. In the application, we define four operations realized by four functions:

1. **A recovery function** to recover the objects backed up by the module.

2. **A backup function** to backup computer objects handled by the module.

3. **A print function** to give various information elements on the backed up objects of the database.

4. **A module info function** to inform the user on the purpose of the module (this is the only function that has to be implemented in a module).

To manipulate its objects[7], a module will of course use the application APIs. But for disaster recovery purpose, it will use all what it wants, i.e. all the specific functions of the concerned platform it wants. The only constraint is the use of the object hierachy. But that is not an unsuperable one.

Application Modules Management

The application will internally maintain three tables. One for the modules file handles, one for the modules functions and one for the modules options. The first one is filled by the main application itself. Once it has loaded a module, the application opens it and puts the module handle in the module table. The application then calls the *init_module()* function of that module.

In order to accomplish disaster recovery operations, the module will register itself with the help of two application functions:

1. *api_register_module(...)* that fills the module functions table. Through this call, the module declares its various disaster recovery functions to the applications.

2. *api_register_option(...)* that fills the module options table. Through this call, the module declares the functions that must be called when the application encounters specific modules options in the application configuration file (described later in this document).

Once all the installed modules have been configured, the disaster recovery functions registered by the modules will be called by the application (following what operation was requested by the operator, of course).

[7]Storing and retrieval in the database

5.4.2 Implementation

Data Types and Variables Definition

```
/*
    The module functions table and the variable that counts the number of
    registered modules
*/
extern Module_Func **Mod_F;
extern int Registered_Mod;

/*
    The module option table and the variable that counts the number of
    registered modules
*/
extern Module_Option **Mod_O;
extern int Registered_Opt;

/*
    The module handle table and the variable that counts the number of
    opened modules
*/
static void **Module;
static int RegMod;
```

Functions Description and Code

```
/*
:Function:       main_mod_init
:Description:    initializes the module handle table
:Arguments:      none
:Return Value:   none
:Pre-condition:  none
:Post-condition: The module handle table is initialized.
*/
void main_mod_init() {
    RegMod = 0;
    Module = NULL;
}

/*
:Function:       main_mod_add
:Description:    initiates the module registration process
:Arguments:      str (char *): a dynamic shared object pathname
:Return Value:   none
:Pre-condition:  - The module handle table is initialized.
```

```
                - Dynamic shared object pathname is correct.
                - The dynamic shared object is a well-constructed object.
:Post-condition: - The module is registered in the module handle table.
                - The module functions are registered in the module
                functions table.
                - The module options (if any) are registered in the
                module options table.
*/
void main_mod_add(char *str) {

    int (*mod_init)();


    RegMod++;
    Module =  (void **) realloc(Module, RegMod*sizeof(void *));


    if (!Module) bserror(ERR_MALLOC,"main_mod_add");

    Module[RegMod-1] = dlopen(str, RTLD_NOW | RTLD_GLOBAL);
    if (!Module[RegMod-1]) {
        fprintf(stderr,"warning: cannot find module %s\n",str);
        fprintf(stderr,"%s\n",dlerror());
        exit(1);
    } else {
#ifdef _DEBUG
        fprintf(stderr,"DEBUG: Calling init_module in module %s\n",str);
#endif
        mod_init = dlsym(Module[RegMod-1],"init_module");
        if ((*mod_init)()) {
            fprintf(stderr,"warning: module %s can achieve its init phase\n",str);
            exit(1);
        }
#ifdef _DEBUG
        else printf("DEBUG: Module %s has been initialized\n",str);
#endif
    }
}

/*
:Function:       main_mod_destroy
:Description:    Closes all the opened modules
:Arguments:      none
:Return Value:   none
:Pre-condition:  The module handle table is initialized.
```

```
:Post-condition: none
*/
void main_mod_destroy() {

    int i;

    i = 0;
    while (i < RegMod) {
        if (Module[i]) dlclose(Module[i]);
        i++;
    }
    RegMod = 0;
}

/*
:Function:       main_api_init
:Description:    initializes the module functions & options table
:Arguments:      none
:Return Value:   none
:Pre-condition:  none
:Post-condition: - The module functions table is initialized.
                 - The module options table is initialized.
*/
void main_api_init() {
    Registered_Mod = 0;
    Registered_Opt = 0;
    Mod_F = NULL;
    Mod_O = NULL;
}

/*
:Function:       main_api_destroy
:Description:    Frees all the ressources allocated to the module functions
                 & options table
:Arguments:      none
:Return Value:   none
:Pre-condition:  - The module functions table is initialized.
                 - The module options table is initialized.
:Post-condition: none
*/
void main_api_destroy() {

    int i;

    i = 0;
```

```
    while (i < Registered_Mod) {
        free(Mod_F[i]);
        i++;
    }
    Registered_Mod = 0;

    i=0;
    while (i < Registered_Opt) {
        free(Mod_O[i]);
        i++;
    }

    Registered_Opt = 0;
}

/*
:Function:        api_register_module
:Description:     This function is called by the modules to register their
                  disaster recovery functions (recover, backup, print, info)
:Arguments:       - name (char *): pointer to module name
                  - save_func (int (*)()): pointer to the module recovery
                  function.
                  - load_func (int (*)()): pointer to the module backup
                  function.
                  - print_func (int (*)()): pointer to the module objects
                  information print function.
                  - info_func (int (*)()): pointer to the module information
                  function.
:Return Value:    (int): always 0
:Pre-condition:   - The module functions table is initialized.
:Post-condition:  - The module (*name) functions (recover,...) are registered
                  in the module functions table.
*/
int api_register_module(char *name, int (*save_func)(), int (*load_func)(),
    int (*print_func)(), int (*info_func)()) {

    if (!info_func) {
        fprintf(stderr,"ERROR: module %s didn't register the mandatory info function\n",n
        exit(1);
    }

    Registered_Mod++;
    Mod_F =  (Module_Func **) realloc(Mod_F, Registered_Mod*sizeof(Module_Func *));

    if (!Mod_F) bserror(ERR_MALLOC,"api_register_module()");
```

```
    Mod_F[Registered_Mod-1] = (Module_Func *) malloc(sizeof(Module_Func));

    if (!Mod_F[Registered_Mod-1]) bserror(ERR_MALLOC,"api_register_module()");

    strcpy(Mod_F[Registered_Mod-1]->name, name);
    Mod_F[Registered_Mod-1]->save_func = save_func;
    Mod_F[Registered_Mod-1]->load_func = load_func;
    Mod_F[Registered_Mod-1]->print_func = print_func;
    Mod_F[Registered_Mod-1]->info_func = info_func;
#ifdef _DEBUG
    printf("DEBUG: Module %s has registered itself\n",name);
#endif
    return 0;
}

/*
:Function:       api_register_option
:Description:    This function is called by the modules to register their
                 options.
:Arguments:      - option (char *): the option name
                 - mod_name (char *): the module name
                 - option_func (int (*)()): the function that handle the
                 function.
:Return Value:   (int): allways 0
:Pre-condition:  - The module options table is initialized.
:Post-condition: - The module (*mod_name) options are registered
                 in the module options table.
*/
int api_register_option(char *option, char *mod_name, int (*option_func)()) {

    Registered_Opt++;
    Mod_O =  (Module_Option **) realloc(Mod_O, Registered_Opt*sizeof(Module_Option *));

    if (!Mod_O) bserror(ERR_MALLOC,"api_register_option()");

    Mod_O[Registered_Opt-1] = (Module_Option *) malloc(sizeof(Module_Option));

    if (!Mod_O[Registered_Opt-1]) bserror(ERR_MALLOC,"api_register_option()");

    strcpy(Mod_O[Registered_Opt-1]->keyword, option);
    strcpy(Mod_O[Registered_Opt-1]->mod_name, mod_name);
    Mod_O[Registered_Opt-1]->option_func = option_func;
#ifdef _DEBUG
    printf("DEBUG: Module %s has registered the keyword %s\n",mod_name, option);
```

```
#endif
    return 0;
}
```

5.5 Remote Objects Database Backup/Restore

To keep it simple, we will use the backup/restoration possibilities of a backup product. During my training period, I used HSMS-CL. So, the backup and the restoration of the objects database will be done through API calls of the HSMS-CL product C-library.

Furthermore, it would be very useful if we could restore all the files backed up by HSMS-CL during normal backup operations. As we can see, our disaster recovery tool is "just"an extension of backup products. Our tool uses them to backup and to restore disaster recovery objects not processed by backup tools. In our case, for example, HSMS-CL does not backup block devices on unices, it is then up to a module of our tool to handle that task.

The reader might want to know why the tool does not permit the backup of the computer files. In fact, I personaly think that backups of files are part of a more general security strategy and have to be done apart. It is also obvious that files change more rapidly than computer objects such as partition tables or the file-system organization. This is why I choose to leave the backup of files out of the tool. The only problem of such implementation is that it is difficult to synchronize a backup and the disaster recovery database[8]. The only solution is to have a good local security policy that ensure that the last disaster recovery database matches the last backup. It is certain that the tool could be greatly enhanced concerning the backup/restore management and the integration with a backup tool (such as HSMS-CL). But it was not the purpose of this paper.

As the implementation of this functionality is particular to the backup tool used, the reader can find the implementation of the backup/restore functions of the database with HSMS-CL in the appendix. More information on the HSMS-CL API, HSMS-CL and HSMS in general can be found in [23, 24, 26, 27].

5.6 Application Configuration File

5.6.1 Motivation

In order to avoid an excessively complex usage of the tool, we will define the layout of a configuration file. It will be local to the protected computer and permit a fine-grained control of what has to be done by the tool.

The recovery operator, after setting up that file, would then only have to execute bsrecov with the commands we described at the beginning of this chapter. And that to accomplish his task.

[8]It's impossible to say if the restoration of a backup set and of disaster recovery objects will match

For a specified computer (and operating system), the configuration file must be set up once. This will make the disaster recovery operation easier. Of course, if the computer crashes completely, the configuration file will be lost. So, this file must be backed up by other means. One other possibility will be presented in section 5.7.

5.6.2 Description

As we said, all the management of the bsrecov application is done through the configuration file. It contains the modules that have to be used, and other parameters as we will see it later.

The file has the following layout:

- The file is organized in sections, beginning with [section].
- Only the main section is mandatory (section beginning with [main])
- The options are given using the *option = value* form.

Each section corresponds to a module. In fact, the main application is itself a module[9], named *main*. The executable is just a wrapper that calls a specific function of this main module.

This particular main module has three options:

Option	Description	Default value
module_path	defines the directory in which the modules are installed (without the trailing slash)	/opt/bsrecov/dso
inc_module	adds a module (path relative to module_path if not beginning with a slash)	none
proc_abs_mod	is used to tell bsrecov if the execution continues (yes) if a module is missing during the recovery or not (no)	no

Note: The order in which the inc_module options[10] are declared IS important as module functions are called in that order.

Other sections are module specific and are not mandatory. You can find an example of a real configuration file below. Empty lines or lines beginning with '#' are not processed by the configuration parser.

A Sample Configuration File

```
# Example of a bsrecov configuration file
# This one correspond to the linux 'port' of bsrecov
# It defines four modules:
```

[9] But it doesn't follow the convention described above in *module skeleton*

[10] We'd better speak about directive here. Because the correct behaviour of bsrecov depends on modules.

```
#
#       - test -> that show how a module works
#       - dpt  -> Disk Partition Table recvovery
#       - fs   -> file system structure recovery
#       - dev  -> device file recovery
#
[main]
module_path = /opt/bsrecov/dso
inc_module = libtest.so
inc_module = libdpt.so
inc_module = libfs.so
inc_module = libdev.so
proc_abs_mod = no

[testmod]
test = zorglub

[dpt]
# save_inv_mbr: Should the module save invalid master boot record?
save_inv_mbr = no

[fs]
#fstab_path: path to the file describing the file-system structure
fstab_path=/etc/fstab
```

5.7 BsRecov for Linux/i386: Practical Example

5.7.1 Presentation

As we saw, and it might be strange, bsrecov cannot perform any disaster recovery operations. It is because it has been developped to be available on various platforms. And to perform a real recovery, bsrecov will use modules adapted to the platform and operating system considered.

But there is also a point on which we have not insisted[11]. How to achieve a recovery after a total crash? Actually, bsrecov can't be executed by itself on a crashed or freshly assembled (or bought) computer. In fact, we cannot boot it up. We then need a way to boot the computer to let bsrecov do its work.

In this particular case (but the principle is valid for other platforms as well), a special linux installation is provided. We will insist on the way this installation was build (see

[11] But we mentioned it in chapter 4

docs in appendix). But we can describe the way it works: A three disk set and a CD-ROM or NFS installation is needed. The first two disks are used to boot the computer, to load needed drivers and to setup the network. Once it is done, the linux special installation furnished on the CD-ROM or accessed through NFS is mounted. The third disk is then accessed to load the computer specific configuration files[12] (bsrecov configuration file, HSMS-CL configuration files,...). The boot process is then terminated and it is up to the operator to launch the disaster recovery operation.

The first thing to do is of course to restore the disaster recovery objects database. Once it is done, the recovery operation can be performed (with a restoration of the files backed up with HSMS-CL, which is a command of bsrecov in its stock implementation). Once bsrecov has terminated, the computer can be rebooted and it should be restored completely. Of course, a very specific installation cannot be fully functional after the reboot and a bit of further administrative tasks could be necessary. But we should notice that modules could then be written to avoid such necessary administrative tasks.

It can happen for example when databases have to be restored. Actually, when the computer is booted with the bsrecov disk set, the database engine is not running. It is then impossible to restore the database. But it should be possible to write a module to handle that task. For example, instead of using the stock restoration possibility of bsrecov for files, we could write a module that restores the files themselves and then launches the database engine. Once the engine is running, another restoration process can be launched to restore the data. This is what was expected to happen when we spoke about the applications and operating systems specific files.

5.8 Conclusion

We have now the template of a disaster recovery tool. Basically, this tool should be easily ported on a variety of hardware and software platforms. But, it's a fact that the modules have all to be rewritten for each platforms. And that certainly a difficult issue. The sole thing to do is to apply the components approach to the new considered platform and to consult the technical documentations. But, providing a template application should greatly enhance the computer recovery facility in heterogenous environments.

Actually, the DR operator will benefit from the framework of this application by allowing him to work with a unified software. He could then concentrate himself on the recovery operations and is not obliged to follow all the reinstallation process needed previously.

Another avantage of the tool is its integration with backup softwares. Indeed, bsrecov and backup softwares together can face any kind of recovery (from simple files to a total computer recovery).

[12]The first two disks are suitable for every Linux/i386 platform and the third one is computer specific

Chapter 6

Conclusion

We are now at the end of this work. And although the title, we have more spoken about computer recovery than disaster recovery. Only because, in computer sciences, the computers seem to lose the focus. And then, when a computer (in its globality) has a problem, the big questions arise.

We work with them, we use them more and more, and we lose their control. I don't say they are intelligent but we are more and more unable to understand their complexity. Even in computer sciences, we use them as tools, without even considering a strong analysis of them. We know principles, and they are just our toys.

If we remember the backup problematic in the introduction of this work, we said the backup policies used nowadays were unappropriate. Because the workstations of every worker is now able to compute and to process a lot of data. Because those workstations are not just dumb terminals. So, when we lose one of them, because of a disaster, we lose more than datas, we lose our **workstation**.

We just get to the point, a computer is a tool, but a tool is important, especially when this tool becomes more and more complicated. If your local garage mechanic lose a screwer, chance is that he can get one back very soon. But if the same local garage mechanic loses the computer he uses to keep his books, clients data and invoices, he risks to have some very bad problems. Because the computer is now an important part of his daily work.

Now, can our local garage mechanic afford to pay to get the protections described in the chapter 1. At least, he will have a tape unit on which he makes backups. But although he makes backups, is he able to restore the data? Of course, a garage mechanic is perhaps not well versed in computers but he encounters the same problem than big enterprises if they lose all their workstations. Because no one can work anymore.

So I personaly think it is time to think about our workstations. Because a local administrator can say you how much time he passed to manage your workstation. Where I made my training period, the computers in use where modern ones, and however, the team that manages them had to intervene at least one time per week – day? –. What would happen to that team if they had to repair or to re-install all the computers installed? Certainly strong headaches and a lot of extra work hours.

Some reader could argue that workstations can be mastered and repaired rapidly. But I am not sure of that. Because a master installation becomes old rapidly, because a master installation needs updates, because some new softwares were installed after the master was made. Furthermore, as complex workstations are in use, they are adapted to every worker. So, in highly dynamic and heterogeneous environments, it is impossible to maintain a master up to date.

This is why I think backup softwares should be updated and really improved to be able to save unoperationnal data and to take into account computer recovery possibilities. This is what we tried to do with bsrecov. And that is what bsrecov do. I admit that bsrecov is not a complete recovery software, but that's just a prototype and I think some concepts could be very interesting for the development of a real computer recovery program.

Beside the fact that workstations are the poor childs of disaster recovery, the price is an important issue and this is why I think the solution we presented is the best we could take. Of course, the software approach can be difficult to implement and sometimes unhelpfull. But that is the only approach that enterprises could afford (and accept) to pay. Did anyone see a workstation with RAID level 1 disks in an average enterprise? I did not.

6.1 Feelings

I'd like to finally conclude with some feelings about this work. More than a final work for my studies, it was really an interesting thing to do. It permits me to write in an other language than my native language and to enhance my knowledge of english. And on a computer sciences point of view, this work brings me to a point that I never reached before. It permits me to rapproach the formal aspects of my studies and the technical ones of computer recovery.

It also permits me to approach the work world by making a training period outside the university. And that makes a big difference for a student like me: these two worlds are so different. I will never regret that. This work was more than a work, it really was a great experience.

Glossary

In order to understand this work correctly, it is mandatory to define a few concepts. The reader must be warned that misinterpretation of this paper could be induced by not keeping those definitions in mind.

Persons

Administrator A person that handles the implementation, the configuration and the maintenance of a computer center. He generally has full access right on any computers.

Operator A person that handles specific administration tasks in a computer center or on a specific computer. He often has privileged rights associated with the aim of his task. (Ex: Backup operator, Database operator)

User A person that works on a computer to accomplish his operational tasks. He generally has no special access rights.

Computers

Computer Center A connected and/or unconnected set of computer resources.

Computer Network A connected set of computer resources.

Operational task A task related to a company activities. For example, a backup is NOT an operational task for a winery, but is an operational task for a remote backup service provider. Accounting and resources management are examples of universal operational tasks.

Server A (powerful) computer that provides one or more services to other computers.

Client A computer that requests one or more services from a server.

Workstation A computer that is used to accomplish operational tasks. This term is often used to designate a client. I will distinguish the two notions here.

Control node A computer that controls one or more computers using a specific software through a network.

Procedures

Backup The action by which important data are saved for later retrieval -in case of deletion or corruption- on middle- or low-cost medias. A backup creates a **backup save-file** that contains the saved data. A property of a backup save-file is that each file contained in it should be available for individual retrieval.

Archival The action by which data are saved for legal reasons or to constitute the memory of an enterprise on low cost medias. This action create **archive save-file** that contains the archived data. Generally, it is not of concern to access individual file in a archive save-file.

Restoration or retrieval The action by which archived or backed up data are retrieved from an archive or a backup save-file.

DR Disaster Recovery.

Disaster Recovery The action by which a resource destroyed by an unexpected fault or external event is brought back to its operationnal state.

Bibliography

[1] Werner Almesberger. *LILO Generic boot loader for Linux Version 20, technical overview*, 1995.

[2] Werner Almesberger and Hans Lermen. *Using the initial RAM disk (initrd)*, 1996.

[3] H. Peter Anvin. *SYSLINUX V1.48: A boot loader for Linux using MS-DOS floppies*, September 1999.

[4] Benjamin Bayart. *Joli manuel pour LaTeX2ε*. ESIEE, 1995.

[5] Rémy Card, Eric Dumas, and Franck Mével. *The Linux kernel book*. John Wiley & Sons, 1998.

[6] J. Case, M. Fedor, M Schoffstall, and J. Davin. A simple network management protocol (SNMP). Technical report, Network Working Group, May 1990.

[7] Z. Cekro. Simple network management protocol (SNMP): Current standards and status. Technical report, Université Libre de Bruxelles, Faculté des Sciences, March 1998.

[8] Cederqvist et al. *Version management with CVS*. Signum Support AB, 1993.

[9] IBM Technology Group. Storage area networks: Putting data to work for e-businesses. Technical report, IBM, June 1999.

[10] Victor Hazlewood. Cluster computing: A survey and tutorial. *SysAdmin*, March 1997.

[11] Intel Corporation. *Intel Architecure Software Developer's Manual Volume 1: Basic Architecture*, 1997.

[12] Intel Corporation. *Intel Architecure Software Developer's Manual Volume 2: Instruction Set Reference Manual*, 1997.

[13] Intel Corporation. *Intel Architecure Software Developer's Manual Volume 3: System Programming Guide*, 1997.

[14] Legato. *Legato NetWorker, Disaster Recovery Guide*, 1998.

[15] Massachusetts Institute of Technology. *MIT business continuity plan*, 1995.

[16] Milan Milenković. *Operating systems, concepts and design.* McGraw-Hill International Editions, second edition, 1992.

[17] A. Philip Nelson. *The GNU Database Manager (GDBM) Man Page.* GNU, 1990.

[18] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). Technical report, Computer Science Division, Departement of Electrical Engineering and Computer Sciences, University of California, Berkley, 1987.

[19] M. Rose. Management information base for network management of TCP/IP-based internets: MIB-II. Technical report, Network Working Group, May 1990.

[20] M. Rose and McCloghrie. Structure and identification of management information for TCP/IP-based internets. Technical report, Network Working Group, May 1990.

[21] Mendel Rosenblum. *The Design and Implementation of a Log-structured File System.* Ph.D. dissertation, University of California, Berkeley, June 1992.

[22] Siemens. *HIPLEX AF V1.0A, Automatic switching of application between BS2000 Systems.*

[23] Siemens. *HSMS / HSMS-SV V4.0A (BS2000/OSD) Hierarchical Storage Management System Volume 1: Functions, Management and Installation, User Guide.*

[24] Siemens. *HSMS / HSMS-SV V4.0A (BS2000/OSD) Hierarchical Storage Management System Volume 2: Statements, User Guide.*

[25] Siemens. HSMS, centralized data backup in enterprises with BS2000/OSD. Brief description.

[26] Siemens. *HSMS-CL V4.0 BS2000 Backup Service for UNIX Systems, User Guide.*

[27] Siemens. *HSMS-CL V4.0 BS2000 Backup Service for Windows NT, User Guide.*

[28] W. Richard Stevens. *Unix network programming.* Prentice Hall, 1990.

[29] Andrew Tanenbaum. *Réseaux.* Prentice Hall, InterEditions, third edition, 1997.

[30] Matt Welsh and Lar Kaufman. *Running Linux.* O'Reilly & Associates, 1995.

[31] Leo A. Wrobel. *Writing Disaster Recovery Plans for Telecommunications Networks and LANs.* Artech House, 1993.

Appendix A

BsRecov/Linux/i386 sources

A.1 C Sources

A.1.1 main-wrapper.c

```
#include <dlfcn.h>


int main(int argc, char *argv[]) {

    void *core;
    char *dler;

    int (*core_entry)(int argc, char *argv[]);

    core = dlopen("/opt/bsrecov/dso/libcore.so", RTLD_GLOBAL | RTLD_LAZY);

    dler = dlerror();
    if (dler) {
        printf("%s\n",dler);
        exit(1);
    }

    if (core) {
        core_entry = dlsym(core, "main_core");
        printf(dlerror());
        if (core_entry) return (*core_entry)(argc, argv);
    } else {
        printf(dlerror());
        printf("\n");
        printf("Cannot open libcore.so\n");
        return 1;
```

```
    }
}
```

A.1.2 main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <gdbm.h>
#if 0
#include <HSMSCL_api.h>
#include <bspitype.h>
#endif
#include "config.h"
#include "modules.h"
#include "api.h"
#include "object.h"
#include "bserror.h"
#include "main.h"
#include "version.h"

#ifdef SINIX
#define strcasecmp strcmp
#endif

Module_Func **Mod_F;
Module_Option **Mod_O;
int Registered_Mod;
int Registered_Opt;
GDBM_FILE file;

GlobalConfig bsConf;

struct CmdLine {

    int backup:1;
    int restore:1;
    int save:1;
    int recover:1;
    int restorefiles:1;
    int info:1;
    int bad:1;
};
```

```
static struct CmdLine options;

static print_help() {

    printf("Usage:\n\tbsrecov [-backup | -restore] [-save | -recover
        [-frestore]]  [-modinfo]\n\n");
    printf("\t-backup    backup the recovery database from the BS2000 server\n");
    printf("\t-restore   restore the recovery database from the BS2000 server\n");
    printf("\t-save      save the recovery information in the recovery database\n");
    printf("\t-recover   recover the recovery information from the recovery database\n");
    printf("\t-frestore  restore all the backed up files from the HSMS-CL server\n");
    printf("\t-modinfo   print information on the installed recovery modules\n\n");
    printf("Note:\n");
    printf("\t-backup and -restore are mutually exclusive\n");
    printf("\t-save and -recover are mutually exclusive\n");
    printf("\tif -backup is combined with other options, it occurs last\n");
    printf("\tif -restore is combined with other options, it occurs first\n");
    printf("\t-frestore can only be used in combinaison with -recover\n\n");
    exit(255);
}

static void process_cmdline(int argc, char *argv[]) {

    int i;

    argc--;
    argv++;

    options.backup = 0;
    options.restore = 0;
    options.save = 0;
    options.recover = 0;
    options.restorefiles = 0;
    options.info = 0;

    if (!argc) {
        printf("Error: No command line argument found\n");
        print_help();
    }
    i = 0;
    while ((i < argc) && (!options.bad)) {

        if (!strcasecmp(argv[i], "-backup")) options.backup = 1;
        else if (!strcasecmp(argv[i], "-restore")) options.restore = 1;
        else if (!strcasecmp(argv[i], "-save")) options.save = 1;
```

```
        else if (!strcasecmp(argv[i], "-recover")) options.recover = 1;
        else if (!strcasecmp(argv[i], "-modinfo")) options.info = 1;
        else if (!strcasecmp(argv[i], "-frestore")) options.restorefiles = 1;
        else {
            options.bad = 1;
            printf("Error: Invalid option: %s\n",argv[i]);
            print_help();
        }
        i++;
    }

    /* mutually exclusive option check */

    if (options.backup && options.restore) {
        fprintf(stderr, "Error: options -backup and -restore are mutually exclusive\n");
        exit(255);
    }
    if (options.save && options.recover) {
        fprintf(stderr, "Error: options -save and -recover are mutually exclusive\n");
        exit(255);
    }
    if ((options.restorefiles) && (!options.recover)) {
        fprintf(stderr, "Error: options -frestore must be specified with -recover\n");
        fprintf(stderr, "If you want to restore files, use bsrest instead\n");
        exit(255);
    }
}

static void create_action_file(char *command) {

    FILE *f;
    unlink("/opt/bsrecov/tmp/bsrproc");
    f = fopen("/opt/bsrecov/tmp/bsrproc","w");
    fprintf(f,"%s",command);
    fclose(f);
}

static void destroy_action_file() {

    unlink("/opt/bsrecov/tmp/bsrproc");
}
static void load_recovery_inf() {

    MODULE_OBJECT module;
    int i;
```

```
    int ret;

    printf("Saving recovery information:\n");
    file = gdbm_open("/opt/bsrecov/db/recovery.db",512, GDBM_NEWDB, 384, 0x0);
    if (!file) {
        bserror(ERR_OPENBD,"main()");
    }
    init_hierarchy();
    i = 0;
    while (i < Registered_Mod) {
        strcpy(module.mod_name, Mod_F[i]->name);
        if ((ret = api_save_object((void *) &module, sizeof(module), TO_MODULE, 0)) < 0)
            bserror(ERR_WRITEDB,"main()");
        if (Mod_F[i]->info_func) (*(Mod_F[i])->info_func)();
        if (Mod_F[i]->load_func) (*(Mod_F[i])->load_func)(ret);
        else fprintf(stderr,"warning: module %s didn't register the load function\n",
            module.mod_name);
        i++;
    }

    save_hierarchy();
    gdbm_close(file);
}

static void recover_recovery_inf() {

    MODULE_OBJECT *module;
    int i;
    long ret;
    int type;
    int size;
    long long where;
    int found;

    printf("Recovering recovery information:\n");
    file = gdbm_open("/opt/bsrecov/db/recovery.db",512, GDBM_READER, 384, 0x0);
    if (!file) {
        bserror(ERR_OPENBD,"recover_recovery_inf()");
    }
    load_hierarchy();
    ret = findfirst_hierarchy(0, &where);
    while (ret) {
        found = 0;
        module = (MODULE_OBJECT *) api_load_object(ret, &type, &size);
        if (!module) bserror(ERR_READDB,"recover_recovery_inf()");
```

```
        i = 0;
        while (i < Registered_Mod) {
            if (!strcmp(module->mod_name,Mod_F[i]->name)) {
                found=1;
                if (Mod_F[i]->info_func) (*(Mod_F[i])->info_func)();
                if (Mod_F[i]->save_func) (*(Mod_F[i])->save_func)(ret);
                else fprintf(stderr,"warning: module %s didn't register the save
                    function\n", module->mod_name);
            }
            i++;
        }
        if ((!found) && (bsConf.Absent_Module)) {
            fprintf(stderr,"ERROR: a necessary module (%s) was not found\n",
                module->mod_name);
            exit(255);
        }
        free(module);
        ret = findnext_hierarchy(0, &where);
    }
    gdbm_close(file);
}


static void print_module_info() {

    int i;

    i = 0;
    printf("Module information:\n");
    while (i < Registered_Mod) {
        if (Mod_F[i]->info_func) (*(Mod_F[i])->info_func)();
        i++;
    }
}

static int completed_ok(char *rpt) {

    FILE *f;
    char str[1024];
    char *strp;
    int status;

    f = fopen(rpt,"r");
    strcpy(str,"");
```

```
    fgets(str, 1023, f);
    while (!feof(f) || strcmp(str,"")) {
        if (!strncmp(str,"RSU",3)) {
            strp = str;
            strp = strp+8;
            if (!strncmp(strp,"WITHOUT",7)) status = 1;
            else status = 0;
            break;
        }
        strcpy(str,"");
        fgets(str, 1023, f);
    }
    fclose(f);
    return status;
}


static void restore_recovery_db() {
#if 0
    bsrest_params bsRest;
    bspi_event_info bsInfo;
    bspi_id id;
    int stop;
    char rptPath[1024];

    const PATHCHAR *inc_files[] = { "/opt/bsrecov/db/recovery.db", NULL };

    printf("Restoring recovery data...");
#ifdef _DEBUG
    fprintf(stderr,"DEBUG: Recovery data restoration in development\n");
#endif
    memset(&bsRest, 0, sizeof(bsRest));

    bsRest.include = inc_files;
    bsRest.exclude = NULL;
    bsRest.recursion = path;
    bsRest.fn_case = case_sensitive;
    bsRest.source = from_backup;
    bsRest.replace = rpl;
    bsRest.path_src = NULL;
    bsRest.path_dst = NULL;
    bsRest.fn_prefix = NULL;
    bsRest.fn_suffix = NULL;
    bsRest.saveversion = sv_latest;
    bsRest.arch_name = NULL;
```

```
    bsRest.report = full;
    bsRest.script = NULL;
    bsRest.req_name = "BSDRREST";


#ifdef _DEBUG
    HSMSCL_set_host("d241p156");
    HSMSCL_set_port("1234");
#endif

    if (stop = HSMSCL_open()) {
        fprintf(stderr,"Error: Cannot open a HSMS connection, code: %d\n",stop);
    }

    if ((id = HSMSCL_rest(&bsRest, public_request)) < 1) {
        fprintf(stderr,"Error: Cannot proceed the restoration, code: %d\n",id);
    }

    stop = 0;
    HSMSCL_status(id, &bsInfo);
    while (!stop) {
#ifdef _DEBUG
        fprintf(stderr,"DEBUG: Waiting for an eventual event\n");
#endif
        switch (bsInfo.state) {
            case req_rejected :
                printf("Error: DR data restoration rejected by the server\n");
                HSMSCL_close();
                stop = 1;
                break;
            case req_available :
                stop = 1;
                break;
            case req_deleted :
                printf("Warning: The request has been deleted\n");
                stop = 1;
                break;
            default :
                sleep(1);
                if (HSMSCL_status(id, &bsInfo)) bsInfo.state = req_deleted;
        }
    }
    if (bsInfo.state == req_available) {
        strcpy(rptPath,"/opt/bsrecov/tmp/bsrecrapres");
        HSMSCL_gimme(id, rptPath);
```

```
        HSMSCL_close();
        printf("\nThe restore rapport is located under %s\n",rptPath);
        if (!completed_ok("/opt/bsrecov/tmp/bsrecrapres")) {
            printf("Error: restoration completed with errors,
                see /tmp/bsrecrapres\n");
        }
#ifdef _DEBUG
        else {
            fprintf(stderr,"DEBUG: restoration completed without error\n");
        }
#endif

    }
    printf("\nDone.\n");
#endif
}

static void backup_recovery_db() {
#if 0
    bsarch_params bsBack;
    bspi_event_info bsInfo;
    bspi_id id;
    int stop;
    char rptPath[1024];

    const PATHCHAR *inc_files[] = { "/opt/bsrecov/db/recovery.db", NULL };

    printf("Backing up recovery data...");
#ifdef _DEBUG
    fprintf(stderr,"DEBUG: Recovery data backup in development\n");
#endif
    memset(&bsBack, 0, sizeof(bsBack));

    bsBack.include = inc_files;
    bsBack.exclude = NULL;
    bsBack.recursion = path;
    bsBack.fn_case = case_sensitive;
    bsBack.backup = full_copy;
    bsBack.erase = noerase;
    bsBack.compress = no_comp;
    bsBack.savefile = std_sf;
    bsBack.sv_name = NULL;
    bsBack.arch_name = NULL;
    bsBack.report = full;
    bsBack.script = NULL;
```

```
    bsBack.req_name = "BSDRBACK";


#ifdef _DEBUG
    HSMSCL_set_host("d241p156");
    HSMSCL_set_port("1234");
#endif
    if (stop = HSMSCL_open()) {
        fprintf(stderr,"Error: Cannot open a HSMS connection, code: %d\n",stop);
    }

    if ((id = HSMSCL_back(&bsBack, public_request)) < 1) {
        fprintf(stderr,"Error: Cannot proceed a backup, code: %d\n",id);
    }

    stop = 0;
    HSMSCL_status(id, &bsInfo);
    while (!stop) {
#ifdef _DEBUG
        fprintf(stderr,"DEBUG: Waiting for an eventual event\n");
#endif
        switch (bsInfo.state) {
            case req_rejected :
                printf("Error: DR data backup rejected by server\n");
                HSMSCL_close();
                stop = 1;
                break;
            case req_deleted :
                printf("Warning: The request has been deleted\n");
                stop = 1;
                break;
            case req_available :
                stop = 1;
                break;
            default :
                sleep(1);
                if (HSMSCL_status(id, &bsInfo)) bsInfo.state = req_deleted;
        }
    }
    if (bsInfo.state == req_available) {
        strcpy(rptPath,"/opt/bsrecov/tmp/bsrecrapbck");
        HSMSCL_gimme(id, rptPath);
        HSMSCL_close();
        printf("\nThe backup rapport is located under %s\n",rptPath);
        if (!completed_ok("/opt/bsrecov/tmp/bsrecrapbck")) {
```

```
            fprintf(stderr,"Error: backup completed with errors, see /tmp/bsrecrapbck\n")
        }
#ifdef _DEBUG
        else {
            fprintf(stderr,"DEBUG: backup completed without error\n");
        }
#endif
    }
#endif
    printf("\nDone.\n");
}



static void recover_all_files() {
#if 0
    bsrest_params bsRest;
    bspi_event_info bsInfo;
    bspi_id id;
    int stop;
    char rptPath[1024];

    const PATHCHAR *inc_files[] = { "/", NULL };

    printf("Restoring files...\n");
#ifdef _DEBUG
    fprintf(stderr,"DEBUG: Recovery data restoration in development\n");
#endif
    memset(&bsRest, 0, sizeof(bsRest));

    bsRest.include = inc_files;
    bsRest.exclude = NULL;
    bsRest.recursion = all_files;
    bsRest.fn_case = case_sensitive;
    bsRest.source = from_backup;
    bsRest.replace = forcerpl;
    bsRest.path_src = NULL;
    bsRest.path_dst = NULL;
    bsRest.fn_prefix = "/recov";
    bsRest.fn_suffix = NULL;
    bsRest.saveversion = sv_latest;
    bsRest.arch_name = NULL;
    bsRest.report = full;
    bsRest.script = NULL;
    bsRest.req_name = "BSDRFRES";
```

```
#ifdef _DEBUG
    HSMSCL_set_host("d241p156");
    HSMSCL_set_port("1234");
#endif

    if (stop = HSMSCL_open()) {
        fprintf(stderr,"Error: Cannot open a HSMS connection, code: %d\n",stop);
    }

    if ((id = HSMSCL_rest(&bsRest, public_request)) < 1) {
        fprintf(stderr,"Error: Cannot proceed the files restoration, code: %d\n",id);
    }

    stop = 0;
    HSMSCL_status(id, &bsInfo);
    while (!stop) {
#ifdef _DEBUG
        fprintf(stderr,"DEBUG: Waiting for an eventual event\n");
#endif
        switch (bsInfo.state) {
            case req_rejected :
                printf("Error: files restoration rejected by the server\n");
                HSMSCL_close();
                stop = 1;
                break;
            case req_available :
                stop = 1;
                break;
            case req_deleted :
                printf("Warning: The request has been deleted\n");
                stop = 1;
                break;
            default :
                sleep(1);
                if (HSMSCL_status(id, &bsInfo)) bsInfo.state = req_deleted;
        }
    }
    if (bsInfo.state == req_available) {
        strcpy(rptPath,"/opt/bsrecov/tmp/bsrecrapfres");
        HSMSCL_gimme(id, rptPath);
        HSMSCL_close();
        printf("\nThe file restore rapport is located under %s\n",rptPath);
        if (!completed_ok("/opt/bsrecov/tmp/bsrecrapfres")) {
            printf("Error: files restoration completed with errors,
```

```
                see /tmp/bsrecrapres\n");
        }
#ifdef _DEBUG
        else {
            fprintf(stderr,"DEBUG: files restoration completed without error\n");
        }
#endif

    }
#endif
    printf("Done.\n");

}



int main_core(int argc, char *argv[]) {

/*
    Local variable definition
*/
/*
    initialization phase
*/
    printf("bsrecov version %s.%s-%s\n\n", VMAJOR, VMINOR, VTYPE);
    process_cmdline(argc, argv);
    main_api_init();
    main_mod_init();
    init_config();

/*
    Configuration applied reading
*/
    if (options.info) print_module_info();
    if (options.restore) {
        create_action_file("RESTORE");
        restore_recovery_db();
    }

    if (options.save) {
        create_action_file("SAVE");
        load_recovery_inf();
    }
    if (options.recover) {
        create_action_file("RECOVER");
```

```
        recover_recovery_inf();
        if (options.restorefiles) {
            recover_all_files();
        }
    }
    if (options.backup) {
        create_action_file("BACKUP");
        backup_recovery_db();
    }

/*
    Global Thermo-Nuclear Destruction phase
*/
    destroy_action_file();
    main_api_destroy();
    main_mod_destroy();
}
```

A.1.3 object.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <gdbm.h>
#include "object.h"
#include "bserror.h"



extern GDBM_FILE file;

/* Key_id generated start at 1 since key_id 0 is reserved for the hierarchy table */

long personal_id=1;

/* data used to represent the object hierarchy */

static long long *h_table;
static long long hierarchy_size;
static long long hierarchy_top;
static long long hierarchy_pos;
```

```
/*
:Function:       generate_key
:Arguments:      none
:Return Value:   unique key (long)
:Pre-condition:  none
:Post-condition: the key generated is unique
*/
long generate_key() {

        personal_id++;
        return (personal_id-1);
}

/*
:Function:       init_hierarchy
:Arguments:      none
:Return Value:   none
:Pre-condition:  none
:Post-condition: the hierarchy table h_table is initialized
*/
void init_hierarchy() {

        hierarchy_size = 0;
        hierarchy_top = 10;
        h_table = (long long *)
            malloc(hierarchy_top*sizeof(long long));
        if (!h_table) bserror(ERR_MALLOC,"init_hierarchy()");
}

/*
:Function:       findnext_hierarchy
:Arguments:      - parent (long) : parent key identifier
                 - where (long long *) : private variable
:Return Value:   a child key identifier (long)
:Pre-condition:  none
:Post-condition: the returned value is a child of parent. If the
                 parent doesn't exist of has no more child, the
                 return value is 0.
*/
long findnext_hierarchy(long parent, long long *where) {

        int ok;
        long value;
        long child, par;
```

```
        ok = 1;
        value= 0;

        hierarchy_pos = *where;
        while ((ok) && (hierarchy_pos < hierarchy_size)) {

                child = h_table[hierarchy_pos] & 0xffffffff;
                par = h_table[hierarchy_pos] >> 32;

                if (par == parent) {
                        ok = 0;
                        value = child;
                }
                hierarchy_pos++;
        }
        *where=hierarchy_pos;
        return value;
}

/*
:Function:       findfirst_hierarchy
:Arguments:      - parent (long) : parent key identifier
                 - where (long long *) : private variable
:Return Value:   the first child key identifier of the parent (long)
:Pre-condition:  none
:Post-condition: the returned value is the first child of parent. If
                 the parent doesn't exist or has no child, the
                 return value is 0.
*/
long findfirst_hierarchy(long parent, long long *where) {

        hierarchy_pos = 0;
        *where = 0;
        return findnext_hierarchy(parent, where);
}

/*
:Function:       add_hierarchy
:Arguments:      - parent (long) : parent key identifier
                 - child (long) : child key identifier
:Return Value:   none
:Pre-condition:  The hierarchy table (h_table) must be initialized.
:Post-condition: the new parent/child pair is in the hierarchy table
*/
void add_hierarchy(long parent, long child) {
```

```
        hierarchy_size++;
        if (hierarchy_size == hierarchy_top) {
                hierarchy_top += 10;
                h_table = (long long *)
                    realloc((void *) h_table,
                    hierarchy_top*sizeof(long long));
                if (!h_table) bserror(ERR_MALLOC,"add_hierarcht()");
        }

        h_table[hierarchy_size-1] = parent;
        h_table[hierarchy_size-1] = (h_table[hierarchy_size-1] << 32)
            | child;
}

/*
:Function:       save_hierarchy
:Arguments:      none
:Return Value:   none
:Pre-condition:  - The hierarchy table (h_table) must be initialized
                 - The GDBM database (file) is opened
:Post-condition: The hierarchy table is saved under the key 0 in the GDBM
                 database (file).
*/
void save_hierarchy() {

        datum key, content;
        long keyid;

        keyid=0;

        key.dptr = (void *) &keyid;
        key.dsize = sizeof(long);

        content.dptr = (void *) h_table;
        content.dsize =  hierarchy_size*sizeof(long long);

        if (gdbm_store(file, key, content, 0))
            bserror(ERR_WRITEDB,"save_hierarchy()");
}

/*
:Function:       load_hierarchy
:Arguments:      none
:Return Value:   none
```

```
:Pre-condition:  GDBM database (file) is opened.
:Post-condition: The hierarchy table is initialized with the hierarchy
                 table found in the GDBM database (file).
*/
void load_hierarchy() {

        datum key, content;
        long keyid;

        keyid = 0;
        key.dptr = (void *) &keyid;
        key.dsize = sizeof(long);

        content = gdbm_fetch(file, key);

        if (!content.dptr) bserror(ERR_OBJNF,"load_hierarchy()");
        h_table=(long long *) content.dptr;
        hierarchy_size = (content.dsize / 8);
        hierarchy_pos = 0;
        hierarchy_top = (content.dsize /8)+1;
}
```

A.1.4 api.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <gdbm.h>
#include <dlfcn.h>
#include "bserror.h"
#include "object.h"
#include "modules.h"


/*
    those variables must be declared elsewhere, in the main application module
    for example
*/

/*
    The module functions table and the variable that counts the number of
    registered modules
```

```
*/
extern Module_Func **Mod_F;
extern int Registered_Mod;

/*
    The module option table and the variable that counts the number of
    registered modules
*/
extern Module_Option **Mod_O;
extern int Registered_Opt;

/*
    The module handle table and the variable that counts the number of
    opened modules
*/
static void **Module;
static int RegMod;

GDBM_FILE file;

/* the following two functions CAN NOT be used by module */

/*
:Function:       main_mod_init
:Description:    initializes the module handle table
:Arguments:      none
:Return Value:   none
:Pre-condition:  none
:Post-condition: The module handle table is initialized.
*/
void main_mod_init() {
    RegMod = 0;
    Module = NULL;
}


/*
:Function:       main_mod_add
:Description:    initiates the module registration process
:Arguments:      str (char *): a dynamic shared object pathname
:Return Value:   none
:Pre-condition:  - The module handle table is initialized.
                 - Dynamic shared object pathname is correct.
                 - The dynamic shared object is a well-constructed object.
:Post-condition: - The module is registered in the module handle table.
```

```
                - The module functions are registered in the module
                functions table.
                - The module options (if any) are registered in the
                module options table.
*/
void main_mod_add(char *str) {

    int (*mod_init)();


    RegMod++;
    Module =  (void **) realloc(Module, RegMod*sizeof(void *));


    if (!Module) bserror(ERR_MALLOC,"main_mod_add");

    Module[RegMod-1] = dlopen(str, RTLD_NOW | RTLD_GLOBAL);
    if (!Module[RegMod-1]) {
        fprintf(stderr,"warning: cannot find module %s\n",str);
        fprintf(stderr,"%s\n",dlerror());
        exit(1);
    } else {
#ifdef _DEBUG
        fprintf(stderr,"DEBUG: Calling init_module in module %s\n",str);
#endif
        mod_init = dlsym(Module[RegMod-1],"init_module");
        if ((*mod_init)()) {
            fprintf(stderr,"warning: module %s can achieve its init phase\n",str);
            exit(1);
        }
#ifdef _DEBUG
        else printf("DEBUG: Module %s has been initialized\n",str);
#endif
    }
}


/*
:Function:       main_mod_destroy
:Description:    Closes the all the opened modules
:Arguments:      none
:Return Value:   none
:Pre-condition:  The module handle table is initialized.
:Post-condition: none
*/
```

```
void main_mod_destroy() {

    int i;

    i = 0;
    while (i < RegMod) {
        if (Module[i]) dlclose(Module[i]);
        i++;
    }
    RegMod = 0;
}

/*
:Function:       main_api_init
:Description:    initializes the module functions & options table
:Arguments:      none
:Return Value:   none
:Pre-condition:  none
:Post-condition: - The module functions table is initialized.
                 - The module options table is initialized.
*/
void main_api_init() {
    Registered_Mod = 0;
    Registered_Opt = 0;
    Mod_F = NULL;
    Mod_O = NULL;
}

/*
:Function:       main_api_destroy
:Description:    Frees all the ressources allocated to the module functions
                 & options table
:Arguments:      none
:Return Value:   none
:Pre-condition:  - The module functions table is initialized.
                 - The module options table is initialized.
:Post-condition: none
*/
void main_api_destroy() {

    int i;

    i = 0;
    while (i < Registered_Mod) {
        free(Mod_F[i]);
```

```
        i++;
    }
    Registered_Mod = 0;

    i=0;
    while (i < Registered_Opt) {
        free(Mod_O[i]);
        i++;
    }

    Registered_Opt = 0;
}

/* Here are the API functions that can be used by modules */

/*
:Function:        api_register_module
:Description:     This function is called by the modules to register their
                  disaster recovery functions (recover, backup, print, info)
:Arguments:       - name (char *): pointer to module name
                  - save_func (int (*)()): pointer to the module recovery
                  function.
                  - load_func (int (*)()): pointer to the module backup
                  function.
                  - print_func (int (*)()): pointer to the module objects
                  information print function.
                  - info_func (int (*)()): pointer to the module information
                  function.
:Return Value:    (int): allways 0
:Pre-condition:   - The module functions table is initialized.
:Post-condition:  - The module (*name) functions (recover,...) are registered
                  in the module functions table.
*/
int api_register_module(char *name, int (*save_func)(), int (*load_func)(),
    int (*print_func)(), int (*info_func)()) {

    if (!info_func) {
        fprintf(stderr,"ERROR: module %s didn't register the mandatory info function\n",
            name);
        exit(1);
    }

    Registered_Mod++;
    Mod_F =  (Module_Func **) realloc(Mod_F, Registered_Mod*sizeof(Module_Func *));
```

```
    if (!Mod_F) bserror(ERR_MALLOC,"api_register_module()");

    Mod_F[Registered_Mod-1] = (Module_Func *) malloc(sizeof(Module_Func));

    if (!Mod_F[Registered_Mod-1]) bserror(ERR_MALLOC,"api_register_module()");

    strcpy(Mod_F[Registered_Mod-1]->name, name);
    Mod_F[Registered_Mod-1]->save_func = save_func;
    Mod_F[Registered_Mod-1]->load_func = load_func;
    Mod_F[Registered_Mod-1]->print_func = print_func;
    Mod_F[Registered_Mod-1]->info_func = info_func;
#ifdef _DEBUG
    printf("DEBUG: Module %s has registered itself\n",name);
#endif
    return 0;
}

/*
:Function:       api_register_option
:Description:    This function is called by the modules to register their
                 options.
:Arguments:      - option (char *): the option name
                 - mod_name (char *): the module name
                 - option_func (int (*)()): the function that handle the
                 function.
:Return Value:   (int): allways 0
:Pre-condition:  - The module options table is initialized.
:Post-condition: - The module (*mod_name) options are registered
                 in the module options table.
*/
int api_register_option(char *option, char *mod_name, int (*option_func)()) {

    Registered_Opt++;
    Mod_O =  (Module_Option **) realloc(Mod_O, Registered_Opt*sizeof(Module_Option *));

    if (!Mod_O) bserror(ERR_MALLOC,"api_register_option()");

    Mod_O[Registered_Opt-1] = (Module_Option *) malloc(sizeof(Module_Option));

    if (!Mod_O[Registered_Opt-1]) bserror(ERR_MALLOC,"api_register_option()");

    strcpy(Mod_O[Registered_Opt-1]->keyword, option);
    strcpy(Mod_O[Registered_Opt-1]->mod_name, mod_name);
    Mod_O[Registered_Opt-1]->option_func = option_func;
#ifdef _DEBUG
```

```
    printf("DEBUG: Module %s has registered the keyword %s\n",mod_name, option);
#endif
    return 0;
}

/*
:Function:       api_load_object
:Arguments:      - ref (long): a object identifier
                 - type (int *)
                 - d_size (int *)
:Return Value:   (void *): a pointer to the object with ref as key
:Pre-condition:  GDBM database (file) is opened.
:Post-condition: - if the object with key identifier ref exists in the
                 database, *type contains the properties of the object,
                 *d_size contains the size of the object in bytes and
                 the returned pointer points to the object itself.
                 - if the object doesn't exist, the returned pointer is NULL
                 and the content of *type and *d_size is undeterminated.
*/
void *api_load_object(long ref, int *type, int *d_size) {

    datum key, content;
    GObject temp;
    void *retval;

    retval = NULL;

    key.dptr=(void *) &ref;
    key.dsize=sizeof(long);

    content = gdbm_fetch(file, key);
    if (!content.dptr) goto err;

    *type = ((GObject *) content.dptr)->type;

    key.dptr = (void *) &(((GObject *) content.dptr)->ref);
    key.dsize = sizeof(long);

    content = gdbm_fetch(file, key);

    if (!content.dptr) bserror(ERR_INCON,"api_load_object()");
    *d_size=content.dsize;

    retval = content.dptr;
```

```
err:
    return retval;
}

/*
:Function:       api_save_object
:Arguments:      - data (void *): a pointer to an object
                 - d_size (int): the size of the object in bytes
                 - type (int): the properties of the object
                 - parent (long): the key identifier of the parent object
:Return Value:   (long): the key identifier of the saved object.
:Pre-condition:  GDBM database (file) is opened.
:Post-condition: - if the return value is > 0, the object is saved in the
                 GDBM database and its key ientifier is equal to the
                 return value. The new parent/child pair (parent/retval)
                 is put in the hierarchy table.
                 - if the return value is equal to -1, the operation has failed.
*/
long api_save_object(void *data, int d_size, int type, long parent) {

      datum key, content;
      GObject temp;
      long key_id;
      int retval;

      retval = -1;
      key_id = generate_key();
      temp.type=type;
      temp.ref=generate_key();

      key.dptr=(void *) &key_id;
      key.dsize = sizeof(long);

      content.dptr=(void *) &temp;
      content.dsize=sizeof(GObject);

      if (gdbm_store(file,key,content,0)) goto err;

      key.dptr=(void *) &temp.ref;
      key.dsize=sizeof(long);

      content.dptr=data;
      content.dsize=d_size;

      if (gdbm_store(file,key,content,0)) {
```

```
            key.dptr=(void *) &key_id;
            key.dsize=sizeof(long);
            if (gdbm_delete(file,key)) {
                bserror(ERR_INCON,"api_save_object()");
            }
            goto err;
        }

        add_hierarchy(parent, key_id);
        retval = key_id;
err:
        return retval;
}
```

A.1.5 config.c

```
#include <stdio.h>
#include <stdlib.h>
#include "api.h"
#include "modules.h"
#include "config.h"
#include "bserror.h"



extern GlobalConfig bsConf;
static char curSection[18];

/*
    If someone wants to get information provided in bsConf, he has to
    provide access function to the wanted structure.
*/


extern Module_Option **Mod_O;
extern int Registered_Opt;


void init_global_config() {

    strcpy(bsConf.Module_Path,"");
    bsConf.Absent_Module = 0;

}
```

```
char *wskip(char *str) {

    while (*str == ' ') {
        str++;
    }
    return str;
}

int iscomment(char *str) {
    char *temp;

    temp = wskip(str);

    if ((*temp == '#') || (*temp == '\0')) return 1;
    else return 0;
}

void string_terminate(char *str) {

    int i;

    i = 0;
    while (str[i] != '\0') i++;

    i--;
    while (str[i] == ' ') {
        i--;
    }
    i++;
    str[i] = '\0';
}

/*
    This function gets the keyword on a line, puts it in result and returns
    a pointer to the parameter (i.e.:  = /usr/lib/libdpt.so)
*/

char  *getkeyword(char *str, char *result) {

    strcpy(result,"");

    while ((*str != ' ') && (*str != '=') && (*str != '\0')) {
        *result = *str;
        result++;
```

```
        str++;
    }
    *result = '\0';
    if (*str != '\0') {
        str = wskip(str);
    }
    if (*str == '=') {
        str++;
        str=wskip(str);
    }
    string_terminate(str);
    return str;
}


int conf_inc_module(char *str) {

    int retval;
    char temp[255];

    retval=-1;

    if (!strcmp(str,"")) {
        fprintf(stderr, "warning: no parameter supplied");
    } else {
        retval=0;
        if ((*str == '/') || (*str == '.')) {
            main_mod_add(str);
        } else {
            strcpy(temp, bsConf.Module_Path);
            if (temp[strlen(temp)-1] != '/') strcat(temp,"/");
            strcat(temp,str);
            main_mod_add(temp);
        }
    }
    return retval;
}

int conf_module_path(char *str) {

    int retval;

    retval = -1;
    if (!strcmp(str,"")) {
        fprintf(stderr, "warning: No parameter supplied\n");
```

```
    } else {
        string_terminate(str);
        strcpy(bsConf.Module_Path,str);
        retval = 0;
#ifdef _DEBUG
        fprintf(stderr,"The module path has been set to %s\n",
            bsConf.Module_Path);
#endif
    }
    return retval;
}

int conf_abs_mod(char *str) {

    int retval;

    bsConf.Absent_Module = 2;

    if (!strcmp(str,"")) {
        fprintf(stderr, "warning: No parameter supplied\n");
    } else {
        string_terminate(str);
        if (strcasecmp(str,"yes")) bsConf.Absent_Module = 1;
        if (strcasecmp(str,"no")) bsConf.Absent_Module = 0;
    }

    if (bsConf.Absent_Module == 2) {
        fprintf(stderr, "warning: invalid option (%s): proc_abs_mod option
            expected yes or no\n",str);
        bsConf.Absent_Module = 0;
    }
}
void register_standard_option() {

    /* here, we register the standard options */

    api_register_option("inc_module", "main", conf_inc_module);
    api_register_option("module_path", "main", conf_module_path);
    api_register_option("proc_abs_mod", "main", conf_abs_mod);
}

int process_line(char *str) {

    char *temp;
    char keyword[18];
```

```
    int i;
    int retval;
    int found=0;

    temp = getkeyword(str, keyword);

    i=0;
    while (i < Registered_Opt) {
        if ((!strcmp(keyword, Mod_O[i]->keyword)) &&
          (!strcmp(curSection, Mod_O[i]->mod_name))) {
            found = 1;
            retval = (*(Mod_O[i])->option_func)(temp);
#ifdef _DEBUG
            if (retval )
            fprintf(stderr,"The handler function for option %s in module %s
                has failed\n", keyword, Mod_O[i]->mod_name);
#endif
        }
        i++;
    }
    if (!found) {
        printf("Invalid option found in section [%s]: %s\n",curSection, keyword);
    }
}


void getcurSection(char *str) {

    char *temp;

    temp = curSection;

    while ((*str != ']') && (*str != '\0')) {
        *temp = *str;
        temp++;
        str++;
    }
    if (*str != ']') {
        fprintf(stderr,"warning: ']' expected in section definition\n");
    }
    *temp = '\0';
}


int init_config() {
```

```
    FILE *conFile;
    char conLine[255];

/*
    FIXME: we should initialize the global config structure
           before doing anything
*/

    register_standard_option();

    conFile = fopen("/etc/bsrecovrc","r");
    if (!conFile) {
        bserror(ERR_NOCONF,"process_line");
    }
    strcpy(conLine,"");
    fgets(conLine, 254, conFile);
    while ((!feof(conFile)) || (strcmp(conLine,""))) {
        if (conLine[strlen(conLine)-1] == '\n')
            conLine[strlen(conLine)-1] = '\0';
        if (*conLine=='[') getcurSection(conLine+1);
        else if (!iscomment(conLine)) process_line(conLine);
        strcpy(conLine,"");
        fgets(conLine, 254, conFile);
    }
    fclose(conFile);
}
```

A.1.6 bserror.c

```
#include <nl_types.h>
#include "bserror.h"
#include <stdio.h>

void bserror(int code, char *where) {

    nl_catd fd;

    fd = catopen("BSRECOV",0);
    if (!fd) {
        fprintf(stderr,"Cannot find error message for error %d\n",code);
        exit(255);
    }
#ifdef _DEBUG
    fprintf(stderr,"In %s:\n",where);
```

```
#endif
    fprintf(stderr,"Error %d : %s\n",code,catgets(fd,1,-code,"No description available"))
    catclose(fd);
    exit(-code);


}
```

A.2 Include files

A.2.1 main.h

```
/*
    Object types are module specific so there is no need to choose a unique
    one
*/

typedef struct {
    char mod_name[18];
} MODULE_OBJECT;

#define TO_MODULE 1
```

A.2.2 object.h

```
typedef struct {
    int type;
    long ref;
 } GObject;

long generate_key() ;
void init_hierarchy() ;
long findnext_hierarchy(long parent, long long *where) ;
long findfirst_hierarchy(long parent, long long *where) ;
void add_hierarchy(long parent, long child) ;
void save_hierarchy() ;
void load_hierarchy() ;
```

A.2.3 api.h

```
void main_mod_init() ;
void main_mod_add(char *str) ;
void main_mod_destroy() ;
```

```
*/

void *api_load_object(long ref, int *type, int *d_size) ;
long api_save_object(void *data, int d_size, int type, long parent) ;

/*
    Object Hierarchy specific API
*/

int findnext_hierarchy(long parent, long long *where) ;
int findfirst_hierarchy(long parent, long long *where) ;
```

A.3 Makefile

```
include Makefile.inc

INCLUDE=api.h.in config.h.in object.h.in bserror.h.in
HEADER=api.h config.h object.h bserror.h
COMP=api config object bserror
MOD=api.o config.o object.o bserror.o main.o
APPNAME=bsrecov
CORE=libcore.so
LIBS=-lgdbm_a -ldl -lstdc++


.SUFFIXES:
.SUFFIXES: .h .h.in .c .o


all: check libcore.so bsrecov
        @echo "Done."

check:
        @if [ ! -f Build ]; then \
        echo "********************** WARNING **********************"; \
        echo "                Use makemake.sh first"; \
        echo "*****************************************************"; \
        exit 1; fi
libcore.so: mkinc $(MOD)
        cd dso ; $(MAKE)
        $(CC) $(INCPATH) $(LIBPATH) $(CFLAGS) $(SHOPT) -o $(CORE) $(MOD) $(LIBS)


bsrecov: main-wrapper.o
```

```
	$(CC) $(CFLAGS) -o $(APPNAME) main-wrapper.o -ldl

.h.h.in:
	cp $< $@
	./makeinclude $(*F).c >> $@

mkinc: $(HEADER) $(INCLUDE)

.c.o:
	$(CC) -c $(INCPATH) $(CFLAGS) -o $@ $<

clean:
	cd dso; $(MAKE) clean
	rm -f *.o
	rm -f $(APPNAME)
	rm -f $(CORE)
	rm Build
install: all
	./install-script
```

A.4 Dynamic Shared Objects Code

A.4.1 libdpt.so

bsrecov/dso/linux/dpt/readdpt.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <errno.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>

#include <linux/hdreg.h>

#include "../../../bsapi.h"
#include "../../../bserror.h"
#include "readdpt.h"
#include "partinfo.h"
#include "dev.h"

struct local_config {
    int process_inv_mbr;
};
```

```
static struct local_config locConf;

static char current_disk[12];

static char *get_partition_type(int id) {

        int ok;
        int i;
        char empty[] = "Unknown type";
        char *value;

        value = empty;

        ok = 1;
        i = 0;
        while ((ok) && (part_type[i].descr)) {

                if (part_type[i].id == id) {
                        ok = 0;
                        value=part_type[i].descr;
                }
                i++;
        }

        return value;
}

static void cylsectconv(struct DPT *dpt) {

  unsigned short int temp;

  temp = dpt->Scylsect & 0x0c0;  /* get 9-8 bits of cylinders */
  temp = temp << 2; /* put them at the right position */
  temp = temp | ((dpt->Scylsect & 0x0ff00) >> 8);
  /* temp now contains the cylinder number */

  dpt->Scyl = temp;
  dpt->Ssect = dpt->Scylsect & 0x3f;

  temp = dpt->Ecylsect & 0x0c0;  /* get 9-8 bits of cylinders */
  temp = temp << 2; /* put them at the right position */
  temp = temp | ((dpt->Ecylsect & 0x0ff00) >> 8);
```

```
  /* temp now contains the cylinder number */

  dpt->Ecyl = temp;
  dpt->Esect = dpt->Scylsect & 0x3f;
}


static void printhdgeometry(char *dev, struct hd_geometry geo) {
      long size;
      printf("%s geometry: \n",dev);
      printf("Heads: %d Sectors: %d Cylinders: %d\n",geo.heads,
      geo.sectors, geo.cylinders);
      size = ((geo.heads) * (geo.sectors) * (geo.cylinders)) * 512;
      size = size / (1024*1024);
      printf("Disk size : %ldMB\n\n",size);
}



static int gethdgeometry(char *filename, struct hd_geometry *geo) {

/* This function assume that we found a CD-ROM if the ioctl call fails.
   This is not what we want and we should include a function that tests
   for the presence of a CD-ROM device
*/


  int descr;
  int retval;

  if (!strcmp(filename,"/dev/fd0")) {
      geo->heads=2;
      geo->cylinders=80;
      geo->sectors=16;
      retval = 0;
  } else {
  retval = 0;
  descr = open(filename, O_RDONLY);
  if (descr < 0) retval = -1;
  else {
      if (ioctl(descr, HDIO_GETGEO, geo)) retval = -1;
      close(descr);
  }

  }
```

```
  return retval;
}

static void printdpt(struct DPT dpt, int num) {

        printf("Partition %d (%s):\n",num, get_partition_type(dpt.id));
        if (dpt.id) {
        printf("Indicator : 0x%02X SH: %d SS: %d ST: %d\n", dpt.Active,
            dpt.Shead, dpt.Ssect, dpt.Scyl);
        printf("id: 0x%d EH: %d ES: %d ET: %d\n",dpt.id, dpt.Ehead, dpt.Esect,
            dpt.Ecyl);
        printf("SP: %d PL: %d\n\n", dpt.psect, dpt.nsect);
        }
}


static void fillinmbr(char *buffer, struct MBR *mbr) {

  int i;

  memcpy((void *) &(mbr->bootcode), (void *) buffer, 446);
  memcpy((void *) &(mbr->sign), (void *) &(buffer[510]), 2);

  i=0;
  while (i< 4) {
    memcpy((void *) &(mbr->dpt[i]), (void *) &(buffer[((DPTADDRESS)+
        (PARTSIZE*i))]), 16);
    cylsectconv(&(mbr->dpt[i]));
    i++;
  }
}

static void process_mbr(FILE *device, int pos, long level) {

  struct MBR_OBJECT mbr_o;
  struct MBR mbr;
  int i;
  long child;


  if (fseek(device, pos, SEEK_SET)) {
      bserror(ERR_READDEV,"dpt:process_mbr");
  }

  if (fread(mbr_o.buffer, SECTOR,1,device) != 1)
```

```
      bserror(ERR_READDEV,"dpt:process_mbr");
  fillinmbr(mbr_o.buffer, &mbr);
  mbr_o.mbr_pos = pos;
  if (mbr.sign == MBRSIGN) {
#ifdef _DEBUG
    fprintf(stderr,"Valid MBR found\n");
#endif

    child = api_save_object(&mbr_o, sizeof(mbr_o), TO_MBR, level);

    i = 0;
    while ((i<4) && (mbr.sign == MBRSIGN)) {
#ifdef _DEBUG
      printdpt(mbr.dpt[i], i+1);
#endif
      if (mbr.dpt[i].id == 0x5) {
                process_mbr(device, ((mbr.dpt[i].psect*SECTOR)+pos), child);
      }
      i++;
    }
  }
#ifdef _DEBUG
  else {
    fprintf(stderr,"Invalid boot sector found, signature: %d\n",level);
  }
#endif
}

static int dpt_load(long par) {

  FILE *device;
  int dev_p;
  int rc;
  struct DISK_OBJECT disk;
  long parent;

  printf("Processing...");
  dev_p = 0;
  while (dev_list[dev_p].file) {

      rc = gethdgeometry(dev_list[dev_p].file, &(disk.geometry));
      if (!rc) {
#ifdef _DEBUG
          printhdgeometry(dev_list[dev_p].file,disk.geometry);
#endif
```

```
            strcpy(disk.name, dev_list[dev_p].file);

            parent = api_save_object(&disk, sizeof(disk),TO_DISK, par);

            device = fopen(dev_list[dev_p].file,"r");
            if (!device) bserror(ERR_OPENDEV);

            process_mbr(device, 0, parent);
            fclose(device);
        }
        dev_p++;
    }
    printf("\nDone.\n");
}

static int save_mbr_option(char *str) {

    int retval;

    retval = -1;

    locConf.process_inv_mbr = 0;
    if (!strcmp(str,"")) {
        fprintf(stderr, "warning: no parameter supplied to save_inv_mbr option\n");
    } else if ((strcasecmp(str,"yes")) && (strcasecmp(str,"no"))) {
        fprintf(stderr, "warning: save_inc_mbr: invalid option %s",str);
    } else {
        if (!strcasecmp(str,"yes")) locConf.process_inv_mbr = 1;
        else locConf.process_inv_mbr = 0;
        retval = 0;
    }
    return retval;
}

static int dpt_info() {
    printf("Disk Partition Table recovery module version 0.1\n");
}

static int dpt_recovery(long par) {

/*    printf("module dpt: recovery not yet implemented\n");
    return 0;
*/
    struct DISK_OBJECT *pdisk;
    struct MBR_OBJECT *pmbr;
```

```
    void *data;
    long long where;
    int type;
    int size;
    long child;
    FILE *disk;
#ifdef _DEBUG
    char buffer[512];
#endif

    child = findfirst_hierarchy(par,&where);
    while (child) {
        data = api_load_object(child,&type, &size);
        switch (type) {
            case TO_DISK: pdisk = (struct DISK_OBJECT *) data;
                          strcpy(current_disk, pdisk->name);
                          dpt_recovery(child);
                          break;
            case TO_MBR: pmbr = (struct MBR_OBJECT *) data;
                         disk = fopen(current_disk, "r+");
                         if (!disk) bserror(ERR_OPENDEV,"dpt:recover");
                         if (fseek(disk, pmbr->mbr_pos, SEEK_SET))
                             bserror(ERR_READDEV,"dpt:recover");
                         printf("Writting MBR to disk %s\n",current_disk);

#ifdef _DEBUG
                         if (fread((void *) buffer, 512, 1, disk) != 1)
                             bserror(ERR_WRITEDEV,"dpt:recover");
                         if (fseek(disk, pmbr->mbr_pos, SEEK_SET))
                             bserror(ERR_READDEV,"dpt:recover");
                         if (memcmp((void *) buffer,
                                 (void *) pmbr->buffer, 512)) {
                             printf("WARNING: MBR not equal\n");
                         } else {
                             printf("Ok, Saved MBR match real MBR\n");
                         }
                         if (fwrite((void *) pmbr->buffer, 512, 1, disk) != 1) {
                             perror("Write devive");
                             bserror(ERR_WRITEDEV,"dpt:recover");
                         }


#else
/*                       if (fwrite((void *) pmbr->buffer, 512, 1, disk) != 1)
                             bserror(ERR_WRITEDEV,"dpt:recover");
```

```
*/
#endif
                        fclose(disk);
                        dpt_recovery(child);
                        break;
            default: bserror(ERR_INCON,"dpt:recover");
        }
        free(data);
        child = findnext_hierarchy(par,&where);
    }
    return 0;
}

static int dpt_recover(long par) {
    int rv;

    printf("Recover: Processing...");
    fflush(stdout);
    rv = dpt_recovery(par);
    printf("Syncing disks.\n");
    sync();
    printf("\nDone.\n");
    return rv;
}


int init_module() {

    api_register_module("dpt", dpt_recover, dpt_load, 0x0, dpt_info);
    api_register_option("save_inv_mbr", "dpt", save_mbr_option);
    return 0;
}
```

bsrecov/dso/linux/dpt/readdpt.h:

```
#define DPTADDRESS 0x1be
#define PARTSIZE   16
#define MBRSIGN    0x0AA55
#define SECTOR 512

#define TO_DISK 1
#define TO_MBR 2


struct DISK_OBJECT {
  char name[12];
```

```
  struct hd_geometry geometry;
  int level;
};

struct MBR_OBJECT {

  char buffer[512];
  int mbr_pos;
  int level;
};


struct DPT {

  unsigned char Active, Shead;
  unsigned short int Scylsect;
  unsigned char  id, Ehead;
  unsigned short int Ecylsect;
  unsigned int  psect, nsect;

  unsigned short int Scyl,Ssect,Ecyl,Esect;

};

struct MBR {

  unsigned char bootcode[446];
  struct DPT dpt[4];
  unsigned short int sign;
};
```

bsrecov/dso/linux/dpt/partinfo.h:

```
struct PART_TYPE {
        int id;
        char *descr;
};


struct PART_TYPE part_type[] = {

        { 0x00 , "Empty" },
        { 0x05 , "DOS 3.3+ Extended Partition" },
        { 0x06 , "DOS 3.3+ 32MB+" },
        { 0x07 , "HPFS, NTFS or Advanced UNIX" },
```

```
        { 0x0b , "Win95 32-bit FAT" },
        { 0x0c , "Win95 32-bit FAT (LBA)" },
        { 0x0e , "LBA VFAT" },
        { 0x0f , "LBA VFAT Extended Partition" },
        { 0x82 , "Linux Swap Partition" },
        { 0x83 , "Linux native file system" },
        { 0x00 , NULL }
};
```

bsrecov/dso/linux/dpt/dev.h:

```
struct device_list {
        char *file;
};

struct device_list dev_list[] = {

#ifdef _DEBUG
        { "/dev/fd0"},
#else
/* IDE Hard Drive */

        { "/dev/hda" },
        { "/dev/hdb" },
        { "/dev/hdc" },
        { "/dev/hdd" },
        { "/dev/hde" },
        { "/dev/hdf" },
        { "/dev/hdg" },
        { "/dev/hdh" },

/* SCSI Hard Drive */

        { "/dev/sda" },
        { "/dev/sdb" },
        { "/dev/sdc" },
        { "/dev/sdd" },
        { "/dev/sde" },
        { "/dev/sdf" },
        { "/dev/sdg" },
        { "/dev/sdh" },
#endif
        { NULL }
```

```
};
```

A.4.2 libdev.so

bsrecov/dso/linux/dev/dev.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <dirent.h>
#include <errno.h>
#include "../../../bsapi.h"
#include "dev_mod.h"


static long parent;

static process_file(char *file, struct stat fStat) {

    DEVS_OBJECT devsObj;
    DEVN_OBJECT devnObj;
    long par;
    int process;

    process = 0;
    switch (fStat.st_mode & S_IFMT) {
        case S_IFBLK:
#ifdef _DEBUG
            fprintf(stderr,"DEBUG: Found a block device\n");
#endif
            process = 1;
            break;
        case S_IFCHR:
#ifdef _DEBUG
            fprintf(stderr,"DEBUG: Found a char device\n");
#endif
            process = 1;
            break;
        case S_IFIFO:
#ifdef _DEBUG
            fprintf(stderr,"DEBUG: Found a named pipe\n");
#endif
            process = 1;
    }
```

```
    if (process) {
        devnObj = file;
        devsObj.fStat = fStat;
        par = api_save_object((void *) devnObj, strlen(devnObj)+1, TO_NDEV, parent);
        api_save_object((void *) &devsObj, sizeof(DEVS_OBJECT), TO_SDEV, par);
    }

}

static process_dir(char *dir) {

    DIR *devDir;
    struct dirent *dirEnt;
    struct stat fStat;
    char compFile[1024];

    devDir = opendir(dir);

    if (!devDir) {
#ifdef _DEBUG
        fprintf(stderr,"DEBUG: Cannot open %s directory\n",dir);
#endif
        return -1;
    }

    dirEnt = readdir(devDir);
    while (dirEnt) {
        if (strcmp(dirEnt->d_name,".") && strcmp(dirEnt->d_name,"..")) {
            strcpy(compFile,dir);
            strcat(compFile,"/");
            strcat(compFile,dirEnt->d_name);
#ifdef _DEBUG
            fprintf(stderr,"DEBUG: we will process %s-\n",compFile);
#endif
            if (lstat(compFile,&fStat)) {
                perror("lstat");
            }

            if (S_ISDIR(fStat.st_mode)) {
#ifdef _DEBUG
                fprintf(stderr,"DEBUG: processing dir %s-\n",compFile);
#endif
                process_dir(compFile);
            } else {
#ifdef _DEBUG
```

```
                fprintf(stderr,"DEBUG: processing file %s-\n",compFile);
#endif

                process_file(compFile, fStat);
            }
        }
        dirEnt = readdir(devDir);
    }
    closedir(devDir);
}

static int load_dev(long par) {

    parent = par;
    printf("Processing...");
    fflush(stdout);
    process_dir("/dev");
    printf("\nDone.\n");
}

static get_dir(char *full, char *dir) {

    int pos;
    pos = strlen(full)-1;
    while (full[pos] != '/') {
        pos--;
    }
    strncpy(dir,full,pos);
    dir[pos] = '\0';
}

static int rmkdir(char *dire) {

    char temp_dir[1024];
    int pos;
    int code;

    if (strcmp(dire,"/")) {
        code = mkdir(dire,384);
        if ((code) && (errno != EEXIST)) {
            pos = strlen(dire)-1;
            while ((dire[pos] != '/') && (pos > 0)) {
                pos--;
            }
            if (pos > 0) {
```

```
                strncpy(temp_dir, dire, pos);
                temp_dir[pos] = '\0';
                if (rmkdir(temp_dir)) code = -1;
                else code = mkdir(dire, 384);
            }
        }
    } else code=-1;
    return code;
}


static int recover_dev(long par) {

/*
    Add error messages when module is wrongly called i.e. Object type doesn't
    match.
*/
    DEVN_OBJECT *pdevn;
    DEVS_OBJECT *pdevs;
    int size;
    int type;
    long long where;
    long long nowhere;
    long nchild;
    long schild;
    char dire[1024];
    char cdire[1024];

    printf("Recover: Processing...");
    fflush(stdout);

    nchild = findfirst_hierarchy(par, &where);
    while (nchild) {
        pdevn = (DEVN_OBJECT *) api_load_object(nchild, &type, &size);
        schild = findfirst_hierarchy(nchild, &nowhere);
        pdevs = (DEVS_OBJECT *) api_load_object(schild, &type, &size);
        strcpy(dire,"/recov");
        strcat(dire,(char *) pdevn);
        switch (pdevs->fStat.st_mode & S_IFMT) {
            case S_IFBLK:
            case S_IFCHR:
            case S_IFIFO:
                        get_dir(dire, cdire);
                        rmkdir(cdire);
```

```
                        mknod(dire, pdevs->fStat.st_mode, pdevs->fStat.st_rdev);
                        break;
            default: fprintf(stderr,"Warning: Unknown file type\n");
        }
        free(pdevn);
        free(pdevs);
        nchild=findnext_hierarchy(par, &where);
    }
    printf("\nDone.\n");
}


static int info_dev() {
    printf("dev module version 0.1\n");
}

int init_module() {
    api_register_module("dev", recover_dev, load_dev, 0x0, info_dev);
    return 0;
}
```

bsrecov/dso/linux/dev/dev_mod.h:

```
typedef struct {
    struct stat fStat;
} DEVS_OBJECT;

typedef char *DEVN_OBJECT;

#define TO_SDEV 1
#define TO_NDEV 2
```

A.4.3 libfs.so

bsrecov/dso/linux/fs/fs.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include "../../../bsapi.h"
#include "../../../bserror.h"
#include "fs.h"

/*
    This module should use the standard system call to get the fstab entry
*/
```

```
static char fstab_path[255];
static char mount_path[128];

static char *wskip(char *str) {

    while (*str == ' ') str++;
    return str;
}

static char *blank_copy(char *source, char *dest) {

    source = wskip(source);
    while ((*source != ' ') && (source != '\0')) {
        *dest = *source;
        dest++;
        source++;
    }
    *dest = '\0';
    return source;
}

static int path_option(char *str) {

    int retval;

    if (!strcmp(str,"")) {
        fprintf(stderr,"warning: fstab_path: value expected\n");
        retval = -1;
    } else {
        strcpy(fstab_path,str);
        retval = 0;
#ifdef _DEBUG
        fprintf(stderr,"DEBUG: fstab path set to : %s\n",str);
#endif
    }
    return retval;
}

static int mount_cmd(char *str) {

    int retval;

    if (!strcmp(str,"")) {
        fprintf(stderr,"warning: fstab_path: value expected\n");
```

```
        retval = -1;
    } else {
        strcpy(mount_path,str);
        retval = 0;
#ifdef _DEBUG
        fprintf(stderr,"DEBUG: mount command set to : %s\n",str);
#endif
    }
    return retval;
}

static int load_fun(long parent) {

    FILE *f;
    char fsLine[255];
    char *temp;
    struct FSTAB_OBJECT fstab;
    int retval;
    long child;


    printf("Processing...");
    f = fopen(fstab_path,"r");

    if (!f) {
        fprintf(stderr,"\nerror: Cannot open fs file %s\n",fstab_path);
        retval = -1;
        goto err;
    }

    fgets(fsLine, 254, f);
    while ((strcmp(fsLine,"")) || (!feof(f))) {
        if (fsLine[strlen(fsLine)-1] == '\n') fsLine[strlen(fsLine)-1] = '\0';
        if (strcmp(fsLine,"")) {
            temp = blank_copy(fsLine,fstab.dev);
            temp = blank_copy(temp,fstab.mount_point);
            temp = blank_copy(temp,fstab.fs_type);
#ifdef _DEBUG
            fprintf(stderr,"Device: %s Mount point: %s FS Type : %s\n",
                fstab.dev, fstab.mount_point, fstab.fs_type);
#endif
            child = api_save_object((void *) &fstab, sizeof(fstab), TO_FSTAB, parent);
            if (child < 0) goto err;
        }
        strcpy(fsLine,"");
```

```
void main_api_init() ;
void main_api_destroy() ;
int api_register_module(char *name, int (*save_func)(), int (*load_func)(), int (*print_f
    int (*info_func)()) ;
int api_register_option(char *option, char *mod_name, int (*option_func)()) ;
void *api_load_object(short int ref, int *type, int *d_size) ;
int api_save_object(void *data, int d_size, int type, int parent) ;
```

A.2.4 modules.h

```
/* The module function table type declaration */

typedef struct {
    char name[18];
    int (*load_func)(int parent);
    int (*save_func)(int parent);
    int (*print_func)(int parent);
    int (*info_func)();
} Module_Func;

/* the module option tabe type declaration */

typedef struct {
    char keyword[18];
    char mod_name[18];
    int (*option_func)(char *str);
} Module_Option;
```

A.2.5 config.h

```
/*
    Each parameter that would be usefull for further
    developpement should be out here for the associated
    function.

    Module specific configuration variable should be stored
    by their own mean.
*/

typedef struct {
    char Module_Path[255];
    int Absent_Module;
} GlobalConfig;

void init_global_config() ;
```

```
char *wskip(char *str) ;
int iscomment(char *str) ;
void string_terminate(char *str) ;
int conf_inc_module(char *str) ;
int conf_module_path(char *str) ;
int conf_abs_mod(char *str) ;
void register_standard_option() ;
int process_line(char *str) ;
void getcurSection(char *str) ;
int init_config() ;
```

A.2.6 bserror.h

```
#define ERR_READDEV -1
#define ERR_WRITEDEV -2
#define ERR_READDB -3
#define ERR_WRITEDB -4
#define ERR_OBJNF -5
#define ERR_IOCTL -6
#define ERR_MALLOC -7
#define ERR_OPENBD -8
#define ERR_OPENDEV -9
#define ERR_INCON -10
#define ERR_NOCONF -11

void error(int code, char *where) ;
```

A.2.7 version.h

```
#define VMAJOR "0"
#define VMINOR "1"
#define VTYPE  "Debug"
```

A.2.8 bsapi.h

```
/*
    Module Specific API
*/

int api_register_module(char *name, int (*save_func)(), int (*load_func)(),
    int (*print_func)(), int (*info_func)());
int api_register_option(char *option, char *mod_name, int (*option_func)());

/*
    Object specific API
```

```
        fgets(fsLine, 254, f);
    }
    child = 0;
err:
    if (f) fclose(f);
    printf("\nDone.");
    if (retval) printf(" (with error(s))\n");
    else printf("\n");
    retval=child;
    return retval;
}

static int info_func() {
    printf("fs module version 0.1\n");
}
static int rmkdir(char *dire) {

    char temp_dir[1024];
    int pos;
    int code;

    if (strcmp(dire,"/")) {
        code = mkdir(dire,384);
        if ((code) && (errno != EEXIST)) {
            pos = strlen(dire)-1;
            while ((dire[pos] != '/') && (pos > 0)) {
                pos--;
            }
            if (pos > 0) {
                strncpy(temp_dir, dire, pos);
                temp_dir[pos] = '\0';
                if (rmkdir(temp_dir)) code = -1;
                else code = mkdir(dire, 384);
            }
        }
    } else code=-1;
    return code;
}

static parseCmd(char *prog, struct FSTAB_OBJECT fstab, char *cmd) {

    while (*prog != '\0') {
        if (*prog == '%') {
            prog++;
            switch (*prog) {
```

```
                case '%' : *cmd = '%';
                           cmd++;
                           break;
                case '1' : strcpy(cmd,fstab.dev);
                           while (*cmd != '\0') cmd++;
                           break;
                case '2' : strcpy(cmd,fstab.fs_type);
                           while (*cmd != '\0') cmd++;
                           break;
                case '3' : strcpy(cmd,fstab.mount_point);
                           while (*cmd != '\0') cmd++;
                           break;
            }
            prog++;
        } else {
            *cmd=*prog;
            cmd++;
            prog++;
        }
    }
}

static int fs_reformat(struct FSTAB_OBJECT fstab) {

    char fmtCmd[128];
    int found;
    int i;
    int rc;

    found=0;
    i = 0;
    strcpy(fmtCmd,"");
    /* We don't stop searching when we find a valid entry, the user
       can then add more commands for a single fs type. Multiple processing
       can then be done.
    */
    while (stdFProg[i].fs_type) {
        if (!strcmp(stdFProg[i].fs_type, fstab.fs_type)) {
            found=1;
            parseCmd(stdFProg[i].prog, fstab, fmtCmd);
#ifdef _DEBUG
            fprintf(stderr,"DEBUG: calling the command: %s\n",fmtCmd);
#endif
            rc=system(fmtCmd);
#ifdef _DEBUG
```

```
            fprintf(stderr,"DEBUG: Called command return code: %d\n", rc);
#endif
        }
        strcpy(fmtCmd,"");
        i++;
    }
    if (!found) {
        fprintf(stderr,"ERROR: a handling command for fs type %s was not found\n",
            fstab.fs_type);
        rc = -1;
    }
    return rc;
}

static int fs_remount(struct FSTAB_OBJECT fstab, char *redir) {
    char cmd[128];
    char rmount[256];
    int rc;

    /* assure that the mount point exist */
    sprintf(rmount,"%s%s",redir,fstab.mount_point);
    rmkdir(rmount);

    /*and finally to the real job */
    sprintf(cmd,"%s %s -t %s %s",mount_path, fstab.dev, fstab.fs_type, rmount);
    rc=system(cmd);
    return rc;
}


static int recover_fs(long parent) {
/*    printf("fs module: recovery not yet implemented\n");
    return 0;
*/
    struct FSTAB_OBJECT *fstab;
    long long where;
    long child;
    int type;
    int size;

    printf("Recover: Processing...\n");
    child = findfirst_hierarchy(parent, &where);
    while (child) {

        fstab = (struct FSTAB_OBJECT *) api_load_object(child, &type, &size);
```

```
        switch (type) {
            case TO_FSTAB:
                    fs_reformat(*fstab);
                    fs_remount(*fstab,"/recov");
                    break;
            default: bserror(ERR_INCON,"fs:recover");
        }
        free(fstab);
        child = findnext_hierarchy(parent, &where);
    }
    printf("Done.\n");
}

int init_module() {
    strcpy(fstab_path,"/etc/fstab");
    strcpy(mount_path,"/bin/mount");
    api_register_module("fs", recover_fs, load_fun, 0x0, info_func);
    api_register_option("fstab_path","fs", path_option);
    api_register_option("mount_cmd","fs", mount_cmd);
    return 0;
}
```

bsrecov/dso/linux/fs/fs.h:

```
#define TO_FSTAB 1

struct FSTAB_OBJECT {
    char dev[18];
    char mount_point[80];
    char fs_type[8];
};

/*
    The prog field contains the name & option of a executable that
    will format a device for a specified type.

    The string can include 2 pattern:
        %1 is replaced by the considered device
        %2 is replaced by the type of the filesystem
*/

struct format_prog {
    char *fs_type;
    char *prog;
};
```

```
/*
    The fs module should allow the user to add any prog he want
    and to overide these
*/

struct format_prog stdFProg[] = {
    { "ext2" , "mke2fs %1" },
    { "swap" , "mkswap -c %1" },
    { "minix" , "mkfs.minix %1" },
    { NULL , NULL }
};
```

A.5 Linux disk set

The exact description of the disk set building process would take to much time and space to put it here. I will only provide the source of the bsrecov init process. I.e. the process executed by the linux kernel just after it finished its initialization phase.

Note that the exact linuxrc executable should be linked statically and that it's a multi-call executable linked with network configuration tools coming from the NET-TOOLS package version 1.53. This package is written by Phil Blundell and Bernd Eckenfels and is distributed under the GPL (GNU General Public License).

The disks set and the BsRecov Light Distribution include the BusyBox package from Erik Andersen and is distributed under the GPL (GNU General Public License).

The disks set boot disk was made using SYSLINUX v1.48, written by H. Peter Anvin. SYSLINUX v1.48 is distributed under the GPL (GNU General Public License).

The Linux kernel is copyrighted by Linus Torvald (and many of other contributors) and is also distributed under the GPL (GNU General Public License).

A.5.1 Bsrecov /Linux/i386 init source

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mount.h>
#include <stdarg.h>
#include <sys/wait.h>
#include <string.h>
#include <ncurses.h>
```

```
        j++;
    }
    move(0,0);
    refresh();
}

void background(char *title, char *status) {
    int x;
    char tile[80];

    sprintf(tile,"BsRecov Version 0.1 - %s", title);
    x = 40 - (strlen(tile) / 2);
    move(0, x);
    printw("%s",tile);
    move(24, 0);
    printw("%s",status);
}


int yn_box(char *str, int YesNo) {

    int text_len;
    int x , y;
    int i;

    curs_set(0);
    text_len = strlen(str);
    x = (COLS/2) - (text_len /2);
    y = (LINES/2) - 1;
    move(y,x);
    printw(str);
    y = (LINES/2) + 1;
    i=0;
    while (i != 13) {
        switch (i) {
            case ' ' : if (YesNo) YesNo= 0; else YesNo=1;
        };
        x = (COLS/2) - 5;
        y = (LINES/2) + 1;
        if (YesNo) {
            move(y,x);
            attrset(COLOR_PAIR(2));
            printw(" Yes ");
            attrset(COLOR_PAIR(1));
            x=x+5;
```

```
/*
    The fs module should allow the user to add any prog he want
    and to overide these
*/

struct format_prog stdFProg[] = {
    { "ext2" , "mke2fs %1" },
    { "swap" , "mkswap -c %1" },
    { "minix" , "mkfs.minix %1" },
    { NULL , NULL }
};
```

A.5 Linux disk set

The exact description of the disk set building process would take to much time and space to put it here. I will only provide the source of the bsrecov init process. I.e. the process executed by the linux kernel just after it finished its initialization phase.

Note that the exact linuxrc executable should be linked statically and that it's a multi-call executable linked with network configuration tools coming from the NET-TOOLS package version 1.53. This package is written by Phil Blundell and Bernd Eckenfels and is distributed under the GPL (GNU General Public License).

The disks set and the BsRecov Light Distribution include the BusyBox package from Erik Andersen and is distributed under the GPL (GNU General Public License).

The disks set boot disk was made using SYSLINUX v1.48, written by H. Peter Anvin. SYSLINUX v1.48 is distributed under the GPL (GNU General Public License).

The Linux kernel is copyrighted by Linus Torvald (and many of other contributors) and is also distributed under the GPL (GNU General Public License).

A.5.1 Bsrecov /Linux/i386 init source

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mount.h>
#include <stdarg.h>
#include <sys/wait.h>
#include <string.h>
#include <ncurses.h>
```

```
#include <linux/unistd.h>
#include "form.h"
#include <signal.h>
#include "module.h"

#ifdef TEST
#include <linux/fs.h>
#endif

#define KERN_TTY "/dev/tty5"
#define EXEC_TTY "/dev/tty4"

/* NFS Global Varaiables */

char nfs_cip[20];
char nfs_sip[20];
char nfs_gip[20];
char nfs_net[20];
char nfs_host[255];
char nfs_dir[255];
char nfs_dev[255];

/* Network Global Var */

char interface[8];
char ip[20];
char netmask[20];
char network[20];
char default_gwip[80];


/* Module Vars */

struct m_info **scsi_mod;
struct m_info **net_mod;
int scsi_mod_len;
int net_mod_len;

void delay(int sec) {

    struct timeval tv;

    tv.tv_sec = sec;
    tv.tv_usec = 0;
```

```
    select(0, NULL, NULL, NULL, &tv);
}

void sigbreak(int num) {
    signal(SIGINT, sigbreak);
}

int execute(char *path,...) {

    int child;
    int status;
    va_list p;
    char **exe;
    int i;

    if (! (child = fork())) {
        close(0);
        close(1);
        close(2);

        open(EXEC_TTY,O_RDONLY);
        open(EXEC_TTY,O_WRONLY);
        open(EXEC_TTY,O_WRONLY);
        exe = (char **) malloc(10*(sizeof(char *)));
        i = 0;
        va_start(p, path);
        while (exe[i] = va_arg(p, char *)) {
            i++;
        }
        va_end(p);
        execv(path, exe);
    } else {
        waitpid(child, &status, 0);
    }
    return status;
}

void clear_scr() {

    int i,j;
    i = COLS*LINES;
    j = 0;
    move(0,0);
    while (j < i) {
        printw(" ");
```

```
        j++;
    }
    move(0,0);
    refresh();
}

void background(char *title, char *status) {
    int x;
    char tile[80];

    sprintf(tile,"BsRecov Version 0.1 - %s", title);
    x = 40 - (strlen(tile) / 2);
    move(0, x);
    printw("%s",tile);
    move(24, 0);
    printw("%s",status);
}


int yn_box(char *str, int YesNo) {

    int text_len;
    int x , y;
    int i;

    curs_set(0);
    text_len = strlen(str);
    x = (COLS/2) - (text_len /2);
    y = (LINES/2) - 1;
    move(y,x);
    printw(str);
    y = (LINES/2) + 1;
    i=0;
    while (i != 13) {
        switch (i) {
            case ' ' : if (YesNo) YesNo= 0; else YesNo=1;
        };
        x = (COLS/2) - 5;
        y = (LINES/2) + 1;
        if (YesNo) {
            move(y,x);
            attrset(COLOR_PAIR(2));
            printw(" Yes ");
            attrset(COLOR_PAIR(1));
            x=x+5;
```

```
            move(y,x);
            printw(" No ");
        } else {
            move(y,x);
            attrset(COLOR_PAIR(1));
            printw(" Yes ");
            attrset(COLOR_PAIR(2));
            x=x+5;
            move(y,x);
            printw(" No ");
            attrset(COLOR_PAIR(1));
        }
        refresh();
        i = getch();
    }
    curs_set(1);
    return YesNo;
}

#define LIST_LEN 10
#define LIST_X 7
#define LIST_Y 7

int process_list(struct m_info **m, char *tile, int *sel, int top) {

    int pos;
    int cur_item;
    int start_item;
    int x=LIST_X;
    int y=LIST_Y;
    int i;
    int key;

    pos=0;
    cur_item=0;
    start_item=0;
    clear_scr();
    background(tile, "[UP] up [DOWN] down [ENTER] select [ESC] abort");
    i = 0;
    while (i < LIST_LEN) {
        move(y+i, x);
        printw("
        move(y+i, x);
        printw("%s", m[start_item+i]->name);
        i++;
```

```
}
move(y+pos,x-1); printw("*");
refresh();
key = getch();
while (key != 27) {
    switch (key) {
        case KEY_DOWN :
            if (cur_item < top) {
                cur_item++;
                move(y+pos,x-1); printw(" ");
                if (pos == LIST_LEN-1) {
                    start_item=start_item+1;
                    i = 0;
                    while (i < LIST_LEN) {
                        move(y+i, x);
                        printw("
                        move(y+i, x);
                        printw("%s", m[start_item+i]->name);
                        i++;
                    }
                } else pos=pos+1;
                move(y+pos,x-1); printw("*");
                refresh();
            }
            break;
        case KEY_UP :
            if (cur_item > 0) {
                cur_item--;
                move(y+pos,x-1); printw(" ");
                if (pos == 0) {
                    start_item=start_item-1;
                    i = 0;
                    while (i < LIST_LEN) {
                        move(y+i, x);
                        printw("
                        move(y+i, x);
                        printw("%s", m[start_item+i]->name);
                        i++;
                    }
                } else pos=pos-1;
                move(y+pos,x-1); printw("*");
                refresh();
            }
            break;
        case 13 : *sel = cur_item;
```

```
                return 1;
            case 27 : return 0;
        }
        key = getch();
    }
    return 0;
}


char *net_selection() {
    int key;
    int sel;
    char full_path[80];

    curs_set(0);
    clear_scr();
    background("Network modules", "[UP] up [DOWN] down [ENTER] select [ESC] abort");
    while ((key = process_list(net_mod, "Network modules", &sel, net_mod_len-1))) {
       sprintf(full_path,"/mnt/net/%s", net_mod[sel]->cmd);
       move(20,35);
       printw("Loading...");
       refresh();
       execute("/bin/insmod","indmod", full_path, NULL);
       move(20,35);
       printw("          ");
       refresh();
    }
    curs_set(1);

}

char *scsi_selection() {
    int key;
    int sel;
    char full_path[80];

    curs_set(0);
    clear_scr();
    background("SCSI modules", "[UP] up [DOWN] down [ENTER] select [ESC] abort");
    while ((key = process_list(scsi_mod, "SCSI modules", &sel, scsi_mod_len-1))) {
       sprintf(full_path,"/mnt/scsi/%s", scsi_mod[sel]->cmd);
       move(20,35);
       printw("Loading...");
       refresh();
       execute("/bin/insmod","indmod", full_path, NULL);
```

```
        move(20,35);
        printw("          ");
        refresh();
    }
    curs_set(1);
}

void unret(char *str) {
    if (str[strlen(str)-1] == '\n') str[strlen(str)-1] = '\0';
}

void my_printf(FILE *f, char *fmt, ...) {
    va_list p;
    va_start(p, fmt);
    vfprintf(f, fmt, p);
    va_end(p);
    fflush(f);
}

void fgetsn(char *str, int size, FILE *f) {
    fgets(str, size, f);
    unret(str);
}


void mount_proc(void)
{
  /* mkdir("/proc",384); */
  mount(0,"/proc","proc",0,0);
  refresh();
}

void mount_ram_dev(void)
{
  /* mkdir("/mnt",384); */
  mount("/dev/ram","/mnt", "ext2", 0, 0);
  refresh();
}

void mount_disk(int ro) {
    int flags;
    flags=0;

    if (!ro) {
        mount("/dev/fd0","/mnt", "ext2", 0, 0);
```

```
    } else {
        flags |= MS_RDONLY;
        mount("/dev/fd0","/mnt", "ext2", flags | MS_MGC_VAL, 0);
    }
    delay(1);
    refresh();
}

int check_disk(char *code) {

    struct stat buf;
    char path[30];
    int rc;

    mount_disk(1);
    sprintf(path,"/mnt/%s", code);
    if (stat(path, &buf)) rc=0;
    else rc=1;
    umount("/mnt");
    return rc;
}

void print_real_root() {

  FILE *f;
  char str[200];
  if (! (f=fopen("/proc/sys/kernel/real-root-dev","r"))) return;
  fgets(str, 199, f);
  printf("Real root: %s",str);
  fflush(stdout);
  fclose(f);
}


int change_real_root(char *root) {

  FILE *f;
  if (root) {
    if (! (f=fopen("/proc/sys/kernel/real-root-dev","w"))) return -1;
    fprintf(f, "%s\n",root);
    fclose(f);
  }
  return 0;
}
```

```
int setup_nfs_root() {

  FILE *f;

  if (! (f=fopen("/proc/sys/kernel/nfs-root-name","w"))) return -1;
  fprintf(f, "%s:%s\n",nfs_sip, nfs_dir);
  fclose(f);
  if (! (f=fopen("/proc/sys/kernel/nfs-root-addrs","w"))) return -1;
  fprintf(f, "%s:%s:%s:%s::%s:none\n",nfs_cip,nfs_sip,nfs_gip, nfs_net, nfs_dev);
  fclose(f);

  return 0;
}


void print_net_config() {

    my_printf(stdout, "Network configuration summary:\n\n");
    execute("/ifconfig", "ifconfig", NULL);
    execute("/route", "route", NULL);
    my_printf(stdout,"\n");
}


void config_network() {
    FORM *net;

    net = init_form();
    add_item(net, 15, 9 , 5, "Interface       : ", interface);
    add_item(net, 15, 10, 16, "IP Address      : ", ip);
    add_item(net, 15, 11, 16, "Netmask         : ", netmask);
    add_item(net, 15, 12, 16, "Network Address : ", network);
    add_item(net, 15, 13, 16, "Default Gateway : ", default_gwip);


    clear_scr();
    background("Network Configuration","[UP] Previous field [DOWN] Next field [BACKSPACE]
        delete char [ENTER] finish");
    process_form(net);
    clear_scr();
    background("Network","[SPACE] Change [ENTER] Select");
    while (!yn_box("Do you accept the network configuration?",1)) {
        clear_scr();
        background("Network Configuration","[UP] Previous field [DOWN] Next field
            [BACKSPACE] delete char [ENTER] finish");
```

```
        process_form(net);
        clear_scr();
        background("Network","[SPACE] Change [ENTER] Select");
    }

    execute("/ifconfig","ifconfig","lo","127.0.0.1", NULL);
    execute("/route","route","add","-net","127.0.0.0","netmask","255.0.0.0", NULL);
    strcpy(nfs_dev, interface);
    strcpy(nfs_cip,ip);
    strcpy(nfs_net,netmask);
    execute("/ifconfig","ifconfig",interface,ip,"netmask",netmask,NULL);
    execute("/route","route","add","-net", network,"netmask",netmask,interface,NULL);
    if (*default_gwip != '\0') {
        strcpy(nfs_gip, default_gwip);
        execute("/route","route","add","default","gw", default_gwip,"metric","1",NULL);
    } else strcpy(nfs_gip, "");

    destroy_form(net);
}

void config_modules() {

  mount_disk(1);

  net_selection();
  scsi_selection();
  umount("/mnt");
}


void config_root_dev() {

  char answer[20];
  char entry[20];
  FORM *root;

  root=init_form();
  add_item(root, 11, 12, 16, "NFS Server IP        : ", nfs_sip);
  add_item(root, 11, 13, 40, "NFS Server Directory : ", nfs_dir);
  clear_scr();
  background("NFS Configuration", "[UP] Previous fiels [DOWN] Next field [BACKSPACE]
      delete char [ENTER] finish");
  process_form(root);
  clear_scr();
  background("NFS Configuration","[SPACE] Change [ENTER] Select");
```

```
  while (!yn_box("Do you accept the NFS configuration?",1)) {
      clear_scr();
      background("NFS Configuration", "[UP] Previous fiels [DOWN] Next field [BACKSPACE]
      process_form(root);
      clear_scr();
      background("NFS Configuration","[SPACE] Change [ENTER] Select");
  }
  setup_nfs_root();
  if (change_real_root("255")) {
      printf("Error: Cannot change the root device, halting\n");
      while (1) {};
  }
}

_syscall3(int, syslog, int, type, char *, bufp, int, len);

void syslog_process() {

    char *buff;
    int tty;
    int l;

    syslog(6,0,0);
    if (!fork()) {
        setsid();
        buff = (char *) malloc(4096);
        tty = open(KERN_TTY, O_WRONLY);
        if ((tty < 0) || (!buff)) {
            printf("Warning: syslog process initialization error\n");
            _exit(0);
        }
        while (1) {
            l = syslog(2, buff, 4096);
            write(tty, (void *) buff, l);
        }
    }
}

int linuxrc_main(int argc, char **argv, char **env) {
    int rc;
    FORM *test;


    syslog_process();
    load_mod_conf();
```

```
signal(SIGINT, sigbreak);
initscr();
keypad(stdscr, TRUE);
cbreak();
noecho();
nonl();

if (has_colors()) {
    start_color();
    init_pair(1, COLOR_WHITE, COLOR_BLUE);
    init_pair(2, COLOR_BLUE, COLOR_WHITE),
    attrset(COLOR_PAIR(1));
    clear_scr();
}
mount_proc();
background("Modules", "[SPACE] Change [ENTER] Select");
rc = yn_box("Do you want to load modules ?", 1);
if (rc) {
    clear_scr();
    curs_set(0);
    background("Modules selection", "[ENTER] Load Disk");
    move(12,26);
    printw("Please insert BSRECOV Disk #2");
    refresh();
    getch();
    while (!check_disk("bsrecov-2")) {
        clear_scr();
        background("Modules selection", "[ENTER] Load Disk");
        move(10, 33);
        printw("Wrong disk ...");
        move(12,26);
        printw("Please insert BSRECOV Disk #2");
        refresh();
        getch();
    }
    curs_set(1);
    clear_scr();
    config_modules();
}
config_network();
config_root_dev();
clear_scr();
curs_set(0);
background("Machine Disk","[ENTER] Load Disk");
```

```
    move(12,26);
    printw("Please insert BSRECOV Disk #3");
    refresh();
    getch();
    while (!check_disk("bsrecov-3")) {
        clear_scr();
        background("Machine Disk", "[ENTER] Load Disk");
        move(10,33);
        printw("Wrong disk...");
        move(12,26);
        printw("Please insert BSRECOV Disk #3");
        refresh();
        getch();
    }
    curs_set(1);
    if (has_colors()) attrset(COLOR_PAIR(0));
    clear_scr();
    move(0,0);
    refresh();
    endwin();
    printf("Recovery configuration done, booting recovery console\n");
    return 0;
}
```

Appendix B

BsRecov /Linux/i386 Manual

B.1 Introduction

Bsrecov is a disaster recovery tool developed to work with HSMS. It consists of a command-line utility, various modules, a three disks set and a small subset of the Linux operating system (with HSMS-CL) on CDROM or NFS. The command-line utility furnishes an API to modules which enables them to load and save Disaster Recovery Data Objects, and get configuration options. All the operations are handled by the command-line utility that give control to the different modules when needed.

The modules are shared objects that can be installed or upgraded without needing a new version of the command line utility. They handle specific parts of the requested disaster recovery operations.

The result of a DRD (Disaster Recovery Data) saving is a small database of hierarchically organized objects that can be automatically backed up or restored from an HSMS server.

The three disks set and the subset of the Linux operating system are used to recover a computer from a complete crash.

B.2 Usage

B.2.1 Bsrecov

Command-line Options

```
bsrecov [-backup | -restore] [-save|-recover [-frestore]] [-modinfo]
```

-backup is used to backup the DRD database on a BS2000 running HSMS. This feature depends on HSMS-CL that must be installed and properly configured (running).

-restore is used to restore the DRD database from a BS2000 to the local workstation (The database is placed in the directory /opt/bsrecov/db). HSMS-CL must be installed and properly configured (running)

-save collect all the DRD by calling the configured modules.

-recover restores all the DRD by calling the configured modules. The files are restored under the directory /recov. It means that the correct disks have to be mounted under that directory (this is the standard mode of operation but this behaviour could be overrided). If -frestore is specified, all the files that were previously backed up by HSMS-CL or HSMS are restored automatically under /recov (instead of /) after all the modules have completed their tasks (TODO: A diagnostic in each module to see what really has to be done.)

-modinfo Gives a short description of the installed and configured modules

B.2.2 Configuration File: bsrecovrc

bsrecovrc is organized in sections, begining with [section]. Only the main section is mandatory ([main]). The options are given using the 'option = value' form.

The main section has for now three different options:

module_path (STRING) that gives the path where the modules are currently installed (default: /opt/bsrecov/dso).

inc_module (STRING) is used to include a module. The path given is relative to the path given in the module_path option.

proc_abs_mod (yes/no) is used to tell bsrecov whether it should continue the recovery process when a module that was used for the saving of DRD is absent.

Other sections are module specific. The options in those sections apply only to one module.

B.2.3 Standard Modules

libcore.so

This is in fact the main application code and is not really a module. This module is MANDATORY.

libdpt.so

This module can save and recover the disks partition table, and other things like the installed disks geometry (to permit the restoration of the table on different disk (STILL NOT IMPLEMENTED)).

This module should be installed first in the module stack (i.e. using the first inc_module directive in the configuration file)

libdpt.so has its options in the [dpt] section:

save_inv_mbr (yes/no) specify whether invalid boot sectors (not ending by 0xAA55) have to be processed or not.

libfs.so

This module saves the file-system organization and the file-system type of each installed disk.

It is capable of reformating the disks (or partitions) correctly and to remount the file-system as it was before a crash occured. (supported file-system types are : ext2, linux swap and minix)

This module has to be installed right after the libdpt.so module.

libfs.so has its options in the [fs] section:

fstab_path (STRING) sets the path of the fstab system file (default: /etc/fstab).

mount_cmd (STRING) sets the mount command used to remount the file-systems

libdev.so

This module saves and restore the files that are not handled by HSMS: named pipe, block and char device.

This modules has currently no options

TODO: add the possibility to parse all the file-system instead of just /dev (should be easy).

...

To be written

B.2.4 Recovery Disks set

Not yet implemented, the disk set is still at the prototype stage.

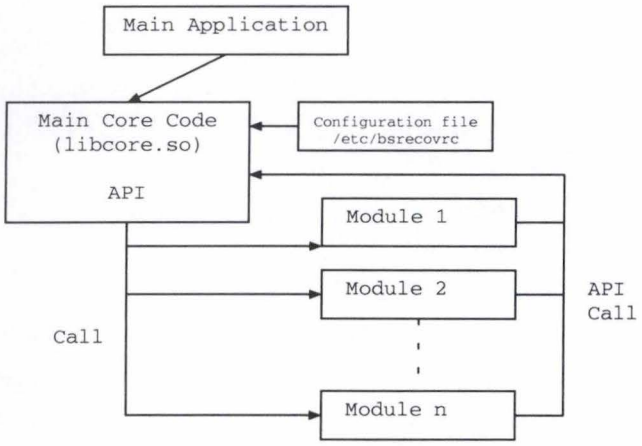

Figure B.1: Bsrecov General Architecture

B.3 Architecture

Description

The general architecture of bsrecov is depicted at figure B.1.

The main application consists in fact of a wrapper that dynamically loads the libcore.so module. That module contains all the core of bsrecov and the API furnished to the modules.

When started, bsrecov reads the configuration file and each time it finds a inc_module directive, loads the concerned module and calls the module exported function *init_module()*. This function registers the module by calling the API function *api_register_module()* and registers the module options by calling *api_register_option()*

If the init_module() function returns 0, the module initialization completed successfully. If it returns a value != 0, the initialization has failed.

The way bsrecov should handle failed module initialization is not currently defined. Just a warning is then issued (but it should be a fatal error).

The registration of a module declares four functions defined in that module:

- The DRD save function
- The DRD restore function
- The DRD info printing function (not implemented yet in bsrecov)
- The module info printing function

Until now, only the info printing function must be registered (although the api_register_module will not complain if it's not declared (BUG?))

When all the configured modules have been registered, the core module processes all the options of the configuration file.

Once the configuration phase is terminated, the function corresponding to the requested operation is called in each module (if it exists).

(For now, calling a non-existant module info printing function will make the application core dump)

Object Database Hierarchy

The DRD database is a hierarchically organized collection of objects. The main application allocates one root object for each declared module. Among those objects, each module is responsible for its object hierarchy.

The structure of each module object hierarchy is hidden to the main application.

To handle the object hierarchy, the core module furnishes the following functions to the modules: findfirst_hierarchy() and findnext_hierarchy().

To respectively load and save object, two functions are available to the modules: api_load_object() and api_save_object().

API Definitions

The functions prototype are in /opt/bsrecov/include/bsapi.h

```
- int api_register_module(char *name, int (*save_func)(int ref),
    int (*load_func)(int ref),
    int (*print_func)(int ref),
    int (*info_func)());

Description:
    Register the module named *name* and declare the save function
    *save_func*, the load function *load_func*, the DRD info printing
    function *print_func* and the module info printing function
    *info_func*.

    The *ref* argument of the three first functions is used to pass
    the root key of the module specific object hierarchy.

- int api_register_option(char *option, char *mod_name,
    int (*option_func)(char *param));

Description:

    Register the option *option* for the module *mod_name* (that corresponds
    to the [*mod_name*] section in the configuration file).

    The *option_func* will be called when the *option* is encountered
```

```
    in *mod_name* section with, as parameter *param*, the option given.

- short int findfirst_hierarchy(short int parent, int *where);

Description:

    Get the key of the first object descending from the root (sub-root) object
    referenced by the *parent* key.

    *where* is a pointer to a int and is a private info that must be passed
    to subsequent call to findnext_hierarchy()

- short int findnext_hierarchy(short int parent, int *where);

Description:

    Get the key of the next object descending from the root (sub-root) object
    referenced by the *parent* key.

    Same notes that in findfirst_key() apply for the *where* parameter.

- int api_save_object(void *data, int d_size, int type, short int parent);

Description:

    Save an object pointed by *data of the private type *type* and of size
    *d_size* descending from the object referenced by the *parent* key
    in the object database.

    This function returns the key assigned to that object in the DRD database.

- void *api_load_object(short int ref, int *type, int *d_size);

Description:

    Load the object identified by the key *ref*. The function return
    a pointer to the object data of private type *type* and of size
    *d_size*.

    The memory allocated for the object must be freed by the module
    programmer.
```

Module Programming Example

Here is a commented template of a bsrecov modules (testmod.c) :

```
#include "/opt/bsrecov/include/bsapi.h"


/*
    All functions and variables definitions are static as global functions
    or variables could interfere with other shared modules.

    This is why the source file must be unique. (Or other source files
    have to be included with #include clauses)
*/

#define MY_OBJECT_TYPE 1;


/* allocate space for our dummy configuration string test */
static char conf[255];

/*
    function called when the option test is encountered in the testmod
    section of /etc/bsrecovrc
*/
static int test_option(char *str) {

    strcpy(conf,str);
    return 0;
}


/*
    function called when bsrecov is invoked with -save
*/
static int load_fun(int parent) {
    char MyObject[20];
    int key;

    printf("My test option is set to : %s\n", conf);
    /* we allocate one object with the test option parameter */
    strcpy(MyObject, conf);
    /* and we put it under this module root key */
```

```
    key = api_save_object((void *) MyObject, 20, MY_OBJECT_TYPE, parent);
    if (key < 0) {
return -1;
    }
    return 0;

}

/*
    function called when bsrecov is invoked with -modinfo
*/
static int info_func() {
    printf("bsrecov test module version 0.1\n");
}

/*
    function called when bsrecov is invoked with -recover
*/
static int test_recover(int parent) {
    char *MyObject;
    short int key;
    int where;
    int type;
    int size;

    printf("recovery operation\n");
    key = findfirst_hierarchy(parent, &where);
    MyObject = (char *) api_load_object(key, &type, &size);
    if ((MyObject) && (type==MY_OBJECT_TYPE)) {
printf("The saved object is : %s\n", MyObject);
printf("MyObject size : %d\n", size);
free(MyObject);
    } else {
printf("Inconsitency found in the database: ");
if (MyObject) {
    free(MyObject);
    printf("Wrong type found\n");
} else {
    printf("No object referenced by key %d\n", key);
}
return -1;
    }
    return 0;
```

```
}

int init_module() {

    /*
register in bsrecov the module called testmod:

recovery function     : test_recover()
save function         : load_fun()
print info function   : null, undefined
module info function  : info_func()
    */

    api_register_module("testmod", test_recover, load_fun, 0x0, info_func);

    /*
register the option test in the testmod section

Function to call when test option is encountered in
section tesmod: test_option()
    */

    api_register_option("test","testmod", test_option);

    /*
initialization succeded
    */

    return 0;
}
```

This module can be compiled using:

```
shell# gcc --shared -fPIC -o libtest.so testmod.c
```

To test this module, copy the resulting shared object (libtest.so) under the module directory (See configuration file) and add an inc_module=libtest.so in bsrecovrc. Comment out all other inc_module, backup your current recovery database (/opt/bsrecov/db/recovery.db) and invoke bsrecov with the following argument in turn: -save, -recover and -modinfo.

After testing, restore your recovery database and uncomment the inc_module statement that you commented in the configuration file.