Package Universes Architecture
Fri, 26 May 2006
(Revision: 7613)

Alexander Spoon (lex@lexspoon.org)

1 Preamble

This is a living document describing the package universes architecture. The canonical web location for this document is currently the Scala Bazaars web site:

```
http://scala.epfl.ch/~spoon/sbaz
```

Scala Bazaars is a running prototype used as infrastructure for the Scala [9, 4] open-source community.

Feedback should be directed either to the author or to the Scala mailing list.

2 Introduction

Many long-term collaborations can be described in terms of developing a shared library of content that is stored in discrete packages. Examples include the Debian's Advanced Package Tool [11] and Squeak's SqueakMap for managing software packages, Super-Swikis for managing Etoys content [12], and Wikis for managing written articles [6]. Frequently the individual packages evolve over time: software gains new features and bug fixes, media content is updated, and articles are edited. Additionally, this content frequently includes dependencies. A software package for email reading might depend on an HTML parser, and an active essay on fractals might require that a 2D plotting library is loaded before the essay.

Processes and access control for such collaborations vary widely from community to community. Many communities want wide-open freedom to edit and contribute. Others, such as software development teams and individual classrooms, allow complete freedom to members of the community but no access to outsiders. Still others have more sophisticated policies where there are different groups allowed to submit, commit, and access content in the shared space. The precise rules depend on the community.

This paper describes *package universes*, an architecture for supporting these package-distribution collaborations. The architecture is unusual for focussing on communal sets of packages instead of on individual packages. Packages are designed to be compatible with a particular set of packages. Users are assumed to work within one single package universe, which sets a context or scope that simplifies several aspects of packages and the distribution system.

3 Philosophy and strategy

3.1 Package universes exist

Philosophically, package universes exist whether or not they are treated explicitly by the architecture. There are at least two fundamental reasons. First, packages are only *convenient* for a user when they have been *configured* consistently with other other packages the user wishes to use. Packages that install files into different or even conflicting locations in a file system cannot be conveniently installed, but instead must fundamentally require configuration after installation. Even if someone could define a convention for every conceivable option (which is not possibly anyway, at least for software package systems), such a set of conventions would constitute a distribution. Other systems would doubtless use different conventions, thus reopening the original problem.

The second reason is that packages are only *reliable* for a user when they are *tested* with respect to other packages the user is interested in. Some software errors only appear in response to combinations of packages. One combination might work fine, while another exhibits new errors.

For both reasons, packages are only convenient and reliable with respect to other packages. Therefore, the set of all possible packages naturally partitions into smaller sets of mutually compatible packages.

3.2 Explicit management of package universes

The package universes architecture goes further and explicitly models package universes. Users must designate which universe their packages will be drawn from. This assumption adds a restriction to how users operate, though hopefully not an onerous one. It is as if each community of users operates within a limited bubble universe, only able to see packages that are within their own universe. If a user wishes to use a package from a foreign universe, then someone—possibly the user—first needs to modify the package to be consistent with the local universe and add the modified version to the local universe.

This assumption yields a number of simplifications in other parts of the architecture. For example, package names can be short. They do not need to include DNS domain names or otherwise attempt to be globally unique. Names can be short because they are all used within the context of one universe, and each universe has a presumably cooperative set of users (communities whose users attack each other have larger problems than their infrastructure!). Debian [8, 2] shows that short human-readable names, names that do not, for example, include embedded DNS host names, are practical for communal libraries at least as large as 10,000 packages.

As an aside, the single universe-wide scope is intended to support a person's workspace. It is not a suggestion or endorsement that the packages themselves should refer to each other via the universe-wide names. Instead, it is intended to allow a human user a terse name for each package in their local library, thereby supporting creative development.

Another simplification is that versions can be totally ordered. Branching versions are not required. Users within the universe are assumed to cooperate and to want the same update policy for their packages. Thus, all users of the universe can update their package in tandem, instead of some users operating on different branches of development from others.

The dependency system, in turn, can be as simple as "depends on B." Versioned dependencies, such as "depends on B newer than version 1.4," are not needed, because users tend to update to the newest version of all packages anyway. A more refined dependency system is described below, but this approach appears sufficient for practical usage. The sufficiency of such a simple dependency system follows from the fundamental design decision that each universe includes a limited set of carefully chosen packages. Because of this design, software can simply select the newest version of any needed package without needing to select among a large number of versions.

It is true that this simple versioning approach does not directly support version changes that intentionally break compatibility. However, because the universe has a limited user base, members of the community can collude to avoid the problem. Users can use multiple package names to represent the incompatible components. For example, if "libc version 5.100" and "libc version 6.101" break compatibility, then they could be considered separate packages by naming them "libc5 version 100" and "libc6 version 101."

In general, any dependency problems that arise can be worked around by modifying the contents of the package universe. That is, instead of requiring the dependency system to handle an arbitrary set of packages, the package universes approach is to let humans engineer the set of packages so that they have simply dependencies. Instead of solving harder problems, community members can work together to make the problems easier.

The explicit universes assumption also provides leverage to enforce a variety of update policies. Gatekeepers for a package universe can refuse to accept any updates that do not conform to the particular universe's policies. A number of examples are given below.

Finally, the design avoids centralized control. There is no central registry of universes. Instead, universes are developed bottom-up by individual communities that want to share code with each other. Additionally, even large widely shared universes do not have complete control over those who participate. Local communities can build a universe based on a larger universe which incorporates tweaks that the local community desires. Examples of such combinations are also described below.

3.3 Evolution and updating

As community members create new content and edit old content, they contribute new packages to the universe. Thus, package universes evolve over time. Inter-package conventions and interfaces can drift as the universe evolves. For the universe's packages to remain compatible with each other, the speed of drift must not be too fast. The drift must be slow enough that users who contribute packages can update their packages to accommodate the drifting conventions.

It is assumed that users update regularly to the newest version of each package they use that is available within their universe. Users who wish to use older software are still accommodated. Such users must subscribe to a universe whose newest versions are essentially the same as the old versions. Note that even in this case, users rarely wish to use precisely the old version. Instead, they wish to use the old version along with any simple bug fixes that have been found over time. Thus, even stable universes containing "old" software tend to have a trickle of updates over time.

It is a conjecture behind the system that most communities only need a small number of universes to accommodate the update policies the users desire. For any community where the conjecture holds, the simplifications of the package universes architecture hold in force. For any community where the conjecture fails, and where users want to use a large number of combinations of old and new software, a more sophisticated approach is required. Further experience is required to determine whether this approach of multiple package universes is effective for most communities. For that matter, further experience is required to determine the most common ways, if any, that actual communities need something more.

4 Overall architecture

The overall architecture is that a package universe implements a *communal library* of *packages* on the network, while individual users have a *workspace* that includes a *local library*. The local library consists of packages selectively downloaded from the communal library.

The software components are a *universe server* maintaining the communal library and allowing it to be manipulated remotely, and a *universe client* that interfaces with human users, local libraries, and universe servers.

A package is the unit of interchange of content. It might contain any form of content, *e.g.* software, media components, or written essays. A package can be loaded and unloaded from the user's local library. Each package has a name, used to refer to the package from the workspace and (possibly) from the content of other packages, and it has a version number, used to distinguish different revisions of the same content.

There are several kinds of universes that are commonly discussed. An *index-server* universe is one whose packages are those stored explicitly on an *index server*. The contents of the index server change over time as authorized developers add, update, and remove packages. An index server corresponds closely to a repository in many Linux distributions.

Universes can be combined and wrapped into other universes in several ways. A *name-based union* combines the packages from an ordered list of universes. Packages in later universes in the list override all same-named packages from earlier in the list, including packages with the same name but a different version. A *pure union* does not override same-named, different-version packages, but such pure unions appear less useful in practice than name-based unions.

A *filter* universe includes a subset of the packages in another universe. The subset is defined by a regular expression which is matched against the package names. Such universes are useful for selecting a few packages—possibly just one package—from an index server.

The *empty* universe has no packages. It is a special case of the union of zero universes. The empty universe is a useful boundary case in theory and in implementation of package universes.

A *literal* universe includes an explicit list of packages in the universe description. Literal universes are practical in two scenarios: test suites for package-universe implementations, and small universes for personal use.

Finally, an *indirect* universe holds a URL that references another universe's description. Indirect universes are useful for communities whose package universes have a complex internal structure. Instead of community members installing complex universe descriptions on their machines, they can refer to a description located on a central community web site.

5 Package dependencies

This section describes a dependency system that appears sufficient while remaining simple. It is only partially implemented. It is described here for a few reasons. It demonstrates how simple it appears a dependency system for a package universe can be, it documents a canonical dependency system for package universes when no other is specified, and it documents the planned dependency system for Scala Bazaars in particular.

Package names are considered unique within a particular universe. Whenever two packages have the same name but a different version number, they are treated as holding different revisions of the same content. The package with the larger version number is preferred by the tools by default.

Packages may have the following relations to other packages:

- A depends on B. Package A absolutely requires that some version of package B be installed for it to function.
- A suggests B. Package A is usually used in conjunction with package B, but it is not a strict requirement.
- A provides B. When package A is installed, it provides the same functionality as package B. Packages that depend on B are satisfied if package A is installed instead. Package "B" can even be a virtual package, a package that is not concretely present in the repository but instead only appears in "provides" dependencies.

To date, only the direct depends-on relation has been implemented in Scala Bazaars. "Provides" and "suggests" are left for future work.

The source package in a dependency is versioned, so that Foo version 1.1 can depend on different packages from Foo version 1.2. However, the target package in a dependency is not versioned. If Foo 1.0 depends on Bar, then it does not get to choose which version of Bar it depends on. It is the responsibility of the community members to ensure that the newest version of Bar—along with every version of Bar if possible—is compatible with every package that depends on it.

A package's list of dependencies may be conjoined with both AND and OR operations. AND combinations allow a package to depend on multiple other packages. OR combinations, which have not yet been implemented in Scala Bazaars, give each user a choice in which packages are installed to satisfy another packages dependencies.

6 Access control

Update policies are enforced by providing an interface to primitive universes based on *capabilities* [7]. Universe servers on the network only allow operations when the requestor presents a sufficient capability to use that request. The servers themselves do not use usernames and passwords nor any other from of access-control lists.

Capabilities can be implemented as unguessably large random numbers. Holding a capability is then equivalent to knowing the large number associated with the capability. A capability can be transmitted to a user (or to a software object) by sending him or her the number. If a capability is revoked on a server, then all holders of that capability lose the authority associated with that capability.

Primitive universe servers respond to capabilities designating the following operations:

1. Download the list of package descriptions.
2. Edit package descriptions where the package name matches a fixed regular expression.
3. Create and revoke arbitrary capabilities.

The first capability is often granted to all community members, but might be denied to some users if there are multiple universe servers for the community and some of them share privileged information.

The second capability allows access to packages based on their names. There is no mechanism to grant access to specific versions of a package, nor to separate editing operations on packages, because these are all expected to be given out in groups. Further, a regular expression is allowed for matching the package name, thus allowing granting access to multiple packages at once. For example, a typical regular expression is `scala-.*`, which grants access to all packages whose names begin with `scala-`.

The third capability has no subdivisions. It is not expected to be frequently useful for communities to grant some but not all capability manipulation commands.

The capabilities approach allows (hopefully) convenient implementation of a wide variety of access-control policies for the community. Simple communities do not need any notion of user, but instead can have officers create, grant, and revoke capabilities manually for the users who should have them. Simple email is sufficient for transmitting keys in simple communities.

More sophisticated communities can implement a capability-handout server that uses existing community infrastructure such as LDAP databases. Since no user model or authentication approach is presumed, it should be straightforward to implement such servers.

The most awkward case is for communities where the defined set of capabilities cannot be combined into the kinds of access allowed for users in the community. For example, if a community wants to give users access to packages based on versions in addition to names, the above set of capabilities is not sufficient. Even for such communities, though, it is possible to implement a proxy server that implements the more awkward operations under the correct policy. However, with such an approach, the proxied operations cannot be performed via standard tools.

7 Sample configurations

The package universes architecture allows convenient implementation of a variety of community organizations and software-engineering processes. Here are a few examples.

- No restrictions. Wikis have proven that effective and useful communities can be built even with no security restrictions at all. This policy can be implemented by posting all of the universe's capabilities on a public web page.

- Full access, but only to community members. Many open-source projects have an organization of this form, including Debian's "Debian Developers" and FreeBSD's "committers." This policy can be implemented by giving the capability-creation and -revocation capabilities to the membership gatekeepers. The last step of admitting a new member to the community is to give them their own capabilities to manipulate the community universes.

- Single provider, multiple users. An individual developer wants to provide a suite of packages to be used by a larger community—possibly the world, or possibly a limited base of subscribers. To implement this policy, the developer can keep a universe's package-adding capabilities for personal use but publish the universe-retrieval capability to the desired user base.

- Moderation queues. Sometimes it is desirable to have a large community contribute suggested packages, but a smaller group to decide on what is actually included. A moderation queue can be implemented as a separate universe where the package-addition capability is made available to whichever community is allowed to contribute suggestions (perhaps the entire world). Moderators would have both a reading capability for the moderation queue and a package-adding capability for the main universe, and could copy packages from the queue to the main universe whenever they are deemed appropriate.

- Private local development. Individual groups can form their own universe for private development without needing to coordinate with any central organization. They simply create the universe and share keys with members of the group according to their own local security policy.

- Public libraries plus local development. The above organization can be refined by allowing developers to use publicly available packages even as they develop packages for private use. This policy can be implemented by taking the union of the local universe with one or more public universes.

- Localized versions of packages. Local groups may want to use something similar to a widely-scoped universe, but override specific packages with localized versions. A particular example is language-specific versions of packages. Users want to use the localized version of a package whenever one is available, but the global version otherwise. This policy is implemented by creating a universe holding the localized packages, and then having users operate in the name-based union of this universe with the public one.

- Stable versus unstable streams. Projects frequently distinguish between stable and unstable streams of development. The stable streams include packages that are heavily tested and deemed to be reliable, while the unstable streams include packages that are newer and more powerful but are not as reliable. A project can implement this policy by having separate universes for each development stream.

- Freezing new stable distributions. A common process for generating stable distributions of code is to take an unstable stream, start testing it, and disallow any patches except for bug fixes. After some point, the distribution is *frozen* and considered a stable release. Such processes can be implemented by manipulating the outstanding capabilities as time passes. The testing phase can be implemented by revoking all outstanding add-package capabilities and switching to a moderation queue process. The actual freeze can be implemented by destroying the moderation queue and revoking the remaining add-package capabilities. The frozen universe can then be duplicated, with one fork hosting a new development stream while the other becomes is designated a stable release.

8 Related work

There has been much work on packages and package distribution. This section is currently overly brief. Feedback is welcome, so that the section can be enriched with comparisons to other systems.

8.1 Packages themselves

The present work focuses on package distribution, not much on packages themselves. The underlying conjecture is that it is more effective for a community to cooperatively tweak packages within a designated set of packages, than it is for individual packages to be so robust and flexible that they can be installed on any machine and with any other packages.

Nevertheless, better package formats reduce strain on the package repositories. For example, a more flexible package format could potentially allow two communities to operate within the same package universe, even when with a simpler format they would each require their own universe. Further, better package formats can reduce accidents that occur as packages are developed semi-independently, much the same way that memory-safe programming languages prevent damage from stray pointers.

Two examples of work on package formats are OSGi packages [5, 3] and the scsh package format [10].

8.2 Package distribution

Debian and its Advanced Package Tool (APT) [11] are the primary inspiration behind the package universes architecture. Package universes develop the design of these tools in a few ways:

- Package universes supports different ways to combine universes. APT only supports simple unions. Particularly interesting are name-based union and the name-based filtering, which support similar functionality to APT's *pinning* mechanism.
- Package universes supports uploading packages and package descriptions in the core architecture, while APT leaves uploading for separate tools.
- In support of uploading, package universes has a refined, policy-neutral approach to access control.

Conary is an ambitious system that aims to provide full version control across distributed ad-hoc repositories [1]. It shares with package universes the design goal of allowing new repositories to be formed from existing ones without needing any kind of permission from the original ones.

It is an open design question whether a package distribution system should include version control. Conary explores the design path of including version control, thus achieving a system with more features. Package universes, on the other hand, intentionally rejects sophisticated versioning. Instead of branching within a repository, the approach in the package universes architecture is to fork a universe into two separate universe. Instead of having users remain at earlier revisions within a repository, the package universes approach is to fork the universe into a fast-moving universe and a slow-moving one. Instead of modelling branching within a single repository, branches can be implemented by forking universes into multiple universes.

Conary pays for internal version control with a large increase in complexity. In the package universes architecture, package universes contain packages which contain files. To contrast, Conary has four ways to group files: groups, packages, components, and filesets. The Conary documentation gives considerable attention to the specific case of a local administrator having a "local shadow", which is accomplished in package universes by the simple and generic mechanism of name-based union.

At the very least, package universes provides a simple conceptual core for package distribution that does not use version control. However, if version control is more complex than worthwhile, package universes may be a better architecture outright. Opinions differ over the desirability of version control within a package-distribution architecture, and it appears that resolving the issue requires more time and experience.

References

[1] Conary web site. `http://wiki.conary.com`.

[2] Debian web site. `http://www.debian.org`.

[3] OSGi web site. `http://www.osgi.org`.

[4] Scala web site. `http://scala.epfl.ch`.

[5] The OSGi Alliance. *OSGi Service Platform, Release 3*. Ios Pr, Inc, 2003.

[6] Bo Leuf and Ward Cunningham. *The Wiki Way: Quick collaboration on the Web*. Addison-Wesley, April 2001.

[7] Mark S. Miller, Chip Morningstar, and Bill Frantz. Capability-based financial instruments. In *Financial Cryptography*, volume 1962 of *Lecture Notes in Computer Science*, pages 349–378. Springer, 2000.

[8] Mattia Monga. From bazaar to kibbutz: how freedom deals with coherence in the Debian project. In *Fourth Workshop on Open Source Software Engineering*, Edinburgh, UK, 2004.

[9] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification. Technical Report 200464, EPFL, Lausanne, Switzerland, January 2004.

[10] Michel Schinz. A proposal for `scsh` packages. `http://lampwww.epfl.ch/~schinz/scsh_packages/`, August 2005.

[11] Gustavo Noronha Silva. APT howto. `http://www.debian.org/doc/manuals/apt-howto/`.

[12] John Steinmetz. Computers and Squeak as environments for learning. In Mark Guzdial and Kim Rose, editors, *Squeak: Open Personal Computing and Multimedia*. Prentice Hall, 2002.