Dependency hell

Dependency hell is a colloquial term for the frustration of some software users who have installed software packages which have dependencies on specific versions of other software packages.[1]

The dependency issue arises around *shared* packages or libraries on which several other packages have dependencies but where they depend on different and incompatible versions of the shared packages. If the shared package or library can only be installed in a single version, the user may need to address the problem by obtaining newer or older versions of the dependent packages. This, in turn, may break other dependencies and push the problem to another set of packages.

Contents

Problems

Dependency hell takes several forms:

Many dependencies

An application depends on many libraries, requiring lengthy downloads, large amounts of disk space, and being very portable (all libraries are already ported enabling the application itself to be ported easily). It can also be difficult to locate all the dependencies, which can be fixed by having a repository (see below). This is partly inevitable; an application built on a given computing platform (such as Java) requires that platform to be installed, but further applications do not require it. This is a particular problem if an application uses a small part of a big library (which can be solved by code refactoring), or a simple application relies on many libraries.[2]

Long chains of dependencies

If `app` depends on `liba`, which depends on `libb`, ..., which depends on `libz`. This is distinct from "many dependencies" if the dependencies must be resolved manually (e.g., on attempting to install `app`, the user is prompted to install `liba` first. On attempting to install `liba`, the user is then prompted to install `libb`, and so on.). Sometimes, however, during this long chain of dependencies, conflicts arise where two different versions of the same package are required[3] (see **conflicting dependencies** below). These long chains of dependencies can be solved by having a package manager that resolves all dependencies automatically. Other than being a hassle (to resolve all the dependencies manually), manual resolution can mask dependency cycles or conflicts.

Conflicting dependencies

If `app1` depends on `libfoo 1.2`, and `app2` depends on `libfoo 1.3`, and different versions of `libfoo` cannot be simultaneously installed, then `app1` and `app2` cannot simultaneously be used (or installed, if the installer checks dependencies). When possible, this is solved by allowing simultaneous installations of the different dependencies. Alternatively, the existing dependency, along with all software that depends on it, must be uninstalled in order to install the new dependency. A problem on Linux systems with installing packages from a different distributor (which is not recommended or even supposed to work) is that the resulting long chain of dependencies may lead to a conflicting version of the C standard library (e.g. the GNU C Library), on which thousands of packages depend. If this happens, the user will be prompted to uninstall all those packages.

Circular dependencies

If `application A` depends upon and can't run without a specific version of `application B`, but `application B`, in turn, depends upon and can't run without a specific version of `application A`, then upgrading any application will break another. This scheme can be deeper in branching. Its impact can be quite heavy, if it affects core systems or update software itself: a package manager (A), which requires specific run-time library (B) to function, may brick itself (A) in the middle of the process when upgrading this library (B) to next version. Due to incorrect library (B) version, the package manager (A) is now broken- thus no rollback or downgrade of library (B) is possible. The usual solution is to download and deploy both applications, sometimes from within a temporary environment.

Package manager dependencies

It is possible[4] for dependency hell to result from installing a prepared package via a package manager (e.g. APT), but this is unlikely since major package managers have matured and official repositories are well maintained. This is the case with current releases of Debian and major derivates such as Ubuntu. Dependency hell, however, can result from installing a package directly via a package installer (e.g. RPM or dpkg).

Diamond dependency

When a library A depends on libraries B and C, both B and C depend on library D, but B requires version D.1 and C requires version D.2. The build fails because only one version of D can exist in the final executable

Package managers like yum,[5] are prone to have conflicts between packages of their repositories, causing dependency hell in Linux distributions such as CentOS and Red Hat Enterprise Linux.

Solutions

Version numbering

A very common solution to this problem is to have a standardized numbering system, wherein software uses a specific number for each version (aka *major version*), and also a subnumber for each revision (aka *minor version*), e.g.: **10**.1, or 5.**7**. The major version only changes when programs that used that version will no longer be compatible. The minor version might change with even a simple revision that does not prevent other software from working with it. In cases like this, software packages can then simply request a component that has a particular major version, and *any* minor version (greater than or equal to a particular minor version). As such, they will continue to work, and dependencies will be resolved successfully, even if the minor version changes. Semantic Versioning (aka "SemVer" [6]) is one example of an effort to generate a technical specification that employs specifically formatted numbers to create a software versioning scheme.

Private per application versions

Windows File Protection introduced in Windows 2000 prevented applications from overwriting system DLLs. Developers were instead encouraged to use "Private DLLs", copies of libraries per application in the directory of the application. This uses the Windows search path characteristic that the local path is always prioritized before the system directory with the system wide libraries. This allows easy and effective shadowing of library versions by specific application ones, therefore preventing dependency hell.[7]

Side-by-side installation of multiple versions

The version numbering solution can be improved upon by elevating the version numbering to an operating system supported feature. This allows an application to request a module/library by a unique name *and* version number constraints, effectively transferring the responsibility for brokering library/module versions from the applications to the operating system. A shared module can then be placed in a central repository without the risk of breaking applications which are dependent on previous or later versions of the module. Each version gets its own entry, side by side with other versions of the same module.

This solution is used in Microsoft Windows operating systems since Windows Vista, where the Global Assembly Cache is an implementation of such a central registry with associated services and integrated with the installation system/package manager. Gentoo Linux solves this problem with a concept called slotting, which allows multiple versions of shared libraries to be installed.[8]

Smart package management

Some package managers can perform smart upgrades, in which interdependent software components are upgraded at the same time, thereby resolving the major number incompatibility issue too.

Many current Linux distributions have also implemented repository-based package management systems to try to solve the dependency problem. These systems are a layer on top of the RPM, dpkg, or other packaging systems that are designed to automatically resolve dependencies by searching in predefined software repositories. Examples of these systems include Apt, Yum, Urpmi, ZYpp, Portage, Pacman and others. Typically, the software repositories are FTP sites or websites, directories on the local computer or shared across a network or, much less commonly, directories on removable media such as CDs or DVDs. This eliminates dependency hell for software packaged in those repositories, which are typically maintained by the Linux distribution provider and mirrored worldwide. Although these repositories are often huge, it is not possible to have every piece of software in them, so dependency hell can still occur. In all cases, dependency hell is still faced by the repository maintainers.[4]

PC-BSD, up to and including version 8.2, a predecessor of TrueOS (an operating system based on FreeBSD) avoids dependency hell by placing packages and dependencies into self-contained directories in */Programs*, which avoids breakage if system libraries are upgraded or changed. It uses its own "PBI" (Push Button Installer) for package management.[9]

Installer options

Because different pieces of software have different dependencies, it is possible to get into a vicious circle of dependency requirements, or an ever-expanding tree of requirements, as each new package demands several more be installed. Systems such as Debian's Advanced Packaging Tool can resolve this by presenting the user with a range of solutions, and allowing the user to accept or reject the solutions, as desired.

Easy adaptability in programming

If application software is designed in such a way that its programmers are able to easily adapt the interface layer that deals with the OS, window manager or desktop environment to new or changing standards, then the programmers would only have to monitor notifications from the environment creators or component library designers and quickly adjust their

software with updates for their users, all with minimal effort and a lack of costly and time-consuming redesign. This method would encourage programmers to pressure those upon whom they depend to maintain a reasonable notification process that is not onerous to anyone involved.

Strict compatibility requirement in code development and maintenance

If the applications and libraries are developed and maintained with guaranteed downward compatibility in mind, any application or library can be replaced with a newer version at any time without breaking anything. While this does not alleviate the multitude of dependency, it does make the jobs of package managers or installers much easier.

Software appliances

Another approach to avoiding dependency issues is to deploy applications as a software appliance. A software appliance encapsulates dependencies in a pre-integrated self-contained unit such that users no longer have to worry about resolving software dependencies. Instead the burden is shifted to developers of the software appliance.

Portable applications

An application (or version of an existing conventional application) that is completely self-contained and requires nothing to be already installed. It is coded to have all necessary components included, or is designed to keep all necessary files within its own directory, and will not create a dependency problem. These are often able to run independently of the system to which they are connected. Applications in RISC OS and the ROX Desktop for Linux use application directories, which work in much the same way: programs and their dependencies are self-contained in their own directories (folders).[10]
This method of distribution has also proven useful when porting applications designed for Unix-like platforms to Windows, the most noticeable drawback being multiple installations of the same shared libraries. For example, Windows installers for gedit, GIMP, and XChat all include identical copies of the GTK+ toolkit, which these programs use to render widgets. On the other hand, if different versions of GTK+ are required by each application, then this is the correct behavior and successfully avoids dependency hell.

Platform-specific

On specific computing platforms, "dependency hell" often goes by a local specific name, generally the name of components.

- DLL Hell – a form of dependency hell occurring on Microsoft Windows.
- Extension conflict – a form of dependency hell occurring on the classic Mac OS.
- JAR hell – a form of dependency hell occurring in the Java Runtime Environment before build tools like Apache Maven solved this problem back in 2004.
- RPM hell – a form of dependency hell occurring in the Red Hat distribution of Linux and other distributions that use RPM as a package manager.[11]

See also

- Configuration management – techniques and tools for managing software versions
- Coupling – forms of dependency among software artifacts
- Dynamic dead code elimination
- Package manager
- PBI
- Software appliance

- Nix package manager
- Left-pad

References

1. Michael Jang (2006). *Linux annoyances for geeks* (https://archive.org/details/linuxannoyancesf0000jang). O'Reilly Media. p. 325 (https://archive.org/details/linuxannoyancesf0000jang/page/325). ISBN 9780596552244. Retrieved 2012-02-16.
2. Donald, James (2003-01-25). "Improved Portability of Shared Libraries" (https://web.archive.org/web/20070926130800/http://www.princeton.edu/~jdonald/research/shared_libraries/cs518_report.pdf) (PDF). Princeton University. Archived from the original (http://www.princeton.edu/~jdonald/research/shared_libraries/cs518_report.pdf) (PDF) on 2007-09-26. Retrieved 2010-04-09.
3. Stevens, Al (2001-05-01). "It's Good Work When You Can Find It; The Dependency Carousel" (https://web.archive.org/web/20110811080730/http://drdobbs.com/blogs/architecture-and-design/228700267). *J-DDJ*. www.drdobbs.com/blog. **26** (5): 121–124. ISSN 1044-789X (https://www.worldcat.org/issn/1044-789X). Archived from the original (http://www.drdobbs.com/blog/archives/2008/12/its_good_work_w.html) on 2011-08-11. Retrieved 2010-04-10.
4. Pjotr Prins; Jeeva Suresh & Eelco Dolstra (2008-12-22). "Nix fixes dependency hell on all Linux distributions" (https://web.archive.org/web/20150708101023/http://archive09.linux.com/feature/155922). linux.com. Archived from the original (http://archive09.linux.com/feature/155922) on 2015-07-08. Retrieved 2013-05-22. "*All popular package managers, including APT, RPM and the FreeBSD Ports Collection, suffer from the problem of destructive upgrades. When you perform an upgrade -- whether for a single application or your entire operating system -- the package manager will overwrite the files that are currently on your system with newer versions. As long as packages are always perfectly backward-compatible, this is not a problem, but in the real world, packages are anything but perfectly backward-compatible. Suppose you upgrade Firefox, and your package manager decides that you need a newer version of GTK as well. If the new GTK is not quite backward-compatible, then other applications on your system might suddenly break. In the Windows world a similar problem is known as the DLL hell, but dependency hell is just as much a problem in the Unix world, if not a bigger one, because Unix programs tend to have many external dependencies.*"
5. "Yum Dependency Hell" (https://web.archive.org/web/20161219072303/http://www.techbrown.com/fix-centos-rhel-fedora-yum-dependencies-hell-problem.shtml). Archived from the original (http://www.techbrown.com/fix-centos-rhel-fedora-yum-dependencies-hell-problem.shtml) on 2016-12-19. Retrieved 2015-12-28.
6. "Project website: semver.org" (https://semver.org).
7. Anderson, Rick (2000-01-11). "The End of DLL Hell" (https://web.archive.org/web/20010605023737/http://msdn.microsoft.com/library/techart/dlldanger1.htm). microsoft.com. Archived from the original (http://msdn.microsoft.com/library/techart/dlldanger1.htm) on 2001-06-05. Retrieved 2010-07-07.
8. Slotting (https://devmanual.gentoo.org/general-concepts/slotting/index.html) on gentoo.org
9. pbiDIR (https://web.archive.org/web/20130327233344/http://www.pbidir.com/)
10. "Application directories" (http://rox.sourceforge.net/desktop/AppDirs.html). Retrieved 7 September 2013.
11. Weinstein, Paul (2003-09-11). "Is Linux Annoying?" (http://linuxdevcenter.com/pub/a/linux/2003/09/11/linux_annoyances.html). linuxdevcenter.com. Retrieved 2010-04-10.

External links

- Context independence (http://c2.com/cgi/wiki?ContextIndependence)
- Dependency walker (http://www.dependencywalker.com/)

- Implicit dependency (http://www.scons.org/doc/0.97/HTML/scons-user/x933.html)
- MacDependency (https://archive.today/20130102230355/http://macdependency.googlecode.com/)
