EUROPEAN SYNCHROTRON RADIATION FACILITY
INSTALLATION EUROPEENNE DE RAYONNEMENT SYNCHROTRON
COMPUTING SERVICES

ESRF/Linux Version Management

Controlling the Release Flow

Petri Mäkijärvi

June 8, 2005

Abstract

The ESRF is searching to improve the long term stability of the Linux based computer installations. The main goal is to increase the homogeneity for the application level support. By limiting the number of Linux installation versions we will also reduce the maintenance costs and we will be able to give better quality system support service.

This document suggests an internal version controlling scheme for a commercial Linux distribution, Red Hat Enterprise Linux. The paper is motivated by the need to better control the operating system release flow so that the application level software can expect long scale stability from the run time environment.

*The outcome of the Red Hat Enterprise Linux distribution and the proposed version control scheme is called **ESRF/Linux**.*

1 Introduction

The ESRF is searching to improve the long term stability of the Linux based computer installations. The main goal is to increase the homogeneity for the application level support. By limiting the number of Linux installation versions we will also reduce the maintenance costs and we will be able to give better quality system support service.

To have a more stable Linux support at the ESRF we need to able to decide the release cycle of the Linux versions ourselves instead of the outside world deciding it for us. The first step toward this goal is to select a commercial Linux distribution with a guaranteed life time and with some proven stability. Starting from that distribution, we would then create our own, regulated distribution. Let's call this strategy of controlling the release flow ***ESRF/Linux***.

The below table gives some idea about the release cycles that we are talking about:

Distribution	***Life Time***	***New major release***	***New minor version***
Software Editor	5 years	18 months	Once per month
ESRF	5 years	3 years	Once per year

1.1 Intended readers

This document targets three groups of readers:

- **System Administrators.**
  The entire document can be used as a specification and as an architectural guideline for the ESRF/Linux version and release management implementation.

- **Application Programmers.**
  The document helps application level programmers in groups such as BLISS and SEG to better understand what will be the impact of the ESRF/Linux version management to their workflow. Suggested sections and chapters are listed below.
  - 3 Version numbering
  - 4.3.1 Kernel
  - 4.3.3 Application level device drivers
  - 5 Managing critical updates

- **Executive Level Summary**
  of this document can be obtained by reading the following chapters:
  - 2 Setting the goals
  - 3 Version numbering
  - 5 Managing critical updates

1.2 Red Hat distribution and this document

The software editor that will be used for the ESRF/Linux is Red Hat[1]. The following products and tools will be discussed in this document:

- Red Hat Entreprise Linux 4 Workstation[2] (***RH4WS***) distribution
- Kickstart[3] – answer file based automated installation for RH4WS.
- Rembo[4] – a network based system cloning tool

[1] http://www.redhat.com/
[2] See http://www.redhat.com/software/rhel/client/
[3] See http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/sysadmin-guide/s1-kickstart2-howuse.html

2 Setting the goals

The ESRF/Linux distribution should fill the following needs:

1. **Promote the application stability**.
   An operating system alone is worth nothing. The main goal of the ESRF/Linux is to increase the stability for the application level support. Each group at the ESRF working with our Linux distribution should feel at home with it and be well informed about the actual status of the ESRF/Linux so that they can build their own packages on top the distribution that they can rely on in the coming years. The ESRF/Linux should promote centralized, compile-farm type of application compilation, system module creation and version control.

2. **Evolve with time**.
   When new hardware will appear, it may occur that new system level software needs to be installed. As a consequence, the ESRF/Linux distribution should publish **minor versions**. When next generation Linux kernel will show up, we should prepare ourselves to switch to a new **major release** of the ESRF/Linux.

3. **Respond to critical events**.
   An unpredictable event may trigger out vulnerability in the installed park of the ESRF/Linux distribution. It may have a major impact either to the security or to the proper functioning of the ESRF computing infrastructure. When a fix is found, either internally or from the software vendor we should be able to distribute the patch on all machines concerned with minimizing the risk of breaking them. There should be **patch levels** in the ESRF/Linux versioning scheme.

4. **Maintain backward compatibility**.
   From the second criteria follows that there will be several different versions of ESRF/Linux installations. Motivated by the system stability criteria for application level software, we will not search to maintain homogeneous installation at the version level but older installations will remain largely untouched. But from the third criteria follows that we should be able to maintain the older versions and – if necessary – bring them at the current level what concerns critical features, without a complete reinstallation.
   We should be able to install the past versions if a need arise.

[4] Read about The Rembo Wizard at http://rembowiz.sourceforge.net/

3 Version numbering

The key to the successful management of the ESRF/Linux is in the careful version numbering. Each installation should be identified with ***[Release].[Version].[Patch]*** tag. For example, the first ever version of ESRF/Linux will have a version number tag **1.1.1**. More details of the fields in the version number tag are explained in following sections.

3.1 Release number

The major version number is the release number, which is reflecting the chosen software vendor's Linux distribution platform. For example, following list can be given to start with:

1. Distribution is Red Hat Enterprise Linux 4 Workstation (2005)
2. Distribution is Red Hat Enterprise Linux 6 Workstation (2008)

There will be no release based on "Red Hat Enterprise Linux 5" provided that the Red Hat release cycle is 18 months and the ESRF release cycle is three years.

3.2 Version number

The minor version number is the ESRF/Linux internal release number. As explained above, the ESRF/Linux internal releases would have the interval of one year, approximately. An internal release would typically contain cumulative security patches, critical patches and other software updates from the software vendor. It may be possible that a new internal release is published in response to an unexpected event, such as a severe virus attack if a patch cannot resolve the problem.

The first ESRF/Linux version number for a major release is **1** (*one*), the version number **0** (*zero*) is reserved for Computing Services internal prototypes.

3.3 Patch level

The software vendor is releasing non-critical and critical patches continuously, sometimes almost daily. Occasionally a patch is needed from the software vendor because a particular type of hardware does not install without it. All the patches that are estimated of important nature are collected under the patch level tag. For example, the machines installed this month will be at patch level 1.1.4, next month an arrival of a new type of machine will make us to switch to the next patch level 1.1.5. From that point onwards, we will continue to install ESRF/Linux 1.1.5 everywhere, until the next patch comes out.

The first ESRF/Linux patch level number for a major release is **1** (*one*), the patch level number **0** (*zero*) is reserved for Computing Services internal prototypes. The patch level reflects the actual status of the patches from the software vendor that have been decided to be installed on ESRF/Linux. The reasons for the generic patch level number maintenance will be explained later in details (*see 4.2.1 Vendor patches*), it is now enough to understand that if there exists an ESRF/Linux 1.3.16, there will exist also versions 1.1.16 and 1.2.16, indicating the same patch and security levels will be made available on all versions.

3.4 Impact to the application level homogeneity

It is important to understand that the application level homogeneity is maintained to its maximum extend. For example, there is no rule that would make an application that has been compiled and tested on an ESRF/Linux 1.1.1 to fail when installed on an ESRF/Linux 1.3.2 system. This principle is called **forward compatibility**. Following important events in the life time of the ESRF/Linux will always be planned, announced ahead and documented appropriately:

- **Kernel change**.
  This is a major event that is avoided to the maximum extent. But if a critical event, major hardware update or such forces the kernel change a new ESRF/Linux version is announced. The reasons for the deployment of the new kernel are explained and a compilation farm type of test environment is set up for the developers and system administrators.
- **C/C++/Fortran/Python/Perl compiler/interpreter or run-time library change**.
  This event cannot occur without a collective agreement between SEG, BLISS, SciSoft, System Administrators and any other application level management body at the ESRF. The initiative for the change can come from any of the groups but its application in the ESRF/Linux would require similar procedure that is explained above for the event of a kernel change.

Exceptions to the above rules are discussed later in this document (see 4.3.1 *Kernel*).

3.5 Obtaining version information

A developer, system administrator or a user login on a system would get a simple ESRF/Linux version message. But in order to manage the entire park of the installed ESRF/Linux systems we need a distributed database solution.

A traditional, device inventory type of database, such as the *jDACE* (under development) would help us to find devices based on their network connection information. Such databases are not suitable to maintain the information about installed operating system and about the actual patch level. Text field data-file based configuration information collections are even less suitable for the job (*ex. /csadmin/common/db/beamline/*.data*).

It is better to keep the exact operating system configuration information within the target system itself and use the technique of distributed databases to manage the version and packaging information. The required version information is stored in local databases within the target systems at two levels:

1. **SNMP database**.
   This database would contain all the necessary administrative information concerning the ESRF/Linux installation. With a few simple commands, the administrators are able to collect information about the entire installed park of the ESRF/Linux machines, their version number, patch levels and other important issues that are needed in the decision making.
   An application programmer's interface would be provided for the application installation management.
2. **RPM database**.
   By the nature of the software vendor's package installation method; there is a local RPM package database present on all machines. It is important to understand that the contents of the packages, and thus the contents of the RPM package database are managed by the ESRF/Linux administrative scripts when the system is needed to be patched. No connection to the software vendor's Internet based update services is made in order to maintain a complete control of the software package database.

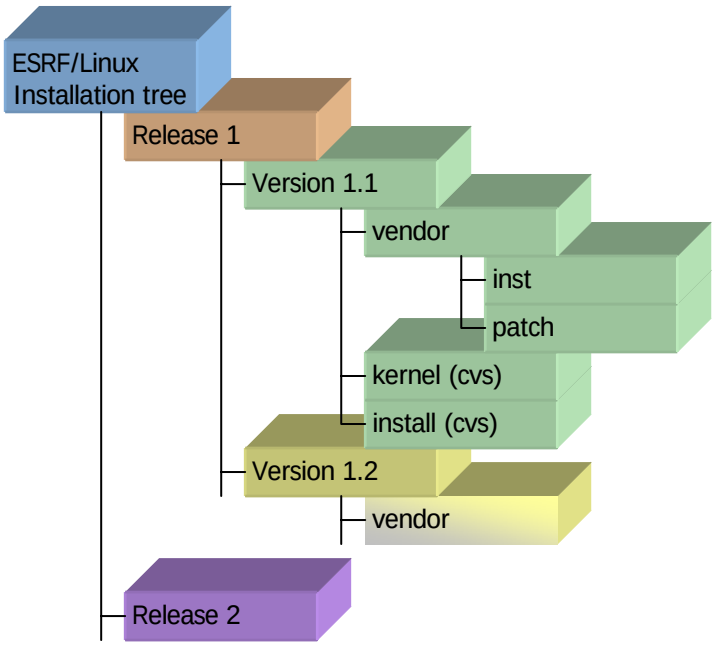
4 Installation tree

In this chapter we are discussing the organization of the installation tree and its hierarchy.

Typically for any installation system, there is a top level entry to the ESRF/Linux installation tree. The exact access mechanism to the ESRF/Linux installation tree's top level entry does not need to be discussed here. It can be almost anything, but most probably it will use a network based access, using protocols such as FTP, NFS or HTTP.

CS Computing Assistance Group is in charge of maintaining the ESRF/Linux installation tree.

4.1 Intallation tree structure



When an installation media makes a reference to the Installation Tree, it would use a path such as

`[protocol]://[server]/ESRF/Linux/[release]/[version]`

For example, `ftp://install.esrf.fr/ESRF/Linux/1/1`. The below sections will explain different elements in the Installation tree and how they will be managed.

4.2 Vendor package management

For each ESRF/Linux version, the vendor packages are collected in *vendor/inst* directory. The structure and the content within this directory are organized as in vendor's installation tree.

4.2.1 Vendor patches

Cumulative security patches, critical patches and other such packages are collected under *vendor/patch* directory.

When the software vendor announces a package it is not automatically added into this directory. The ESRF/Linux Maintainer decides which packages from the vendor patches should actually go into the vendor patch directory according to their usage at the ESRF.

Vendor patches are maintained also in earlier published versions. As explained in section *3.3 Patch level*, the patch number indicates a general level of vendor patches applied for the ESRF/Linux. For example, if the actual installing version of ESRF/Linux is 1.2.5, the general patch level is 5. From this follows, that there must exists also an upgraded version of an earlier version, 1.1.5. This is to indicate that both the version 1.1 and 1.2 distributions have the same critical patches applied.

4.3 System modules

This section discuss about the system modules and their management within the ESRF/Linux.

4.3.1 Kernel

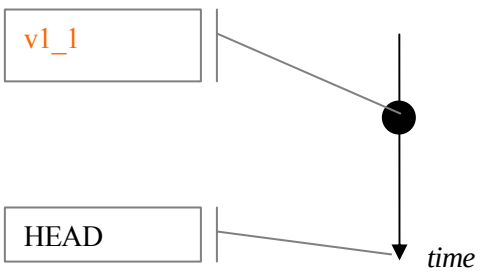
The motivation behind the ESRF/Linux distribution is to promote the overall stability while allowing exceptions when the official distribution cannot do the required job. By learning from the past we can point out that most of the exceptions will fall in the category of the ESRF/Linux distribution's kernel not being usable with a specific detector hardware device driver or such. Although we are promoting to use dynamically loadable, modular device drivers whenever possible, sometimes it is not feasible. Therefore we have to have a controlled way to manage a reasonable number of kernels within the ESRF/Linux distribution and also outside of the distribution.

Therefore the kernel management should be kept outside of the vendor distribution management. Custom built, tailored kernels should be integrated to the kernel management. To answer to these requirements, the kernel management for the ESRF/Linux should be managed outside the Installation Tree, using an external version control repository on the ESRF/Linux compilation farm arrangement. The version control system to be used for the kernel management is Concurrent Versions System, or ***CVS***.

CVS and its usage is too vast subject to discuss in this document, recommend reading for developers is "Open Source Development with CVS"[5], for everybody else "CVS for New Users"[6]. The following explanation tries to be as generic as possible, without going into the details of the CVS usage.

Version control systems are intended to manage directories full of source code modules, or files. A Linux kernel contains hundreds and hundreds of these source code modules and it would be an impossible task to set up and maintain a Linux kernel tree under any version control system. Luckily Linus Torvalds is taking care about that already. We would use a CVS repository just to store bz2 encoded (compressed) tar-balls (source code archives). This is better followed with an example.

When the first ESRF/Linux will go out, it has a revision and version number set to 1.1. After creating the Installation Tree for ESRF/Linux 1.1 we will probably have installed kernel version *2.6.9-5.0.5.EL-i686*. The directory with that name in the development system (a compilation farm machine) is left as it is, with compiled binaries to reduce the compiler incompatibility problems that may occur on later installations and to maintain maximum information about the kernel installation in the repository. Without going into the details of the CVS instructions, the initial contents of the ESRF/Linux kernel maintenance repository would look like as illustrated in the below picture.



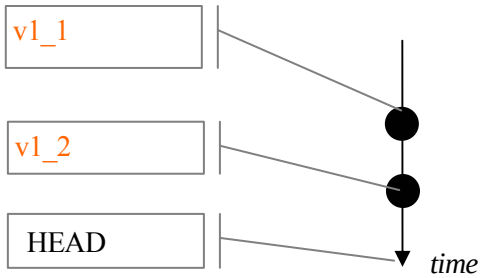
v1_1 is called a ***tag***. It is used to permanently mark a position of all files in the repository at a given time. In this case it is determining the ESRF/Linux revision (1) and version (1).

[5] 1st edition on the web http://cvsbook.red-bean.com/cvsbook.html
[6] From cvs.org https://www.cvshome.org/new_users.html

Instead of multiple source file, the point contains just a single file, which we will always call ***esrflinuxkernel.tar.bz2***. It is binary file, a compressed archive that, when unpacked will give us the entire contents of the directory *2.6.9-5.0.5.EL-i686*.

The directory in our example, *2.6.9-5.0.5.EL-i686* will be placed in the directory *[protocol]://[server]/ESRF/Linux/1/1/kernel*. The directory is left as it is, with binaries and without ever recompiling it. To help installation between different versions, a symbolic link will be created within the directory (*esrflinuxkernel -> 2.6.9-5.0.5.EL-i686* in our example).

It is to be reminded that if we **change the kernel**, it is a major event in the ESRF/Linux distribution and that it will automatically lead to a new version number in the distribution. Let's take an example of the new kernel *2.6.18-6.0.7.EL-i686* that everybody agrees being the right one to resolve all our problems. On our development (compile farm) machine will check out the HEAD from the CVS version control system. We will get a single file, *esrflinuxkernel.bz2*. When unpacking it, we will find again the same *2.6.9-5.0.5.EL-i686* directory. We will ignore it and concentrate our development efforts to the new kernel. Once it is ready for the distribution, we recreate *esrflinuxkernel.tar.bz2* archive, this time with the contents of the *2.6.18-6.0.7.EL-i686* directory. We will check it in to the CVS control system and tag it with the new ESRF/Linux version number (2). The ESRF/Linux kernel maintenance repository would look as illustrated in the below image.

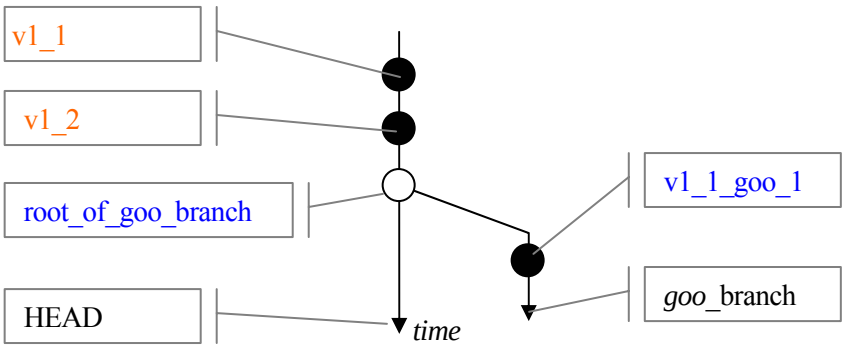


As with the kernel for the ESRF/Linux version 1.1 we can get the ESRF/Linux 1.2 kernel out from the repository using the tag **v1_2**.

The directory in our example, *2.6.18-6.0.7.EL-i686* will be placed in the directory *[protocol]://[server]/ESRF/Linux/1/2/kernel* and a symbolic link *esrflinuxkernel* will be set to point into it.

What about ***kernel exceptions***? Let's imagine the following example: A new detector, called *goo* has arrived with a device driver that ***must*** be compiled into the kernel, or otherwise… A BLISS support person for the beamline must take care of this task. To complicate the task, the computer on which the detector's interface card will be installed is ESRF/Linux version 1.1.3, while we are installing already version 1.2.10 everywhere else.

The BLISS support person would use a machine in the ESRF/Linux compile farm with the same version number (1.1) that is running in the target system. The support person would extract the kernel used in the computer from the ESRF/Linux kernel maintenance repository using the tag **v1_1**. After putting in the *goo* device driver, this person would recompile the kernel and test it. Once satisfied with the result, the BLISS support person would put in the kernel to the ESRF/Linux kernel maintenance repository. This time, a branch is created because this is a specific kernel. For those familiar with CVS, let's add that the ***branch is never merged*** with the trunk (the main tree). The below image illustrates the contents of the ESRF/Linux kernel maintenance repository after the first version of the kernel with the incorporated *goo* device driver has been checked in and documented.



What about with **exceptions of exceptions**? For example, can I use **v1_2** instead of **v1_1** for the ESRF/Linux 1.1 run-time? The answer is that the version management of the ESRF/Linux does not set any limitations to this kind of operations as long as the *goo* branch is not merged with the ESRF/Linux kernel trunk.

4.3.2 Vendor device drivers

Vendor device drivers will not be separated from the vendor's package management (see 4.2). Therefore their installation and version management is not different to that of the other vendor packages.

It may occur that a particular hardware needs exceptional measures and that the software vendor can provide a solution to the problem in form of a device driver. The vendor based device driver installation should be managed as any other software vendor patch installation (see 4.2.1 *Vendor patches*).

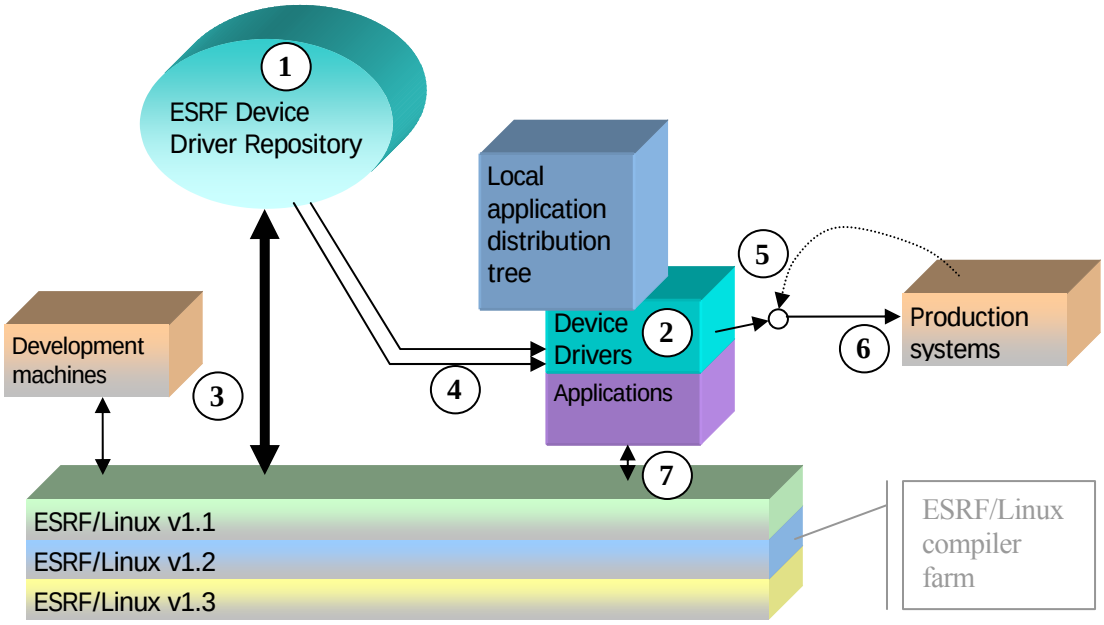
4.3.3 Application level device drivers

Application level device drivers, such as ESRF-developed I/O board drivers are part of the application software and they will be managed outside the ESRF/Linux version management system and rules. But any device driver is a system module, having dependencies of the kernel used in the ESRF/Linux and therefore there must be a mechanism to synchronize the application level device driver version management and the ESRF/Linux version management. The control on the actual functionality of the application level device drivers should be left to the groups responsible of the entire application level implementation, such as BLISS and SEG.

In the search of stability of the run-time platforms, there will be probably but one, or in the worst case a few ESRF/Linux versions on which the application level device drivers need to run. With the time, the device drivers will get out of synchronization from the ESRF/Linux, which will continue to evolve. For example, when a new vendor kernel version needs to be installed, the ESRF/Linux version will change. Although the application level software will probably not suffer from the change, the application level device driver modules need to be recompiled for the new ESRF/Linux version.

As long as there is no need to update the run-times on which the application level device drivers are used, there is no problem. But sooner or later we will be faced with the problem of the application level device driver recompilation and testing for the new ESRF/Linux version. For example, a major critical event (a virus attack, a huge NFS bug or such) may force us to plan an upgrade of the installed park of the ESRF/Linux systems. Hopefully, a patching plan is sufficient to deal with the upgrade. In the opposite case, we may have to be forced to upgrade the kernel version. It would be a valuable asset to be sure that we have at least the knowledge that the application level device drivers would compile and run for the new kernel, even if would not have the hundred per cent guarantee that they would work everywhere. We have already discussed about this approach, the **forward compatibility**.

The answer to the forward compatibility problem is to work using the tools and the methods of the any Open Source software project. The main component of the solution is a centralized **software repository** and the secondary component is a **compiler farm**. The below picture illustrates the proposed method for the synchronization of the application level device drivers and the ESRF/Linux version management.



1. Repository is a CVS version control system located externally to the ESRF, at SourceForge[7]. This is equivalent to the approach of the Tango control system development. It is motivated by the better collaboration between the laboratories using developing Tango.
2. The local application level distribution tree would remain as before. It is just that it would be filled by extracting a device driver **package version** from the ESRF Device Driver repository. For example, "get me the version **v1_1**".
3. Different versions are created by checking out a development version of a given device driver into the **ESRF/Linux compiler farm** for compilation and for testing on a development machine.
   Once there will be more versions than just the ESRF/Linux v1.1, make-file methods will be developed to allow the compilation of any device driver on all ESRF/Linux compiler farm machines.
4. Apart of the ordinary version control over the device driver package, there will be "obligatory" package versions (**v1_1**, **v1_2**), released together with each new ESRF/Linux version.
5. All run-time systems of the ESRF/Linux have version information incorporated to the installation image in a way that it can be used as a switch…
6. …when installing the device drivers (and applications) on the system. The installation script, for example can select from **v1_1**, **v1_2** and so on. Please note that there is no dependency whatsoever to the ESRF device driver repository but the installation is done in the "traditional way" over the NFS file system using the existing scripts, slightly modified.
7. In a similar manner, the compiler farm would be available for the application level programs, but this discussion is already beyond the scope of this document.

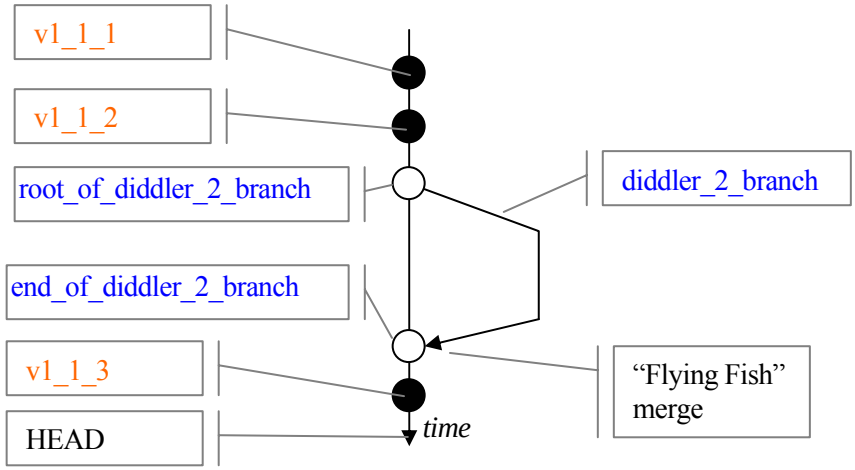
The CVS versioning of the device drivers is now "real" source code management when compared to the archive binary management of the kernel versions (see 4.3.1). The repository at the SourceForge

[7] See http://sourceforge.net/projects/tango-cs/

would be maintained by the undersigned. I would also take the responsibility to issue the "obligatory" package version every time when a new ESRF/Linux version is published. Each driver has its own responsible, who should resolve the problems if I am not able to do it. For example, the functional issues are managed completely outside of the ESRF/Linux version management.

A device driver is supposed to compile on all ESRF/Linux versions but it is **not necessarily tested** on other ESRF/Linux versions but only on the "common" run-time platform (*i.e.* ESRF/Linux 1.1). If there are incompatibility issues in the device drivers between the different versions, they will be managed at the source code level using compilation switches. This would allow keeping the device driver development at the trunk of the version control system, without creating excess amount of branches. The branching technique to use for new developments, bug fixes and such is the Flying Fish technique.

Let's take the following example – without going into the details of the CVS instructions – to better understand how an individual developer would work with the ESRF device driver repository. The below image illustrates a situation where an imaginary device driver ***diddler*** needs to be modified.



The key to the successful management of the source code projects with CVS is to follow strict tagging rules and forgot the old RCS revision numbering (which is still used behind the scene). Here we have made our first public release of the ESRF Linux device drivers. We call it – and tag it – as v1_1_1. The release contains also the first version of our example, the *diddler* device driver. Some times later we have corrected some bugs and released the next version of our package, the v1_1_2. The version means "*ESRF/Linux v1.1 compatible device driver package, second edition*". Now the *diddler* device driver developer needs to make some modifications to the driver. The responsible person creates a *Flying Fish* branch for the development work. The following should be noted:

- The repository is modular; the developer needs to check out only the module *diddler*.
- The ***C***VS is concurrent so that other people can happily work with the other modules, or even with the same one what comes to that.
- The branch follows the rules set for the tag names. All tag names are documented in the trunk's head so that anybody can extract whatever version from the repository.

Now, our developer is finished with the modifications and testing of the second version of the *diddler* device driver. The developer will merge the modifications back to the trunk, making sure that after the successful merging the return point is – you guessed it – tagged. The developer informs the repository manager (the undersigned) who will create a new release of the ESRF Linux device driver package, v1_1_3.

4.3.4 Other system modules

Other system level components, such as network protocol modules are most often provided by the software vendor. They are managed as described in section 4.3.2 *Vendor device drivers*.

It may occur that there will be system level components that are either developed at the ESRF or that are maintained by our own means. In this case these modules are managed as described in section 4.3.3 *Application level device drivers*.

4.4 Installation boot media management

The traditional installation boot media devices, such as software vendor's CD-ROM are not suitable for the ESRF/Linux installation. The installation boot media needs some ESRF specific tailoring, if not only for the possibility to install older versions but also we need to add new versions. Managing CD-ROMs, USB-keys or even floppy disks would quickly become too cumbersome.

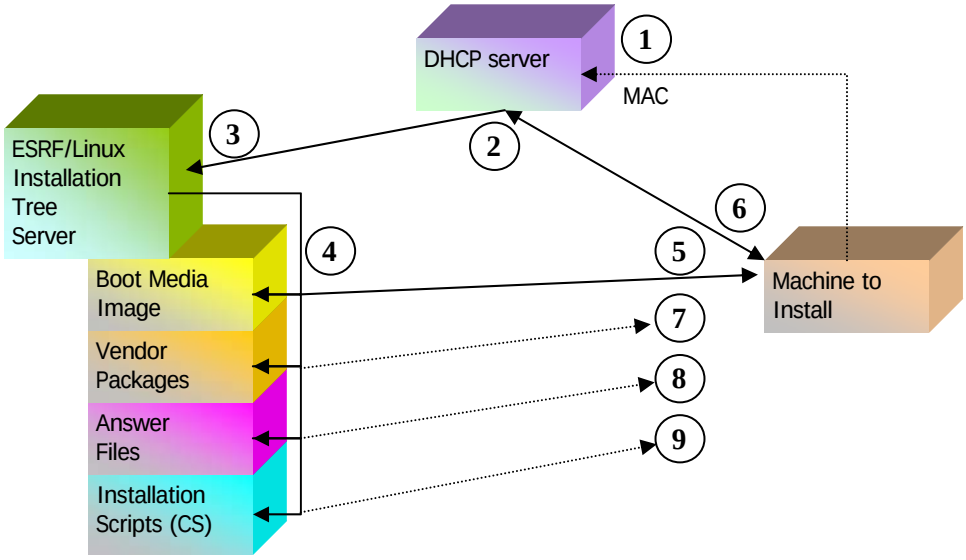
The answer to the problem is built-in about all PC computers. It is a BIOS extension called PXE (Pre eXecution Environment) which allows the computer to boot from the network instead of from the local devices. During the past five years, we have built a complete DHCP (Dynamic Host Configuration Protocol) and PXE supporting infrastructure at the ESRF. It is natural that we would design the ESRF/Linux Installation boot media management based on the existing DHCP/PXE infrastructure.

The **network based installation boot media** management has two advantages:

1. ***Create and maintain an installation database on the DHCP server***.
   ESRF/Linux is a Red Hat Enterprise Linux installation. We must know each installation of the ESRF/Linux if for nothing else but for the licensing reasons. For home users we would suggest to use Fedora distribution so that they do not need any installation media or disks for the ESRF/Linux installation.
2. ***The installation boot media can be modified on a single place***.
   New versions, bug fixes, they are all there; just reboot and try again!

4.4.1 One-off installation using Answer Files

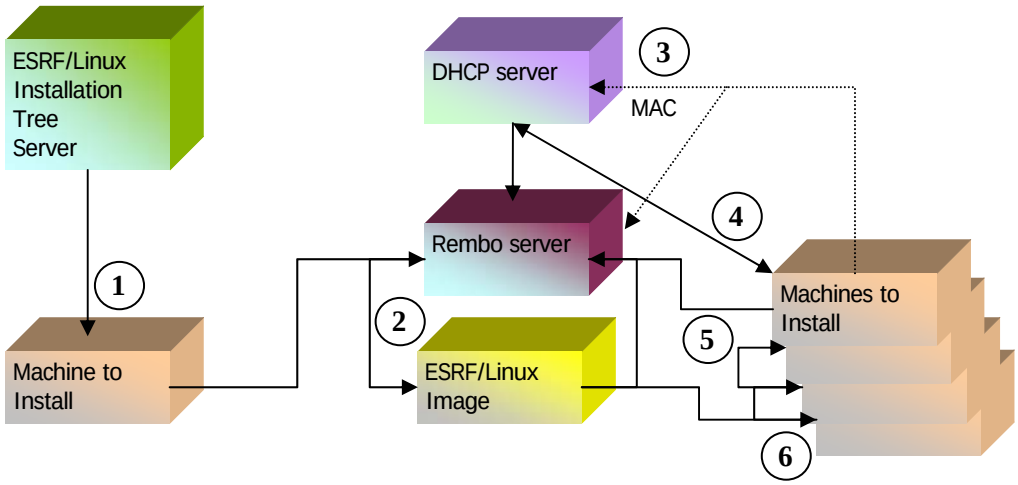
The below picture illustrates a typical installation procedure of ESRF/Linux using network based boot media and the role of the Answer Files in the installation procedure.



1. A network identity is obtained from the network group for the machine to be installed. Its MAC address and its network identity name are declared on a DHCP server.
2. The machine is started with PXE network boot. The DHCP server recognizes the MAC address and allocates a boot server for the PXE client.
3. In our case the boot server is the very same machine that is holding the ESRF/Linux Installation Tree.
4. On the server, we maintain not only the different versions of the ESRF/Linux but also a continuously evolving Boot Media Image for ESRF/Linux installation. The image is very much the same provided by the software vendor but added with the ESRF specific menus and such.
5. The installation personnel in front of the machine to be installed gets a simple menu-like structure to select from:
    a. Installation Type (beamline, machine, laboratory, office and so on).
    b. ESRF/Linux Version (by default the latest version and latest patches gets installed).
    c. Once done, the installation starts, **running automatically** from this point onwards.
6. The installation starts by a request to the DHCP server to set the network identity of the machine to be installed. This is an all automatic procedure that may involve also some additional network parameters, such as NFS file system servers to use, and other similar instructions based on the Installation Type.
7. Vendor packages and vendor patches are installed.
8. To finalize the installation, the vendor's installation program gets automatically answered to all of its questions related to the system configuration. The answers are taken from the Answer Files that are located on the installation server and in the Installation Tree. The Answer Files is selected according the selected Installation Type. For example, Red Hat has named this commonly known installation method as ***Kickstart***.
9. Computing Service's Installation Scripts will be executed to configure the system to for best integration to the ESRF computing infrastructure. There will be other, application level installation scripts to be executed, such as BLISS installation scripts but they are beyond the scope of this document.

4.4.2 Cloning based deployment using Rembo servers

As explained in the previous section we have improved the system installation stability by creating a semi-automated installation method for the ESRF/Linux. But since the underlying file system is dynamic, containing vendor patches and others, there is still room for improvement concerning the system stability. On server clusters, control systems and such, computers are likely to have exactly (or almost) the same hardware. Since they are identical, the question arises why the operating system on them cannot be identical as well. This can be achieved using system cloning techniques for the operating system deployment. During the past years, Computing Services has installed an infrastructure that allows cloning based operating system deployment over the network for Windows and Linux operating systems. The infrastructure's core elements are Rembo servers[8] and a deployment application called Rembo Wizard[9], which has been developed at the ESRF. The below picture illustrates the method that is used when systems are installed using cloning based deployment methods.



1. A machine of the selected type is installed manually with an ESRF/Linux version, either manually or using a generic Answer File, as explained in the previous section.
2. Using The Rembo Wizard, an image of the hard disk is taken and stored on a Rembo server.
3. New machines of the same type are declared both on the DHCP server and on the Rembo server using their MAC addresses.
4. They are set to boot from the network temporarily, either by modifying the BIOS settings or by hitting the F12 key when the system is starting (on Dell machines)
5. The Rembo Wizard gets in execution and launches an installation of the ESRF/Linux image. The installation is automatic and independent of the ESRF/Linux Installation Tree.
6. Once the installation is finished, the Computing Service's Installation Scripts will be executed. This is usually done manually, following the post-install instructions for the image. Application level installation scripts, such as BLISS installation scripts and other system configuration that follows is beyond the scope of this document.

4.5 Computing Services installation scripts

This section discuss the installation and configuration scripts and in particular those maintained by the Computing Services. Generally speaking, the scripts are automated tasks to tailor an installed system for specific needs, depending of the location and usage of the installed system. A typical example of such a script is the printer installation script.

[8] Rembo Toolkit product page http://www.rembo.com/products_toolkit.htm
[9] The Rembo Wizard home page http://rembowiz.sourceforge.net/

4.5.1 Configuration scripts

Configuration scripts have the task to configure the system to for best integration to the ESRF computing infrastructure. The specific needs for a machine are expressed in a configuration database file. For example, a beamline computer should do a NFS mount on certain servers.

The installation scripts are most often shell scripts than anything else. They have one common design parameter; they must be **reentrant** so that in case of doubt they can be executed "just to be sure" without breaking anything. If no action needs to be taken, the scripts just exists silently, otherwise they would report the action that was taken.

4.5.2 Patch level installation script

A new type of installation script has to be developed for the ESRF/Linux version management. The script would be responsible of the patch level installation and version management. As an example of its usage, we can imagine that we run it on an ESRF/Linux v1.1.3 system and ask it to bring the system to the patch level v1.1.6. The script should be intelligent enough to deal with the gap between the patch levels and recursively patch the system until it is at the requested level.

The patch level installation script can have an "undo" feature but only if the software vendor's patch management allows the patch removal. Otherwise this type of removal operation is too complex to manage within a script.

The fair amount of complexity of such a script requires that a higher level scripting language, such as Perl is used. The traditional library incompatibility problems are not an issue here, seeing that such a script can be used only on ESRF/Linux distributions and not on other operating systems.

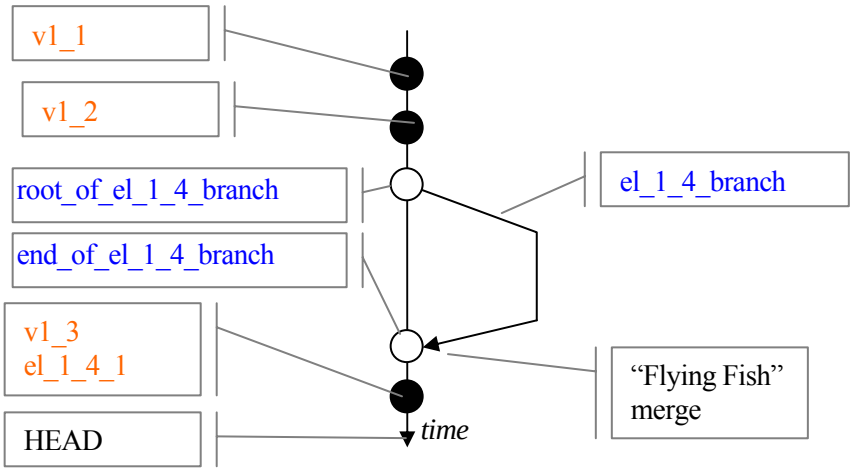
4.5.3 Application level installation scripts

BLISS and SEG would maintain their own application level installation scripts. They would remain very much the same that they are today, but the scripts must – in long term – become aware about the ESRF/Linux run-time's version number. The version number is available in the run-time in a standardized manner (see 3.5 *Obtaining version information*) and the application level installation scripts should use this information as a selector to their own, internal installation tree structure.

4.5.4 Script version management

Computing Services installation scripts for ESRF/Linux should be version controlled under the CVS version management system. The repository does not need to locate on an ESRF/Linux Installation Tree server. It is even recommended to use an external repository, so that our colleagues from the Computing Infrastructure would feel motivated to use the same repository and thus share their knowledge.

The CVS ***project*** in the CVS repository would contain **all** the Computing Service's installation scripts, and the ESRF/Linux installation scripts would be a CVS ***module*** in the repository. Therefore the version numbering of the ESRF/Linux cannot be dominating in the repository. As usual in the CVS repositories, ***tag naming rules*** will be used to distinguish the different versions and relations between the modules. Another technique to use is the *Flying Fish* branching. The below picture illustrates a management example.



The scripts project in the repository follows version numbering v1.1, v1.2, v1.3 (v1_1, v1_2, v1_3 tags), and so on, without probably never arriving to version "2". Let's take an example where we decide to publish ESRF/Linux version 1.4. We know that we need to modify a few scripts and also we have to touch some of the common scripts, used by all the scripts within the project. During the development work, we would create a *Flying Fish* branch. When merged to the trunk for publication, we would tag (and document) that spot in the project's timeline **both** as v1_3 (for the entire project with all scripts) and as el_1_4_1 for the project's approval to be used with ESRF/Linux v1.4. If an "*oops*" occurs, there can be, in a similar manner, a tag for el_1_4_2.

Once ready to publish the new ESRF/Linux v1.4 version in the Installation Tree, we would **extract** the module containing the ESRF/Linux installation scripts and the common modules using the tag el_1_4_2.

5 Managing critical updates

The policy: do not modify existing ESRF/Linux system without an explicit request.

For example, vendor patches are not applied if they are not asked for. The above principle will lead to a situation, where there are a large number of machines with ESRF/Linux with slight differences between them. This is not a problem in such, since from the ESRF/Linux versioning philosophy (see 3 *Version numbering*) follows that the differences are insignificant for all practical purposes.

But as Linux in general, and Red Hat Linux in particular is gaining popularity in the world, the ESRF/Linux installation scheme should be prepared for worse. For example, a sudden outburst of computer viruses targeted for Linux systems may require a site-wide action in order to keep the institute's computing infrastructure operational. This paper has already explained several techniques how this type can be managed. There is no doubt from the system administrator's point of view that we can rapidly upgrade the entire park of the installed ESRF/Linux running computers to, say version ***1.4.2***.

But what about the applications that were running previously only on ESRF/Linux version ***1.1.4***, will they still run? That is the real challenge which cannot be separated from the ESRF/Linux version management.

5.1 Application level forward compatibility

To reply to an eventual need to upgrade systems from early versions of ESRF/Linux to the latest one containing the critical updates, we must do the necessary to assure the application level compatibility with the future versions of the ESRF/Linux. Without pretending that it would be possible to prepare a flawless approach, we have already discussed some techniques that allow us be relatively confident on this matter:

- **Conservative policies**.
  Following very much the principle "if it works, do not fix it" the ESRF/Linux is an additional buffer between the installed park and the software vendor's (Red Hat) distribution, already built for long term stability.
- **Limited number of kernels**.
  Software vendor is already limiting the number of kernels it publishes and the ESRF/Linux is using new kernels only when necessary.
- **Application level device drivers and their kernel level follow up**.
  When a new kernel is published, it means automatically that a new ESRF/Linux version is published. New application level device driver modules will be compiled systematically for new ESRF/Linux versions. Although they cannot be tested in real life conditions there is more chance that they would eventually work because irritating compilation problems can be resolved in advance.
- **Stable compiler and run time libraries**.
  If new kernels will be introduced in limited numbers, we will be even stricter what comes to the changes with compilers and run time libraries. A common agreement with all concerning groups is required to any change in this area.
- **Compiler farm**.
  For each ESRF/Linux version there will be a reference compiling machine (no matter if the compiler and the run time libraries are supposed to be same between the two versions). The compiler farm gives the application developers tools to maintain her software in best conditions. If an urgent upgrade need will arise, there is an infrastructure already available to resolve occasional run time incompatibility problems.

6 Conclusion

The proposed version control strategy makes it possible to obtain medium term stability with the Linux operating systems at the ESRF. Instead of following the public domain distributions' three month release cycle, we base our installations on a commercial Linux product, Red Hat Enterprise Linux with much longer expected life time. The Linux installation stability is increased further with a versioning scheme, internal to the ESRF. The outcome is called *ESRF/Linux*.

The ESRF/Linux can be considered as a version control strategy, which has a commitment to provide an extremely stable run time platform for various Linux application used at the ESRF. This is not done by sacrificing the dynamic upgrade path to the latest development in the Linux world. New versions are published regularly and ESRF/Linux compiler farm machines will provide a platform to keep the applications up to date.