mancoosi
managing software complexity

Survey of the state of the art technologies for handling versioning, rollback and state snapshot in complex systems

Deliverable 3.1

Nature : Deliverable
Due date : 30/09/2008
Actual submission date : 19.11.2008
Start date of project : 01.01.2008
Duration : 36 months

SEVENTH FRAMEWORK PROGRAMME

Information Society Technologies

Specific Targeted Research Project

Contract no.214898

Seventh Framework Programme: FP7-ICT-2007-1

A list of the authors and reviewers

Project acronym	MANCOOSI
Project full title	Managing the Complexity of the Open Source Infrastructure
Project number	214898
Authors list	Paulo Barata Paulo Trezentos : paulo.trezentos@caixamagica.pt Inês Lynce : ines@sat.inesc-id.pt Davide di Ruscio : diruscio@di.univaq.it
Internal Review **Workpackage number**	Arnaud Laprevote Sophie Cousin WP3
Deliverable number	1
Document type	Report
Version	1.0
Due date	30/09/2008
Actual submission date	19/11/2008
Distribution	Consortium, Commission and Reviewers
Project coordinator	Roberto Di Cosmo : roberto@dicosmo.org

Table of contents

1 Introduction

Managing the complexity of Open Source software components is the main goal of the Mancoosi project. The research work has an impact on different topics and Workpackage 3 is specifically focused in Transactional Upgrades.

As in other systems, like databases, transactional upgrades of software components have desirable characteristics. We should be able to revert to a previous working state of the system when a installation / removal has a non expected behaviour and impact.

Moreover, this rollback may not only be performed after the problem has occured, but also any time in the future given that we cannot predict when the error will be discovered.

This could be accomplished by storing every change in the system, binaries, documents and configuration files. However, the cost of storing such an amount of information is not feasible in a system in permanent evolution.

With the aim of developing and enhancing tools to keep track of system evolution, we proposed to start this workpackge by capturing the state of the art of existing techniques and tools.

This deliverable analyzes various techniques. Each of them focuses on one or more of the following axes:

Domain : What can and should be undone about failures (e.g., binary files, configuration files, user files) ?

Time : For how long a specific upgrade can be undone?

Granularity : Does the undo of one element imply the undo of all other elements to that state in time? Should the undo unit be a file, a package, an upgrade run, or a file system?

We hope that new ideas and new approaches will arise from this discussion and will be accomplished in the other phases of this workpackage.

1.1 Structure of the deliverable

This document contains five chapters.

- Chapter 1 gives a brief introduction to the work developed and the topics addressed by giving an overview of the structure of the deliverable.
- The concepts of Software component management are depicted in **chapter 2**. This chapter presents main lines of research in academic and scientific fields for component management in the last two decades. The state of the art is presented in three sections: run-time reconfiguration, dynamic adaptation and components selection.
- **Chapter 3** introduces the concept of Linux packages which are a particular kind of components. The DEB and RPM file formats are also presented in detail.
- **Chapter 4** describes the state of the art of rollback components, and presents an overview of different components critical for setting up a solution of transactional upgrades. In this chapter, some of the components may not apply directly to transactional upgrades but as bleeding edge technology in their fields they give guidance about trends and solutions for similar problems in other scopes.

- In **chapter 5**, various existing tools are analyzed and their strengths and weaknesses stated. Typically, these tools are intrinsically inter-connected with the operating system.
- Finally, the Conclusion, **chapter 6**, points out to future directions and different possible approaches.

2 Software component management - a scientific state of the art

In software engineering, component-based development (CBD) [34, 62] is an important emerging topic. CBD aims at composing systems from prebuilt software units or components. A system is developed as a composite of subparts rather than a monolithic entity. Such an approach reduces production costs by composing a system from existing components instead of building it from scratch. It also enables software reuse since components can be reused in many systems. More precisely, Szyperski defines a component as *"...a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties..."* [62]. As a corollary of this definition, he states that *"software components are binary units that are composed without modification"*.

According to [60] software components can also be defined in terms of a number of properties which are summarized below. In particular, a software component can be seen as a *unit of*:

- *abstraction,* a component provides some functionality that can be accessed by means of a contract which consists of interfaces. Moreover, a component usually hides its implementation, state and resources, which are needed to perform its tasks;
- *composability,* a component is a reusable software building block, i.e. a pre-built piece of encapsulated application code that can be combined with other components and with hand-written code to rapidly produce a custom application;
- *deployment,* a component can be packaged and delivered independently from other software components. Furthermore, it can be installed onto the nodes of the target infrastructure and loaded into and prepared to execution in its execution environment (called *container*);
- *system management,* each component can be individually managed and configured in terms of its quality of service, such as performance, provided security or fault tolerance.

Any CBD methodology depends on an underlying *software component model* [39], which defines what components are, how they can be constructed, how they can be composed or assembled, and how they can be deployed. In particular, a software component *model* is given by the definition of the *semantics* of components (that is, what components are meant to be), the *syntax* of components (how they are defined, constructed and represented), and the *composition* of components (how they are composed or assembled). Current component models can largely be divided into two categories [39]: *i)* models where components are objects, as in object-oriented programming, and *ii)* models where components are architectural units, as in software architectures [14]. Enterprise JavaBeans (EJB) [45], COM [16], .NET [64], and architecture description languages (ADLs) [21] are examples of these categories.

The development and management of component-based software are typically based on processes like the one depicted in Fig. 1. It consists of six main phases: *system specification requirements*, *design*, *component selection and adaptation*, *component deployment*, *system run-time*, and *system upgrade*. In particular, once the requirements of the software system being developed have been defined, a *design* phase is performed by taking into account already existing components which will be properly selected, adapted, and validated in order to be composed during a *component deployment* phase. At this stage, the software components are deployed into a target environment by delivering them and installing them in their destination.

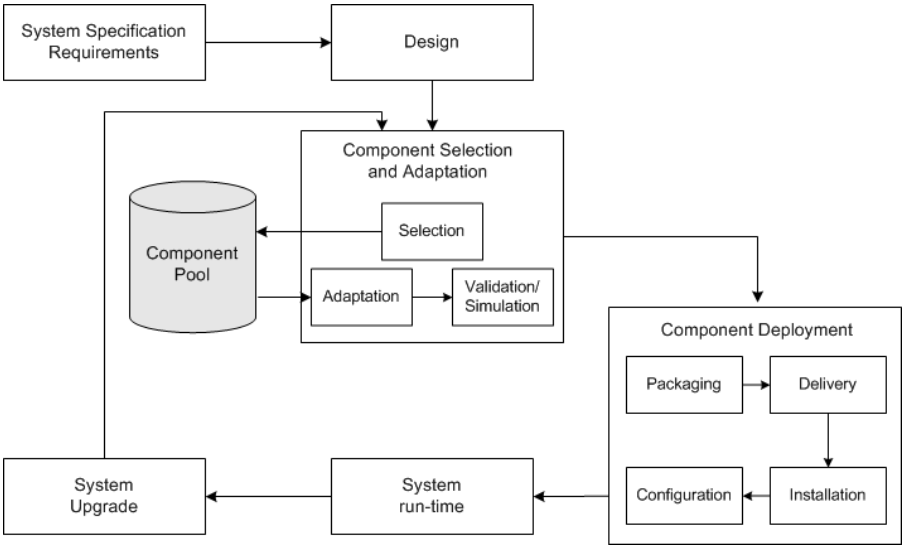

Figure 1: Component-based software system lifecycle

If needed, the component is configured according to the deployment criteria of the local run-time environment and the preferences of the local system users. After that, all the composed components can be activated and the integrated system can be started.

Systems are made up of components that can be modified by replacing the components with their new versions or by withdrawing components which are not needed any more (see the *system upgrade* phase in Fig. 1). Instead of applying the changes to the system by shutting the system down, the components can be replaced on-line, so that the maximum availability of the system functionality is reached. The changes are applied so that both the scope of the system that has to be stopped and the time of the system maintenance are minimized. Such a dynamic upgrade has a number of intricacies and represents a complex operation which relies on the capabilities provided by the deployment system, like releasing new versions of software in the distributed system, delivering software components to their destinations, allowing installation and uninstallation of software components. Moreover, these mechanisms include storing the state of the old version components and transferring it to the new ones as discussed in the rest of the section.

2.1 Dynamic upgrades of component-based systems

A dynamic upgrade is a particular case of deployment and a particular case of software life-cycle management, in which the deployed code of the component is replaced with a new version and, consequently, runtime instances of the software component are replaced with instances of the new one. In particular, dynamic upgrades consist of basic actions which can be summarized as follows [35]:

– *replacing or upgrading the application components* that define the system functionality. The change is further referred to as component upgrade and involves replacing the implementation of a system component with a new one;

– *modifying the logical structure (topology) of the system*, which adds or removes the system components and the connections between them;

– *altering the physical structure of the system*, which defines how the system components are mapped onto the underlying system physical resources, like assignment of components to network nodes.

All these operations give place to a number of challenges which have to be overcome. In particular, preserving the consistency of a running software is highly important when upgrading it. Consistency preservation is a multi aspect issue which can be investigated taking into account two main issues that are *compatibility specification* (Section 2.1.1) and *upgrade validation* (Section 2.1.2). *State transfer* and component *deployment* are two additional issues that have to be managed; they are respectively described in Section 2.1.3 and Section 2.1.4.

2.1.1 Compatibility

The compatibility issue is related to the notion of component substitutability [59], that is the new component has to "contain" (possibly by interacting one another) the same functionalities implemented by the old one. Furthermore, there must exist a mapping between the different component interfaces. This mapping can be empty, meaning that it is not required, since the message names of the architectural component interface match with the ones of the actual components' interface (i.e., a syntactical and semantic one-to-one correspondence exists). It can be a trivial mapping, meaning that there is a one-to-one correspondence between messages of the architectural component interface and the ones of the actual components interface except for their names (i.e., a semantic one-to-one correspondence exists). In this case, the needed message relabeling is realized by means of trivial component wrappers that directly delegate messages without any particular message routing logic. In the most general cases mapping can also be complex, meaning that there exists a many-to-many correspondence between messages of different components. In this case, the component wrappers implementing such a mapping also have a complex message routing logic. In general, this interface mapping cannot be built automatically and requires to develop component wrappers solving, e.g., syntactical mismatches (by hand).

Finally, the selection of suitable components, as required by an upgrade, must often be performed taking into account non-functional aspects, as will be detailed in Section 2.2.3.

2.1.2 Upgrade validation

Upgrade validation consists of checking that the upgrade is valid, i.e., the new and the old software components fulfill the substitutability constraints previously discussed, and that the upgrade has terminated successfully, i.e. the system consistency is preserved once the system is upgraded. Moreover, it is necessary that the upgraded system behaves according to the expectations and that the introduced changes do not make the system incompatible with respect to its old version. Upgrade validations can be performed in different moments and, in this respect, they can be distinguished into:

– *online validation,* which is performed during the upgrade process and immediately after

the upgrade is done over a limited period of time. During such a phase, the new version of the system may be run in a testing mode in which it performs a possibly constrained set of its functions and the results of its computations are compared to the results of the old version. If the checks are positive, the new version may fully replace the old version which can be then deactivated. Otherwise, the upgrade is not valid and the system has to be rolled back in order to resume the old version;

– *offline validation,* which is done before the actual upgrade is carried out in the running system. The new version of the code is validated and tested in an environment simulating the real system. The new version of the component can be used to upgrade the target system only if the validation succeeds.

2.1.3 State transfer

Another problem related to dynamic upgrade concerns component state transformation between the old and the new version of the considered software component. In particular, to preserve the consistency of the whole system, the state of the component being upgraded has to be transferred to the new version. This means that the computational state of a component has to be stored somewhere temporarily and subsequently restored by the new component. In this way, the new version of the component can correctly carry on the computations from the point where they were stopped because of the upgrade. In this respect, a well-defined protocol is required even for transferring the component state from a newer version back to the old one in case the old version has to be resumed. An example of such a standardized protocol is GIOP [28] or the one supported by Java serialization mechanism [42].

2.1.4 Component deployment

The software components which have to be deployed need to be packaged prior to transfer and delivery to the target nodes. The package must contain the software components in terms of the code and a meta-data description of the system including its requirements and dependencies. Versions of the component implementation also have to be managed in order to perform the system upgrade. The mechanism supporting this is called *versioning* and is usually achieved by assigning a version number to each component version released by the software provider. There is a number of versioning schemes applied in different development and deployment environments. The version numbering schemes can be as simple as monotonously increasing integer numbers whenever a new component version is released [29]. A component version number can also be defined as a n-tuple of integer numbers, each of which is increased with regard to a different aspect of the system that has been changed in the new software version [55]. More complex versioning schemes are based on description of component set-valued features to model relationships between component versions sets [8]. In any case, version numbering is related to the software release and should be orthogonal to other aspects of component, like naming of runtime instances of the component or naming scheme for the internal software artifacts comprising the component implementation. Additionally, since the interface and code of a component implementation can evolve independently of each other, these two versioning mechanisms are frequently separated.

2.2 State of the art

The approach of the dynamic and automatic composition of software components is related to a large number of other problems that have been considered by researchers over the past two decades. We organized the description of these works into three main research areas: component frameworks to the *run-time reconfiguration*, described in Section 2.2.1, *dynamic adaptation* of component-based systems, discussed in Section 2.2.2, and *components selection*, discussed in Section 2.2.3.

2.2.1 Component frameworks to the run-time reconfiguration

Jadda (Java Adaptive component for Dynamic Distributed Architecture) [30] is a framework that relies on architecture specifications given by means of xADL [4] to support dynamic reconfiguration. Jadda support for ad-hoc reconfiguration is accomplished via a console that is used to submit a xADL file with the change specification. Jadda is limited to ad-hoc reconfiguration with no formal support. It thus lacks an automatic support to guarantee the change consistency with respect to the system properties of interest.

In Fractal [3], a component is both a design and a runtime entity with explicitly provided and required interfaces. Each component is composed of a finite number of other components, which are under the control of the controller of the enclosing component. The Fractal framework [3] is a Java software framework that supports component-based programming according to the Fractal model. It is an open framework that comprises a *core* and several *increments*. The core defines the minimal concepts and APIs necessary for Fractal-based component programming. Increments define additional concepts and APIs which extend the core framework to allow for different forms of component composition, configuration, and administration. A Java implementation of the Fractal Framework has been developed. It is called Julia [2] and offers three different forms of configuration: static, dynamic, and partially dynamic.

SOFA [20] is a component model with a number of advanced features. It allows for dynamic evolution of architectures at runtime. The controlled evolution of the SA is driven by well-defined evolution patterns. These patterns are supported by the runtime environment which handles reconfigurations according to them. A *factory* pattern is used to create and add a new component. A *removal* pattern is used instead to destroy a component that was previously created. In SOFA/DCUP [54] (Dynamic Component UPdating), each component defines one component manager (CM) and one component builder (CB), that are responsible for managing the associated component. A component may have several implementation objects and/or sub-components that provide its functionality. A component is divided into two parts: a *permanent part* and a *replaceable part*. Therefore, it provides two kinds of operation: control operations and functional operations. Adapting a component means replacing its replaceable part by a new version at run-time. When one of the sub-components of a global component has to be adapted, the whole component is affected, and its replaceable part is re-deployed, therefore, the entire application has to be re-deployed. DCUP does not provide any degree of automation, i.e. all the adaptation operations must be done by the administrator.

Batista et al. in [15] propose a meta-framework called “Plastik” which supports the specification and creation of component-based systems by facilitating and managing the run-time reconfiguration of such systems while ensuring integrity across changes. This meta-framework is the integration of an architecture description language (an extension of ACME/Armani) and a reflective component framework called OpenCOM [25]. Plastik generates component

systems that can be dynamically reconfigured either through programmed changes or through ad-hoc changes. The run-time level is based on the OpenCOM framework. Components are encapsulated units of functionality and deployment that interact with their environment (i.e., the other components in the system) exclusively through interfaces. By default, components are written in C++. The OpenCOM framework supports a set of so-called *reflective meta-models* which facilitate reconfiguration of systems by permitting different system aspects to be inspected, adapted and extended at run-time. A *run-time configurator* is responsible for managing the OpenCOM run-time layer. Since OpenCOM considers components as white-box entities, it does not allow the handling of third-party and black-box components.

The OSGI Platform [7] provides standardized primitives that allow applications to be constructed from small, reusable and collaborative components (implemented in Java). These components can be composed into an application and deployed. The OSGI technology provides a Dynamic Software Architecture (or Service-Oriented) with functions to change the components composition dynamically without requiring restarts. The basic unit of deployment is a *bundle*, which provides services and which can depend on and use other services. A bundle can be seen as a primitive component and its services as interfaces of the component. Moreover the OSGI framework allows bundles to select an available implementation at run-time through the framework service registry. Bundles register new services, receive notifications about the state of services, or look up existing services to adapt to the current capabilities of the device. This aspect of the Framework makes an installed bundle extensible after deployment: new bundles can be installed for adding features or existing bundles can be modified and updated without requiring the system to be restarted. Once a bundle is started, its functionality is provided and services are exposed to other bundles installed in the OSGI Service Platform. For each bundle installed in the OSGI framework, there is an associated *Bundle Object* [7] that is used to manage its life cycle.

2.2.2 Dynamic adaptation of component-based systems

As part of the RAPIDware project, Zhang et al. [66] introduced an aspect-oriented approach to add dynamic adaptation infrastructure to legacy programmes in order to enable dynamic adaptation. They separate the adaptation concerns from the functional ones of the programme, resulting in a clearer and more maintainable design. This concept of separation of concerns is crucial to perform adaptation, especially when it has to be performed at run-time.

Kulkarni et al. [38] propose a distributed approach to compose distributed fault-tolerant components at run-time. They use theorem proving techniques to show that during and after an adaptation, the adaptive system is always in correct states with respect to satisfying specified transitional-invariants. Their approach, however, does not guarantee the “safeness” of the adaptation process in the presence of failures during the application of the adaptation strategy.

Appavoo et al. [12] proposed a hot-swapping technique that supports run-time object replacement. In their approach, a quiescent state of an object is the state in which no other processes are currently using any function of the object. We argue that this condition is not sufficient in cases where a critical communication segment between two components includes a series of function invocations. Also, they did not address global conditions for safe dynamic adaptation.

Amano et al. [10] introduced a model for flexible and safe mobile code adaptation, where adaptations are serialized if there are dependencies among adaptation procedures. Their work focuses on the dependency relationships among adaptation procedures supporting the use of assertions for specifying pre-conditions and post-conditions for adaptation, where violations

will cancel the adaptation or roll back the system to the state prior to the adaptation.

Mechanisms used to interconnect components and their ability to cope with architectural mismatches is another related research area. An architectural mismatch occurs when the assumptions that a component makes about another component or the rest of the system are not correct. In the Gacek's Ph.D. dissertation [31] a static mismatch detection has been proposed. The overall idea of this work is that by analysing the characteristics of the architectural elements to be integrated and the styles from which these elements were derived, the system architects are able to localize architectural mismatches during development time. Finally, the Architect's Automated Assistant (AAA)[1] tool has been used to detect mismatches during component composition. The limit of this approach is that it is so specific to the context of software development that it cannot be used for run-time detection of error caused mismatches. Lemos et al. [27] starting from the previous work have presented how these mismatches can be tolerated during run-time at the architectural level. They have applied general principles of fault tolerance to deal with architectural mismatches. However, it is only a preliminary analysis showing a number of particular mismatch tolerance techniques that can be developed depending on the application, architectural styles, types of mismatches, redundancies available, etc. The applicability of this approach to real systems is still an open issue and the authors are trying to define in a more rigorous way the applicability of the approach and its set of general mismatch tolerance techniques.

Synthesis [63] aims at giving an answer to one of the main problems in component assembly: the problem of establishing properties on the assembly code by only assuming a limited knowledge of the single component properties. Synthesis suggests an architectural approach in which the software architecture imposed on the assembly prevents black-box integration anomalies. The basic idea is to build applications by assuming a "coordinator-based" architectural style. Synthesis then operates on the coordinating part of the system architecture to obtain an equivalent version of the system which is failure-free. A failure-free system is a deadlock-free one that does not violate any specified coordination policy. A coordination policy models those interactions of components that are actually needed for the overall purpose of the system. The approach is tool supported [13].

The main idea of [65] is to modify the interaction mechanisms that are used to glue components together so that compatibility is achieved. This is done by integrating the interaction protocol into components by means of adaptors. However, they are limited to only consider syntactic incompatibilities between the interfaces of components and they do not allow the kind of interaction behaviour that is supported in [63]. Moreover, they require a formal specification of the adaptor dictating, for example, a mapping function among events of different components.

Spritznagel et al. in [61] propose an approach to specify wrappers, independent of any particular context of use and exploit this specification to understand things such as impact of its use, the effects on the communication protocol between components, compositional properties, etc. The class of wrappers that they consider are connector wrappers that are primarily designed to affect the communication between components. They define a connector wrapper formally as a protocol transformation and adopt an approach based on process algebras (e.g., FSP). To describe a connector protocol in FSP, they use an approach similar to Wright [9]; a connector is defined as a set or processes. Moreover three classes of properties are analyzed (e.g., soundness, transparency and compositionality).

Other work from Bracciali, Brogi and Canal [17] in the area of component adaptation show how to automatically generate a concrete adaptor from: (i) a specification of component inter-

[1] http://sunset.usc.edu/available_tools/AAA/

faces, (ii) a partial specification of the components interaction behaviour, (iii) a specification of the adaptation in terms of a set of correspondences between actions of different components and (iv) a partial specification of the adaptor. The key result is the setting of a formal foundation for the adaptation of heterogeneous components that may present mismatching interaction behaviour. Analogically to the work in [65] this work provides a full formal definition of the notion of component adaptor, although in both cases assuming a specification of the adaptation to be given in terms of a set of correspondences between methods (and their parameters) of two components requires to know many implementation details (about the adaptation).

Other strictly related approaches are in the *"scheduler synthesis"* research area. In the discrete event domain they appear as *"supervisory control"* or *"discrete controller synthesis"* problem [40, 56] addressed by Wonham, Ramadge et al. In particular, in the scheduler synthesis approaches the possible system executions are modeled as a set of event sequences and the system specification describes the desired executions. The role of the supervisory controller is to interact with the system in order to meet system specification. The aim of these approaches is to restrict the system behaviour so that it is contained in a desired behaviour, called the *specification*. To do this, the system is constrained to perform events only in strict synchronization with another system, called the *supervisor* (or *controller*). This is achieved by automatically synthesizing a suitable supervisor with respect to the system specification. There is one main assumption to deal with deadlocks: in order to automatically synthesize a *supervisor* which avoids deadlocks, they need to consider a specification of the deadlocking behaviours of the base system (i.e., the event sequences that might cause deadlocks). This is a problem because, for large systems, the designers might not know the deadlocking behaviours since they can be unpredictable.

Promising formal techniques for the compositional analysis of component-based design have been developed in [26, 53]. The key of these works is the modular-based reasoning that provides a support for the modular checking of behavioural properties. In [26], De Alfaro and Henzinger use an automata-based approach to capture both input assumptions about the order in which the methods of a component are called, and output guarantees about the order in which the component calls external methods. The formalism supports automatic compatibility checks between interface models, and thus constitutes a type system for components interaction. The purpose of this work is different from ours. The authors check that two components have compatible interfaces when a legal environment letting them correctly interact exists. Each legal environment is an adaptor for the two components. They provide only a consistency check among components interfaces, i.e. they do not deal with automatic synthesis of component interface adaptors (i.e., automatic synthesis of legal environments). However in [53] De Alfaro, Henzinger, Passerone and Sangiovanni-Vincentelli use a game theoretic approach for checking whether incompatible component interfaces can be made compatible by inserting a converter between them which satisfies specified requirements. This approach is able to automatically synthesize the converter. With respect to deadlock-freedom, the specification of the converter's requirements is assumed to be correct. Thus, if, for instance, the specification erroneously introduced deadlocks, they would not be prevented by the converter that it is synthesized in order to be completely compliant to its requirements specification. In other words, a *deadlock preventing* specification of the requirements to be satisfied by the adaptor has to be provided by delegating to the user the non-trivial task of specifying it.

2.2.3 Components selection

Several interesting approaches have been introduced to support the activity of selection of COTS components (see, for example, [48, 37, 58, 44, 47] and references therein). In a component-based development process the selection of components is an activity that takes place over multiple lifecycle phases that span from requirement specifications through design to implementation and integration. Obviously, in different phases, different assumptions are valid and different granularity of information are available, as a consequence different procedure should be used in the selection process and an automated tool support for an optimized component selection would be very helpful in each phase.

As highlighted in [11], some approaches are based on the architecture of the system, whereas others support the selection of COTS given a set of requirements, so such approaches can also be used in the requirements phase, in which the component selection activity is related to the definition of the (functional and non-functional) requirements. However, all these approaches basically provide guidelines to select the components. Only a few of these approaches, some of which are detailed below, support automation for this task.

In [32] the DesCOTS (Description, evaluation and selection of COTS components) system is presented. DesCOTS is made up of four tools that collaborate with each other. It is designed to support quality requirements only, and the component selection criterion is based on quality models, while the possibility that a requirement can be satisfied by a COTS assembly is not considered.

The DEER framework is based on an optimization model. The optimization techniques resolve some drawbacks of the Analytic Hierarchy Process (AHP) and the Weighted Scoring Method (WSM) that are typically used by the component selection approaches. In fact, "both methods come with serious drawbacks such as the combinatorial explosion of the number of pair-wise comparisons necessary when applying the AHP, the need of extensive a priori preference information such as weights for the WSM, or the highly problematic assumption of linear utility functions in both cases." [49]. [49] presents an approach that can be embedded in a process of COTS selection. They suggest to solve an optimization model that, through a multi-objective optimization, provides a set of COTS solutions. This so-called Pareto solution of the model is the one that maximizes a set of objectives (e.g. functionality, usability) under some constraints, such as the resources available (e.g. maintenance cost). Moreover, they suggest how to analyze and explore the space of solutions in order to find the preferred solution. This approach does not consider the possibility that a requirement can be satisfied by a COTS assembly, whereas our approach does.

Few other experiences have been based on optimization problems to automatize the development of a component-based system. For example, in [24] an optimization model is generated and solved within CODER (Cost Optimization under DElivery and Reliability constraints), a framework that supports "build-or-buy" decisions in selecting components. Getting as input value the software architecture and the possible scenarios of the system, CODER suggests the best selection for each component of the system, either one of the available COTS products or an in-house developed one. The selection is based on minimizing the costs of construction of the system under constraints on delivery time and reliability of the whole system. In addition, for each in-house developed component CODER suggests the amount of testing to perform in order to achieve the required level of reliability. CODER supports the selection of (COTS and in-house) component in the design phase, whereas DEER does so in the requirements phase. While CODER suggests the best assembly of components (i.e. the best software architecture), DEER selects a set of COTS (or assembly of more COTS). Since the architecture of the sys-

tem has not yet been built in the requirements phase, DEER does not suggest how to make the best architectural decisions, but it helps checking whether the requirements that do not depend only on the architecture of the system are satisfied by a COTS component or an assembly of more COTS components.

Based on the dynamic monitoring of a system, in [43] an optimization model has been suggested for the best allocation of the components on the hardware hosts while maximizing the availability of the system under some constraints, such as the allocation of two components on the same host.

In [33] an optimization model is formulated to allow the best architectural decisions, while maximizing the satisfaction of the quality attributes of the system using multi-objective optimization under some constraints, such as budget limitations.

In [23] the authors analyze the assumptions and propose the selection procedure in the requirements phase. The selection criterion is based on cost minimization of the whole system while assuring a certain degree of satisfaction of the system requirements that can be considered before designing the whole architecture. For the selection and optimization procedure we have adopted the DEER (DEcision support for componEnt-based softwaRe) framework, previously developed to be used in the selection process in the design phase. The output of DEER indicates the optimal combination of single COTS (Commercial-Off-The-Shelf) components and assemblies of COTS that satisfy the requirements while minimizing costs.

2.3 Conclusions

This section provided an overview of the software component based systems and their management. More attention has been paid to the dynamic upgrades of such systems which have to be maintained consistent during the overall upgradability process. Next chapter will focus on Linux distribution packages that are a particular kind of components which are ”packaged” by distribution editors.

3 Linux distribution packages - a software component point of view

Linux distributions are among the most complex component based software systems made of tens of thousands of components that evolve rapidly without centralized design. Software components are provided in ”packaged” form by distribution editors. Packages are a convenient way for users to get new (or updated) software installed on their computer. A package is more than just a collection of files to be installed. It usually contains additional information required for the proper installation and/or uninstallation: other packages on which this package depends, directories for files to be installed, menu items for the desktop environments, scripts to be executed before/after installing/uninstalling the package, and more. Packages are usually not installed manually by the user, but using a package manager. The role of the package manager is to automate the process of installing (as much as possible), upgrading, configuring, and removing software packages from the user's computer. Packages may be either binary packages or source packages. Binary packages contain the files needed for installation and proper functioning of the software, but not the source files. The files in the binary package are precompiled and are then usually expected to work on a limited set of machine architectures. Source packages contain the files needed for compilation of the software on the user's computer. The source package contains a compilation script (typically in form of a Unix Makefile) which automates the compilation and (afterwards) installation processes. It is considered good practice to enable an uninstallation procedure in the makefile. These source packages, which are originally intended for compilation and (de-)installation of the software on the target machine, are used by F/OSS distributors as basis for their own source packages. The derived distribution-specific source package contains a compilation script which compiles the source files and arranges all relevant files into bundles constituting the binary packages. Source packages are more flexible, because the user may choose to tweak the source and because the compilation will usually be optimized for the user's architecture at compile time. However, most users are not expected to be able to cope with software compilation on their machines and to fix problems with the compilation. Moreover, local compilation can slow down the machine quite considerably.

3.1 Debian Packages

The package management system used is Dpkg and .deb is the file extension. There are two types of packages: *binary packages* and *source packages*. Binary packages contain files that can be installed directly from the package file; source packages contain source code that can be used to create binary packages - it is possible to create multiple binary packages from one source package.

3.1.1 DEB file format

A DEB package (binary or source) is an `ar` file which has three members: the package version (which nowadays is 2.0) and two `gzip`-compressed tar archives containing, respectively, metadata (in proper DEB terminology, the *control data*) and the files that are to be installed as part of the package. The control archive contains all metadata. Besides a *control file* in which

the metadata are stored, it contains MD5 sums for the package data, as well as scripts that are to be run when installing or removing the package.

3.1.2 DEB metadata

Binary packages
The *control file*, which contains all metadata, is a text file which consists of *paragraphs*, each of which consists of *fields*. The paragraphs are separated by blank lines. A field is usually a single line which contains the field name, followed by a colon and the field contents. It is possible to include fields that span multiple lines; in that case, the second and further lines start with a space.

A list of all possible fields (except for dependencies) that occur in the control file of a binary package follows: *Package, Source, Version, Section (main, contrib, non-free), Priority (required, important, standard, optional, extra), Architecture (i386, sparc, all), Essential, Installed-Size, Maintainer, Description*. [1]

Then follow the package dependencies. In the DEB format, there are several different types of dependencies: Depends, Recommends, Suggests, Enhances, Pre-Depends, Conflicts, Replaces.

A dependency can also limit the versions of the package that satisfy it. There are five different 'version operators': = Exactly equal; <= Earlier or equal; >= Later or equal; << or < (deprecated) Strictly earlier; >> or > (deprecated) Strictly later.

Virtual packages
It is also possible to declare dependencies on *virtual packages*. A virtual package is a package that does not exist in itself, but must be *provided* by another package. A more complex example would be a virtual package named `web-server`. A package that needs a web server, but is not interested in any particular web server, could declare a dependency on `web-server`. Any package that provides `web-server` could then satisfy that dependency. Virtual packages do not have versions, but the possibility to add this functionality to later versions of the DEB format is specifically left open.

3.1.3 Source packages

As mentioned several times earlier, the DEB format also has source packages. From one source package, it is possible to build several binary packages. This is reflected in the fact that the control file for a source package consists of one general paragraph and several paragraphs for the binary packages that can be created from the source package. The control information of a source package is different from the one used in binary packages. A source package may *build*-depend on or *build*-conflict with other packages, thus expressing requirements for the source package to compile. Since the source package is in general common to several architectures it contains schemata for the control information of the binary packages which are instantiated at compilation time. For instance, the *architecture* may now either consist of a *list* of architectures (where any abbreviates the list of all supported architectures), or `all` for an architecture-independent binary package. Dependencies in the schema for the control information of a binary package may be qualified by an architecture specifier. The schema

may also contain *variables* which are instantiated at compilation time.

3.1.4 DEB version numbers

A DEB version number consists of three components: First, the *epoch*, a single integer number. This is the most important component; whatever the rest of the version number, a package of epoch `n+1` will always be of higher version than a package of epoch `n`. It is intended to be used in case of a change in version numbering scheme, or if a mistake is made. The epoch is optional (if not present, epoch 0 is assumed), and separated from the upstream version by a colon. If there is no epoch, the upstream version may not contain any colon. Then, the `upstream version`: this usually is the original version of the software that has been packaged. It may contain letters, digits, periods, plus and minus signs and colons. It should start with a digit.

Next comes the optional `Debian revision`, separated from the upstream version by a minus sign; if the Debian revision is not present, the upstream version may not contain a minus sign. The Debian revision, which is of the same form as the upstream version, indicates the version of the Debian package based on the upstream version; therefore, it changes if the Debian package is changed, but the upstream version does not. It is conventional to reset the Debian revision to 1 every time the upstream version is changed. Version comparison is done from left to right; first the epoch, then the upstream version and finally the Debian revision are compared. For any two strings that must be compared (epoch to epoch, upstream version to upstream version or Debian revision to Debian revision), firstly the initial parts that contain only non-digit characters are determined and compared lexicographically. If there is no difference, the initial parts of the remainders that contain only digits are compared numerically. This process (comparing non-digit strings lexicographically and digit strings numerically) is repeated until either a difference is found or both strings are exhausted.

3.1.5 DEB sections

The `main` section
The packages in the `main` section all comply with the Debian Free Software Guidelines[9]. Furthermore, packages in the `main` section do not have any 'positive' dependencies (`Depends`, `Recommends` or `Build-Depends` dependencies) on packages outside the `main` distribution. They also conform to a certain quality standard ("they must not be so buggy that we refuse to support them").

The `contrib` section
The `contrib` section contains packages that do conform to the Debian Free Software Guidelines, but that do not satisfy the requirement of having no dependencies on packages outside the `main` section. The quality standard is the same as for the `main` section.

The `non-free` section
Packages that do not conform to the Debian Free Software Guidelines can be placed in the `non-free` section.

Non-US sections
Each of the three sections mentioned above has a non-US subsection. Packages that are in the `main` section cannot depend on packages that are in the non-US subsection of `main`, but it is possible for packages in the non-US subsection of `main` to depend on packages that are in the `main` section. The same applies to the `contrib` section.

3.2 RPM packages

In this section we give details of the format of RPM packages [6], starting from their structure and focusing particularly on the attributes which detail the metadata associated with the package and, in particular, attributes which represent the relationships with other packages. RPM packages can be of two different types: *Binary* packages, that contain a compiled and ready to install/run packages software and *Source* packages, that contain the source code to build and package the software into a binary package. In this document, we will address binary packages only.

3.2.1 RPM package naming convention

RPM packages follow a well defined naming convention in order to maintain consistency between the name of the package file, and the information encoded in the RPM package format and contained in the file itself. This naming convention is also used by all the tools that support RPM package creation. Given that all the information regarding the RPM package is self contained in the package itself, an RPM package will continue to be usable even if its file name is renamed to some other file name which does not follow the naming convention. Moreover, it is important to notice that the same naming convention is also used in some metadata fields in the RPM package format (see Section 3.2.2). The standard RPM package naming convention is the following: name, version, release and architecture (name-version-release.architecture.rpm).

Here are some examples of package names found in various Linux distributions:
`mc-4.6.1-0.pre3.2mdk.i586.rpm`, `gedit-0.9.7-2.i386.rpm`,
`gaim-1.3.0-1.fc4.i386.rpm`, `kphone-4.1.1-1.fc4.x86 64.rpm`.
Notice how the `release` field is often used, not only to show the actual release number, but also to indicate the distribution the package belongs to (i.e., Mandriva/ Mandrake (`mdk`), Fedora Core4 (`fc4`), etc.).

3.2.2 RPM file format

RPM packages are bundled in a binary format. The format is the same for both binary and source packages. The current version of the RPM format is 3.0 and it is used by all the RPM package managers since version 2.1. An RPM package is divided into four logical sections: *Lead, Signature, Header, Payload*.

Being a (multi-platform) binary file format, RPM has been designed in order to be correctly handled by the RPM package manager, in spite of the actual platform in use. In particular the reference byte-ordering is the one defined for the Internet (network byte order).

The `Lead` section
The `Lead` section of an RPM package is basically used as a "signature" in order to identify the file as an RPM package. For example, the Unix `file`[2] command uses this information in order to recognize the format. RPM package managers and other RPM oriented tools use this information as well. Much of the information that is present in the `Lead` section is obsolete and is actually ignored by current RPM package managers. Moreover, that information is duplicated in the `Header` section. It is maintained only for backward compatibility of the file format with older tools. The structure of the `Lead` section is represented by the data structure `rpmlead`[3] and is made of the following fields: `magic (0xED 0xAB 0xEE 0xDB), major, minor, type (binary: type == 0, source: type == 1), archnum, name, osnum, signature`.

The `Header` structure
The `header` structure defines the format of the `header` and `signature` section of an RPM package file. The choice of the names is a bit confusing and is maintained for historical reasons. The `header` structure is quite complicated and models a small database where it is possible to store and retrieve arbitrary data by the means of keys, called tags. The `header` structure is composed of several header entries that logically provide the actual data. An entry is characterized by the following attributes: `tag, type, count, magic (0x8E 0xAD 0xE8), version + reserved, entries count, data size, index, data`.

The `Signature` section
The `signature` section contains one or more digital signatures to assess the origin of the package. The `signature` section is stored using the `header` structure format described in Section 3.2.2. The `signature` section may contain multiple signatures. However, every RPM package must have at least one signature that specifies the size of the package (identified by the tag `RPMTAG SIGSIZE`) and one that gives the MD5 hash of the package (identified by the tag `RPMTAG SIGMD5`). Multiple cryptographic signatures, identified by the relative tags (e.g., `RPMTAG GPG, RPMTAG PGP`, etc.) could be present, but are not required.

The `Header` section
The `header` contains all the metadata information regarding the RPM package itself. It is stored by using the `header` structure format described in 3.2.2 and provides all the information needed to handle a given RPM package. A detailed description of the relevant metadata attributes is presented in Section 3.2.2

The `Payload` section
The `payload` section contains the actual archive with all the files belonging to the RPM package. The format of the payload is a gzipped `cpio` archive which is uncompressed, depending on the directives specified in the package metadata, when the package is actually installed. The format of the `cpio` archive is `SVR4` with a CRC checksum.

[2]The `file` command tries to associate the format/type to the file which is passed as its argument
[3]Defined in `lib/rpmlib.h` in the RPM source tree [14]

3.2.3 RPM package metadata

In this section, we will examine the most relevant package metadata that are present in the `header` section (3.2.2) of an RPM package. In particular we will focus on those metadata describing package relationships with the other RPM packages (i.e., dependency information). It is clear that all the metadata are encoded using the `header` structure format described in Section 3.2.2 by means of tags and their associated data. For the sake of clarity, in order to refer to package metadata we will use descriptive names instead of actual `tag ID` associated to the data. Moreover, the descriptive names are the ones used in `spec` files. These are files used by automated packaging tools in order to create RPM packages. A `spec` file contains all the directive and the metadata information specified in a textual and readable format. Starting from the `spec` file package tools are able to build a standard RPM package in the format described in Section 3.2.2. In the following section we will use the syntax and the tags taken from the standard spec file format to describe how to build RPM packages. We will not describe all the directives of the spec file format because this will be out of scope for this report. However it is possible to find a quite complete description of these directives in [14].

3.2.4 Descriptive and naming metadata

Descriptive metadata allows the packager to specify informational attributes regarding the package itself. The most relevant metadata to be expressed as package specification tags are: `name, version, release, epoch, description and summary, group`.

Package version format
As already hinted in Section 3.2.1, every package is characterized by a version that is used extensively in package metadata in order to specify relationships between packages. Generally a complete version specification has the following format: `epoch, version, release ([epoch:]version[-release])`.

Obviously, package versions impose an ordering on packages which is used when it is necessary to specify package relationships with other packages. The comparison algorithm breaks the package version and is basically a segmented comparison. The version is broken up in segments, each of them containing either alphabetical characters or digits. The segments are compared in order, with the rightmost segment being the least significant. The alphabetical segments are compared using a lexicographical ascii ordering, while the digit segments are compared the same way after having removed any leading zero. If the two digit segments are equal, the longer the bigger. No additional knowledge is embedded in the algorithm, so a version number `5.6` will be older than `5.0000503` because the comparison will be made between `6` and `503` (i.e. `0000503` without leading zeroes), and `503 > 6`.

3.2.5 Dependency metadata

Dependency metadata are used to establish relationships between packages. Those relationships are used in order to ensure that once the packaged software is installed, the system will provide anything it needs to work properly (i.e., other packaged software, libraries, etc.) By (correctly) specifying dependency metadata it is possible to guarantee that package manage-

ment operations (see Section 3.2.7) will not break the consistency of the system when they are performed. In this section we describe the metadata tags that are used in RPM packages in order to clarify relationships between packages, and we will detail their semantics.

Dependency specification
A dependency relation is always specified by using the name of a package and possibly some additional constraint defined using arithmetic comparison operator.

This is possible because, as described in Section 3.2.4, version number are totally ordered. The RPM package format allows the usage of the following comparison operators when specifying dependency relation constraints: `<, <=, =, >=, =`. The semantics of those operators is the standard one, applied to version numbers. Dependency relation specifications establish relations between the current package in its current version and the set of other packages entailed by the dependency specification: i) if only a package name `P` is specified in the dependency relation, then there is a dependency between the current package in its current version and package `P` in *all* or *any* (depending on the semantics of the relation) of its versions; ii) if a package name `P` and a constraint `C` on its version number is specified (e.g., `>= 2.3`) , then there is a dependency between the current package in its current version and package `P` in *all* or *any* (depending on the semantics of the relation) version that satisfies the constraint `C`.

The tags used for packaging dependency are: `requires, prereq, conflict, provides, obsoletes`.

Automatic dependencies
When building a RPM package, a set of dependency relations are implicitly declared. In order to do so, starting from the list of the files that make up the package, the following operations are performed for each file in the list: *i) if the file is executable, it is examined using the* `ldd`[4] *command in order to find out what are the shared libraries it needs. The names of these shared libraries are actually added to the RPM package as* `Requires` *dependencies, ii) if the file is a shared library, then its* `soname`[5] *is added to the capabilities the RPM package provides using the* `Provides` *tag*.

Even if automatically provided and required library names may seem file names, they are actually capability identifiers that are not related to actual file names contained in the package itself.

3.2.6 Script metadata

In an RPM package it is possible to find metadata that provide an operational behaviour that is executed at some stage, after having issued an operation on a package. These metadata simply specify shell-script or script written in some other interpreted language, and are executed by the RPM package management system. Many script metadata are used to handle source RPM packages, in order to automate the building process. The following ones, however, are used when action are performed on binary packages, in particular, during installation and removal of RPM packages. The tags used for these operations are: *%pre, %post, %preun, %postun*.

[4]The `lddprints` are the shared libraries required by each program or shared library specified on the command line

[5]The `soname` is the name used by a shared library to determine compatibility between different versions of the same shared library

3.2.7 RPM package management system

The RPM package management system is build upon a single command line utility rpm, which provides the user with all the functionalities to: build RPM package starting from source code, install RPM packages, remove RPM packages, upgrade RPM packages, query uninstalled RPM packages for information, and query the installed base of RPM packages.

`rpm` make use of a central database where it stores all the information about the packages that are already present in the system. Each operation provided by rpm queries this database in order to perform consistency checks with respect to package dependencies.

3.2.8 RPM package installation

When `rpm` executes a package installation it performs the following steps: *dependency check, conflict check,* `%pre` *script execution, configuration files processing, payload archive unpacking,* `%post` *script execution, central database update*.

3.2.9 RPM package removal

When rpm executes a package removal it performs the following steps: *dependency check,* `%preun` *script execution, config file backup, file check and removal,* `%postun` *script execution, central database update*.

3.2.10 RPM package upgrade

When `rpm` executes a package upgrade it basically performs an installation of the package first and then a removal of the upgraded ones taking care of correctly handling the various config files that are present in the packages.

4 State of the art of rollback components

4.1 ZFS

Filesystem technology implemented by ZFS claims that provides capacity (128-bit), maintains data integrity and consistence on-disk format, makes self-optimizing performance and a real-time remote replication. ZFS breaks with the concept of volumes used in traditional filesystems. It uses a common storage pool for consistence of writable storage media. This pool can manage media (hard disks or others) in order to provide more capacity of storage as needed and the way it can be performed. It grows and shrinks dynamically as needed without the need to re-partition underlying storage. A copy-on-write (COW) transaction model is used to maintain a consistence format on-disk and ensure that data is never overwritten and all updates are done atomically.

4.1.1 History

ZFS stands for Zettabyte File System and has been designed, developed and implemented by Sun Microsystems. Announced in September 2004 and integrated into the main trunk of Solaris development in October 2005, it was released by OpenSolaris in November 2005. Sun decided that ZFS would be included in Solaris 10 in June 2006.[6]

4.1.2 ZFS Pooled Storage

ZFS storage pool uses malloc/free abstraction to manage memory and does not have partitions to manage. It grows and shrinks automatically, gives all bandwidth always available, and all storage in the pool is shared.

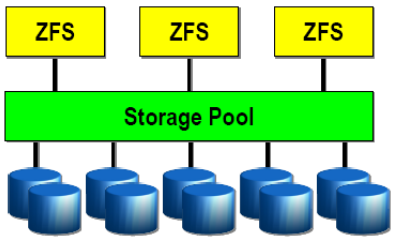

Figure 2: `ZFS Storage Pool image [46]`

4.1.3 Copy on write

The ZFS filesystem has copy-on-write, transactions and checksum as its basic building blocks. Copy on write does not write data modified directly to the place where it comes from, it writes

[6] `http://en.wikipedia.org/wiki/ZFS#Snapshots_and_clones`

on another place that is allocated for instead, using a superblock called *uberblock*. This means that original data is never overwritten in place but it creates a transaction that only saves changes made in one place and keeps original data where they are. Using this transaction process, the overhead of writing data can be reduced. A transaction groups several amounts of data modified and stores it to disk as a whole block of data, in data blocks allocated for that purpose. Using transactions, data can be committed or rolled back. This allows snapshot and clone among data, including the whole filesystem, that can be managed in the future. Transaction can be committed if data is correct or not committed if some changes are not correct, or if the system crashes.

There are two processes that can be done with the created data in those transactions: store data on disk as a part of a modified file (data modified) or store data on disk as a snapshot for future rollback/commit file or filesystem. The data modified is stored using tree nodes linked to the original tree with pointers between the old unmodified data and the new modified data (as shown in Fig. 3).

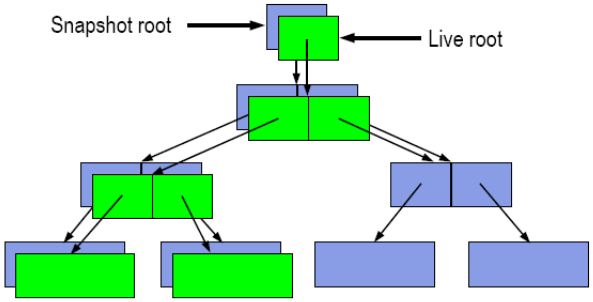

Figure 3: `COW Example [46]`

As depicted in Fig. 3, when a COW is performed, a snapshot is performed automatically. This means that there are two versions of the same file (or filesystem): the original one and the modified one.

4.1.4 Snapshots

The snapshot [7] mechanism is provided by the DSL (Dataset and Snapshot Layer) and by relationships between other components and object sets. A clone is similar to a filesystem with the exception of its origin. Clones are originated from snapshots and their initial content is identical to the originated snapshot. A snapshot is a read-only version of a filesystem, clone, or volume at a specific point in time.

Snapshots can be taken by using the usual commands, or as seen in the previous item, just by modifying some dataset. Snapshot is a read-only copy of a filesystem at a specific point in time. It is created instantaneously and has no number limit. It does not take any significant space, but if some change happens in one block, that block is copied. It can be accessed through *.zfs/snapshot* in the root of each filesystem and allows users to recover files without sysadmin intervention.

[7] `http://opensolaris.org/os/community/zfs/docs/ondiskformat0822.pdf`

4.2 UnionFS

UnionFS is a type of file system that allows to have files physically separated but together in a logical way. A set of directories put together is called a ”Union”, and each directory is called a ”Branch”. UnionFS can bring together filesystems that were formatted with different filesystems (ext2/3, nfs,...). This process is made using an abstract layer that is called stacking. UnionFS can not be mounted as a root file system but rather as a directory.

To *merge* two directories into one, unionfs assigns a precedence to each branch. A higher precedence branch overrides a lower precedence branch. Unionfs works over directories, and if there are two files with the same name in two different directories, the higher precedence is used. This means that a file in the lower branch precedence is not used.

Example:
In this case, there is a file called ’honda’ in separated directories, but because it has the same name, only the one that belongs to the branch with the highest precedence is viewed in the mounted point (merged directory - `/mnt/veiculos`).

The example below illustrates the process of merging two directories:
`/unionfs/example/cars/` and `/unionfs/example/motorbikes/`; formatted in ext3, into a new one (/mnt/motors) mounted as a unionfs. There are two files with the same name in the two original directories, one stands for a car (a Honda car) and the other for a motorbike (a Honda motorbike). When unionfs merges those two files with the same name, only the one in the directory with the major precedence (usually the first one) is sent to the merged directory.

```
mount -t unionfs -o dirs=/cars:/motos none /mnt/veiculos
```

Ext3		**UNIONFS**
/example/cars/renault	`send file`	**/mnt/motors/renault**
/example/cars/honda	`send file`	**/mnt/motors/honda**
/example/motorbikes/ducatti	`send file`	**/mnt/motors/ducatti**
/example/motorbikes/honda	`don’t send file`	`because there is already another file with the same name ’honda’`

4.2.1 Snapshot

UnionFS can freeze a directory at a specific point in time, which is the way it implements a snapshot. When the `unionctl` command is used, any branch configuration can be changed on the fly. It could be a removal operation, an addition, a change from read-write to read-only,

etc. With this possibility, a filesystem snapshot can be created.

To create a snapshot, the location, a new branch for the files of snapshot, needs to be specified in the first step.

The `/myCar` directory is used to make a snapshot of the system.

With this operation, this mount point is used in some control commands such as `unionctl` and is required to add new branches in the process of creating Snapshots.

```
mount -t unionfs -o dirs=/myCar none /myCar

unionctl /myCar --add /snapshot/0
```

With this action, a new branch named `/snapshot/0` will be added.

The files created in the /myCar directory are also stored in the `snapshot/0`.

Once a new directory `snapshot/1` is created and added to `/myCar`, pointers to all the files created will be stored in this new branch, and the `/snapshot/0` remains as it was before this operation.

A snapshot was created as `/snapshot/0` but still with no accessible information from `/myCar` [8] mount point/directory. This is the second step in snapshot creation.

```
unionctl /myCar --add /snapshot/0
touch    /myCar/test123

unionctl /myCar --mode /snapshot/0 ro
unionctl /myCar --add /snapshot/1
echo     "Test123">>/myCar/test123

unionctl /myCar --mode /snapshot/1 ro
unionctl /myCar --add /snapshot/2
echo     "Test123">>/myCar/test123
```

Once a new branch is created and added to the mount point, the older one keeps its data unchanged.

The next figure shows this behaviour in three steps.

```
     touch test123   echo "Test123">>test123  echo "Test123">>test123
--------------|-----------|-------------------------|-----------> time

snapshot0   --add      --mode (ro)              --------
            test123    test123                  test123
            size 0     size 0                   size 0

snapshot1     X        --add                    --mode (ro)
              X        test123                  test123
              X        size 9                   size 9

snapshot2     X          X                      --add
              X          X                      test123
              X          X                      size 18
```

[8] This information is not accessible from /myCar directory, but can be accessed directly from other place by the system administrator just like the access of any other common directory

4.2.2 Copy-on-write

The copy-on-write semantics on unionfs is implemented by using the snapshot method, as shown below. This aims at changing a directory to read-only and adding a new branch:

```
unionctl /myCar --mode /snapshot/0 ro
unionctl /myCar --add /snapshot/1
```

to receive the new set of files created in the mount point `/myCar`.

4.3 LVM

LVM stands for Logical Volume Management. With LVM, users can create Volume Groups that can manage several disks as a whole space together. It has the capability to manage mass devices for storage, in a single volume called Volume Group(VG). It can bring together a large number of disks and create Volumes Group with those disks. In that way, users can access the information as they only have one single place for dataset. Volume group can put together several mass storage elements (as hard disks, flash memories, optical disks, etc.) all together in a single Volume. All the space that is available for this VG can be used by Logical Volumes(LV) as they are accessing only one ”hard disk”. The system looks into these LV as they belong to a single volume (VG) and the administrator can extend or shrink the size of the LV’s as needed. On the other hand, he can add another storage device to the VG (increasing the size of the VG), without interfering with any of the logical structure already created, and use it as new requirements.

4.3.1 History

LVM1 was an implementation of LVM, integrated in Linux Kernel in 1997. There is another implementation of LVM, called EVMS, made by IBM, and competing with LVM2 for its entrance on Linux kernel, but LVM2 ”won” the competition. LVM is now on version LVM2, and was integrated on 2.6 kernel tree. Both EVMS and LVM use Device Mapper for their implementation to communicate to kernel land and userland. Snapshot is a new implementation of LVM and was integrated on 2.6 kernel version.

4.3.2 Device Mapper (dm)

Device Mapper is a framework to support and implement LVM by the kernel. This framework makes the definition of ranges of sectors as blocks possible. It processes data from a virtual block device into another block device. This information is stored in a mapping table, and consists of an ordered list of rules. It can be accessed by shell scripts (in the userland) using dmsetup tools and libdevmapper.

4.3.3 How LVM works

All the Physical Volumes (PV) devices connected to the system can be grouped together in one or more VG.

See Fig.4 (LVM Structure):

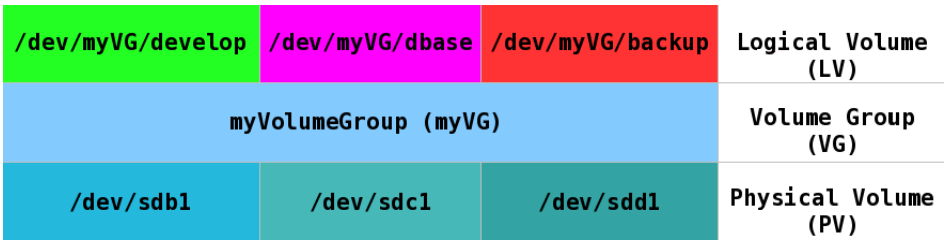

Figure 4: LVM Structure

In the figure, there are three Physical Volumes (/dev/sdb1, /dev/sdc1 and /dev/sdd1), and one Volume Group - myVolumeGroup - created on the PV. This VG is a logical structure mapping all three storage devices as only one storage block. Over myVolumeGroup, a Logical Volume can be created as wanted (taking care of mass storage physical limits). The size of each Logical Volume has to be defined at creation time, and can be ”extended” or ”shrunk” later.

The output of commands execution for all this creations are as follows:

- Creation of PV (Physical Volume)

```
pvcreate /dev/sdb1 /dev/sdc1 /dev/sdd1
```

- Creation of VG and put all PV’s into VG

```
vgcreate myVolumeGroup /dev/sdb1 /dev/sdc1 /dev/sdd1
```

- Creation of LV’s and put all LV’s into VG

```
lvcreate --name develop --size  35G myVolumeGroup
lvcreate --name dbase   --size  30G myVolumeGroup
lvcreate --name backup  --size  20G myVolumeGroup
```

Three logical volumes were created. It is necessary to format those LV with some filesystem format to manage information in them. For instance, formatting this LV’s with ext3 filesystem:

```
mkfs.ext3 /dev/myVolumeGroup/develop
mkfs.ext3 /dev/myVolumeGroup/dbase
mkfs.ext3 /dev/myVolumeGroup/backup
```

Making directories for mounting:

```
mkdir /var/develop /var/dbase /var/backup
```

Now the logical volumes formatted can be mounted as shown below:

```
mount /dev/myVolumeGroup/develop /var/develop
mount /dev/myVolumeGroup/dbase   /var/dbase
mount /dev/myVolumeGroup/backup  /var/backup
```

After that output, the creation process for a rollback, which can be mounted over the system above, should be shown. Rollback is a mount point created to receive the Snapshot mounted in the original LV some time later.

4.3.4 Snapshot

LVM can create snapshots that are a 'frozen image' of some logical volumes allowing the revert of changes made by some filesystem operation. A LVM snapshot is a copy of a partition, saving all its data in another logical volume. This snapshot could be used to create a backup of the whole system, rollback to a previous state of the filesystem or a volume, create copies of the system very fast, without stopping it (using backup tools that are slower and must not stop the system), etc. On average, the space needed for a snapshot is about 15%-20% of the original space.

When a snapshot of a specified LV (`/dev/myVolumeGroup/develop`) is taken at a specific point in time, it comes to freezing the specified LV at that moment. This volume is created in runtime, and is called `/dev/myVolumeGroup/developsnapshot`, that is the name used in the command line when executing:

```
lvcreate -L20G -s -n developsnapshot /dev/myVolumeGroup/develop
```

To mount `/dev/myVolumeGroup/developsnapshot`
on `/mnt/myVolumeGroup/developsnapshot`, that directory must be created:

```
mkdir -p /mnt/myVolumeGroup/developsnapshot
```

Then, it is possible to mount the snapshot just created, at the mount point:

```
mount /dev/myVolumeGroup/developsnapshot /mnt/myVolumeGroup/developsnapshot
```

The snapshot information is usually accessible executing what follows:

```
ls -l /mnt/myVolumeGroup/developsnapshot
```

Merge a Snapshot

Snapshots are mostly used to back up information and rollback to a previous state, preventing an installation of a package that may break the system. This capability is used to make a snapshot before the installation of a package. In this case, if the transaction fails, it is possible to restore the whole system by merging the snapshot over the original system. Therefore, the snapshot becomes the main volume again.

To merge a snapshot the following command should be executed: `lvconvert -M lv_snapshot`. After this execution, and once the merge finished, the snapshot is removed automatically.

The merging types are:

- `nameorigin`, that does not allow mounting the origin nor the snapshot at the starting moment of merge. Once started, and if not finished yet they can be mounted;
- `namesnapshot`, that does not allow mounting the origin but the snapshot can be mounted, when merging starts;

- `onactivate`, selects the next merging snapshot when the system is rebooting. It makes the merge and starts the root filesystem with this snapshot;

The following steps should be done in order to create a snapshot and merge it onto the system, and avoids a potential problem:

1. Before installing a package, create a snapshot
2. Install the package
3. (A transactional upgrade problem occurs)
4. Create a new snapshot
5. Activate the first snapshot with the `--onactivate` option
6. Reboot

4.4 Git

Git is an open source code software to manage versions of source code. This project was created by Linus Torvalds and was initially meant to manage Linux Kernel source development. It was also designed to be a core (low-level) for GUI's front ends, that can be much user-friendly. Git was inspired by two version control systems: BitKeeper and Monotone.

Git has become a complete distributed revision control system, and is used by several projects to manage its version control, such as WINE, X.org, SAMBA, Firefox, and Thunderbird. Git specifically focuses on speed [20], efficiency and real-world usability on large projects. But it also works very well for small personal projects.

Several features are included in Git.

File contents in Git are stored as blobs that are leaf nodes in a tree. A blob is a name generated with the SHA1 hash algorithm, and has an `id` made up of the content and size of that file. This `id` is always the same for the same content, wherever it is generated. It is a unique and arbitrary number, which is stored in the repository. If there are more than one file with the same content, it will create a hard link for this `id` and will remain in the repository until it has a link to this `id`.

All commits in Git will generate new tree objects that have all files associated to this commit. This will allow easy comparison of files content and their previous state. All operations over a file or a directory will generate a SHA1 (40 bits) hash id unique name.

4.4.1 History

The development of Git by Linus Torvals started in April 2005. The major reason for its development was because the BitKeeper (another revision control system) stopped its out-of-charge open source tool.

4.4.2 Features

- Distributed development[9]
  Git, as many other version control systems, creates a local copy of the entire development history for each developer, and copies the changes made from one repository to another. These changes are imported as additional development branches that are merged into a local development branch as they were in the main repository. The repositories are accessed through Git protocol, which is very efficient, or just using HTTP. The repository can be published anywhere without requiring any specific webserver configuration.

- Non-linear development
  Git supports branching and merging, including tools for visualizing and navigating a non-linear development history. In its core it is assumed that merging operations happens more often than writing, since tipically OSS projects have often different people working on the document at the same time.

- Large projects
  Git is known to be fast and scalable for large projects and long histories[20]. It is an order of magnitude[21], which is faster than most other version control systems [22], and several orders of magnitude, which are faster on some operations[20]. It also uses an efficient packed format for long-term revision storage.

- Cryptographic authentication of history
  The Git history is stored in such a way that the name of a particular revision (a "commit" in Git terms) depends upon the complete development history leading up to that commit. Once published, it is not possible to change the old versions without it being noticed. Moreover, tags can be cryptographically signed.

4.4.3 Components

- Repository

  The repository is an archive that shows how the working tree looks like at different times. All updates are stored there. It also allows comparison of those updates through the working tree and moves it, past snapshots, to the current branch.

- The index file

  The index file is used to register all changes made in the working tree before it commits to the repository. It is a way of keeping the job safe before committing. Since the user is required to confirm the commit, all changes (atomic operation) will be recorded in the repository at once. The index file has all the information necessary to commit, such as the differences between the files to be committed. To save this information in the index file, we use the following: `git add <filename>`

[9] `http://git.or.cz/`

- Working tree

  The sub-directory `.git` indicates that a working tree exists on the filesystem with the structure of the repository. This `.git` sub-directory includes all the files and sub-directories needed for managing the repository.

- branch

  A branch is a name for a commit. It can be called a reference too. Branch is a tip or a parent of a commit that defines its history. It is a top of `branch of development`.

- master

  The branch named `master` is the mainline of development in most repositories.

- HEAD

  HEAD is a file that refers to the most recent commit of the last branch checked out. The master branch would be equivalent to the HEAD when the user is working on the master branch. The HEAD content can be modified to refer to another branch as current.

4.4.4 Implementation

When a new repository is created, Git creates a local `.git` directory, where configuration files and all the information needed for managing the repository is stored. Once the repository is created, all the changes, creations and other operations over files or directories made in the working tree are identified by the git system. They can be committed to the repository by running the `add` and `commit` command respectively. With the `add` command, those changes are saved in the `index` file (metadata information about files) and with the `commit` command they are stored in the repository. Once the `commit` executed, all the saved information in the index file is used to store data in the repository, so the `index file` is like a stage, or a middle point (temporary file) between the user changes and the commits.

To create a repository and its default structure, the following commands must be executed:

```
[root@localhost git_repos]# git init
Initialized empty Git repository in .git/

[root@localhost git_repos]# ls -A
.git/

[root@localhost git_repos]# git status
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)

[root@localhost git_repos]# ls -l .git
total 36
drwxr-xr-x 2 root root 4096 2008-07-03 19:37 branches/
-rw-r--r-- 1 root root   92 2008-07-03 19:37 config
-rw-r--r-- 1 root root   58 2008-07-03 19:37 description
-rw-r--r-- 1 root root   23 2008-07-03 19:37 HEAD
```

```
drwxr-xr-x 2 root root 4096 2008-07-03 19:37 hooks/
-rw-r--r-- 1 root root  352 2008-07-03 19:38 index
drwxr-xr-x 2 root root 4096 2008-07-03 19:37 info/
drwxr-xr-x 8 root root 4096 2008-07-03 19:38 objects/
drwxr-xr-x 4 root root 4096 2008-07-03 19:37 refs/
```

4.4.5 Manage a repository

Once Git is installed, the latest development version of Git can be downloaded, via "git protocol":

```
git clone git://git.kernel.org/pub/scm/git/git.git
```

If there are problems with the connection over port 9418, that is used by the git protocol, the access to the repository can be done over the HTTP protocol (using HTTP is slower but works even behind firewalls and such):

```
git clone http://www.kernel.org/pub/scm/git/git.git
```

```
[root@localhost git]# git clone git://git.kernel.org/pub/scm/git/git.git
Initialized empty Git repository in \
/root/Documentos/cxa/git/repos/git_source/git/.git/
remote: Counting objects: 77575, done.
remote: Compressing objects: 100% (25761/25761), done.
remote: Total 77575 (delta 56281), reused 70698 (delta 50421)
Receiving objects: 100% (77575/77575), 17.49 MiB | 94 KiB/s, done.
Resolving deltas: 100% (56281/56281), done.
```

The way git works

Git has a HEAD file that stores the pointer to the head of the newest branch in the repository, and every time the head of the repository tree changes it has a corresponding reference. Every commit is thus done over this head. Each commit is a tree of objects, has a SHA1 identifier of this commit, a parent of the branch created that is the previous commit. With this sequence of commits and parents, any user could point his head file to a previous commit, and consequently look at it as a rollback or a vision of a snapshot previously created. Each commit could be interpreted as a snapshot that could be pointed at any moment. Another very important point in git is the merging capability, a feature of every SCM's (Source Control Manager), but git merges more efficiently than other tools or applications. Git executes several commands to show the structure created by the last commit executed before the source shown below is downloaded.

This is the output of all the objects committed into the repository once the last commit has been executed. The transaction, a whole block of information, is like a new tree.

```
[root@localhost git]# git-ls-tree HEAD
100644 blob 6b9c715d21d5486e59083fb6071566aa6ecd4d42 .gitattributes
100644 blob 8054d9ddb8be3630f500be2ff37228aa17e839b1 .gitignore
100644 blob f88ae77a1f298004320bb8cc9e6a2bd9b0cafd06 .mailmap
100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3 COPYING
040000 tree 8ca8e12bb38c186b4db27966b75fddbcb76a1596 Documentation
100755 blob cb7cd4b53827fa6820e84b1318572d0115b3b17f GIT-VERSION-GEN
```

```
100644 blob 0efb4b7801ca79d7ddec90c49826336d6d943294 INSTALL
[...]
100644 blob e4c8e225fd232dfd642aa13d7ae5b64b9827c915 write_or_die.c
100644 blob 7a7ff130a34942506e6068105ac5946c9404bf18 ws.c
100644 blob e7d42d0491743d577f84dc262ead721eb82c4785 wt-status.c
100644 blob 78add09bd67c727babb61cd1eaa773bcd0c6e55e wt-status.h
100644 blob 61dc5c547019776b971dc89d009f628bbac134fd xdiff-interface.c
100644 blob f7f791d96b9a34ef0f08db4b007c5309b9adc3d6 xdiff-interface.h
040000 tree 88b6f65753131f1f2c0dbceb1f37950e3494833a xdiff
```

It also has a SHA1 `id`, that identifies the transaction, the head of that tree.

```
[root@localhost git]# git rev-parse HEAD
44701c67fd1d5d771b440c8646b7b268d4f1402d
```

The transaction type can also be shown.

```
[root@localhost git]# git cat-file -t HEAD
commit
```

The following command shows the `id` of the tree created with this commit and the parents of this commit.

```
[root@localhost git]# git cat-file commit HEAD
tree aa64a8e791d43ce75333e822fd751963c9cab868
parent af9a01e1c2f6a8814b817eb7f3d78814389a3212
parent 3ecb171d2b9aca7b09bf112594977d00925893b4
author Junio C Hamano <gitster@pobox.com> 1215329713 -0700
committer Junio C Hamano <gitster@pobox.com> 1215329713 -0700

Merge branch 'qq/maint'

* qq/maint:
  Fix "config_error_nonbool" used with value instead of key
```

Using the `tree id` shown in the command above (`git-ls-tree HEAD`), the same output is easily seen. This means that an `id` created by the system can be accessed directly to see the content of such a commit. In this case, it is the last one.

```
[root@localhost git]# git ls-tree aa64a8e791d43ce75333e822fd751963c9cab868
100644 blob 6b9c715d21d5486e59083fb6071566aa6ecd4d42  .gitattributes
100644 blob 8054d9ddb8be3630f500be2ff37228aa17e839b1  .gitignore
100644 blob f88ae77a1f298004320bb8cc9e6a2bd9b0cafd06  .mailmap
100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3  COPYING
040000 tree 8ca8e12bb38c186b4db27966b75fddbcb76a1596  Documentation
100755 blob cb7cd4b53827fa6820e84b1318572d0115b3b17f  GIT-VERSION-GEN
100644 blob 0efb4b7801ca79d7ddec90c49826336d6d943294  INSTALL
[...]
100644 blob e4c8e225fd232dfd642aa13d7ae5b64b9827c915  write_or_die.c
100644 blob 7a7ff130a34942506e6068105ac5946c9404bf18  ws.c
100644 blob e7d42d0491743d577f84dc262ead721eb82c4785  wt-status.c
100644 blob 78add09bd67c727babb61cd1eaa773bcd0c6e55e  wt-status.h
100644 blob 61dc5c547019776b971dc89d009f628bbac134fd  xdiff-interface.c
100644 blob f7f791d96b9a34ef0f08db4b007c5309b9adc3d6  xdiff-interface.h
040000 tree 88b6f65753131f1f2c0dbceb1f37950e3494833a  xdiff
```

The user can switch from a branch to another, and create a new branch too. Many branches can be created in one repository, and the user can switch from one to another. The changes made in such a branch belong to that branch, keeping those changes from one to another.

```
[root@localhost testeRepos]# git checkout master
Switched to branch "master"

[root@localhost testeRepos]# git branch
* master
  novo

[root@localhost testeRepos]# git checkout novo
Switched to branch "novo"

[root@localhost testeRepos]# git branch
  master
* novo
```

If any commit is done at this moment, it is linked to the branch called 'novo'. So the new commit becomes the head of the 'novo' branch. At this point, if the user switches to the branch called master (that is a default, in every repository) he will not see the commit done over the branch called 'novo'. It thus behaves as a snapshot over the repository.

The information available to be committed in the repository is stored in the index file, and the data files are stored in the filesystem. So, there is 'something' between them (repository and filesystem) that is called 'index' file.

Files

Git applies a SHA1 hash to create a blob of a file content, that is unique for the same content of a file. If there are many files with different names, but with the same content, they are stored once, but have many hard links to this SHA1 blob (stored in the repository). If someone wants to view the content of these files they can use `git-cat-file blob db2a1ac1`, but it has to be already stored in the repository. If the content of that file changes, the blob also changes.

The next example shows the SHA1 hash of the file 'f1', and once its content changed, it is easy to see that its SHA1 hash has changed. The content of a file can be seen by using the 'git-cat-file' command, but if the file has not already been committed to the repository, it is not possible to see it. It must then be committed to the repository, before its content can be shown.

```
[root@localhost testeRepos]# git-hash-object f1
db2a1ac1b02d56df7840c81aeff29cf6b471805b

[root@localhost testeRepos]# echo "mais um teste" >> f1

[root@localhost testeRepos]# git-hash-object f1
1a60379eb0f54078119db17d8b641df0052b071c

[root@localhost testeRepos]# git-cat-file blob db2a1ac1
teste do f1
alteracao sobre o file f1
user: test a acrescentar coisas manualmente
```

```
change test
pb change
mais um teste

[root@localhost testeRepos]# git-cat-file blob 1a60379eb0f
fatal: Not a valid object name 1a60379eb0f

[root@localhost testeRepos]# git commit -m "alterei o f1" -a
Created commit c8de651: alterei o f1
 1 files changed, 1 insertions(+), 0 deletions(-)

[root@localhost testeRepos]# git-cat-file blob 1a60379eb0f
teste do f1
alteracao sobre o file f1
user: test a acrescentar coisas manualmente
change test
pb change
mais um teste
mais um teste
```

Merge

One of the most important features of git is the capability of merging 'branches'. Merge is used to bring the content of another branch into the current branch (even from a non-local repository). If the merge process refers to a different repository, it will fetch the remote branch and merge the result into the current branch, respectively. When the user combines a fetch and a merge, it is called a pull. Merging is an automatic process that applies the changes as a whole operation (atomic operation) after identifying all changes occurred in those branches. If there is a conflict it should be solved by a manual intervention, but git usually solves this conflict unless it is a huge one.

The place where the merge is done should be identified by using a checkout to change the HEAD reference. The branch is then merged on top of this head, and becomes a new head. The example below shows a sequence of commands to perform a 'branch merge': first i) change the head of the working directory, changing the branch in use, and secondly ii) merge another branch `old branch` to the current branch.

```
i)  [root@localhost testeRepos]# git checkout novo
ii) [root@localhost testeRepos]# git merge old
```

4.5 SVN

"This open-source project aims at producing a compelling replacement for the Concurrent Versions System (CVS)" [10]

The version control has been a critical tool for developers, who typically make changes over a set of files, and have to rollback those changes few days later. There are a lot of tools that intend to solve this problem, and one of the most commonly used is CVS (Concurrent Version System). In fact, this kind of problem is not just encountered by developers, but is a

[10] Linux Journal (http://www.linuxjournal.com/article/4768

shared problem when there are many people accessing the same set of data and changing it in one repository for its storage. In order to examine those changes, one may keep a historical of changes and all versions of files stored in the repository. It is no longer a problem that concerns developers only, but anyone that works on the same project from different places (this could be the case for a book, written by various authors, or any other kind of distributed work).

4.5.1 History

Subversion (also known as SVN) is an open source version control system. It allows a multiuser management of the files and directories stored in a repository.
CollabNet started this project called Subversion to fix the bugs of CVS (Control Version System) and misfeatures. It started in 2000, and is now maintained independently by the open source community and is used by several open source code projects. SVN started hosting itself in 2001.
SVN is used in several projects, like Google Code, and is recognized as the sole leader in the Standalone Software Configuration Management (SCM).

4.5.2 Objective

The objective of this technology [11] is to manage versions of files that are used by several persons in different places and at different times. Anyone can access and modify files at the same time, so that the changes made by one person should not be lost if there is another person changing the same file at the same time. This system may control the different versions of files, the history of the files and keep them consistent.

SVN original design team did not intend to make a new version control methodology, they only meant to fix CVS problems. The decision was to match the features of CVS, keep the same development model, and avoid known flaws of CVS. They did not want to change CVS processes, because they wanted people who have been using CVS to be able to migrate their information to SVN with little effort. There are thus a lot of similarities with CVS.

4.5.3 Features

SVN uses a centralized repository where all versions of files uploaded are stored, together with a history of those files and their modifications.

- Directory versioning

  Subversion implements a ”virtual” versioned filesystem that tracks changes that have been made to the whole directory over time. Files and directories are versioned. [22]

- True version history

[11] `http://svnbook.red-bean.com/en/1.0/svn-book.html`

”Since CVS is limited to file versioning, operations such as copies and renaming which might happen to files, but which are really modifications of the content of some directory are not supported in CVS. Additionally, in CVS it is not possible to replace a versioned file with some new file with the same name without the new item inheriting the history of the old file that may be completely unrelated. With SVN, you can add, delete, copy, and rename both files and directories, and every newly added file is created with a fresh, clean history of its own”. [22]

- Atomic commits

  ”A collection of modifications either goes completely into the repository, or not at all. This allows developers to construct and commit changes as logical chunks, and prevents problems that can occur when only a portion of a set of changes is successfully sent to the repository”. [22]

- Versioned metadata

  ”Each file and directory has a set of property keys and the values associated to it. You can create and store any arbitrary key/value pairs you wish. Properties are versioned over time, just like file contents”. [22]

- Choice of network layers

  Subversion has an abstract notion of repository access, making it easy for people to implement new network mechanisms. Subversion can plug into the Apache HTTP Server as an extension module. This gives SVN a big advantage in stability and interoperability, and instant access to existing features provided by that server authentication, authorization, wire compression, and so on. A more lightweight, standalone Subversion server process is also available. This server uses a custom protocol which can be easily tunneled over SSH.” [22]

- Consistent data handling

  ”Subversion expresses file differences using a binary differencing algorithm, which works identically on both text (human-readable) and binary (human-unreadable) files. Both kinds of files are stored compressed in the repository, and differences are transmitted in both directions across the network”. [22].

- Efficient branching and tagging

  ”The costs of branching and tagging need not be proportional to the project size. Subversion creates branches and tags by simply copying the project, using a mechanism similar to a hard-link. Thus these operations take only a very small, constant amount of time.” [22]

- Hackability

  ”Subversion has no historical baggage; it is implemented as a collection of shared C libraries with well-defined APIs. This makes Subversion extremely maintainable and usable by other applications and languages.” [22]

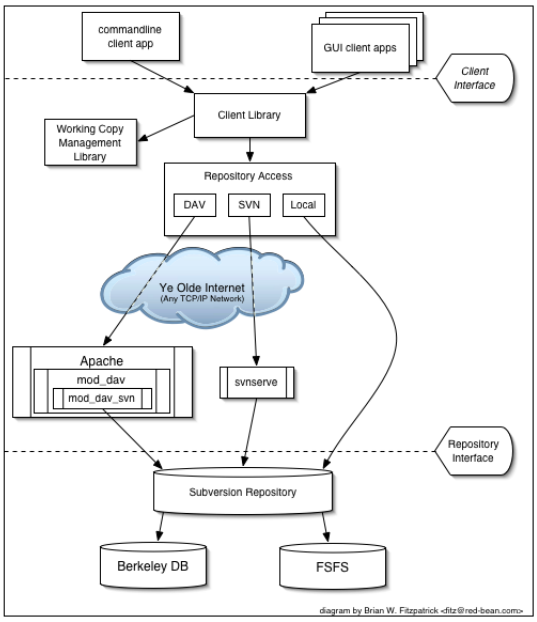

Figure 5: SVN Architecture [22]

4.5.4 Architecture and Components

4.5.5 Repository

Subversion has a central repository, that can be created in a local directory, where it can process to the version control of files and directories. Users can have their working copies, by checking out the content of the repository and their work to that central repository. The command to create a central repository in a local machine, is the following: `svnadmin create /usr/local/pbRepos`, and the structure of that repository will be created.

Creation of a Central Repository

```
[root@localhost repos]# svnadmin create /usr/local/pbRepos
```

Structure of the Central Repository

```
[root@localhost repos]# ll /usr/local/pbRepos/
total 28
drwxr-xr-x 2 root root 4096 2008-07-21 17:33 conf/
drwxr-xr-x 2 root root 4096 2008-07-21 17:33 dav/
drwxr-sr-x 5 root root 4096 2008-07-21 17:33 db/
-r--r--r-- 1 root root    2 2008-07-21 17:33 format
drwxr-xr-x 2 root root 4096 2008-07-21 17:33 hooks/
drwxr-xr-x 2 root root 4096 2008-07-21 17:33 locks/
-rw-r--r-- 1 root root  229 2008-07-21 17:33 README.txt
```

Information of the Central Repository

```
[pbarata@localhost pbRepos]# svn info
Caminho: .
URL: file:///usr/local/pbRepos
Raiz do Repositorio: file:///usr/local/pbRepos
UUID do repositorio: 8bcbbea2-852e-4e87-8778-8951acdf88b8
Revisao: 2
Tipo de No: diretorio
Agendado: normal
Autor da Ultima Mudanca: pb
Revisao da Ultima Mudanca: 2
Data da Ultima Mudanca: 2008-07-01 16:06:13 +0100 (Ter, 01 Jul 2008)
```

Creation of a Working Copy

```
[root@localhost repos]# svn co file:///usr/local/pbRepos/
Gerado copia de trabalho para revisao 0.

[root@localhost repos]# ll
total 4
drwxr-xr-x 3 root root 4096 2008-07-21 17:33 pbRepos/

[root@localhost repos]# ll pbRepos/ -A
total 4
drwxr-xr-x 6 root root 4096 2008-07-21 17:33 .svn/
```

Server repository places holding versions data

- /usr/local/pbRepos/db/revprops/0
- /usr/local/pbRepos/db/revprops/1
- /usr/local/pbRepos/db/revprops/2

Server repository places holding contents of files and changes made

- /usr/local/pbRepos/db/revs/0
- /usr/local/pbRepos/db/revs/1
- /usr/local/pbRepos/db/revs/2

4.5.6 Working copies

The working copy [12] is the structure created by SVN when the user executes the following:
`svn checkout file:///local/repository` (for a local repository)
or
`svn checkout http://host/repository` (for a remote repository).
This structure builds the version control of the working copy in the user machine. In SVN every directory under the repository is managed by the svn tools and has one `.svn` subdirectory with the information needed for its management.

- entries file

  This is the most important file in the `.svn` directory. It is an XML document that contains most of the administrative information about a versioned resource in a working copy directory.

[12] `http://svnbook.red-bean.com/en/1.1/ch08s03.html`

4.5.7 Creating branches

SVN Copy
Creating branches is very simple, it can be achieved using the `svn copy` command. In the following example we copy the `my_trunk` directory as a new branch called `my_repos/my_merge_branch`. After that, the new branch needs to be committed as usual.

```
[pbarata@localhost pbRepos]# svn copy my_trunk my_repos/my_merge_branch
[pbarata@localhost pbRepos]# svn status
A + my_repos/my_merge_branch
```

The `svn copy` command will copy recursively `my_trunk` working directory to a new working directory, `my_repos/my_merge_branch`. The `svn status` command shows that there is a new directory scheduled for addition into the repository. The "+" after the letter A, indicates that this is a copy rather than some new item.

SVN Checkout
A branch can also be created by checking out an entire repository - or just a part of it - by executing a `svn checkout` to bring the structure and data of a repository into a local repository.

```
[pbarata@localhost pbRepos]# svn checkout file:///usr/local/pbRepos
A    pbRepos/teste1
A    pbRepos/teste1/f1
A    pbRepos/teste1/f2
A    pbRepos/teste1/f3
A    pbRepos/teste1/f333
Gerado copia de trabalho para revisao 10.
```

This branch was created in the current directory of the current repository.

4.5.8 SVN problems

A well known problem occurs in SVN when it comes to renaming a file or a directory. SVN does not rename the file/directory, but makes a `copy` and a `delete` of that file/directory. The name changes but all the historical information remains the same, which means that the older versions in the tree remain the same, so that the references from the new file (renamed file) to the old file (original file) will be lost.

The fact that SVN stores a lot of data on the local machine, can cause a problem if the users are working on huge projects, or for developers that are usually working on multiple branches simultaneously. The `.svn` directories stored on local machines can be corrupted and confuse the information.

4.5.9 Scenario (Commit - Conflict)

One of the most common problem occurs when SVN users use a copy that is not up-to-date, and try to commit their own changes to the repository. This can generate a conflict in the

repository and users receive a notification of that conflict. The example below shows how it happens and how to solve this problem.

There are two users (pb, test) using the same repository. One user (pb) changes a file `f1` in his working copy and commits it to the repository, no problem occurs then. The second user (test), who has the same version of the file in his working copy, makes some changes too, and tries to commit his work to the repository. An error occurrs, and he makes an `svn update` that generates several files (`f1 f1.mine f1.r4 f1.r5`), shown below in `After Update`. To solve this problem, the second user should check what changes he must make and make them manually. After that he can commit changes made in the working copy by using the `svn resolved f1` command. This commit will not encounter any problems and the right version will be in the repository. Now the other user must run `svn update` to keep his working copy updated.

```
User: pb
[pbarata@localhost pbRepos]$sudo svn commit ../teste1/ -m "Commit test pb"
Enviando       teste1/f1
Transmitindo dados do arquivo .
Commit da revisao 5.
```

Below is the result, the content, of the committed file.

```
[pbarata@localhost pbRepos]$ cat teste1/f1
teste do f1
alteracao sobre o file f1
user: test a acrescentar coisas manualmente
pb change
```

Below is shown the conflict of the user 'test', as explained above.

```
User: test

[pbarata@localhost testRepos]$ sudo svn commit teste1/ -m "Commit test"
Enviando       teste1/f1
svn: Commit falhou (detalhes a seguir):
svn: Desatualizado: '/teste1/f1' na transacao '5-1'

[pbarata@localhost testRepos]$ cat teste1/f1
teste do f1
alteracao sobre o file f1
user: test a acrescentar coisas manualmente
change test
```

The following happens after Update

```
[pbarata@localhost teste1]$ svn update
C    f1
Atualizado para revisao 5.

[pbarata@localhost teste1]$ cat f1
teste do f1
alteracao sobre o file f1
```

```
user: test a acrescentar coisas manualmente
<<<<<<< .mine
change test
=======
pb change
>>>>>>> .r5

[pbarata@localhost teste1]$ ls -A
f1  f1.mine  f1.r4  f1.r5
```

The content of the 'f1' files that was created by the conflict, in order to be solved by the user, is shown below.

```
[pbarata@localhost teste1]$ cat f1.mine
teste do f1
alteracao sobre o file f1
user: test a acrescentar coisas manualmente
change test

[pbarata@localhost teste1]$ cat f1.r4
teste do f1
alteracao sobre o file f1
user: test a acrescentar coisas manualmente

[pbarata@localhost teste1]$ cat f1.r5
teste do f1
alteracao sobre o file f1
user: test a acrescentar coisas manualmente
pb change
```

This output is the result of the difference between the 'f1' file in the repository and the 'f1' file in the working copy.

```
Index: f1
===================================================================
--- f1  (revisao 5)
+++ f1  (copia de trabalho)
@@ -1,4 +1,8 @@
 teste do f1
 alteracao sobre o file f1
 user: test a acrescentar coisas manualmente
+<<<<<<< .mine
+change test
+=======
 pb change
+>>>>>>> .r5
```

The following must be done to commit the file with the solved conflict:

```
[pbarata@localhost teste1]$ svn resolved f1
```

The commit is now done and the right version of the file is in the repository.

5 State of the art of rollback tools

5.1 NexentaOS Free and Open Source Operating System over OpenSolaris kernel

Figure 6: `Nexenta desktop`

Nexenta is a combination of OpenSolaris, the GNU utilities, and Ubuntu. It is a free and open source operating system made over OpenSolaris kernel using GNU applications. Nexenta Operating System can run on several different hardware platforms such as Intel/AMD 32/64bit and can be distributed as a single installable CD. Upgrades can be installed from a NexentaOS CD using Advanced Packaging Tool, and other components can be downloaded from Debian/GNU Linux and Ubuntu Linux network repositories available [19, 51].

NexentaOS is based on a Nexenta Core Platform 1.0 (NexentaCore or NexentaCP), which is now available. NexentaCore includes complete OpenSolaris kernel and runtime, along with Debian and GNU tool chains. NexentaCore will serve as a stable platform for future Nexenta development. It can be downloaded as an ISO image and burnt onto a CD [19].

NexentaCore is a minimal (core) foundation that can be used to quickly build servers, desktops, and custom distributions tailored for specialized applications. Similar to NexentOS desktop distribution, NexentaCore combines reliable state-of-the-art kernel with the GNU userland, and the ability to integrate open source components in no time. However, unlike NexentaOS desktop distribution, NexentaCore does not aim at providing a complete desktop. The overriding objective of NexentaCore is - stable foundation [19].

Differences between NexentaCore and others OpenSolaris distributions [19]:

- Debian environment provided

- Ubuntu APT software repository provided
- Distribution Builder - platform for derivatives - provided
- Software optimized for server usage
- Simplified software upgrades via Debian tools
- Community-built, strong non-Sun centric community support
- Wider range of HW support in some areas (e.g. Networking)
- ZFS-bootable with transactional upgrades integrated
- Small memory requirements - 256MB

5.1.1 OpenSolaris kernel

OpenSolaris Core has enterprise hardware support, rock solid stability, binary compatibility, and technologies such as ZFS, DTrace and OpenSolaris Containers [50].

ZFS

The `ZFS filesystem` inserts a lot of features that have the capabilities of the filesystems becoming administrative control points. This will allow: snapshots, compression, backups, privileges, etc. The possibility of doing snapshots of the system and doing rollback by accessing those checkpoints are the functionality used by the `apt-clone` (see next). These snapshots are read-only copies of the filesystem at a specific point in time created in an unlimited number. They are accessible through .zfs/snapshot in `root` of each filesystem and allow users to recover files without sysadmin intervention.

5.1.2 `Apt-clone`

Apt-clone uses capabilities of ZFS commands and `apt-get`, with its own features. It uses snapshot's capability to manage the system and creates one checkpoint every time a user installs a package. Once the installation done and working (live), one can proceed to the rollback. A reboot is necessary in case the system requires it, unless a safe `-s option` is explicitly specified. Apt-clone manages Grub menu and ZFS system pool filesystems. It saves upgrades, using the cloning method of ZFS, of a current and active filesystem, and performs the actual upgrade by changing into chroot later. One checkpoint can have two status: activate or current. Activating a checkpoint is an option flag to be the default line when Grub menu starts. Current flag is that one which was selected to start the system. The file used to fetch packages is `/etc/apt/sources.list`, as used for the `apt-get`.

5.1.3 Tests of NexentaOS rollback capabilities

The main tool used in this section is `apt-clone`. It is used to make snapshots of the system and create checkpoints, while the installation of packages is in progress. Once the instruction `apt-clone install emacs` executed, it will do the following actions:

- Ask whether we want to upgrade the system using ZFS capabilities;
- Update the APT sources;
- Download upgrades and check whether reboot is required;
- Check whether there is available space for installation in the system;
- Create checkpoint

These steps show how `apt-clone` works and how the system reacts to rollback for a certain checkpoint. The set of tests were those that are in Nexenta site, and new ones made by our team.
Outputs will show the results of packages installation and rollback the created checkpoints.

Step 0: `Installation with apt-clone`
There is a configuration that must be performed on `/etc/apt/sources.list` file to access `Debian/GNU Linux` and `Ubuntu Linux` network repositories.
The content of `/etc/apt/sources.list` file should be, at minimum:

- `deb-src http://archive.ubuntu.com/ubuntu dapper main`
- `deb-src http://archive.ubuntu.com/ubuntu dapper-updates main`
- `deb-src http://security.ubuntu.com/ubuntu dapper-security main`

This content can be extended to:

- `deb-src http://archive.ubuntu.com/ubuntu dapper main universe multiverse`
- `deb-src http://archive.ubuntu.com/ubuntu dapper-updates main universe multiverse`
- `deb-src http://security.ubuntu.com/ubuntu dapper-security main restricted`

Furthermore, we can add other lines to search other repositories that could have more packages needed to the system.

How to install packages
`apt-clone`, an implementation of the features of ZFS command, is used for package installation.

```
apt-clone install joe
```

This command will check repositories defined in `/etc/apt/sources.list` to start getting packages for installation.

Step 1: `List of ZFS checkpoints of the system`
This is the initial state of NexentaOS. As shown below, there is only one initial checkpoint.

```
root@myhost:~# zfs list
NAME                            USED  AVAIL  REFER  MOUNTPOINT
syspool                         806M  6.53G    23K  none
syspool/rootfs-nmu-000          806M  6.53G   796M  legacy
syspool/rootfs-nmu-000@initial  9.90M      -   794M  -
root@myhost:~#
```

Step 2: `System Upgrade`

When the user tries to upgrade the system, there is only one package to install, and it has been installed. Note, that it was not necessary to reboot the system, and it created a checkpoint to a possible rollback.

```
root@myhost:~# apt-clone dist-upgrade
This operation will upgrade your system using ZFS capabilities.
Proceed ? (y/n) y

Updating APT sources ...
Downloading upgrades and checking if reboot will be required.
This may take a few minutes. Please wait...
Verifying free space...
Success. Upgrade requires 0.12MB of available free space.
Upgrade is in progress. Please DO NOT interrupt...
Creating Rollback Checkpoint...

Rollback Checkpoint has been created: rootfs-nmu-001

Use 'zfs list -r syspool' command to list all available
upgrade/rollback checkpoints

(Reading database ... 26629 files and directories currently installed.)
Preparing to replace dput 0.9.2.28nexenta2
        (using ...dput\_0.9.2.28nexenta3\_all.deb) ...
Unpacking replacement dput ...
Setting up dput (0.9.2.28nexenta3) ...
(Reading database ... 26629 files and directories currently installed.)
Installing new version of config file /etc/dput.cf ...
```

Step 3: `Trying to upgrade again`

This is just a test to check if the system overwrites the previous upgrade. The following shows that the system detects that it is actually upgraded and does nothing as a result ('no upgrades found.').

```
root@myhost:~#
root@myhost:~# apt-clone dist-upgrade
This operation will upgrade your system using ZFS capabilities.
Proceed ? (y/n) y

Updating APT sources ...
Downloading upgrades and checking if reboot will be required.
This may take a few minutes. Please wait...
Verifying free space...
Success. Upgrade requires 0.00MB of available free space.
No upgrades found.
```

Step 4: `List of ZFS checkpoints of the system again, after dist-upgrade`

This step shows the list of checkpoints available after upgrading the system.

```
root@myhost:~# zfs list
```

```
NAME                              USED  AVAIL  REFER  MOUNTPOINT
syspool                           816M  6.52G  23.5K  legacy
syspool/rootfs-nmu-000            815M  6.52G   801M  legacy
syspool/rootfs-nmu-000@initial   10.3M      -   794M  -
syspool/rootfs-nmu-000@nmu-001   1.60M      -   801M  -
syspool/rootfs-nmu-001           36.5M  6.52G   801M  legacy
```

Step 5: `List again checkpoints`

```
root@myhost:~# zfs list
NAME                              USED  AVAIL  REFER  MOUNTPOINT
syspool                           828M  6.51G  23.5K  legacy
syspool/rootfs-nmu-000            827M  6.51G   810M  legacy
syspool/rootfs-nmu-000@initial   10.3M      -   794M  -
syspool/rootfs-nmu-000@nmu-001   1.61M      -   801M  -
syspool/rootfs-nmu-000@nmu-002   1.89M      -   804M  -
syspool/rootfs-nmu-001           36.5M  6.51G   801M  legacy
syspool/rootfs-nmu-002           74.5M  6.51G   804M  legacy
```

Step 6: `apt-clone -l`

```
root@myhost:~# apt-clone -l
A C BOOTFS         TITLE
    rootfs-nmu-001 Rollback Checkpoint [nmu-001 : Mai 06 22:13:59 2008]
o o rootfs-nmu-000 Nexenta Core Platform "Elatte" [initial]
    rootfs-nmu-002 Rollback Checkpoint [nmu-002 : Mai 06 22:19:25 2008]
```

This is the state of the system after some new installations and a few rollbacks. In the meantime, VIM and EMACS have been installed.

There are two kinds of structure:

`syspool/rootfs-nmu-005` - the checkpoint itself

`syspool/rootfs-nmu-005@nmu-005` - changes between previous checkpoint that have not been rollback yet, and should be saved (there is not much information about it).

```
root@myhost:~# zfs list
NAME                              USED  AVAIL  REFER  MOUNTPOINT
syspool                          3.32G  4.00G  24.5K  legacy
syspool/rootfs-nmu-000            237M  4.00G   813M  legacy
syspool/rootfs-nmu-001           67.6M  4.00G  2.57G  legacy
syspool/rootfs-nmu-002           1.02G  4.00G   807M  legacy
syspool/rootfs-nmu-002@initial   10.3M      -   794M  -
syspool/rootfs-nmu-002@nmu-001   1.61M      -   801M  -
syspool/rootfs-nmu-002@nmu-002    182K      -   804M  -
syspool/rootfs-nmu-003           36.5K  4.00G  1.22G  legacy
syspool/rootfs-nmu-004           71.5K  4.00G  2.51G  legacy
syspool/rootfs-nmu-005           2.00G  4.00G  2.53G  legacy
syspool/rootfs-nmu-005@nmu-003   11.3M      -  1.22G  -
syspool/rootfs-nmu-005@nmu-004   8.49M      -  2.51G  -
syspool/rootfs-nmu-005@nmu-005    296K      -  2.53G  -
```

Step 7: `Show Bootfs: active and current`

This list made by apt-clone shows the checkpoint that was used for the startup of the system (`C - current`), and the checkpoint that is active (`A - active`). The current checkpoint means that the system has made a rollback to that checkpoint and all the things made after its creation have been destroyed, and the system works normally as though nothing had happened. The active checkpoint is the one that will be activated as a default in GRUB menu at the start of the system. In the example below, the system started with the GRUB default and therefore C is equal to A, but C (current) and A (active) could become different between the time when one activates a snapshot and the bootstrap of the system with that snapshot.

```
root@myhost:~# apt-clone -l
A C BOOTFS         TITLE
    rootfs-nmu-001 Upgrade Checkpoint    [nmu-001 : May  6 22:13:59 2008]
    rootfs-nmu-003 Rollback Checkpoint   [nmu-003 : May  7 19:39:06 2008]
o o rootfs-nmu-005 Nexenta Core Platform [nmu-005 : May 13 02:03:56 2008]
    rootfs-nmu-004 Rollback Checkpoint   [nmu-004 : May 13 01:59:55 2008]
    rootfs-nmu-002 Upgrade Checkpoint    [nmu-002 : May  6 22:19:25 2008]
```

Installation of JOE text editor.

```
root@myhost:~# apt-clone install joe
This operation will upgrade your system using ZFS capabilities.
Proceed ? (y/n) y

Updating APT sources ...
Downloading upgrades and checking if reboot will be required.
This may take a few minutes. Please wait...
Verifying free space...
Success. Upgrade requires 1.32MB of available free space.
Upgrade is in progress. Please DO NOT interrupt...
Creating Rollback Checkpoint...

Rollback Checkpoint has been created: rootfs-nmu-006

Use 'zfs list -r syspool' command to list all available
upgrade/rollback checkpoints

Selecting previously deselected package joe.
(Reading database ... 69804 files and directories
     currently installed.)

Unpacking joe (from .../joe_3.5-1.1_solaris-i386.deb) ...
Setting up joe (3.5-1.1) ...
```

JOE packages are in dpkg repository.

```
root@myhost:~# dpkg -l | grep joe
ii  joe     3.5-1.1   user friendly full screen text editor
```

After the installation of JOE, the packages will be copied to /var/cache/apt/archives.

```
root@myhost:~# ll /var/cache/apt/archives/ | grep joe
-rw-r--r-- 1 root root    359226 Jun 26  2007
                          joe_3.5-1.1_solaris-i386.deb
```

Below is the list of checkpoints after the installation of JOE.

```
syspool/rootfs-nmu-005@nmu-006  5.05M      -  2.53G  -
syspool/rootfs-nmu-006          75.5K  3.99G  2.53G  legacy

root@myhost:~# zfs list
NAME                             USED  AVAIL  REFER  MOUNTPOINT
syspool                         3.34G  3.99G  24.5K  legacy
syspool/rootfs-nmu-000           237M  3.99G   813M  legacy
syspool/rootfs-nmu-001          67.6M  3.99G  2.57G  legacy
syspool/rootfs-nmu-002          1.02G  3.99G   807M  legacy
syspool/rootfs-nmu-002@initial  10.3M      -   794M  -
syspool/rootfs-nmu-002@nmu-001  1.61M      -   801M  -
syspool/rootfs-nmu-002@nmu-002   182K      -   804M  -
syspool/rootfs-nmu-003          36.5K  3.99G  1.22G  legacy
syspool/rootfs-nmu-004          71.5K  3.99G  2.51G  legacy
syspool/rootfs-nmu-005          2.02G  3.99G  2.53G  legacy
syspool/rootfs-nmu-005@nmu-003  11.3M      -  1.22G  -
syspool/rootfs-nmu-005@nmu-004  8.49M      -  2.51G  -
syspool/rootfs-nmu-005@nmu-005   384K      -  2.53G  -
syspool/rootfs-nmu-005@nmu-006  5.05M      -  2.53G  -
syspool/rootfs-nmu-006          75.5K  3.99G  2.53G  legacy
```

5.1.4 Remove checkpoints

Now, the user wants to remove checkpoint (syspool/rootfs-nmu-001) and see what happens. Removing checkpoint is a way of getting more space on the system, and has no impact on the filesystem.

5.2 etckeeper

etckeeper[13] is a set of tools developed to keep the `/etc` directory under version control. Even though the tools are more generic, i.e. they can be used to keep under version control any file system directory, they have been designed with `/etc` in mind to handle specificities of the directory mandated to store system-wide configurations.

Besides plain version control, etckeeper keeps track of file permissions which are essential to properly handle /etc. This avoids risks like checking out `/etc/shadow` as being world readable instead of more constrained with appropriate file permissions. Such a feature is rarely handled properly by plain version control systems and not in its complete extension, for example with Subversion the executable bit is handled properly, but other file permissions are not.

Moreover, etckeeper is to some extent independent from the underlying version control systems. At the time of writing it supports Git (which is also the default version control system; see Section 4.4 for more information on Git), mercurial, and bzr.

etckeeper not only offers initialization and storage of meta information not handled by the underlying version control system, but also stays out of the way and lets the system administrator

[13] `http://joey.kitenet.net/code/etckeeper/`

play with `/etc` using the legacy interface of the used version control system. For example, rolling back to previous states. This means that the commands to be used are the ones used in version control system, and no additional knowledge is required.

The final ingredient offered by etckeeper is integration with the package manager used by the distribution. By default etckeeper hooks into Debian's apt, but support for other package managers is available as well. The provided hooks take care of distinguishing the changes introduced by a given package upgrade (or installation, removal, ...) from those manually performed by the system administrator using the legacy commit interface of the underlying version control system. By default, hooks are provided to automatically commit left-over (i.e. non committed) changes before a given run of the package manager, and to automatically commit changes accumulated during package upgrades at the end of the upgrade run. This way the history always shows with distinguished commits the changes introduced by an upgrade run and, depending on the capabilities of the version control system, the system administrator is free to selectively rollback such changes. This can of course be either right after an upgrade run or arbitrarily later in time (provided possible conflicts are solved of course).

etckeeper is actively developed and often released as tags in a public accessible git repository.[14] It is also packaged in Debian. Getting started with etckeeper using the default configuration (that is: using Git as a version control system and hooking into apt) is as easy to execute as root:

```
apt-get install etckeeper
etckeeper init
cd /etc
git commit -m 'initial checkin'
```

5.3 Conary - Package Management System

Conary is a distributed software management system developed by rPath. Conary is a package manager that has the same objective as dpkg and rpm. It can install or remove software on/from a computer, searching the versions of installed software, and find some package installation available for the system. Conary, after downloading and first installing a package, updates faster than other tools because it only requires downloads to update a file and does not require the full binary [57].

Conary is the tool used to update the system as it was after its default installation. There are two ways of doing it, 1) one is using the graphical user interface - PackageKit, and 2) the other is by command line using conary commands (conary updateall).

Conary is more than a Package Management System, it can also be a package creator, a repository of software, and a versioning tool.

It has the capability of doing rollback. Everytime a user installs a package that creates a rollback point, the tool takes a snapshot.

5.3.1 Linux distribution's using Conary

There are two Linux distributions using Conary: rPath and Foresight.

[14] `http://git.kitenet.net/?p=etckeeper.git`

- rPath is a Linux Distribution, that uses conary capabilities
- Foresight is a Linux Distribution based on rPath, that uses conary capabilities

5.3.2 Conary system-based

A Linux Distribution based on rPath Linux is a Conary-based system. Conary was developed by the rPath community and is implemented as the base system of managing packages of the rPath Linux. Foresight Linux is based on rPath Linux, it is thus a conary system-based. It uses all the functionalities of conary to manage packages.

5.3.3 Installation and rollback with conary

This amount of tests were made over a Virtual Machine using the Foresigth Linux Distibution.

Before starting the installation process with conary, it is made of a list of packages installed originally by the system. One thousand packages are installed in the example.

```
[pbarata@pc-00150 ~]$ conary q | wc
   1000    1000   24985
```

`conary update` is the command to update/install packages.

Install Emacs

A lot of execution messages could not be caught, because they overwrite each other's outputs.

```
[pbarata@pc-00150 ~]$ sudo conary update emacs
Resolving dependencies...The following updates will be performed:
  Install emacs(:data :doc :elisp :runtime :supdoc)=23.0.0cvs20080306-0.1-1
continue with update? [Y/n] y
Applying update job:
  Install emacs(:data :doc :elisp :runtime :supdoc)=23.0.0cvs20080306-0.1-1
```

List of all rollbacks (just one) at this moment, after installing 'emacs'.

```
[pbarata@pc-00150 ~]$ conary rblist
r.0:
    installed: emacs(:data :doc :elisp :runtime :supdoc)
 foresight.rpath.org@fl:2/23.0.0cvs20080306-0.1-1
```

Now this shows the number of packages installed (1001) and with `grep emacs` a list of emacs packages installed. There were 1000 before the installation and 1001 after the installation.

```
[pbarata@pc-00150 ~]$ conary q | wc
   1001    1001   25015

[pbarata@pc-00150 ~]$ conary q | grep emacs
emacs=23.0.0cvs20080306-0.1-1
```

Installation of Joe (text editor)

```
[pbarata@pc-00150 ~]$ sudo conary update joe
Resolving dependencies...The following updates will be performed:
    Install joe(:config :doc :runtime :supdoc)=3.5-1-0.1
continue with update? [Y/n] y
Applying update job:
    Install joe(:config :doc :runtime :supdoc)=3.5-1-0.1
```

The list of rollbacks appears once JOE is installed.

```
[pbarata@pc-00150 ~]$ sudo conary rblist
r.1:
    installed: joe(:config :doc :runtime :supdoc) \
      conary.rpath.com@rpl:2-qa/3.5-1-0.1

r.0:
    installed: emacs(:data :doc :elisp :runtime :supdoc) \
  foresight.rpath.org@fl:2/23.0.0cvs20080306-0.1-1
```

Rollback

After installing these two packages, a list of rollback possibilities is created. Once the rollback process executed, all packages installed after this point will be removed. In this case, rollback `r.0` point will remove all the installations made after this point by jumping the system to that point. It means that the system will be as it was before installing `joe`.

```
[pbarata@pc-00150 ~]$ sudo conary rollback r.0 --verbose
Applying update job 1 of 2:
  Erase   joe(:config :doc :runtime :supdoc)=3.5-1-0.1
Applying update job 2 of 2:
  Erase   emacs(:data :doc :elisp :runtime :supdoc)=23.0.0cvs20080306-0.1-1
```

As we can see, the number of packages installed is still the same as it was at the starting point.

```
[pbarata@pc-00150 ~]$ conary q | wc
   1000    1000   24985
```

The `conary updateall` command is made after this rollback. This command updates all the software installed on the system. The result of this update is an update of 93 points. There should have been a lot of rollback points, but the system did not take any, so when the user makes an `updateall` he cannot remove it. A short/truncated list is shown below.

```
[pbarata@pc-00150 ~]$ conary updateall
[...]
Applying update job 17 of 93:
    Update  gnome-speech(:data :lib :runtime) (0.4.18-0.1-1[~!builddocs]
 -> 0.4.19-0.1-1[~!builddocs])
    Update  gnome-volume-manager(:config :data :locale :runtime)
       (2.22.1-0.3-1[~!builddocs]
 -> 2.22.5-0.1-1[~!builddocs])
Applying update job 18 of 93:
    Update  group-drivers (2.0.1-0.2-1 -> 2.0.1-0.28-1)
    Update  group-pidgin (2.0.1-0.2-1 -> 2.0.1-0.28-1)
    Update  group-printing (2.0.1-0.2-1 -> 2.0.1-0.28-1)
```

```
    Update  group-xorg-fonts (2.0.1-0.2-1 -> 2.0.1-0.28-1)
[...]
Applying update job 92 of 93:
    Update  icedtea-jre(:doc :runtime) (1.6.0.0_b08_1.0+r782-0.1-1
 -> 1.6.0.0_b09_1.1+r853-0.2-1)
    Update  xulrunner(:config :runtime) (1.8.1.11-0.2-1 -> 1.9-0.13-1)
Applying update job 93 of 93:
    Update  group-desktop-common (2.0.1-0.2-1 -> 2.0.1-0.28-1)
    Update  group-gnome-dist (2.0.1-0.2-1 -> 2.0.1-0.28-1)
    Update  group-internet (2.0.1-0.2-1 -> 2.0.1-0.28-1)

[pbarata@pc-00150 ~]$ sudo conary rblist
```

`emacs` and `joe` were installed again. Then we tried to install `vim`, but it was already installed. The decision was then to remove it, with the `conary erase` command. As we can see, it was removed and it created a rollback point (`r.5`). Some checks were made to verify whether `vim` was realy removed/erased.

```
[pbarata@pc-00150 conary]$ sudo conary update vim
no new troves were found

[pbarata@pc-00150 conary]$ sudo conary erase vim
Resolving dependencies...The following updates will be performed:
   Erase   vim(:config :data :doc :locale :runtime)=7.1.213-1-0.1
continue with update? [Y/n] y
Applying update job:
   Erase   vim(:config :data :doc :locale :runtime)=7.1.213-1-0.1
[pbarata@pc-00150 conary]$ vim
bash: vim: command not found

[pbarata@pc-00150 conary]$ sudo conary rblist
r.5:
    erased: vim(:config :data :doc :locale :runtime) \
         conary.rpath.com@rpl:2-qa/7.1.213-1-0.1

r.4:
    installed: joe(:config :doc :runtime :supdoc) \
         conary.rpath.com@rpl:2-qa/3.5-1-0.1

r.3:
    installed: emacs(:data :doc :elisp :runtime :supdoc) \
     foresight.rpath.org@fl:2/23.0.0cvs20080306-0.1-1
```

Now it has been rolled back to point (`r.5`), which was an `erase`, so the process of rollback should be an installation. The following happened:

```
[pbarata@pc-00150 conary]$ sudo conary rollback r.5
Applying update job 1 of 2:
    Install vim(:config :data :doc :locale :runtime)=7.1.213-1-0.1
Applying update job 2 of 2:
    Update  vim(:config :data :doc :locale :runtime)
    (/conary.rpath.com@rpl:devel//2-qa/7.1.213-1-0.1  \
    -> /conary.rpath.com@rpl:devel//2-qa//  \
```

```
    local@local:ROLLBACK/7.1.213-1-0.1)

[pbarata@pc-00150 conary]$ sudo conary rblist
r.4:
    installed: joe(:config :doc :runtime :supdoc)  \
         conary.rpath.com@rpl:2-qa/3.5-1-0.1

r.3:
    installed: emacs(:data :doc :elisp :runtime :supdoc)
 foresight.rpath.org@fl:2/23.0.0cvs20080306-0.1-1
```

Another way to install packages is to use the name of the repository where this package is available. Let us install the `scribes` (text editor) package.

```
[pbarata@pc-00150 conary]$ sudo conary update \
       scribes=foresight.rpath.org@fl:1-contrib

The following dependencies could not be resolved:
    scribes:data=0.3.2.4-1-1:
    python: gtksourceview
    scribes:python=0.3.2.4-1-1:
    python: gtksourceview


[pbarata@pc-00150 conary]$ sudo conary update  \
       scribes=foresight.rpath.org@fl:1-contrib

The following dependencies could not be resolved:
    scribes:data=0.3.2.4-1-1:
    python: gtksourceview
    scribes:python=0.3.2.4-1-1:
    python: gtksourceview

[pbarata@pc-00150 conary]$ conary q | grep scribes

[pbarata@pc-00150 conary]$ sudo conary rblist
r.4:
    installed: joe(:config :doc :runtime :supdoc)  \
         conary.rpath.com@rpl:2-qa/3.5-1-0.1

r.3:
    installed: emacs(:data :doc :elisp :runtime :supdoc)  \
         foresight.rpath.org@fl:2/23.0.0cvs20080306-0.1-1

[pbarata@pc-00150 conary]$ sudo conary update  \
       scribes=foresight.rpath.org@fl:desktop

The following dependencies could not be resolved:
    scribes:python=0.2.5-1-1:
    python: gtksourceview

[pbarata@pc-00150 conary]$ sudo conary update  \
       scribes=foresight.rpath.org@fl:1-devel
```

```
The following dependencies could not be resolved:
    scribes:python=0.2.9.86-3-1:
    python: gtksourceview
```

There were several conflicts in python packages that could not be resolved during the installation of `scribes` and the installation was not successful.

We then installed the `gconf-editor` successfully.

```
[pbarata@pc-00150 conary]$ sudo conary update gconf-editor
Resolving dependencies...The following updates will be performed:
    Install gconf-editor(:data :doc :locale :runtime  \
                         :supdoc)=2.22.0-0.1-1[~!builddocs]

continue with update? [Y/n] y
Applying update job:
    Install gconf-editor(:data :doc :locale :runtime  \
                         :supdoc)=2.22.0-0.1-1[~!builddocs]

[pbarata@pc-00150 conary]$ sudo conary rblist
r.5:
    installed: gconf-editor(:data :doc :locale :runtime :supdoc)  \
     foresight.rpath.org@fl:2/2.22.0-0.1-1

r.4:
    installed: joe(:config :doc :runtime :supdoc)  \
         conary.rpath.com@rpl:2-qa/3.5-1-0.1

r.3:
    installed: emacs(:data :doc :elisp :runtime :supdoc)  \
     foresight.rpath.org@fl:2/23.0.0cvs20080306-0.1-1
```

Below are some commands to help users find information about packages and files installation, repositories, etc.

```
[pbarata@pc-00150 conary]$ conary rq gconf-editor --all-versions
gconf-editor=2.20.0-0.2-1[~!builddocs]
gconf-editor=2.22.0-0.1-1[~!builddocs]
```

The `--info option`, shows the information on the package. The label identifies the repository from where the package was installed.

```
[pbarata@pc-00150 conary]$ conary q --info gconf-editor
Name      : gconf-editor       Build time: Tue Mar 11 14:01:44 2008
Version   : 2.22.0-0.1-1       Label     : foresight.rpath.org@fl:2
Size      : 1556962
Pinned    : False
Flavor    : ~!builddocs is: x86
Category  : GNOME
Category  : GTK
Category  : System
Summary   : Configuration Editor
Description:
    Directly edit your entire configuration database
```

The `--full-version option`, shows the information on the package, essentially from where the package was installed.

```
[pbarata@pc-00150 conary]$ conary q --full-versions gconf-editor
gconf-editor=/foresight.rpath.org@fl:devel//2/2.22.0-0.1-1[~!builddocs]
```

Below is a list of all the files and directories where the package is installed and configured.

```
[pbarata@pc-00150 conary]$ conary q --ls gconf-editor
/etc/gconf/schemas/gconf-editor.schemas
/usr/share/applications/gconf-editor.desktop
/usr/share/icons/hicolor/48x48/apps/gconf-editor.png
/usr/share/omf/gconf-editor/gconf-editor-C.omf

[...]  (Output truncated)

/usr/share/doc/gconf-editor-2.22.0/COPYING
/usr/share/doc/gconf-editor-2.22.0/ChangeLog
/usr/share/doc/gconf-editor-2.22.0/INSTALL
/usr/share/doc/gconf-editor-2.22.0/NEWS
/usr/share/doc/gconf-editor-2.22.0/README
/usr/share/doc/gconf-editor-2.22.0/docs/ChangeLog
/usr/share/doc/gconf-editor-2.22.0/po/ChangeLog
```

The `--path` option is used to determine what package is managed by a specific file.

```
[pbarata@pc-00150 conary]$ conary q --path /usr/bin/gconf-editor
gconf-editor:runtime=2.22.0-0.1-1[~!builddocs]
```

When the name of the package has a colon and another name, such as `gconf-editor:data` and `gconf-editor:doc`, this refers to a component. When conary builds the package, it separates files into different components. These belong to the same file, but represent a logical group of files within the package. This gives flexibility for other packages to solve dependencies problems making components instead of entire packages. Thanks to this users can remove components that they do not need and get more space without removing entire packages.

The `--troves` option, lists all the components that are installed as a part of the package. A `trove` is a part of the package that can be removed or installed and is a unit of system base of Conary or a repository. Troves includes packages and their individual components. The example below shows the troves installed for `gconf-editor` package.

```
[pbarata@pc-00150 conary]$ conary q --troves gconf-editor
gconf-editor=2.22.0-0.1-1[~!builddocs]
  gconf-editor:data=2.22.0-0.1-1[~!builddocs]
  gconf-editor:doc=2.22.0-0.1-1[~!builddocs]
  gconf-editor:locale=2.22.0-0.1-1[~!builddocs]
  gconf-editor:runtime=2.22.0-0.1-1[~!builddocs]
  gconf-editor:supdoc=2.22.0-0.1-1[~!builddocs]

[pbarata@pc-00150 conary]$ conary q --troves emacs
emacs=23.0.0cvs20080306-0.1-1
  emacs:data=23.0.0cvs20080306-0.1-1
  emacs:doc=23.0.0cvs20080306-0.1-1
```

```
emacs:elisp=23.0.0cvs20080306-0.1-1
emacs:runtime=23.0.0cvs20080306-0.1-1
emacs:supdoc=23.0.0cvs20080306-0.1-1
```

One then needs to check the changes on packages version made over the emacs package since it was firstly installed. In this case, no change appears.

```
[pbarata@pc-00150 conary]$ conary verify emacs
Update  emacs (/foresight.rpath.org@fl:devel//2/23.0.0cvs20080306-0.1-1
 -> /foresight.rpath.org@fl:devel//2//local@local:LOCAL/23.0.0cvs20080306-0.1-1)
Update  emacs:data (/foresight.rpath.org@fl:devel//2/23.0.0cvs20080306-0.1-1
 -> /foresight.rpath.org@fl:devel//2//local@local:LOCAL/23.0.0cvs20080306-0.1-1)
Update  emacs:doc (/foresight.rpath.org@fl:devel//2/23.0.0cvs20080306-0.1-1
 -> /foresight.rpath.org@fl:devel//2//local@local:LOCAL/23.0.0cvs20080306-0.1-1)
Update  emacs:elisp (/foresight.rpath.org@fl:devel//2/23.0.0cvs20080306-0.1-1
 -> /foresight.rpath.org@fl:devel//2//local@local:LOCAL/23.0.0cvs20080306-0.1-1)
Update  emacs:runtime (/foresight.rpath.org@fl:devel//2/23.0.0cvs20080306-0.1-1
 -> /foresight.rpath.org@fl:devel//2//local@local:LOCAL/23.0.0cvs20080306-0.1-1)
Update  emacs:supdoc (/foresight.rpath.org@fl:devel//2/23.0.0cvs20080306-0.1-1
 -> /foresight.rpath.org@fl:devel//2//local@local:LOCAL/23.0.0cvs20080306-0.1-1)
```

Configuration file for packages repository

```
[pbarata@pc-00150 ~]$ cat /etc/conaryrc
installLabelPath foresight.rpath.org@fl:2
pinTroves kernel.*
includeConfigFile /etc/conary/config.d/*
```

Configuration file for packages repository

```
[pbarata@pc-00150 ~]$ cat /etc/conary/config.d/foresight
installLabelPath foresight.rpath.org@fl:2-kernel
     foresight.rpath.org@fl:2 conary.rpath.com@rpl:2-qa
interactive         True
autoResolve         True
#pinTroves      .*(\-kernel)$

[pbarata@pc-00150 ~]$ cat /etc/conary/config.d/kernel
pinTroves kernel.*
```

5.4 NixOS

NixOS: A Purely Functional Linux Distribution

Nix is a package management software used in NixOS. This package management system proposes a purely functional approach where packages never change once created and are built in a deterministic way through Nix expressions. Nix expressions is a simple functional language. The main focus is on solving broken dependencies by identifying packages uniquely under `/nix/store/hash-packagename-version`. The hash is made upon the package dependencies graph. A package with exactly the same files but with a modified dependency is a

different package stored under /nix/store/. Packages are not removed after the installation of a new version. Therefore, a package is never broken because its dependencies are never removed. Nevertheless, authors state that you can rollback an unsuccessful installation. After the rollback you have to run the garbage collector. The configuration files (/etc) are maintained as symbolic links to /nix/store. Still, it is not clear how the versions of manually edited files are preserved.
NixOs is a Linux distribution with no production intention, although it is a stable distribution of a linux based operating system.

5.4.1 History

NixOS started around 2004 with a PhD Thesis, as an experience to use a purely functional way, i.e. deterministic functions that keep their states after their building. It was developed by Eelco Dolstra at the university of Utrecth in the Netherlands with the intention to solve package dependencies problems.

5.4.2 Structure of repositories

Store information on nix/store
All packages are stored in a local repository, the /nix/store/hash-packagename-version, and all the actions are performed over this nix store. For more than one installation of an application the system creates directories, files and links with unique values that can be accessed individually. This is seen as a profile. With this technique NixOS can have more than one version of the same application as well as more than one user accessing and installing the same application.

DB with information of the packages installed and stored in the system
There is db information in the /nix/var/nix/db that matches the storage repository, but with more information such as the date and hour of package installation, and the links to those packages. It also stores the references to the packages.

Nix channel
A Nix channel is a way to save some URLs that point to some repositories, in order to get the Nix expressions, and the Manifest for such a package version. For example: nixpkgs-stable. This package is stored in a remote repository and the system needs to know where to get the packages it needs, it must then add a URL as a 'channel' to that repository. The command nix-channel --add http://nix.cs.uu.nl/dist/nix/channels-v3/nixpkgs-stable, will provide the access to that repository in order to download the files for installation.
After adding a URL to the channel, the command nix-channel --update is used to download the information for the local repository, where the nix expressions and the manifest is available, and is used mainly by the nix-env command for package installation. Manifest has the information about packages, the nix expressions and the sources.

```
[pbarata@nixos:~]$ nix-channel --update
```

```
obtaining list of Nix archives at \
‘http://nixos.org/releases/nixpkgs/channels/\
        nixpkgs-unstable/MANIFEST.bz2’...

3873 store paths in manifest
downloading Nix expressions from \
‘http://nixos.org/releases/nixpkgs/channels/\
        nixpkgs-unstable/nixexprs.tar.bz2’...

unpacking channel Nix expressions...
building path(s) ‘/nix/store/znqcmwjkxk5....hqmr09ahqp-channels’
unpacking channel nixpkgs-unstable

/nix/store/md2hjc11ks0nc093b7ar12xzslv1wrqp-gnutar-1.20/bin/tar: \
nixpkgs-0.12pre12803/VERSION: time stamp 2008-09-05 08:07:11 \
  is 80072.033519025 s in the future

/nix/store/md2hjc11ks0nc093b7ar12xzslv1wrqp-gnutar-1.20/bin/tar: \
nixpkgs-0.12pre12803/COPYING: time stamp 2008-09-05 08:07:11 \
  is 80072.001866821 s in the future

/nix/store/md2hjc11ks0nc093b7ar12xzslv1wrqp-gnutar-1.20/bin/tar: \
nixpkgs-0.12pre12803/relname: time stamp 2008-09-05 08:07:11 \
  is 80071.977534403 s in the future

[...]

/nix/store/md2hjc11ks0nc093b7ar12xzslv1wrqp-gnutar-1.20/bin/tar: \
nixpkgs-0.12pre12803/doc/outline.txt: time stamp 2008-09-05 08:07:11 \
  is 80045.509736249 s in the future

/nix/store/md2hjc11ks0nc093b7ar12xzslv1wrqp-gnutar-1.20/bin/tar: \
nixpkgs-0.12pre12803/doc: time stamp 2008-09-05 08:07:15 \
  is 80049.50089663 s in the future

/nix/store/md2hjc11ks0nc093b7ar12xzslv1wrqp-gnutar-1.20/bin/tar: \
nixpkgs-0.12pre12803: time stamp 2008-09-05 08:07:57 \
  is 80091.500555226 s in the future
```

5.4.3 Packages

Packages are made using Nix purely functional model, i.e. packages are built by functions, they have a well defined structure and they do not depend on any package outside the Nix Store.

The packages used in NixOS are built by Nix purely functional approach and once built they never change. The package building is supported by functions that just depend on the function arguments passed to them in order of generate an output. These functions use nix expressions to construct and describe the dependency graph of the package. The outputs are always the same, for the same input.

The file `default.nix` has the specification of the package, the arguments they need to access a function to perform all the actions to be made with the package, the scripts for installation using variables and the dependencies. They generate a file called derivation (`package.drv`) with the capability to install that package. The `nix-env` can execute this derivation, as shown later in this document, and install the package and all its dependencies. The `default.nix` is stored in the package cache for nix expression in the right directory, e.g. `pkgs/applications/emacs-22/` .

Structure of packages
Any package needs a Nix expression to be built. This expression describes a package, and is used to generate a derivation file with the packages needed to install that application. This derivation is a build script that has the environment variables and the dependencies between the components used by the application.

Nix expressions (`default.nix`) for `w3m-0.5.2` package
The format of the nix expression is: { `args` } : `body` . The `args` are the arguments needed for this package to build it. The `body` is the area where the execution is made and the construction happens.

```
{ stdenv, fetchurl, sslSupport ? true, graphicsSupport ? false,
  ncurses, openssl ? null, boehmgc, gettext, zlib, gdkpixbuf ? null } :

   assert sslSupport -> openssl != null;
   assert graphicsSupport -> gdkpixbuf != null;

   stdenv.mkDerivation {
  name = "w3m-0.5.2";
  builder = ./builder.sh;
  src = fetchurl {
  url = mirror://sourceforge/w3m/w3m-0.5.2.tar.gz;
  md5 = "ba06992d3207666ed1bf2dcf7c72bf58";
  };

   inherit openssl boehmgc;

   buildInputs = [
  ncurses boehmgc gettext zlib
  (if sslSupport then openssl else null)
  (if graphicsSupport then gdkpixbuf else null)
   ];
 }
```

The components used in the nix expression are used to generate the `.drv` file, shown below, and the function that will make this is called `mkDerivation`, using all the variables within the nix expression. This function belongs to the stdenv package, that gives the basic Unix environment tools.

This nix expression receives several arguments { `args` } :
{ `stdenv, fetchurl , sslSupport ? true, graphicsSupport ? false, ncurses, openssl ? null, boehmgc, gettext, zlib, gdkpixbuf ? null` }
that will not be used now but are necessary to build the package.

Other attributes such as `name, builder, src, buildInputs` are also needed to generate the package.

- `name` (is the name of the package)
- `builder` (is the shell script to install the package)
- `src` (is a tarball with the source)
- `buildInputs` (dependencies of this package)

Derivation File (`w3m-0.5.2.drv`)
Derivation is a file created by executing the default.nix of the `w3m-0.5.2` file, in this particular case, `stdenv.mkDerivation` using the nix expression with the environment variables (name, builder, src and buildInputs). This file is very important, because it builds the dependency between packages needed to make their applications work. It is the one which will perform the installation of the package over the system, and once in the nix store, it can be the following derivation:

```
Derive(
[
("out","/nix/store/zri8vnwsvgnr34jb2c5sb43ww63myp60-w3m-0.5.2","","")
],

[
("/nix/store/05kv8a8al38...6648vl3g-gettext-0.17.drv",    ["out"]),
("/nix/store/0k7x9sd4fsi...4xpahp7a-ncurses-5.6.drv",     ["out"]),
("/nix/store/2g82813q533...d0blsmq2-w3m-0.5.2.tar.gz.drv",["out"]),
("/nix/store/hinl1ka0wm1...gymkzs3c-boehm-gc-7.0.drv",    ["out"]),
("/nix/store/l86dzzvpi1j...ywlyggs9-openssl-0.9.8h.drv",  ["out"]),
("/nix/store/ny12shdnsfc...bl9bxf05-stdenv-linux.drv",    ["out"]),
("/nix/store/v77sqh7znk3...w2jyykda-bash-3.2-p39.drv",    ["out"]),
("/nix/store/vdx5ndfl89k...66mjsq50-zlib-1.2.3.drv",      ["out"])
],

["/nix/store/1i9c73i8ns35vglmfaxvsrh8kcrxb4wz-builder.sh"],

"i686-linux",

"/nix/store/vdzbaimmqgnzbv41jlz04h0pfwxi9vc6-bash-3.2-p39/bin/sh",

["-e","/nix/store/1i9c73i8ns35vglmfaxvsrh8kcrxb4wz-builder.sh"],

[
("boehmgc","/nix/store/jaqvg2incp...a8hx4p7qarlvkm7-boehm-gc-7.0"),

("buildInputs",
"/nix/store/c9pw6rdlfg4rgc5qygvj3bjlfnxg974j-ncurses-5.6
/nix/store/jaqvg2incp55dz0xaa8hx4p7qarlvkm7-boehm-gc-7.0
/nix/store/ylsv1ic73vs867scnh6yhrirysh6gkrq-gettext-0.17
/nix/store/rv9xzahcch1235w9fnzxkqdgwv6v0c7v-zlib-1.2.3
/nix/store/f2qv21j393bx57640a3hdda3lfrj2kvx-openssl-0.9.8h
"),
```

```
("builder","/nix/store/vdzbaimmqgnz...wxi9vc6-bash-3.2-p39/bin/sh"),

("name","w3m-0.5.2"),

("openssl","/nix/store/f2qv21j393bx...dda3lfrj2kvx-openssl-0.9.8h"),

("out","/nix/store/zri8vnwsvgnr34jb2c5sb43ww63myp60-w3m-0.5.2"),

("src","/nix/store/0m6a2qic31rxb8h0z1xnw8sh4nc1v9sm-w3m-0.5.2.tar.gz"),

("stdenv","/nix/store/5vfzs8lw12lf2rnmcv220i0gqwwg7qbc-stdenv-linux"),

("system","i686-linux")

]
)
```

Nixpkgs

This package format allows NixOS to install a package not stored in the local repository, but using information about the main components of a package, such as a download repository and the derivation file of the package, name of the package and architecture. This file is one line file with all the needed components for installation and has to be stored in the local machine, as it contains all the directives.

Inside the unzip-5.52-i686-linux.nixpkg

```
NIXPKG1 http://nixos.org/releases/nixpkgs\
        /nixpkgs-0.12pre12803-0qgw3s1x/MANIFEST \
        unzip-5.52 \
        i686-linux \
        /nix/store/xallrgyjhwmsmm2hxgb5f8zcaacdkhl6-unzip-5.52.drv \
        /nix/store/52y6jd9xbvr6w49k6vd7cf6kmgm2jn02-unzip-5.52
```

Execute the nix install command

```
[pbarata@nixos:~]$ nix-install-package --non-interactive \
                   unzip-5.52-i686-linux.nixpkg

Pulling manifests...
obtaining list of Nix archives at \
‘http://nixos.org/releases/nixpkgs/\
        nixpkgs-0.12pre12803-0qgw3s1x/MANIFEST.bz2’...
############################################################ 100.0%
3873 store paths in manifest

Installing package...
installing ‘unzip-5.52’
substituting path \
‘/nix/store/52y6jd9xbvr6w49k6vd7cf6kmgm2jn02-unzip-5.52’ \
using substituter \
‘/nix/store/9cv476i6kd...0yc1rjnx-nix-0.12pre12178/libexec/\
 nix/download-using-manifests.pl’
```

```
*** Trying to download/patch '/nix/store/52y6jd9xbv...2jn02-unzip-5.52'

*** Step 1/1: downloading \
'http://nixos.org/releases/nix-cache/1vf8knw6a2...lah0pggs1.nar.bz2' \
into '/nix/store/52y6jd9xbvr6w49k6vd7cf6kmgm2jn02-unzip-5.52'

  downloading archive...
############################################################ 100.0%
  unpacking archive...

building path(s) '/nix/store/p6iri1zaz809...gmwcigg2d-user-environment'

created 1030 symlinks in user environment
```

It pulls the MANIFEST from the remote repository and installs the package in the system.

Packages Installation

To proceed with the installation of a package, NixOS uses the command `nix-env`, one of the main command in NixOS. This command looks into the `.nix-channels` that have pointers to the repository where the packages are stored. Nix Channel is a pointer to a repository of packages we want to access. The repository contains a set of Nix expressions and a manifest. This pointer information is stored into a file located in `/.nix-channels` FOOTNOTE (works as sources.list for RPM). The installation of a package can also be performed by calling the derivation file of a package, for instance `nix-env -i /nix/store/fi214zh...83xkc-w3m-0.5.2.drv` or just `nix-env -i w3m-0.5.2`.

Install two packages of the same application

If the installation is made from the same package, i.e. the same version, it replaces the older one by the new one. The same applies to a newer version of the same package. If the user wants to install a new version of a package and wants to keep the older version installed under the system, it must explicitly indicate so to the package installer tool. For example, the command below must be executed to install a version of `gawk-3.1.6` and keep the previous installed. Otherwise it could generate a conflict between packages (view section Packages conflict).

```
nix-env --preserve-installed -i gawk-3.1.6
```

5.4.4 Rollback

The rollback process in NixOS consists in decreasing the number of the current generation. Generation is a snapshot of the system after the installation process. Every installation has a generation and the user can switch from a snapshot to another, removing the package installed in the latter installation by running the rollback command. This removal is not permanent, unless we run the garbage collection to remove all the generation but the current one. The current generation is the one that appears as a default in the grub menu when the system starts, and when the user runs a rollback (nix-env –rollback) or a switch command to an earlier generation (nix-env –switch-generation 2).

The example below shows all generations (snapshots) of the system at that moment

```
[pbarata@nixos:~]$ nix-env --list-generations
    1   2008-08-06  22:50:27
    2   2008-08-06  22:52:35
    3   2008-08-06  23:56:54
    4   2008-08-06  23:02:19
    5   2008-08-06  23:02:52
    6   2008-08-06  23:52:35   (current)
```

Execute a Rollback, that decreases the current position to the previous one

```
[pbarata@nixos:~]$ nix-env --rollback
switching from generation 6 to 5

[pbarata@nixos:~]$ nix-env --list-generations
    1   2008-08-06  22:50:27
    2   2008-08-06  22:52:35
    3   2008-08-06  23:56:54
    4   2008-08-06  23:02:19
    5   2008-08-06  23:02:52   (current)
    6   2008-08-06  23:52:35

[pbarata@nixos:~]$ nix-env --switch-generation 2
switching from generation 5 to 2

[pbarata@nixos:~]$ nix-env --list-generations
    1   2008-08-06  22:50:27
    2   2008-08-06  22:52:35   (current)
    3   2008-08-06  23:56:54
    4   2008-08-06  23:02:19
    5   2008-08-06  23:02:52
    6   2008-08-06  23:52:35
```

With the `--delete-generations old` option, all the generations will be deleted permanently, except the current one.

```
[pbarata@nixos:~]$ nix-env --delete-generations old
```

5.4.5 Dependencies

There are files that save all dependencies between packages. Broken dependencies are not a problem in NixOS, because they create a new entry for such a package and a package is never removed automatically. The removal of a package from the system should be done manually by the user.

5.4.6 Packages conflict

If the user installs a package but wants to keep an older version, it must explicitly inform the package installer. We tried to install *gawk-3.1.6* and keep the previous version of that package, the *gawk-3.1.5*, but encountered a conflict, which we solved, then we installed the new package and kept the older one installed. Several steps appear on the image below: a list

of packages installed by user, preserved installation of the older gawk, the conflict problem is shown, the conflict problem is solved, and the new package is installed keeping the older, by sequence.

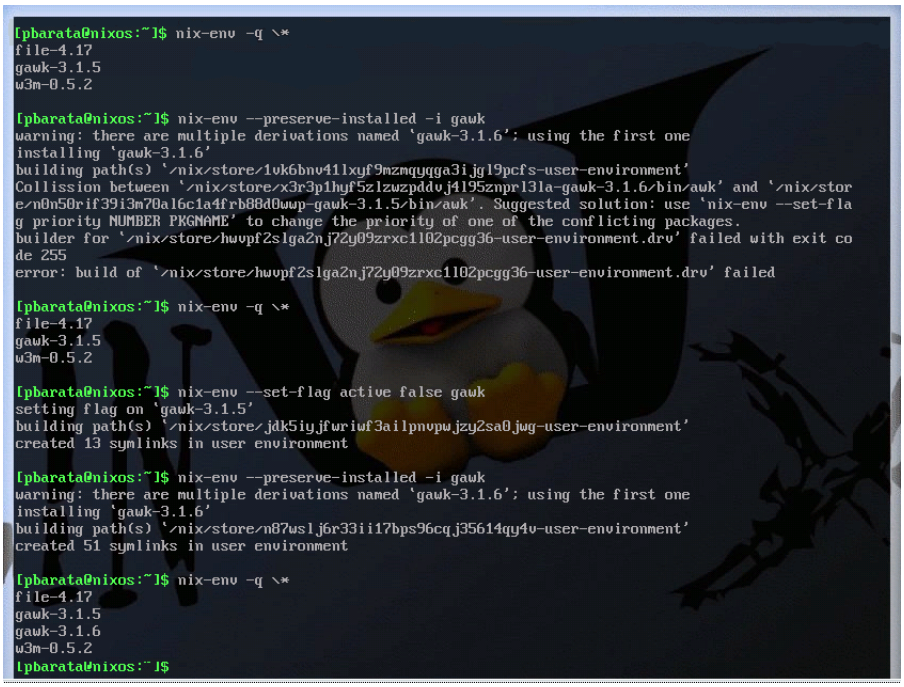

Figure 7: `Nixos Gawk conflict`

5.4.7 Profiles

When a user tries to install an existing package that is already installed by another user, a lot of symbolic links to those packages are created, but a reference between the package and the user is maintained. It is thus possible to have some application linked to a user, and a lot of users with some applications installed or shared within the filesystem and the NixOS repository.

5.4.8 Advantages

An advantage is that there are no broken dependencies. Different versions of the same package are installed at the same time. There is consistency between packages, because of their build process, with purely Nix Functions, and also stateless of the package.

5.4.9 Disadvantages

Space available for the system is an issue, because packages are never removed, unless the user does it manually in order to reduce the used space. The storage space for packages is highly consuming but the garbage collection mechanism can be executed to remove unreferenced packages, and free an important amount of disk space.

5.5 Apple MacOS X - Time machine

The main purpose of Time Machine [41] is to make backups of the filesystem as easy as possible for the users. Users do not usually have much technical knowledge to make backups and they sometimes do not know what to back up, nor even how to do it. Therefore, Apple developed Time Machine as a mechanism to help users. This tool forced Apple to make some changes over the file system HFS+ to support new features, mainly the use of several ”hard links” to one file, that they call ”multi links”.

5.5.1 Problem to Solve - Backups

Time machine is specially adapted to the cases where:

- No need to store things that did not change, but rather the ones that have changed
- Backups are usually spread in several disks, that make the restore process very hard for a disk or even for the whole system
- It is difficult to save all the changes along time and their spreadness through several disks
- It is difficult to know what to save

5.5.2 FSEvents

Time Machine has a monitoring system that saves all events that happen, called MAC OS X FSEvents. In that way it can look into FSEvents to see what was saved into the system without any user interference. When the Time Machine starts a backup, it does not need to search through all the disks to know what has changed, it asks the FSEvents which changes have occurred and saves those files only. The system does not backup temporary files and others that are not needed. It has an intelligent mechanism to do it automatically, so neither users or programmers need to care about this.

5.5.3 Where Backup

To make a backup with Time Machine the user only needs to connect an external device, via the Firewire or the USB device, and the configuration process starts. It is a simple and safe

way of doing it. It can also be made in a system disk, but according to Apple this is not the right way of doing it, because of the space consumption. Time Machine also allows users to select what they want to backup.

5.5.4 When to make a Backup

Time Machine makes a full backup when it first finds a source device to do it. After that, it starts making backups each hour, thanks to a scheduler designed for it. It drops the earlier hour backup for this day backup at the end of the day. It does the same thing for the weekly backups and the monthly backups. It removes the first day backup of the week at the end of the week, and so on. This is made that way, because of the need for space available to save new backups. Each backup takes just a few seconds to proceed, unless the user has written a lot of information in the last hour and over a lot of files. If the user creates and deletes a file in the period between backups, it will not be backed up.

5.5.5 Doing Backups

There are three ways of doing backups: differential, incremental and full backups.[15] [16] Time Machine makes full and differential backups every hour. In order to support this new functionality Apple changed the HFS+ MAC filesystem. This change is about the multi-links functionality (explained later in this document) that is similar to hard links in Unix like systems.

5.5.6 References to files

Hard links

"A Mac multi-link is a second 'hard link' record that points to data or to a directory".[17]

Hard links in one way look like a "ghost" and in other way look like a "clone". Therefore, when a new hard link is created it just refers to an existing file, but does not take additional space on disk. The system can create several hard links for the same file. When deleting a file, it is not removed from the filesystem, but a hard link counter decreases its counter of hard links references. To entirely remove a file, all the hard links must be removed.

Multi Links

HFS+ was changed to support Time Machine in order to create Multi-Links to file or directories. Multi Links is the capability of doing several hard links to the same files or directories.

The first full backup of Time Machine is a bunch of regular files saved in a "time-stamped folder". Afterwards, Time Machine makes an hourly backup, that seems a full backup but it is a copy of the new files created after the backup and a secondary hard link to the files already

[15] http://www.acronis.com/resource/solutions/backup/2005/incremental-backups.html
[16] http://en.wikipedia.org/wiki/Incremental_backup
[17] http://www.appleinsider.com/articles/07/10/12/road_to_mac_os_x_leopard_time_machine.html

backed up. The existing files that are already backed up (but not changed) will have one more hard link, called multi link. The original file can be removed without any problem, since the new hard link points to the data stored in the filesystem. The data stored in the filesystem is usually related to a file name, and can be the target of several hard links. When the last hard link to that file is deleted, this file is removed from the filesystem and is deleted for good.

5.5.7 Advantages

The use of hard links allows to do backups without using too much storage space. It also allows the user to see the whole system and what is stored in the backup at any time. For example, the user can see the state of his system two days before, or a week or even a month earlier, at a very precise point in time. With this capability, any version of any file can be seen at any time. Still, the user can browse any backup, and it will appear on Desktop and can be changed or restored easily.

5.5.8 Back up to the Future

Time Machine is a backup system, but it also allows the user to search files (photos, movies, documents, etc.) and much more. The user can also access his mailbox and see previously deleted emails, as well as contacts recently deleted from his address book. Time Machine is thus a backup system that visualizes the data in the system. This is user-friendly and made possible thanks to the changes made over the design of HFS+. As far as developers are concerned, they can also implement this functionality for those applications.

5.6 Zumastor

The main difference between Zumastor and other tools is the way the process of snapshot and replication is implemented. Zumastor has a limit of 64 snapshots per volume. The Zumastor team claims that this number will grow in the next releases. Zumastor is scalable when it writes to a volume, but has a `significant constant overhead` [36]. This does not depend on the number of snapshots it writes. If the administrator uses a RAID controller caches, this overhead will decrease.

5.6.1 History

Zumaster was first released to the public in December 2007 and the version 0.8 was issued in May 2008.

5.6.2 Block device snapshot and replication

The `ddsnap`[18] has the capability to make several snapshots efficiently at the same time. It distinguishes which are the differences from one snapshot to another and sends those differences to the snapshot volume. It can write blocks, on a device block used for snapshots, in a low-level way. That is why it is used in Zumastor.

5.6.3 LVM2

LVM, described in section 4.3, is a technology used with LVM to create snapshots, make backups and create logical volumes. Zumastor also does those things, but LVM is more simple. Zumastor has a garbage collector mechanism that removes old snapshot and mount/unmount volumes.

5.6.4 Snapshots

Zumastor creates volumes and mounts them under `/var/run/zumastor/mount` and creates its snapshots under `/var/run/zumastor/snapshot`. The snapshots are taken from the mounted volumes, called origin mounts, and mounted in snapshots volumes, called snapshot volumes. The snapshot constructs its name in the form of `YYYY.MM.DD-HH:MM.SS`, and keeps several Symbolic Links for each of those snapshots. All these volumes (origin and snapshot) are automatically mounted by a zumastor init script at boot time.

Zumastor can make backups every hour or/and day or other, so that it can be scheduled. It is easy and helpful because zumastor can do this by stopping the system for one or two seconds, exactly the time need to make a snapshot.

Zumastor has a snapshot store and a volume store to save data and keep this useful information available.

Zumastor uses `ddsnap` to access the snapshot store via virtual devices. It also uses LVM2 to create the volumes to mount origin volumes and snapshot volumes.

5.6.5 Remote replication

Replication is the process of sending a snapshot from one machine to another, typically from a local machine to a remote one. The replication process is over when this snapshot is entirely sent. This proccess can be scheduled to be sent hourly, or at any other frequency, in upstream. Once done in down stream and the snapshot downloaded, the main volume will be pointed to the new one.

Zumastor recognizes what blocks have changed from one snapshot to another and sends those changed blocks to the next snapshot.

Definitions

[18] `http://zumastor.org/man/ddsnap.8.html`

With Zumastor, the administrator can define which volumes to use, their name, size, mount points, the snapshots, and the replication machine.

Creating a Snapshot

First create a Physical Volume as shown below

```
pvcreate /dev/zumast
```

Creation of a Volume Group

```
vgcreate zumastvg /dev/zumast
```

Creation of a Logical Volume for an Original volume

```
lvcreate --name test --size 200MB zumastvg
```

Format the Logical Volume created above

```
mkfs.ext3 /dev/zumastvg/test
```

Creation of a Logical Volume for a Snapshot volume

```
lvcreate --name test_snap --size 200MB zumastvg
```

The creation above is made using LVM2 commands (`pvcreate, vgcreate, lvcreate`). Master is used to define the start process of snapshotting by the scheduled time definition (by user or by default).

```
zumastor define volume zumastor_test /dev/zumastvg/test /dev/zumastvg/test_snap
--initialize
```

```
zumastor define master zumastor_test
```

5.7 Apt-RPM

Apt-RPM[5] is a meta-installer that selects, retrieves and installs RPM packages. Apt-RPM is a fork project of APT.

The original APT (Advanced Packaging Tool) project was initially developed to extend some of the capabilities of the *dpkg* command, which allows the installation of local DEB files. The DEB files contain applications to be installed.

To some extent, this is the same path taken by other meta-installers like up2date / YUM or YaST / YOU but these related to RPM files.

The installer level (rpm / dpkg) lacks a lot of features that are important to the user, such as:

- **Retrieving files**: meta-installers like Apt-RPM support different methods for retrieving the file such as FTP, HTTP, local file or peer-to-peer.
- **Solving dependencies**: meta-installers are able to identify whether a package needs

another package (dependency relation) or has a conflict. Once identified which packages are needed, the version constrains to the universe of available packages must be applied.

- **Searching for packages**: since the repositories are registered in the meta-installer, we can use it to search for a specific package by name, category or description.

5.7.1 Rollback

RPM Rollback feature

Related with transactional upgrades, RPM has a rollback feature since version 4.0.3[52]. The transactional rollback feature was developed by Jeff Johnson in 2002. This feature allows a backup of all files of the version that has to be replaced to be stored in the disk in each transaction. This files are kept in a RPM package associated with TID (transaction ID). This feature is very interesting but has two major problems: a) a large space is needed in the disk since all files, even binaries, are maintained and b) changes performed by installation scripts outside of the set of RPMs are not stored.

An evolution of this approach was proposed in 2005 by Caixa Mï¿½gica Software in the framework of the EDOS FP6 project.

Caixa Mï¿½gica Rollback feature

Caixa Mï¿½gica has developed several contributions to Apt-RPM[6]. The most interesting in this scope is transactional rollback at meta-installer level.

This feature allows the user to revert any transaction (remove, install or upgrade of an RPM package). To support it, Apt-RPM has two sources of information: a) a SQLite database with the information about all transaction and b) configuration files of each transaction.

Since binary files are not supposed to be changed by the user, Apt-RPM downloads the previous RPM file when a rollback is required, unpacks it and applies the configurations to it. In this way, the need of disk space is much smaller and all properties of transaction rollback at RPM level are maintained. The requirements for this feature are: SQLite support and access to RPM repositories that maintain old versions.

6 Conclusion

State of the art technologies for reverting changes in the operating system have very different stages of development and goals. While the pure functional approach is still in a proof-of-concept phase, others like transactional rollbacks and snapshotting techniques are deployed over millions of systems world wide.

Nevertheless, they share the same goal: *let the user revert an action in the system*.

While some are more oriented towards a total backup of the system (applications and user data), others are specifically concerned with installed applications.

Table 1 depicts some of the technologies presented in this deliverable.

File System	Transactional rollback	Functional Approach
NexentaOS	Conary	NixOS
MacOS Time machine	Apt-rpm	
Zumaster		

Table 1: Classification of rollback tools

This classification is further extended in table 2 where each tool is analysed according to the granularity and the time period for the changes monitored.

Although a more intensive work in the scope of Mancoosi's Workpackage 3 has to be performed, a first approach to specification of a transactional upgrade system with rollback capabilities should follow the high level specification[18] in table 3.

A brief description of the mandatory elements of the high level specification follows:

- **Respect ACID properties**: the system should respect the ACID properties: Atomicity, Consistency, Isolation and Durability.
- **Transaction independence**: One transaction should be independent of other transactions. This means that a rollback of a transaction has to be executed without needing other transactions to be rollbacked as well.
- **Package level granularity**: the level of granularity of the transaction is the package. This does not block the possibility of maintaining state of files independently and allow some operations to be performed at that level.
- **Monitoring domain**: A rollback system should monitor all files related with the package. This files comprise not only the files included in the package but all files created, modified or deleted during the transaction.
- **Device independence**: the rollback capability should be available in different devices like servers, desktops, mobile devices and others.
- **Minimum disk space use**: the used disk space for storing transactional information should be limited to the absolute necessary. The disk space should be limited to 5% to 10% of the total disk space used by the system.

Rollback tool	Granularity	Reported time period
NexentaOS	All file system	Since last snapshot
Conary	Set of files contained in a package	Triggered by the upgrade of the package
NixOS	Set of files contained in a package	Permanent
MacOS Time machine	All file system	Since last snapshot. Triggered by user, time or changes.
Apt-RPM	Set of files contained in a package	Triggered by the upgrade of the package
Zumaster	Filesystem	Since last snapshot

Table 2: Analysis of different snapshot / rollback tools

#	Requirement	Designation
M1	REQUIRED	Transaction respect ACID properties
M2	REQUIRED	Transaction Independence
M3	REQUIRED	Package level granularity
M4	REQUIRED	Monitoring domain
M5	REQUIRED	Device independence
M6	REQUIRED	Minimum disk space use
M7	REQUIRED	User transparency
N1	MUST NOT	File-system Dependency
O1	MAY	Permanent change monitoring
O2	MAY	User interaction

Table 3: High level specification of transactional upgrade system with rollback capabilities.

- **User transparency**: If activated, the support for rollback transactions should not oblige the user to specific actions or decisions.

The following element must not be mandatory:

- **File system dependency**: a transactional upgrade system must not depend on a specific file system.

The following elements may be implemented but are not mandatory:

- **Permanent change monitoring**: permanently monitoring the supervised files without being triggered by the user or the transactional system may be an option of implementation.
- **User interaction**: the transactional system may allow the user to add new files to be supervised.

Workpackage 3 will now refine this specification and develop an intensive research work in the critical topics towards a secure, safe and complete transaction upgrade system with rollback capabilities.

References

[1] EDOS deliverable 3.1: A system testing framework. http://www.edos-project.org/xwiki/bin/download/Main/D3-1/edos-d3.1.pdf.

[2] Objectweb. Fractal - Julia. http://fractal.objectweb.org/julia/index.html.

[3] Objectweb. Fractal - Specification. http://fractal.objectweb.org/specification/index.html.

[4] xADL 2.0 Architecture Description Language. http://www.isr.uci.edu/projects/xarchuci/, 2005.

[5] Apt-rpm project, 2008. http://apt-rpm.org/.

[6] Caixa magica apt-rpm contributions, 2008. http://aptrpm.caixamagica.pt/.

[7] Open services gateway initiative (osgi). http://www.osgi.org/, July, 2002.

[8] Zeller A. Versioning software systems through concept descriptions. Technical report, Technical Report 97-01, Universität Braunschweig, Jan 1997.

[9] Robert John Allen. *A formal approach to software architecture*. PhD thesis, Pittsburgh, PA, USA, 1997. Chair-David Garlan.

[10] Noriki Amano and Takuo Watanabe. A software model for flexible & safe adaptation of mobile code programs. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, pages 57–61, New York, NY, USA, 2002. ACM.

[11] Anneliese A. Andrews, Andreas Stefik, Nina Picone, and Sudipto Ghosh. A cots component comprehension process. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 135–144, Washington, DC, USA, 2005. IEEE Computer Society.

[12] J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis. Enabling autonomic behavior in systems software with hot swapping. *IBM Syst. J.*, 42(1):60–76, 2003.

[13] Marco Autili, Paola Inverardi, Alfredo Navarra, and Massimo Tivoli. Synthesis: A tool for automatically assembling correct and distributed component-based systems. In *ICSE*, pages 784–787, 2007.

[14] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, December 1997.

[15] Thaís Vasconcelos Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In *EWSA*, pages 1–17, 2005.

[16] D. Box. *Essential COM*. Addison-Wesley, 1998.

[17] Andrea Bracciali, Antonio Brogi, and Carlos Canal. Systematic component adaptation. *Electr. Notes Theor. Comput. Sci.*, 66(4), 2002.

[18] S. Bradner. Key words for use in rfcs to indicate requirement levels, 1997.

[19] J.Z. Brockmeier. Nexenta combines opensolaris, gnu, and ubuntu. http://www.linux.com/articles/57628, October 2006.

[20] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. *sera*, 0:40–48, 2006.

[21] Paul C. Clements. A survey of architecture description languages. In *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design*, page 16, Washington, DC, USA, 1996. IEEE Computer Society.

[22] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. What is subversion? http://svnbook.red-bean.com/en/1.4/svn.intro.whatis.html, 2008.

[23] V. Cortellessa, I. Crnkovic, F. Marinelli, and P. Potena. Experimenting the automated selection of cots components based on cost and system requirements. *Journal of Universal Computer Science (JUCS). Accepted for publication.*

[24] Vittorio Cortellessa, Fabrizio Marinelli, and Pasqualina Potena. Automated selection of software components based on cost/reliability tradeoff. In *EWSA*, pages 66–81, 2006.

[25] Geoff Coulson, Gordon S. Blair, Paul Grace, Ackbar Joolia, Kevin Lee, and Jo Ueyama. A component model for building systems software. In M. H. Hamza, editor, *IASTED Conf. on Software Engineering and Applications*, pages 684–689. IASTED/ACTA Press, 2004.

[26] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5):109–120, 2001.

[27] Rogério de Lemos, Cristina Gacek, and Alexander Romanovsky. Architectural mismatch tolerance. In Rogério de Lemos, Cristina Gacek, and Alexander B. Romanovsky, editors, *WADS*, volume 2677 of *Lecture Notes in Computer Science*, pages 175–194. Springer, 2002.

[28] OMG Document. formal/02-06-01: The common object request broker: Architecture and specification, revision 3.0, July, 2002.

[29] OMG TC Document. Orb interface type versioning management, request for proposal, 1996.

[30] P. Falcarin and G. Alonso. Software architecture evolution through dynamic aop. In *EWSA 2004*. LNCS 3047. Springer-Verlag,, 2004.

[31] Cristina Gacek. *Detecting architectural mismatches during systems composition*. PhD thesis, Los Angeles, CA, USA, 1998. Adviser-Barry W. Boehm.

[32] Gemma Grau, Juan Pablo Carvallo, Xavier Franch, and Carme Quer. Descots: A software system for selecting cots components. *euromicro*, 00:118–126, 2004.

[33] Lars Grunske. Identifying "good" architectural design alternatives with multi-objective optimization strategies. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 849–852, New York, NY, USA, 2006. ACM.

[34] George T. Heineman and William T. Councill, editors. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[35] Christine R. Hofmeister and James M. Purtilo. A framework for dynamic reconfiguration of distributed programs. Technical report, College Park, MD, USA, 1993.

[36] Dan Kegel. Zumastor howto. URL: http://zumastor.googlecode.com/svn/branches/0.8/doc/zumastor-howto.html, 2008.

[37] Gerald Kotonya and John Hutchinson. Viewpoints for specifying component-based systems. In *CBSE*, pages 114–121, 2004.

[38] Sandeep S. Kulkarni and Karun N. Biyani. Correctness of component-based adaptation. In *CBSE*, pages 48–58, 2004.

[39] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Transactions on Software Engineering*, 33(10):709–724, 2007.

[40] F. Lin and W.M. Wonham. Supervisory control of timed discrete-event systems under partial observation. *Automatic Control, IEEE Transactions on*, 40(3):558–562, Mar 1995.

[41] Prince McLean. Road to mac os x leopard: Time machine, 2008.

[42] Sun Microsystems. Java language specification, second edition, April 2000.

[43] Marija Mikic-Rakic, Sam Malek, Nels Beckman, and Nenad Medvidovic. A tailorable environment for assessing the quality of deployment architectures in highly distributed settings. In *Component Deployment*, pages 1–17, 2004.

[44] Mohamed, Ruhe, and Eberlein. Cots selection: Past, present, and future. *ecbs*, 00:103–114, 2007.

[45] R. Monson-Haefel. *Enterprise JavaBeans, fourth ed.* O'Reilly & Assoc, 2004.

[46] Bill Moore. Zfs - the last word in file systems. http://www.sun.com/software/solaris/zfs_lc_preso.pdf, 2008.

[47] Fredy Navarrete, Pere Botella, and Xavier Franch. How agile cots selection methods are (and can be)? In *EUROMICRO '05: Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 160–167, Washington, DC, USA, 2005. IEEE Computer Society.

[48] C. Ncube and N. A. M. Maiden. Pore: Procurement-oriented requirements engineering method for the component-based systems engineering development paradigm. In *Proc. of International Workshop on Component-Based Software Engineering*, 1999.

[49] Thomas Neubauer and Christian Stummer. Interactive decision support for multiobjective cots selection. In *HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, page 283b, Washington, DC, USA, 2007. IEEE Computer Society.

[50] Nexenta. About nexenta project. http://www.nexenta.org/os/AboutNexenta.

[51] Nexenta. Nexenta operating system. http://www.nexenta.org/os, 2008.

[52] James Olin Oden. Transactions and rollback with rpm. *Linux Journal*, May 2004. http://www.linuxjournal.com/article/7034.

[53] Roberto Passerone, Luca de Alfaro, Thomas A. Henzinger, and Alberto L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: two faces of the same coin. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 132–139, New York, NY, USA, 2002. ACM.

[54] F. Plásil, D. Bálek, and R. Janecek. Sofa/dcup: Architecture for component trading and dynamic updating. In *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*, page 43, Washington, DC, USA, 1998. IEEE Computer Society.

[55] Riggs R. Java product versioning specification, part of the sun's jdk 1.2 specification sun, Dec. 1997.

[56] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, 1987.

[57] rPath.com. Conary. http://wiki.rpath.com/wiki/Conary, 2008.

[58] Günther Ruhe. Intelligent support for selection of cots products. In *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*, pages 34–45, London, UK, 2003. Springer-Verlag.

[59] R. O. Sinnott. An architecture based approach to specifying distributed systems in lotos and z. PhD thesis, Department of Computing Science and Mathematics, University of Stirling, 1997.

[60] Marcin Solarski. *Dynamic Upgrade of Distributed Software Components*. PhD thesis, Technischen Universität Berlin, 2004.

[61] Bridget Spitznagel and David Garlan. A compositional formalization of connector wrappers. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 374–384, Washington, DC, USA, 2003. IEEE Computer Society.

[62] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[63] Massimo Tivoli and Paola Inverardi. Failure-free coordinators synthesis for component-based architectures. *Sci. Comput. Program.*, 71(3):181–212, 2008.

[64] A. Wigley, M. Sutton, R. MacLeod, R. Burbidge, and S. Wheelwright. Microsoft .net compact framework (core reference). Microsoft Press, January 2003.

[65] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.

[66] Ji Zhang, Betty H. C. Cheng, Zhenxiao Yang, and Philip K. McKinley. Enabling safe dynamic component-based software adaptation. In *WADS*, pages 194–211, 2004.