

# SOFTWARE DEPLOYMENT

*Markus W. Weissmann* <[mww@opendarwin.org](mailto:mww@opendarwin.org)>

Georg Simon Ohm University of Applied Sciences,  
12/21/2005

## Abstract

Software Deployment is required by every computer user. Whether you install a simple word processor or the latest compiler on a computer, you always have the same set of problems to solve. While operating systems are a well researched field of computer science, the task of caring about some thousand installed programs on a computer is left to the administrator without further investigation.

This paper is an effort to materialize and condense the problems of a software deployment system that has to care about real software. All the software that is programmed all over the globe does not behave according to some predefined rules but follows different ways to achieve it's goals. Sometimes these procedures are rather weird and inconsistent. Though they have to be brought together to leverage the potential of thousands of programs.

Everybody trying to understand the workings of tools like the RPM package manager<sup>1</sup>, NetBSD's pkgsrc<sup>2</sup> or the VISE installer<sup>3</sup> should first try to have a look at the goals that these systems are trying to achieve. Programmers that are going to make use of certain libraries or code-generators also are advised to gain a basic knowledge in this process, so they can avoid pitfalls when their work finally is put into action. Administrator even more need to understand their tools of trade, and they all are going to achieve the same goals, be it on a Windows operating system, Linux or Solaris. Operating system designers can make the user experience of their OS much brighter when they provide their users with a powerful and easy to use software deployment system.

---

<sup>1</sup> RPM package manager, <http://www.rpm.org/>

<sup>2</sup> NetBSD pkgsrc, <http://www.netbsd.org/Documentation/pkgsrc/>

<sup>3</sup> MindVision VISE, <http://www.mindvision.com/>

<i>Introduction</i>	4
<i>Software Deployment</i>	4
<i>Problems of Software Deployment</i>	4
<i>General Solving Approaches</i>	6
1. Get the Sources	6
2. Verify the Sources	6
3. Required Patched	6
4. Required Build Tools	6
5. Required Libraries and Servers	6
6. Parameterization of Build Environment	7
7. Build Resources	7
8. Detect Conflicts	7
9. Install	7
10. Upgrade	7
11. Uninstall	8
12. Robustness	8
<i>DarwinPorts</i>	8
Overview	8
User Experience	9
Internal mechanisms	9
Summary of port(r)	10
<i>Summary</i>	<b>II</b>

## Introduction

At first glance, software deployment seems trivial. The foundations of modern operating systems are well understood, from file systems to dynamic libraries. Build tools like compilers are also a successful field of research for decades now. The only step missing from transforming the source code into an installed program is copying some files around, right? Probably every serious system administrator will disagree.

As soon as the theoretic fields are left, a plethora of problems appear: Will your program only run on your development machine? What is needed to get it up and running on a users system? What about other programs that the user may use? If the user has a different operating system - maybe just a different version - will your software work, too? How do you handle earlier installations of your program?

Clearly software deployment is the field of pragmatics. For all operating systems that deploy software in a large scale there are automation frameworks to handle at least some of the problems. For Microsoft Windows there are some graphical installation programs, Linux mostly uses package managers; Solaris, SCO-Unix, Mac OS X and HP-UX rely on primitive package formats while the BSD-Unices use a system called ports collection. Most of these systems are hacks that were the quick solution to a concrete problem and then evolved over years to handle other deployment problems, too.

This paper will try to outline a theory for software deployment, create a requirement analysis for fulfilling the users needs. After describing the problems that need to be solved, some solutions will be shown. We will focus on already proven to work solutions that are used by existing frameworks. Then the product DarwinPorts will be analyzed in how far it solves the challenges.

## Software Deployment

### PROBLEMS OF SOFTWARE DEPLOYMENT

As said, there is no grand unified theory of software deployment yet. Nevertheless current software deployment frameworks seem to get a grip on the problem. Therefore it seems like a good start to collect the problems that they are trying to solve. From several years of experience with systems of this kind, I collected the following list of problems which appear to cover all steps that a perfect deployment framework should solve:

1. Get the Sources
2. Verify the Sources
3. Required Patches
4. Required Build Tools

5. Required Libraries and Servers
6. Parameterization of Build Environment
7. Required Build Resources
8. Detection and Handling of Conflicts
9. Installation
10. Upgrade
11. Uninstall
12. Robustness

This list of tasks takes you from the point where you finished programming to the point when your end-user has the software up and running on her machine (points 1 to 9). The lists though goes further to also solve the problems the user might have after the initial installation (points 10, 11, 12). When looking at closed source software, most often the points 1 to 7 are already solved for the users - at least when using the intended operating system and environment. Therefore to not focus on the problems 8 to 12 only, we will concentrate on open source systems where the whole process is more transparent for us.

To fill these problems with some more life, lets walk through a simple example: Imagine a friend of you calling you late at night, because he has a programming task due to the next day. He has to write a hello-world program and remembered that you wrote a hello-world library some time ago. You agree on mailing him the library source, solving problem 1. As he knows your mail-address and just talked to you about it, he can be quite sure that the sent code is the source he wants (2). As your program was written for lets say Linux and you know your friend is on Solaris, you write into your mail that he needs to change a certain include-statement - this will solve 3. Luckily for him, he has the required C-compiler installed on his system already and your plain-C code doesn't need any libraries, so challenge 4 and 5 are of no worry. As a bonanza for your friend your code builds fine (6) in just a few seconds (7), as your source is really short. He then copies (9) your library and header file into his home directory - there is no file named "libhello.a" there yet (8). After getting his program done, your friend just deletes the library (11). Fortunately he didn't forget the header file (12).

As we see on this simple example, solving all these problems by hand can get quite out of hand. If you want to install the library again on a different machine, you have to repeat all steps again. This alone is error prone, and initially your friend got a very good start with you telling him about the pitfalls in advance. Also this example didn't have curious requirements, neither on libraries nor on build tools. Your friend knew in advance which library he needed, it was very small and therefore quick to compile and easy to install and

uninstall. It is easy to imagine that with more complex software to be deployed, this problem is no more a task for a human but for a computer.

## GENERAL SOLVING APPROACHES

The goal in solving the problems of software deployment is to automate the process as far as possible. The ultimate solution would take as input only the wish of it's user and then deploy the demanded software. This should take as least time as possible, should be revertible, make updated easy and brake very rarely. Let's have a look at the problems one by one.

### 1. Get the Sources

The omnipresence of the internet makes it possible to reduce this problem to simply providing an URL. This URL though needs to have some key/value pairs that make it possible for the user to identify the software in the first place. The user must have a way of querying this database to get from the abstract URL to more useful information, information on which she then can decide which programs to deploy. When the user made her decision, it is technically quite easy to fetch a file from an URL from the internet or a local file system.

### 2. Verify the Sources

To make sure that the referenced sources are the demanded ones, there are basically two possibilities: Trust the location or URL alone. This is possible if the URL is under the deployment systems control, e. g. if it is on a local file-system, for example a trusted DVD-ROM. The second way of bringing trust in the relationship between the user and the referenced URL is to verify the downloaded file. This can be done via it's size, by having a checksum or by checking the signature of the file - if it is signed. The information like the checksum has to be provided by the deployment system.

### 3. Required Patched

As most software only is made for and tested on certain machines, it is quite often needed to patch it, so it will run on another platform. This is a task that can barely be fully automated. In practice humans, who know about both, the original and the destination platform, write patches. Due to this, the remaining solution is to collect these patches in a database and then provide them automatically to the user.

### 4. Required Build Tools

To get into an executable form, most programs need some build tools, be it a compiler, an interpreter or simply a generator tool like make. It is in general hard to guess what tools a certain piece of software needs to get compiled. If the build is successful one may know what tools were needed, but in general, this has to be looked up somewhere, by reading the source, browsing the documentation or by querying a database which knows.

### 5. Required Libraries and Servers

The goal to success in this area is very similar to 4.: You may find out which libraries you need by looking at the source automatically, though this would require a very powerful analyzer for each supported programming language and a large database of known libraries and their signatures, like C-header files or Java class-names. Apart from being a mammoth project to generate the database and analyzers, the database probably would not be useable as signatures of libraries are not unique: The real world features quite some conflicting libraries, at least different but incompatible versions of the same library would lead to conflicts here. The only reasonable solution again is a database, which knows about these dependencies.

#### 6. Parameterization of Build Environment

Automation again can practically only be solved by depositing the information in some kind of database. The result could be guessed by trying popular environments, but if none of them works, there is no repair-algorithm which would provide useful settings.

#### 7. Build Resources

The amount of resources needed to build a certain piece of software can be vast. Especially if it has a large code-base in a language like C, which needs to be compiled. To reduce the amount of time a user needs to compile such a program you can either use faster compilers and faster computers in his environment, or remove the burden of compiling from him altogether. This can be achieved by providing him the binaries directly e. g. via downloadable archives.

#### 8. Detect Conflicts

Conflicts may happen if multiple pieces of software try to use the same non-shareable resources on the local installation. This might be just file-paths, network ports or even whole devices. This can partially be discovered in advanced, as least as far as duplicate files are concerned. By remembering the already installed files and checking for conflicts in advanced, this problem can be solved rather easily if it is known in advanced which files will be installed.

#### 9. Install

Many open source projects provide their programs with installation routines, a circumstance that makes solving the installation-problem much easier. In fact the framework only needs to know which files to place where on the local filesystem. For some software there are additional requirements though, e. g. creation of local user accounts. If the provided installation routine covers the steps needed, this problem can be shoved off to them.

#### 10. Upgrade

The problem of software-upgrades seems easy to solve if both install and uninstall are available. This is true in theory but may well be inappropriate on productive systems. The

downtime of such a computer has to be kept as low as possible, therefore the upgrade of a software has to happen as fast as can. It is in general not an option to wait for steps 1 to 8 get solved by the system while in the meantime e. g. a database server is unavailable. Two options are available: Either having pre-packaged archives or preparing the successor-version concurrently while the predecessor is available.

## 11. Uninstall

The uninstall-operation basically has to revert the state of the system back to the one before the installation. Remembering the installed files is rather easy; more complicated tasks are how to handle changed files, currently running processes, shared resources that are owned by multiple deployed entities. Apart from just deleting the installed files, this process is full of exceptions for special cases. This strongly depends on the kind of software installed.

## 12. Robustness

While problems 1 to 11 are a rather linear, process-orientated list, robustness is a more general requirement. Requesting this feature from a software deployment system means that it needs to take care of more hideous problems. This are is very wide and ranges from overcoming user errors, over handling external effects to minimizing effects that come from programming errors in the deployment system itself. The system can for example keep hashes of installed files to check if they were changed by an external incident. Backups of the deployment systems own databases, that keep track of software installed, can help restore the system if it crashes during database-operations or if the file-system was damaged.

## DARWIN PORTS

### Overview

DarwinPorts<sup>4</sup> was initially brought to life by Apple Computers. Apple released their new Unix-based operating system Mac OS X in 2001. Back then not much software was available for it. In an effort to changed this situation at the open source software frontier, a system similar to the FreeBSD-ports<sup>5</sup> was launched. As the core-system of Mac OS X<sup>6</sup>, which actually is open source itself, is called Darwin<sup>7</sup>, this system was called DarwinPorts. A port references a ported piece of software, basically a database entry on how this



Hexley, the DarwinPorts mascot

---

<sup>4</sup> DarwinPorts project, <http://www.darwinports.org/>

<sup>5</sup> FreeBSD ports, <http://www.freebsd.org/ports/>

<sup>6</sup> Apple Mac OS X, <http://www.apple.com/macosx/>

<sup>7</sup> OpenDarwin project, <http://www.opendarwin.org/>

program can be deployed successfully on the Darwin operating system.

Currently this project is run by an elected board of 3 so-called port-managers. There are round about 50 developers working on the system itself, on keeping ports up-to-date, creating new ones and also on writing documentation. Meanwhile the number of ports has risen to over 3000, making this system probably the biggest open source deployment system for the Mac.

### User Experience

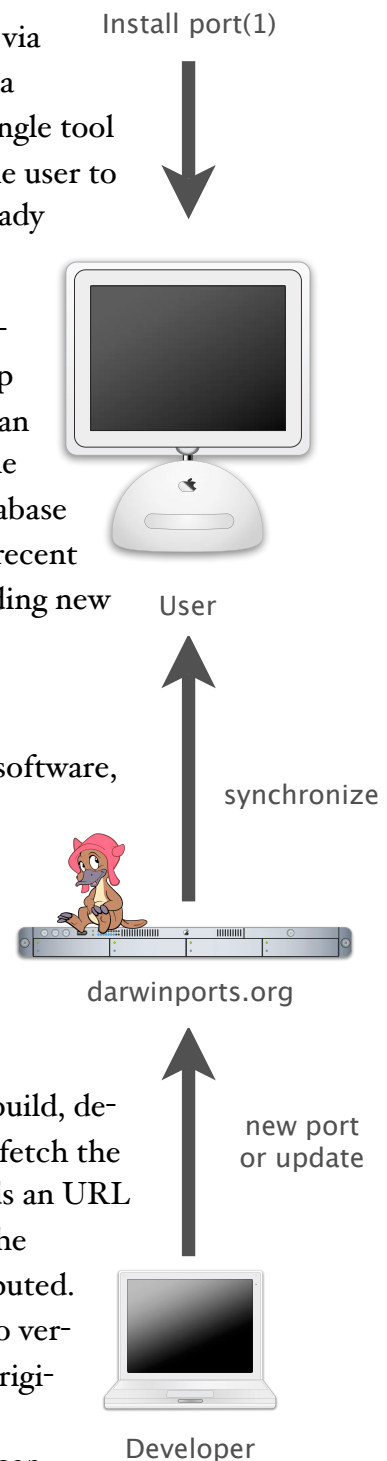
To bootstrap the system initially, it has to be installed either via Mac OS X native but rather primitive package manager or via installing it from source manually. DarwinPorts provides a single tool named port(1) that performs all required actions. It allows the user to search for ports in the database of the system, check for already installed software and of course deploy programs.

DarwinPorts makes heavy use of it's internal database to perform the required actions. Because of this it is crucial to keep it up-to-date. The port-tool makes this easy for the user: It can synchronize its local database with the online database on the darwinports.org server. The developers keep this central database up-to-date by updating the port-descriptions with the most recent versions of the programs, fixing bugs in them and also by adding new ones.

### Internal mechanisms

To describe the required actions to install a certain piece of software, DarwinPorts uses a meta-language interpreted by port(1). When installing software, port(1) runs through eight stages. There are default actions for each stage that can be overridden in a so called Portfile. If one stage fails to complete due to an error, port(1) halts, requiring user intervention to continue.

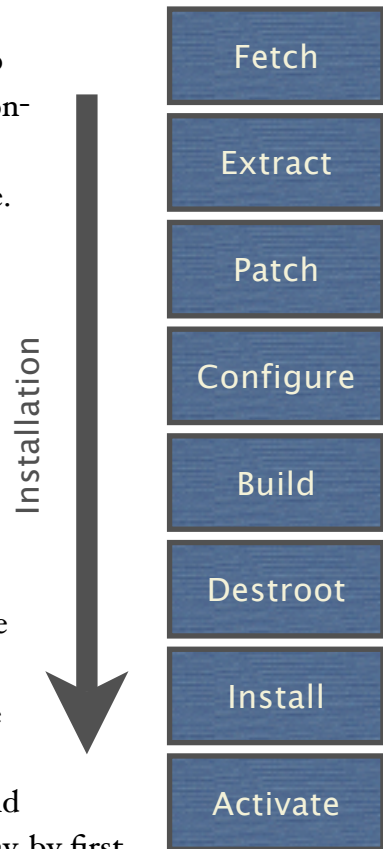
The stages run through are fetch, extract, patch, configure, build, destroot, install and activate. In the first phase, port(1) tries to fetch the source of the port to be installed. The internal database holds an URL for this operation, so solving challenge 1. Before extracting the source-archive, a checksum of the downloaded file(s) is computed. It has to match the provided checksum from the database, to verify that the downloaded file is identical to the one that the original port author worked with (solving problem 2). After that, patches can be applied if required. In this phase, the author can



care about problem 3 and solve it without user interaction.

The missing step, before the compilation can be started, is to setup the environment. There also is a stage for this called configure, caring about problem 6. The compilation itself is currently burdened on the user's machine during the build phase. So port(1) does not solve challenge 7.

When the compilation is finished, port(1) copies the files into a so called "destrout". This is a directory that looks very much like the final destination directory of the files. With this approach it is quite easy to find out which are the files that belong to a certain port, by remembering all that are currently in the destrout. The final installation process happens after the destrout stage, during the install phase. Here the files are copied to their destination in the file system. Afterwards there is a final stage called activation. It makes links from the files to the point in the file system were they are expected by other software. This approach makes it easy to keep software installed while being able to activate and deactivate it quickly. Upgrading ports is also very easy this way, by first installing the successor version, deactivating the predecessor and then activating the already installed most recent installation. This mechanism gives a solution for the problems 9, 10 and 11.



Before any port is being build, the system also checks for required dependencies. If not already installed, the required programs get pulled in recursively via port(1) itself. The build tools may be uninstalled afterwards, while the libraries and servers that are listed as requirements in the port-database must not. Here the requirements problems 4 and 5 are taken care of.

In terms of robustness, port(1) has only few to offer. The most-useful feature in this area is, that a user may quickly heal an activated port if she discovers errors with the installed links. By deactivating and re-activating a port, the links get refreshed, repairing damaged files.

### Summary of port(1)

DarwinPorts features good solutions to 10 out of 12 problems in the field of software deployment. It lacks in not providing pre-compiled archives and cannot do any kind of self-examination nor any self-healing operations.

For not mission-critical systems, DarwinPorts is quite appropriate, especially given the fact that many commercial software deployment systems are by far less user-friendly, robust and feature-rich.

## Summary

Software deployment is an existential problem with modern operating systems and the huge software collections that want to get administrated there. We saw that the challenges provided by it are non-trivial but solvable by process-orientated divide and conquer. Quite some of the twelve presented challenges need a database-lookup to get solved, few though can be solved fully by the computer on it's own. Perhaps some more of the described problem can be automated further or described more easily by a maintainer of a program.

The correct use of software deployment also makes it possible to install, keep track-of and keep up-to-date a much larger software base than a manual processing could do. But not only makes it bigger systems manageable, but also reduces the amount of time a system administrator has to spend on a system. Faster updates also mean faster bug-fixes, a point that can save millions of Euros if otherwise - without a quick update - an enterprise-wide server would be damaged by a worm or if business-relevant data got stolen by an intruder.

The problem that remains after all is keeping some database up-to-date. Developing even better domain specific languages for describing deployable programs will make it possible for few people to care about a database of some thousand programs, providing their users with an excellent systems.