CPAN6

Mark overmeer*

July 29, 2007

Abstract

People are predestinated to collect things, whether it is food, postal stamps, or digital information. On our hard drives, we collect software, photos, development sources, documents, music, e-mail, and much more. The typical application sees this 'collecting' as secundary problem to their main task, offering little help in administering the data produced with it. CPAN6 focusses purely on collection needs, and can therefore improve the way people work in general.

This paper shows how CPAN6 can simplify the task of releasing Linux distributions, how administrating ftp-servers can be made simple, how software module libraries can be built as joined effort. As far as the author is aware of, there is no comparible system yet. More information can be found on the project's website `http://cpan6.org`.

Introduction

The Perl community has a very succesfull module archive, named CPAN[1]. Over 11 years, it has collected more than 11,000 different open source modules, contributed by thousands of authors.

Many other programming language communities have tried to create their own version of CPAN, but none was so successful yet. One of the disadvantages of CPAN is that it is very much entangled with Perl5, so it cannot be used to implement other archives. And this starts to bite the Perl community itself as well: with the upcoming release of Perl6 (with Parrot as large product aside), the old infrastructure needs a major overhaul.

In the process of abstracting the archive's functionality –combining the features of CPAN with modern needs like authentication– it was discovered that the resulting design could benefit close to *all* applications: every time someone collects any form of digital information. Even, to organize ones home directory. Now, the target for CPAN6 has become to be included as Open/Save option in any application.

*`mark@overmeer.net` (The Netherlands) `http://solutions.overmeer.net`

[1] See `http://search.cpan.org`): the *Comprehensive Perl Archive Network*

1 Considerations

In our daily work, we use digital collections everywhere. But how to we collect, and how do we organize the collections we have?

My Linux system provides me with a home directory, in which I can do whatever I want. So, I have a sub-directory containing all e-mail I ever received, a directory with photos, a directory used to develop software, and so on. Descending into the development directory, each of my projects has its own sub-directory. And after some steps down in the directory hierarchy, we end-up somewhere where the project content is.

How I organize my directory tree is my own choice; although I am not a fan of a *Clean Desk Policy*, my 'Clean Directory Policy' works. For many people, this is more of a challange. One of the observations I made was that, moving through this directory tree, the meaning of the directory level changes: from 'me' at home, the development work as a whole, development by project, project versions, onto project data. This organization very ad-hoc, and needs human interpretation based on the content of the directory.

Another collection. At home, we have three people, eight computers, two photo cameras, and a film camera. How and where do we collect all these pictures and films? Asking around, it sems that usually the most advanced computer user in the household creates a network disk where he (or she) uploads the pictures from all equipment to be available to everyone. The other people are told how this disk can be accessed and where on the disk the data is kept. Very ad-hoc. And how do we share this data with people outside our protected home network? Use Flickr?

Yet another collection. Some central ftp-server administrator mirrors many other ftp-servers. The data is copied on regular intervals with some (rsync) script. The disk and directory where the copy of the data is located are determined ad-hoc, as is the mirror frequency. The people who like to download the information cannot find-out when the last update was. They cannot easily figure-out which mirror of some data is fastest, and where it is on that server. Gladly, some dedicated applications attempt to improve the situation a bit, like rpmfind.

Just to add a last example. Linux distributions are distributed via ftp mirrors. When a new release is published, thousands of people want to download it the same day, overloading these servers. Therefore, the distribution has to be released under embargo to the mirrors first, and a few days later that embargo is lifted. There is no standard way of doing this.

Besides, most distributions add some authenticity checks in their packages. Often, this is based on some (recently very broken) MD5 checksums. Why does everyone have their own solution, where the problem is with the distribution process itself?

2 Overview on CPAN6

All examples listed above demonstrate that we could use some help in organizing our digital collections. CPAN6 is designed to do precisely and purely that: to help applications to structure and exchange information.

2.1 Components

The implementation has three conceptual components:

CPAN6 is the name of the **infrastructure**. It defines how *archives* (collections) exchange their *releases* (digital information). You may very well call this the meta-archiving level: playing games with collections of archives;

Pause6 is the name of one of the possible archive **administration** implementations. It adds higher level concepts like organization, purpose, and trust;

applications access the archives much like they access directories or ftp-servers. Like with directories, you have commands like `cp`, and a graphical `open file`.

2.2 Terminology

The administration of an archive (Pause6) uses the following terms:

- one or more persons (or processes) are generating digital information to be collected in the archive. These are the **authors**;
- this information is organized a coherent set of files in directories, as a whole named a **release**[2];
- an **archive** is a set of releases, grouped together for a certain purpose.
- each release has a **project** name, and often a **version** indicator. The project name and version string uniquely determine the release within an archive.
- one of the authors will upload the release to the archive, and is therefore named the **publisher** of the release.
- inside the archive, a release is in a certain **release-state**, which can be *uploading* (incomplete), *published* (upload ready, only available to the authors), *embargo* (not for public download yet), *released* (for public access), *deprecated* (should be avoided), *rejected* (must be removed), or *expired* (will be removed).
- one or more people form the the **board** of the archive: they are responsible for its well-being.

[2] Some people suggest to use Version Control Systems as distribution mechanism, however: end-users are totally not interested in all the intermediate development steps. When you read a book, you do not wish to know how the author faught with the text: you only need the result. Besides, a lot of features which make releases manageable do not fit in the VCS concept.

- the board set the rules for the archive, the **constitution**. This sounds very formal, and this very well may be: the board can determine how release states can change. For instance, they can enforce that any release requires at least two digital authographs from a selected set of people before it is released to the general public. The board can also decide to refrain from any kind of security or control.
- authors can always retrieve the releases of their projects. The board can access all releases in the archive. Other archive **users** are restricted to released and deprecated material. Of course, other archive access restriction rules can be set-up as well.
- the authors, publisher, board members, and users (either humans or processes) are represented by an **identity**, which is used for authentication.

Above definitions are all part of the Pause6 implementation, which will evolve over time, or be replaced by an other concept. The design is made in such a way that different archive implementations can be used on the same CPAN6 infrastructure.

CPAN6 defines the world more abstract:[3]

- on the CPAN6 infrastructural level are releases just collections of files with a name, and archives simply collections of releases.
- uploads are made to the **commissioner** (a daemon which executes the constitution set by the board)
- downloads are available from any of the **deployers** of the archive (the administrator daemons of the mirrors).
- **scribe** processes are used to copy information from commissioner to the deployers, but also to publish and download.

2.3 Decissions

CPAN6 tries to impose as little restrictions as possible, but archive implementations may do so.

- project names and version identifiers are utf8 encoded unicode strings of unlimited non-zero length;
- archive names are IRI (an URI where the path part can contain encoded utf8 characters). A part of the IRI is the name of the daemon which handles the commissioner role;
- the archive name in combination with the project name and version string provide a world-wide unique address for a release;
- the transport protocol used for copying releases does not matter to the archive, whether it is ftp, e-mail, local disk, or DVD is only interesting on the CPAN6 infrastructural level;

[3] On purpose, these terms are distinguishable from existing terminology, in an attempt to indicate that meta-archiving is different from the usual archiving.

- the compression and packaging used is also only interesting to distribution process. Therefore, it is part of the automatic exchange negotiations, exactly like the `Accept-Encoding` / `Content-Encoding` pair works in web browsers;
- authentication and signatures are integral parts of all components, but there is no choice made about what algorithm is used: anywhere from 'none' to shared keys per release. The better the exchange of keys and algorithm, the more **trust** a release gets.

The board of an archive can set restrictions on many things, and project authors (the owners of the project's name-space within the archive) can set those even more tight. For instance, the project-name and version string can be restricted in size and with a regular expression. Permitted authentication methods and licences, release mime-types, and packaging and compression preferences can be restricted too.

Of course, you still can release a '`vim-7.8.1.tar.gz`' file, but there is no need for it: let the scribes figure-out what the most convenient way of transport is. Better just release the content of the directory structure `'vim/'` itself[4]. With minimal clients, where `tar` and `gzip` are (not yet) installed, you can still access the data, on expense of a larger transport.

The interpretation of the quality of the used signatures is personal: how much *trust* do we need to have in the authenticity of the material? For instance: can we verify the signatures ourselves directly because we are on-line and how do we value the used algorithm?[5] The end-user should be able to configure a trust 'threshold', below which downloading and installation is not done without explicit permission.

2.4 Everything is a project!

Reading through this paper, you may more and more get the impression that the implementation is complex. Well, actually the core is very simple! Let me try to convince you.

Pause6 defines various components which need configuration: a project has authors, an archive has a constitution, we have identities for people and processes, and there will be a way to let one archive help you contact the next one (like sub-directories within a directory). But what is a configuration? Well: some syntax which wraps some semantics, stored in a single file or set of files. So, it falls under our definition of a release: zero or more files. Consequently, we can treat configuration files are releases, and configuration changes as sequential releases of the configuration 'project'.

Let's play an example. When we start a new archive, we 'magically' get a space to release material into. Inside this archive, there is one release occupying the name-space (project name) `pause6-constitution`, which contains a file describing the default configuration of an initial archive. It is the last release in that name-space, so

[4] The project name and version identifier have to be (derived as) explicit meta-data anyway, during the opload process.

[5] MD5 signatures got cracked recently, so at once we should trust the releases for which we only checked the MD5-sum much less. The user must be informed when the system gets less secure.

the one which is *active*. In that configuration, you are listed as only *author*, therewith the only *board* member. You download the release, edit it to add more people (identities) to the archive board. Precisely said: you add a few addresses (the combination of an archive IRI, project name and version identifier) of releases which contain metadats which descrines someone or some processes (most importantly the authentication information) to the list of authors of the constitution project.

At the same time, you can set a wide scala from permissions. For instance, you can say: each new project needs a signature of consent by a board member. You can also configure: any new release in a project needs a signature by more than half of the authors. As example, the project `pause-constitution` itself could be made to require a majority of signatures by its authors, the board members. Configure three board-members with the need of majority vote for releasing a new version of it. Then, release the new constitution under the active (is 'old') rules, which means that one signature suffices to effectuate it. When the commissioner daemon sees that the new configuration gets into the 'released' state, it will automatically load the new one. From then on, two board members are needed to change the archive rules.

The "game" with releases is simple to implement, but must be presented to end-users of the CPAN6/Pause6 system in a very different ways dependent on the project type. An average computer user will not understand that uploading a set of photos to the archive is the same concept as changing the rules of that archive. So, the whole CPAN6 project is primarily an interface challange.

2.5 All collections are archives!

The commissioner, deployers, and scribes need configuration. Commissioners and deployers are basically the same[6]. One daemon will execute all the commissioner, deployer and scribe tasks. To achieve this, it needs the configuration for each of those tasks, which is actually kept in the constitution (project) for each of the archives. So, the daemon needs to find the archives which are available in the system, to collect it.

Locating archives is more simple than it looks, because one of the special project type is introduced: the 'archive reference' type. It is just (again) some project name with releases, with as release data some details about how to contact a remote or local archive.[7]

The daemon configuration simply an archive itself, which restricts the types of the contained projects to archive references. The constitution of that archive defines who can add new archive references and whether this needs the consent of the board.

All release collections are archives. For instance, the installation tools can create an archive to maintain the installation logs. Each time you copy the pictures from your camera into an archive, it is one release for your home wide available archive. A PGP public-key server is just an archive of identities.[8]

[6]For a new archive, the commissioner and deployer are the same: only when mirrors are configured the roles will be split. The only real difference is found in some access rules.

[7]When you have so many archives around, configuring each of them by hand becomes a hassle... so when you have contacted one archive, you can get in touch with related archives by simple installing the last release of the archive reference installation which is contained in the first.

[8]There is no need to transfer the existing PGP infrastructure into a Pause6 archive: with a simple

3 Some key features

Those who would like to see all the features of the system should read the detailed design documents. Here, only a few remarkable features get introduced in random order.

3.1 Release versioning

Project names group sets of releases. One of those releases is the initial (naught) release, which is the start of a **strand** of releases which extend the information each time. For documents, it is the logical sequence of newer versions of that document. For software, a strand is formed by releases which extend the interface without breaking it.

Next to these *follow-up* releases, you can also have *diverting* releases: where the interface does get broken. When a user asks for a newer release for a project, it should never automatically jump into the diverting release strand. For instance, your mysql 4.2.12 will not be upgraded into mysql 5.1 (without asking), but will upgrade into 4.2.25. Diverting releases (for instance development releases) can have different rules (more freedom, other authors) than the main strand of releases. They can also be merged in, back into the main strand: when people upgrade their alpha version, they come back into the following official release.

Each developer(-community) has its own versioning scheme. In combination with the diverting releases, there is no way for the archives to determine the release relations based on the version string. Therefore, a new release must (helped by the application used for the upload) explictly contain this relation detail. Client applications can ask deployers how the actual structure of strands and releases is.

3.2 Deployers

When an (ftp-like) mirror manager wants to invite an archive to be copied onto his system, he simply uploads some authentication information about his configuration onto that archive (sends it to the commissioner). It may require a board signature of that archive to accept it (get it into the released state), but once the mirror details are released, then all end-users can see that mirror and use it.

Normal end-users have very restricted search capabilities in the commissioner of an archive: they can ask for the defined deployers, and for project data where they are the author of (because that may not be distributed yet). Deployers offer more ways to search to end-users, comparible to the features linux installation tools provide you about the content of rpm's. The deployers may use databases or other enhancements to speed-up the searching. There is no need to download huge indexes to the client anymore.

wrapper, the data exchange with the PGP public-key infrastructure can get its own place on top of CPAN6, as 'brother' from Pause6 implementations.

3.3 Scribes

The scribes are responsible for the transports. First, they organize the upload of a release by the publisher; at the end the download to the end-user's system. Inbetween, they are responsible for copying the releases from commissioner to the deployers.

Scribes implement the transport protocols (FTP, HTTP, rsync, disk), the optional compression (gzip, bzip2), the optional packaging (tar, cpio, zip). Scribes may do information inspection and reshaping. They may change the used protocol, compression, and packaging.

Scribes are also capable of filtering. For instance, a scribe can be configured to say: copy only the latest release for each project. Filtering can be based on project-name, mime-type, release size, authors, trust, a manually picked list of project names, and so on.

3.4 Hierarchies

The flexibility of the scribes makes the following possible. As user, you have your own archive in which you collect all software releases you ever produced. A scribe automatically copies the last release for each of your projects onto the departments development archive. There it awaits a signature of one of the managers to be accepted. Once that signature was given, the scribe on company level picks it up and other departments (maybe also the outside world) sees it.

The same can be done the other way around. Often people complain that Perl's software archive CPAN contains too many low quality modules. On company level, someone manually picks out a sub-set of the whole CPAN and configures a scribe to regularly collect the latest releases of each of those modules. Only when the internal regression tests run without failure, the module is visible on the internal company archive. On company level, you can decide to never expire releases you use, or to reject them... things you can not do with CPAN itself.

3.5 Trusting releases

The end-users may like to have some certainty that the release they get is the same as what the publisher has uploaded. The end-user could be a bank, and the publisher an external contractor. The bank really would like to have an indication how large the chance is that a third party changes the content. That value is expressed in a **trust** value between 'none' and 'full trust'.

Each component in the infrastructure leaves its signatures on releases (more precisely: on the release-states). This starts with the publisher, but also commissioner, deployers, and scribes. Each component has a certain trust, based on the quality of the algorithm used (MD5?) and the quality of the key delivery process (PGP?).

When a release is received over the network, the overal trust can be calculated based on that of the publisher: check his PGP information directory. Off-line (for instance, while installing Linux from CD), you trust that the distribution creator did the checking for you.

How to calculate trust exactly needs more research. Important to realize is that Pause6 does not decide about the signature types or trust algorithms: anything between fully open archives upto extremely complex secure processes can be used.

4 Use cases

Now some examples as use-cases. This should demonstrate how CPAN6 can simplify everyone's life. The first one is especially for programming communities, especially the Perl people.

4.1 CPAN6 for Perl6

As already explained in the introduction of this paper, the arrival of Perl6 adds complications to the Perl6 CPAN software archive which is currently in use. The most eminent problem is the name-space problem: there will be an Perl6 implementation of a data-base interface (DBI), and how will we keep that apart from the Perl5 DBI implementation? And Perl6 is based on Parrot, which may produce a PIR implementation of DBI. Perl6 supports precompiled distributions, how do we distribute the pre-compiled version of DBI? And so on.

The only way to crack that nut, is to create diffent archives (different name-spaces) for each of those DBI implementations. Each of these archives may enlist archive references to the related archives, or one archive can be founded dedicated to that task. Pause6 defines the concept of **views**, where the some project name is used for related content in different archives.

During the international YAPC conferences, a few well-known community members sit together to form the board, writing the constitution of the archive. Key-signing parties can be used to improve everyones trust in the signatures found on the constitution of the archive.

4.2 Ftp servers maintainers

In the Pause6 design, the release data is kept in **stores**. Some stores will use archive name, project name, and version number to create a directory structure, in which the released information can be found. Other stores may just assign a sequencial number to each following release and keep the other information in some database. The latter has advantages as you realize that all names are unicode and unlimited in length. Stores can be made to just fill-up disks, because the releases does not need to be kept in a logical structure.

If you start a new mirror service, like current ftp-servers, you simply install CPAN6 and Pause6, configure a store, assign it to be used in the default top-level daemon archive, and start sending-out mirror invitation requests to the archives you would like to mirror. This can either be done in a one-liner or via a graphical interface. Cleaning-up a mirror also just a one-liner.

When you already have a functioning ftp-server set-up, then use the store type which monitors the changes on disk. Pause6 can still be used to provide the deployer func-

tionality and the exchange of trust information. In this case, the scribe does not need to copy the releases itself, because some external rsync script is already doing that.

4.3 Managing family pictures

An average Linux user gets CPAN6/Pause6 installed automatically with the next upgrade[9]. When he inserts a photo or film camera into his USB port, a pop-up offers not only the option 'copy fotos to disk', but also 'copy fotos to archive'. Next to a 'My Photos', we have an 'Our Photos', which is an archive limiting access to local network users. Other family members see the same 'Our Photos'.

The image processing applications have an 'Open' from disk, and 'Connect' to archive menu entry. The respective pop-ups look very much alike. When the selection is made, the application will see the extracted release information on the local disk. The application does not need to be changed much.

5 Status

On the project's website, you can read the current status of the project. In the previous year, a quite detailed design was made, and components explained for short in this paper are described in those designs in much more detail.

The data structures are worked-out as XML-schemata, which certainly will be extended when the project evolves. These schemata are used in SOAP message exchange protocols. A better XML/SOAP library was implemented for Perl to get this to work.

The archive implementation is starting-up, written in Perl5. This permits the most flexible framework for extension. The initial code is already capable of language translations, with a new module especially developed to become part of this framework.

A command-line interface is under development. In parallel, an AJAX graphical interface is created, based on the jQuery library. This way, end-users do not need to install extra graphical interfaces on their system, but just use their recent browser. Other graphical libraries may in the future use this as a reference implementation.

Financial support is coming from the Dutch foundation Stichting NLnet, which donates working hours for the development, but especially for promotional activities. More help, preferrably from people who want to particate in the imlpementation, is very welcome.

[9] In the ideal world for the developers of CPAN6, of course.

6 About the author

Mark Overmeer graduated as MSc in Computer Science at the University of Nijmegen in the Netherlands in 1990. After being a UNIX administrator for six years, followed by being a UNIX (and related subjects) trainer for an additional six years, he decided to become a free-lance programmer.

Being very active in the Perl community, he also recently created a huge satellite imagery archive, maintains websites and network infrastructures, and is the scriptor for the nl-TLD.