CPAN6 and Pause6

Mark overmeer*   Sam Vilain†

July 28, 2009

Abstract

CPAN6 distributes collections of released material as *archives* over Internet. Each archive may contain software releases for any programming language, but also tutorials and other documentation, rpm-packages, or software patches. CPAN6 as concept is a meta-archiving system: a uniform way to create and spread archives.

This project also implements one archive administration implementation for CPAN6 named Pause6. Pause6 offers trust, administration tools, and namespace control. As third component, an install helper application for Perl5 users (CPAN6.pm) is created, as first front-end to the new infrastructure.

Contents

1 CPAN (for Perl5) 3
- 1.1 FTP-servers . . . 3
- 1.2 The structure of CPAN . . . 3
- 1.3 CPAN Administration . . . 4
- 1.4 Archives aside . . . 6
- 1.5 Emerging problems . . . 6

2 Development plan 9
- 2.1 Three developments at once . . . 9
- 2.2 Decisions . . . 9

3 CPAN6 Structural 11
- 3.1 Components . . . 12
- 3.2 Physical . . . 14
- 3.3 Hierarchy . . . 16
- 3.4 Processes . . . 17

*markov@cpan.org (The Netherlands) http://solutions.overmeer.net

†samv@cpan.org, Catalyst IT (New Zealand) http://catalyst.net.nz/

4 Pause6 Organizational **19**
4.1 Additional components . . . 19
4.2 Projects . . . 21
4.3 Web of Trust . . . 24
4.4 Archives . . . 27
4.5 Configuration . . . 28

5 Epilogue **29**
References . . . 30
List of figures . . . 30

Introduction

CPAN celebrates its 10th birthday this year, in 2006. Ten years of Perl5, a universal dependency. Many techniques have evolved and technologies appeared in this decade, like XML and .NET. Even more new dangers emerged, like *spam* and *phishing*. CPAN survived these roaring years, but not by adaption and development: only a few minor adjustments were made over the years. Can we use CPAN for the coming ten years? What must be better and what can be made better?

How can we extend CPAN to serve new requirements? For instance, multiple programming languages (Perl5, Perl6, PMC, etc), more than one developer for a single project, and security. This CPAN6 project took off rather small, designing fixes to the existing infrastructure, but rapidly grew into a whole new concept of *meta-archiving*.

The “CPAN6 project” designs a CPAN replacement, but on three different levels:

- a general infrastructure to create and distribute archives: **CPAN6**,
- one implementation of archive management: **Pause6**, and
- one transport mechanism to install Perl modules from Pause6 archives **CPAN6.pm**

In this paper, we first discuss the things to learn from the current CPAN. Then in the second part, a new design is made for each of the three components. Follow-up documents are created to describe implementation specifics: file-system structure, protocols, and use-cases.

The Project’s Names

The names used to describe the parts of this project are chosen to honor the incredible usefulness of the original work –as dedication to CPAN and Pause as created by Andreas König.

The ‘6’ has something to do with Perl6. It also refers to the “Comprehensive Perl5 Perl6 Parrot PBC Python PHP Archive Network”, with alternative abbreviations into CP^6AN, or $(CPAN)^6$. Or does the P stand for “Pasm Pir PIL PIL2 Pie-ton ParTCL Pathological”?

1 CPAN (for Perl5)

Why do we design CPAN6, as a replacement for CPAN? First, let us take a look at the core of CPAN. The basic advantages of CPAN should be kept intact, and shortcomings of the system must be improved upon.

1.1 FTP-servers

All over the world, you can find ftp-servers to get publicly available data from. These ftp-servers use a very simple structure, as shown in figure 1. The publisher of the

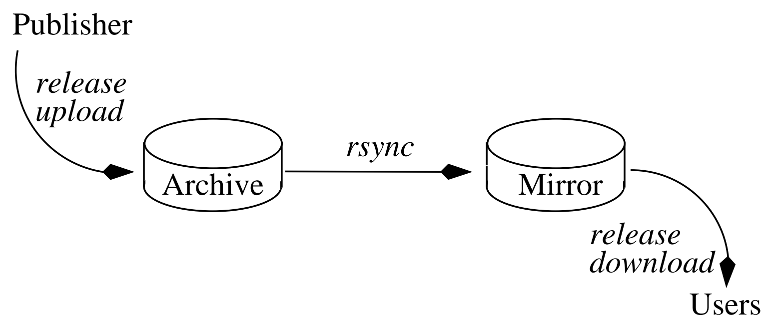

Figure 1: Simple ftp-server distribution structure.

information has write access to some ftp-server's directory and uploads a new release there. Daemons run at regular intervals on the other ftp-servers to replicate the contents of the source directory onto their local disk using *rsync*, *wget*, or some other copy tool.

Ftp-archives are very loosely coupled. In some cases (like some Linux distributions), access to the source archive is restricted to avoid performance problems when their new release comes out: end-users can only access the mirrors. In general, however, there is no such limitation: any user can use either the source archive or any mirror to download from.

Downloads from mirrors are usually faster, because those machines can be closer to the end-user –a shorter hence faster Internet link. There is no way to find that mirror automatically. It is hard to determine which mirror is the fastest. One mirror server usually contains data from many sources: it can be hard to find location of a mirrored release on the server because everyone organizes them in a different way.

1.2 The structure of CPAN

CPAN offers people a way to publish their own code, without direct write access to the archive source directory. CPAN has a daemon process named *Pause* which writes to the ftp-server source directory and maintains the CPAN name-space. The CPAN infrastructure adds this Pause component to the general ftp-server structure as shown in figure 2.

New releases are inserted into CPAN via the Pause system, a web site. After uploading, the releases are mildly checked, then stored, and indexed in a module list. The module list is published every few hours. Modules can be deleted, reassigned, and have some

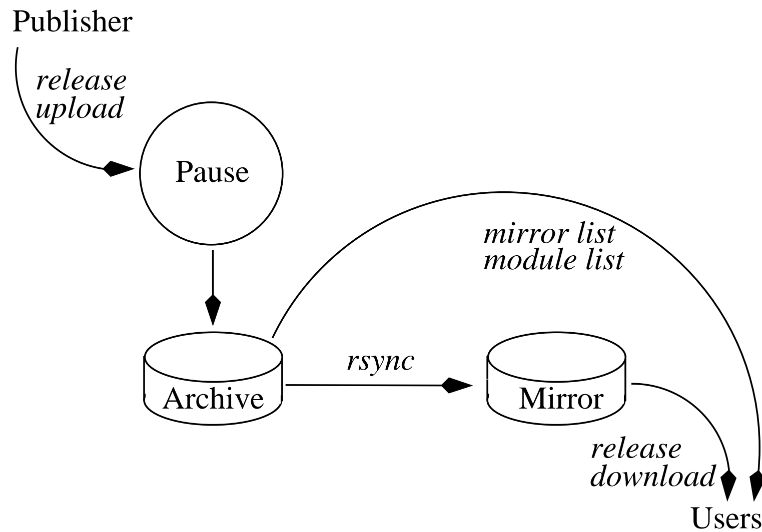

Figure 2: CPAN/Pause set-up.

meta-data added via a few maintenance web-forms.

The PAUSE content is mirrored onto a huge network of ftp-servers. The installation tools are quite successful at determining the nearest ftp-server to download a module from.

1.3 CPAN Administration

CPAN is very simple, it has a Pause daemon to administer releases ('distributions') and people ('authors'). It keeps track of all versions of modules.

Distributions

CPAN contains Perl5 software releases, referred to as *distributions* and *bundles*. Modules in bundles are listed separately in the index, whereas distributions encapsulate the included modules tighter together. There is no technical difference between those two packaging methods.

A distribution (or bundle) is distributed as `tar.gz` file (package) which contains

modules
: a set of `.pm` files, usually stored in the `lib` directory within the package.

manuals
: each module file may produce a manual-page at installation time. In most cases, manual pages are generated from the docs contained in the `.pm` files, but in rare cases, separate `.pod` files are added to the package as well.

scripts
: kept in the `bin` directory. When installed, manual pages are extracted from them as well (if present).

install help

the included Makefile.PL controls the installation; a script producing a Makefile. That Makefile needs to be run to build and install the software. Furthermore, usually some install documentation is provided in README, INSTALL, Copyrights, and License text files. Bundles also include a .ppd file.

tests

one script file named test.pl or a directory t/ full of test-scripts are run just before each software installation. These tests take some time to run, because there are usually many of them.

The tests are run for each installation on each platform where the software gets installed. The advantages of running tests everywhere is that there is a better chance that (platform dependent) code problems (including missing dependencies) are spotted. In general, this procedure improves the quality of the released code.

The core purpose for a distribution is: grouping a software into a release. Some people like to release many small distributions, other prefer a smaller number of larger distributions.

Versioning

A distribution has a version, which is derived by the Makefile.PL script from the version number found in one of the distributed files.

Some people put their development trace in a Version Control System (VCS) during development, which usually leaves version numbers in multiple files. These numbers may and may not have a relation to the distribution release version number. The ChangeLog (or Changes, or even Changes.pod) file shows the changes between releases in general terms, in abstract words from the author; not the change notes from the VCS.

CPAN also supports alpha quality software releases on CPAN, using special version number syntax. Because of that number, the release will not be automatically selected as ”latest” version of the code, so not automatically installed in the user’s environment.

A voluntary rule says that newer releases are always better than the older releases, and that the programming interface of modules are only extended, not breaking the existing code. Many modules have seen compatibility breaking changes over time, however. The installation tools do not signal that: you will find-out the hard way.

Authors

Authors are the responsible people behind the distributions. Everyone can simply register and then publish code. Pause uses a very simple username/password validation scheme, where anyone can request any non-existing username (PAUSE-ID) without validation. There is no guarantee that the author’s intentions are honest, but this has not caused known problems yet.

CPAN keeps track of who has released what software. It is not possible to use the same module name as someone else already has in use; an author claims a *name-space*.

1.4 Archives aside

Besides the 'official' CPAN infrastructure, people are free to set-up related services. The main additional sources of information which have appear in recent years are:

Backpan

: released distributions which got explicitly deleted by the authors from CPAN are still available from Backpan. The existence of this archive is not widely known.

Search

: a few indexed search facilities are implemented, the most popular is `http://search.cpan.org`. Pause only produces a distribution list, which does not contain sufficient information for a lot of the questions users want to ask; the search system is much more helpful.

RT

: The report tracking system manages bug-reports for all distributions in CPAN, available via `http://rt.cpan.org`.

Smoke

: smoke testers compile development versions of Perl and some of the modules with varying configuration parameters on a large number of different platforms. Smokers run the extensive test suites to early detect portability problems of Perl. Perl as language itself and Perl's core modules should work on all of the officially supported platforms[1].

CPANTS

: does automatic *kwalitee* evaluation on all CPAN modules; an attempt to judge the quality of releases, thereby improving the quality awareness.

Search, RT, Smoke, and CPANTS are typical examples of initiatives which grew around the CPAN/Pause set-up. Some even share password knowledge with Pause.

1.5 Emerging problems

What are the shortcomings? How can we improve the CPAN concept?

multiple programming languages

: As Perl6 gains adoption, we need a place to store modules written for it. But as Perl6 is based on Parrot, we must also be able to collect Parrots pasm, pir, and pbc modules in the same central archive. Besides, other languages based on Parrot (ParTcl, Lua, etc.) produce code which need to be distributed in some way as well.

[1] See manual-page `perlport(1)` section PLATFORMS for the current list of supported platforms.

CPAN has only one name-space, for Perl5 modules. How do we handle the situation of a Perl6 and Perl5 implementation of the same, with the same name in the archive? Do we need to prepend a perl6 indicator to all names used?

For instance, the `Tk` graphical library wrapper is named `Tk` in CPAN. Do we need to name Perl6's wrapper `Perl6::Tk`? This would extend all new module names with an extra name level.

multiple developers

over time, people pass on module ownership to other people. It currently is possible to assign one project to multiple people, but only one can release new versions. When some module's author stops responding (what quite often happens) only manual intervention by the Pause administrator can re-assign the name-space to someone else.

security

Pause has a simple password protection system, which is far too weak for current best practices. Everyone can anonymously add fake user-ids, and therewith spam CPAN. Who is personally responsible for which released component?

See the Linux kernel development procedures, where changes are signed-off by key figures: that adds trust to the code base. A better code base will attract serious users.

packaging

part of CPAN's success is the super simple packaging: `tar.gz`. This works well if you look to Perl itself, but it doesn't work well together with packaging systems which are native to the operating system where Perl is used: like `rpm` or `deb`.

For system administrators, the blunt standard installation procedure of Perl is cause for many headaches.

"brute" installation

When a module is upgraded, the whole distribution is reloaded. Often a whole bunch of dependencies are upgraded as well. A very verbose transport, configuration, testing, and (finally) installation procedure starts off.

For system-administrators, it is impossible to oversee the consequences for their system beforehand. It is impossible to trace all changes during the process. Which modules are installed or upgraded? How much disk-space will be used? Which licenses are used (and silently changed)?

alternative sources

Linux distributions often give you the opportunity to specify a list of archives which are not equivalent, but complementary or overlapping. With CPAN installation tools, there is only one uniform archive. It is should be easy to publish additional modules on system or company level.

versioning

CPAN's versioning system is not integrated with version control systems people use locally, like CVS or SVN. Software releases use versions which do not need to relate to any version control system used to develop the release, so by concept these systems are distinct. However, it would be nice to have a single command with releases a tagged version from CVS/SVN directory into CPAN.

backpan

backpan should be merged back into CPAN: Perl6 will allow coders to request very explicit versions of modules. It is simple, so people will start using it. As a result, distribution owners must not be able to remove releases anymore: only deprecate them when they are considered *bad*.

dependencies

more automatic dependency checks for the author when a distribution is released would be appreciated. It is not enough to say "use v5.8.2" in your Perl module, because you may require a special Parrot version as well. You may even borrow modules from other programming languages (based on Parrot); those language implementations may have to be installed as well.

It is much hassle to get the list of dependencies correct and complete. Therefore, it would be nice to get automated help to build these lists. For instance an indication about what the code publisher has installed on its development system.

integration in environment

we should fix things with the CPAN6 installer; it should register installed packages with the system package database, be it SysV, RPM, deb, Windows Installer, fink, etc. It should also allow admin operations to be backed out with confidence.

A lot of things to worry about. CPAN/Pause is very popular because it works! But there are many ways to improve it.

Example: using modules in Perl6

As an example for emerging problems, this example for (near?) future Perl6 requirements. In a Perl6 program, a simple "`use Module;`" will refer to a perl6 module, and a "`use perl5:Module;`" to its Perl5 counterpart. But we may also link `pir`, `pasm`, or `pbc` modules into our program. We may even link to `parTCL`, `Pie-ton`, or whatsoever. Where do those modules come from? How do we denote dependencies between those modules?

On the moment, CPAN has a simple task, only serving Perl5 and sometimes a little bit of XS code (hoping you have a C compiler at hand). The new complex modular structure is much more demanding. What should happen when a module `perl6:AAA` requires a module `partcl:BBB`? Should we install (the syntax module for) parTCL as well? Or download its compiled PBC equivalent? If we install parTCL on Parrot,

do we require Perl to be installed for the installation process, or do we distribute that as compiled pbc? Do we really expect users to understand all these dependencies, or can the insert procedures to figure this out?

If we want people to be able to install a pbc alternative to the Perl6 module, do we require each module author to prepare that pbc, or will there be CPANTs like service providers which prepare those packages? And how do we support them? How do you find them? How do tools figure that out automatically?

Above questions provide more than sufficient worries to rethink CPAN thoroughly.

2 Development plan

2.1 Three developments at once

CPAN/Pause has a straightforward approach, simply installing Perl5 modules. To host all future software in one CPAN archive will cause name-space conflicts, but also archive size problems. Let us break up the monolithic CPAN archive into smaller CPAN-like archives, each with a different purpose, its own name-space, and its own regulations.

To be able to create multiple archives for CPAN6, the project (and terminology) has to be split into three parts:

CPAN6 is not a new version of CPAN: CPAN6 is a concept, an idea how to create, manage, and distribute archives in a network. It is more on the level of ftp-server configuration, where the set of mirrored servers is configured.

Pause6 is one possible implementation of the CPAN6 ideas. Over time, multiple implementations of the CPAN6 concept may be introduced in the network. A single Pause6 instance manages one archive; one collection of releases. It implements the features of the existing Pause interface, but safer and primarily with a command-line interface.

CPAN6.pm is created to use Pause6 in a way similar to the existing `CPAN.pm`. It is implementing a different transport protocol and search system on top of the Perl5 CPAN implementation.

Experience has shown that the above definitions do not come easily: there are no other meta-archiving systems like CPAN6 available that we know of. Therefore, the design needs to define clear terminology.

2.2 Decisions

Learning from the current CPAN and other existing applications, we decide for the following as basic requirements for the CPAN6/Pause infrastructure which we will detail later.

Archive content

CPAN is about Perl5 software distributions, where XS (C library bindings) and Perl6 are added in various hackers' suppositious way. CPAN's functionality is, however, not at all specific to the Perl5 language. The actual type of content of shouldn't bother any archiving software: it's just an end-user application issue.

CPAN6 concentrates on distributing releases: sets of information packaged as releases, which are transported between archives. Pause6 administers archive content, sets of releases. Also Pause6 has no detailed knowledge about the released materials it maintains; like library index-cards do not bother about the content of the described book.

Only the front-end is (release) application specific, using the CPAN6 infrastructure are transport mechanism to retrieve releases, for instance to download releases of Perl5 modules.

Instead of trying to get the various releases (Perl5, XS, Perl6, etc) into one archive (one name-space), with CPAN6 you build separate archives, each with their own name-space - each archive with its own administrative rules.

Simplicity

CPAN is a little less free and simple as a general ftp-server infrastructure, but still easy to use. The authors need a bit more knowledge to be able to use it –get acquainted with Pause–, the users not. On the other hand, CPAN is too simple to be safe, which is required by some groups of users.

CPAN6, as network of archives, should make it easy to start and distribute your own archive. Pause6 should make it easy to have web-based and command-line based archive administration. On the end-user application level, the archives are simply transport mechanisms, like HTTP and FTP, with additional search features.

Security and Trust

CPAN only uses username/password authentication during software upload to Pause. It is commonly known that this scheme can not withhold attacks against abuse. And can we trust that all mirrors of CPAN contain the same data as the source archive? Are we sure to download the same data as we expect to get? No, CPAN is very insecure compared to modern needs.

The target is to get the highest level of *trust* for a user who installs the code. Security will primarily be achieved with cryptographic signatures. Every transmission is signed by the sender, and you define how well you trust each sender.

Before you install software on your system, you get an impression about the level of trust, the licenses, and the size of the code to be installed.

CPAN6 enables security features in the network of archives. Pause6 implements connection to various security mechanisms. The client-side applications should check the available security information.

Network

CPAN6 should behave like a real network player: developed with tasks which can run distributed over multiple systems, redundant where possible.

Being a network player also means platform independent, which translates into: must be able to work on all current operating systems. At least, it should avoid complications which block a porting effort to problematic systems.

Language

Protocols will use English as language. All administrative files and all protocols will use the UTF-8 character encoding.

3 CPAN6 Structural

Small pieces of the puzzle will be discussed in later chapters, one by one and in detail, but let us start with the whole picture. Most importantly: define the names we will be using.

Figure 2 showed the structure of CPAN/Pause as used nowadays. To solve the need for more control over the distribution of the releases –especially for tracking security information–, the mirror archives need to be monitored as well. Figure 3 shows the simplified picture for CPAN6. In CPAN6, each of the archive's replications is monitored by a daemon. The release accepting process (Pause in CPAN) only provides a list of mirrors to the users, where all other user requests (like searches) must be directed to the deploying archive copies. This implies that daemons must run for the mirrors, where ftp-mirrors currently do not require them.

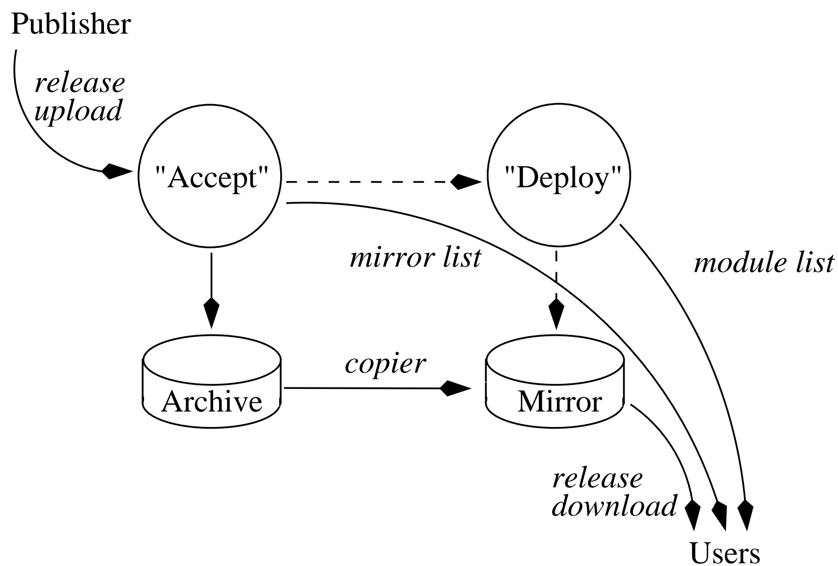

Figure 3: CPAN6 simplified general structure.

The ad-hoc copy processes for ftp-servers is replaced by a formalized copying process, which both transports the released data and updates the deployers information.

3.1 Components

Figure 3 is shown as extension of the existing CPAN structure. For our purpose, we need to redefine the terminology to add flexibility. Figure 4 shows the processes as defined in CPAN6, each of which will get detailed below.

For the CPAN6 concept of meta-archive, we define the following basic components

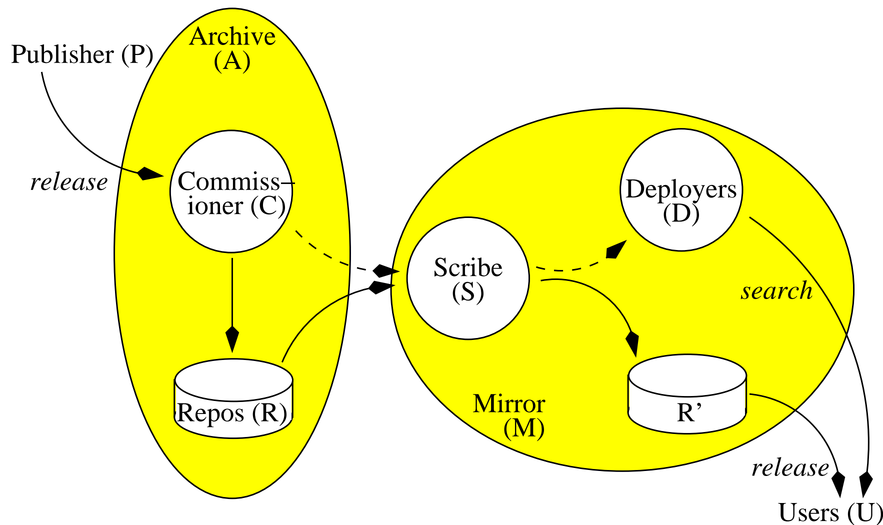

Figure 4: CPAN6 general structure.

and roles:

1. Some person who has software, documentation, or other information to be made public is named the **Publisher** of that information.

2. Published information is built on a directory containing **Files**. Files are defined as sequences of bytes, labeled with an unique name: the filename[2]. The content of the files is kept outside the archive, on a separate storage. The archive administration is only maintaining the location of the file[3].

3. Files grouped in a directory are uploaded together to form a **Release**. As far as CPAN6 is concerned do single files within a release have no versions: only the whole set has a version. Releases are created for a purpose, whereas files are only made to contain data.

[2]Files used by different projects, different project versions, or different archives may very well be the same, and then stored only once.

[3]Pause6 adds cryptographic checksums of the file contents to its administration. This way, the authenticity of the downloaded data can be checked before its use.

4. Releases have a name and usually a version indication. The name of the release is the defined as the **Project** name. When two releases share a name, than (some way) one will have a higher (more recent) version indicator than the other[4].

5. A set of projects is grouped into an **Archive**. An archive collects projects which have some common ground, for instance all projects that use the same programming language. Or, all papers and tutorial materials produced at a certain conference.

6. An archive sets the rules how its **Name-Space** is managed: releases with the same name are part of the same project. Rules about who is permitted to add releases, start projects, change owner, and so on, are part of the name-space policy for that archive[5].

7. In different contexts (different archives), the same project name can be used to indicate related releases[6]. For simplicity, we define a project definition within one archive as a **View** on the project. The related project name in a different archive is a different view on the same project. In case of accidental name-space collisions, where the projects carry the same name but are not related, those projects are not considered views of the same.

8. The data-files can be stored on local disk, ftp-servers, CD-ROM, anywhere. They may be stored compressed. They should be simple to transport. The collection of files for one archive is called a **Repository**.

9. One server can host multiple archives, either as source and as mirror location of the data. These repositories together are typically kept inside one directory hierarchy, named the **Store**.

10. New releases will be submitted by the publisher to an archive maintainer daemon, called the **Commissioner**. The CPAN equivalent of the commissioner is the Pause indexer daemon.

11. The commissioner is the single point of failure of an archive. Even worse, it might get very busy when it is also used for downloading. Therefore, ‘normal’ users access the archive via **Deployer**s, say mirrors. A deployer contains a copy of the archive administration and has its own repository, located in its own store.

12. Replication is performed by a process called the **Scribe**. It is the Scribe’s duty to perform the physical copying of releases and to be sure that all the indexes are updated correctly, as well as verifying signatures.

    Scribes are used to transfer data from the commissioner to its deployers. Scribes are also used to extract and combine archives into new ones. Scribes can extract one release to get installed on a user’s system, but also publishing (uploading)

[4] The meaning of the version indicators is interpreted by the archive administrator, i.e. Pause6.

[5] Pause6 implements various policies, from very generous to very strict rules.

[6] For instance, the ’c-lib’ archive may define a project ‘Tk’ to contain a set of `.c`, `.h`, and `Makefile` files which are used to create a version of the Tk graphical library. But at the same time, the ‘perl5’ archive can define a set of files, containing a `lib/Tk.pm`, `Makefile.PL`, and some XS wrappers with the same project name. Those are related.

releases are tasks for a scribe.

There will be many versions of scribes. They have an reading side, to download information from the source, and a writing side which uploads information to some destination.

13. Uploaded or downloaded information needs to be checked for consistency and authenticity. This will be done in the commissioner, when it receives a new release, in the deployer, and in intelligent scribes. These general applicable checks are **Auditor**s.

14. Finally, on the end of the line are the **User**s: the people or automated install tool which requires releases to be installed on a system. The User will have a `cpan6` program, which offers general CPAN6 interrogation and maintenance features.

Pause6 –as we will see in later chapters– adds more components to the infrastructure. Those components are very useful, for instance adding security to the archives, but not part of the CPAN6 concept.

3.2 Physical

To start any CPAN6 archive, the physical location of the following four component classes have to be defined:

Commissioner

is a task for a daemon which is able to make changes in the archive's source repository. It will write received files into the repository directly, although that repository does not need to be located on the same physical system.

Deployer

is also a task for a daemon. Deployers administer full mirrors of the archive, although their information may be outdated by a few hours. They provide search facilities to the users. Deployers need write access to their own repositories, which are not required to be on the same physical system either.

Store

a network accessible directory which contains the repositories for commissioners and deployers.

Scribes

are processes which implement replication. Some will be started with `cron`, but it may also be started manually. Scribes talk to commissioners and deployers. They are also used to upload releases from publishers and download releases to users.

In a typical situation, the deployers will run on different hardware than the related commissioner of that archive. The repository is usually kept in a store on the same

system as where the related deployer or commissioner daemon runs to simplify configuration. Scribe processes will usually run on the system of the destination process: that simplifies the access configuration considerably.

In very small set-up, everything can be located on a single system: one daemon which plays as commissioner and deployer for an archive, where the data is stored locally. No scribe is needed in this case.

Off-line install tools will behave like a one-shot combined deployer process and scribe.

Combining tasks

Usually, one physical server will run only one CPAN6 daemon instance. Figure 5 shows that commissioners and deployers can be joined in one daemon. The archive

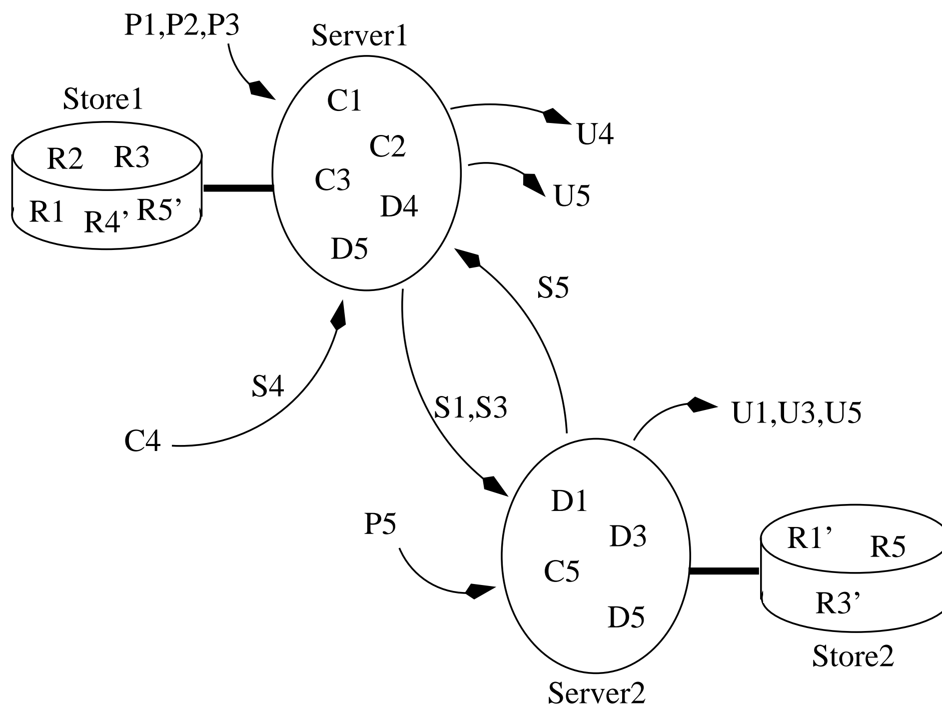

Figure 5: CPAN6 allocation of systems.

implementations within one daemon may differ, but the CPAN6 protocol ensures that they can be combined. This is required to be able to smoothly transition to new archiver implementations in the future.

Complexity

People who want to install software on their system need to have the pre-requisites of the projects involved. This includes the software which implement the client side of the archiving software (initially a pause6 client). They also need to have read access to one of the archive stores.

There is no need for any version administration knowledge, or such like: that is implemented in the search queries, run by the deployer. For off-line installation, a very

simple search engine (like SQLite) will suffice.

The server-side (commissioner and deployers) require the server-side implementation of the archiver, which may involve version control and probably databases to speed-up searches.

3.3 Hierarchy

There is no archive hierarchy defined by CPAN6: no pre-defined relation between one archive and the other. However, archives can be built using selective merging of other archives and own data; they can be configured to simulate hierarchies.

Archive relations are implemented by the scribes as shown in figure 6. CPAN6 itself

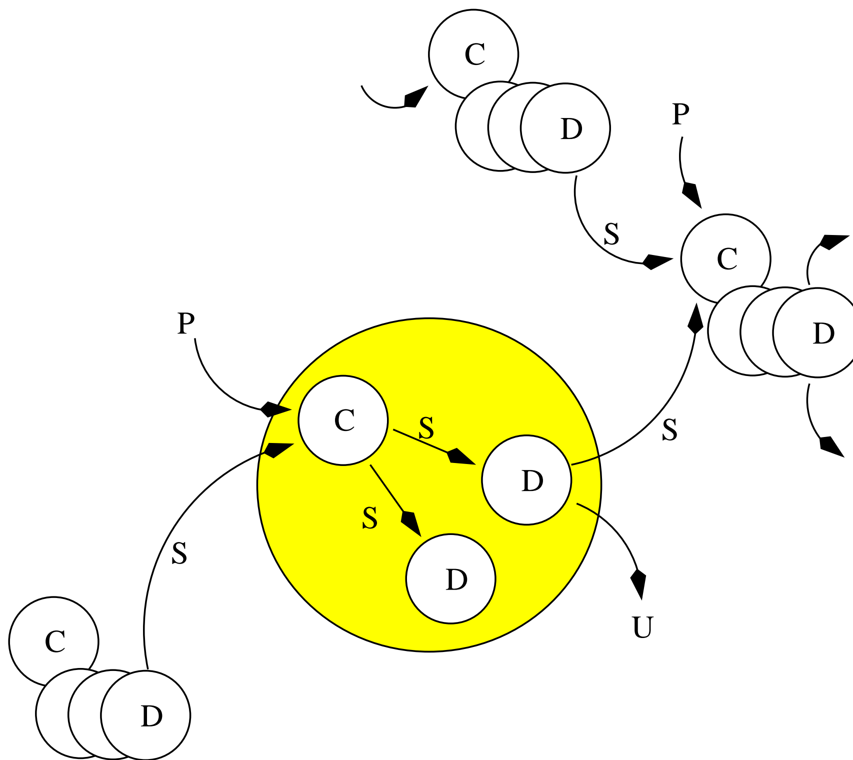

Figure 6: CPAN6 archives as web.

does not know about archive hierarchies, it only facilitates their use. By creating the right configuration, you can define your own virtual archive hierarchy: an ordered list of archives to visit to solve the search for a certain release.

Merging and Filtering

New archives can be build as sub-set or merge of other archives. Besides, an archive may accept new releases directly from a publisher.

For instance, you can configure a local archive to contain only the latest releases of the projects in the Perl5 archive. Define a scribe which searches the original archive

for the latest versions, and copy the discovered new releases into your own archive. Flag all updated project releases as expired, and eventually their will be removed from your repository.

To get this to work, the following is provided:

- scribes can source from one or more commissioners and deployers;
- scribes can be configured to filter releases (reduced search);
- one archive may be fed by multiple scribes and publishers.

Fake a Hierarchy

It is easy to construct a ‘traditional’ hierarchy. For instance, the default configured hierarchy for the ‘perl5’ archive will probably be a listing of a location in your own home-directory, one on the system-wide level, and then the CPAN6 core server.

In serious development environments, the number of (pseudo-)levels may grow. It could be:

- personal development archive
- personal installation archive
- system-wide installation archive
- department-wide installation archive
- company-wide installation archive
- commercial value adder (like ActiveState or CPANTS)
- CPAN6 central archive

In most situations, you would not configure all of these locations between each user, as searches queries can be configured to get redirected to other archives.

The “company-wide archive” would probably have rules which determine which internal releases are allowed to propagate outside of the company, probably only after permission by the project management.

3.4 Processes

One daemon can serve as commissioner for multiple archives (even of different implementations), and at the same time as deployer for other archives, as shown in figure 5. Furthermore, CPAN6 defines scribe processes.

This section will explain the basic ideas of these processes in more detail. For details on the protocol level, and other implementation issues, refer to the paper [1] “CPAN6 implementation”.

Commissioners

The commissioner of an archive takes care of the name-space of the archive. Intelligent archive administrators (like Pause6) will add a stringent name-space control and security mechanism to provide trust for the users. However, unsecured name-spaces may come available as well.

The commissioner provides services to publishers of information, but tries to avoid contact with the users. So, the following tasks are to be implemented:

- list deployers for the archive for users
- start a project by an author
- upload a release by a publisher

For predefined scribes only, the commissioner permits services which the deployers provide to normal users. For instance, the scribe can freely search the commissioners administration.

Deployers

The difference between commissioner and deployer is artifical; on the configuration level only: who is permitted to ask which questions. The deployer can ask the commissioner to get releases, where normal user's cannot.

The deployer receives its updates from a scribe process. So: the scribe is for the deployer like a publisher for the commissioner. It will use the same mechanisms for access control.

Deployers provide the following services to users:

- search the archive
- download release meta-data

Remember that the release data is kept in the repository, inside some external store. The only information that the user gets from the deployer is a list of file meta-data. For each file, one url is listed where the content can be found[7].

Scribes

Scribes are used to transfer data. Scribes are also used to extract and combine archives into new ones. Even extracting one release to get installed on a user's system is the task for a scribe.

CPAN is distributed by ftp-mirroring scripts, usually simple rsync calls. They copy the new releases and the text file which contains the module-list. Such a script is just one implementation of a scribe. CPAN6 scribes are in concept more powerful than an rsync script. Scribes

[7]The Pause6 implementation adds security to this protocol. It will administer crypto-checksum on the release list, and on the content of each file. Some other implementations may not be that protective.

- administer the *trust* and license of the copied data,
- they can copy release by release, such that the destination archive can be a subset of the source archive.
- their reading may use a different transport protocol than the writing. For instance, read (download) by HTTP and write to local disk. Or read from local disk, and write via SOAP.

Like Internet traffic routers, scribes can interfere with the communication on different levels of understanding: from a very dumb rsync implementation which does not inspect the transported data, upto high-level, firewall-like inspection of each copied release by starting auditor processes.

Daemon configuration

The daemon configuration itself is also kept as a simple archive, containing local archive references. That archive's projects each describe one locally kept commissioner, deployer, or scribe. This way, the general configuration gets all the archive features:

1. searching in locally kept archives;
2. adding and removing of 'real' archives;
3. release management over archive configurations; and
4. trust definitions.

The initial daemon implementation will require that the global configuration is kept in a Pause6 archive.

4 Pause6 Organizational

For Pause6, we introduce quite a few extra features on top of the base requirements for CPAN6. Pause6 builds on CPAN's heritage, a trustworthy public archive.

4.1 Additional components

Based on the infrastructure rules of CPAN6 (as shown in figure 4), Pause6 introduces extensions leading to the situation of figure 7.

In addition to the terms defined in section 3.1, Pause6 adds the following:

1. People (like the publisher) and processes (commissioners and deployers) will add their cryptographic **Signature** to release information. This way, the end-user can check the authenticity of the downloaded release meta-data by verifying the signatures.

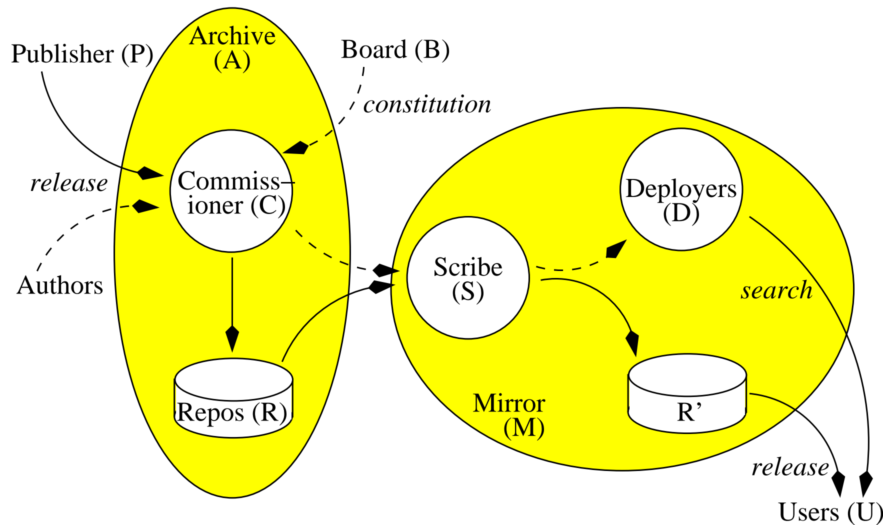

Figure 7: Pause6 extensions to CPAN6 base.

2. Where each release has only one publisher (the person who uploaded the data), each project will have one of more **Authors**. It is possible to require that certain number of authors have to validate (put their signature on) a new release before it will become the leading release for the project. It is also possible to accept releases from a publisher who is not one of the authors.

3. The content of the files is not important as such, but is indirectly validated. Each release has a list of files and their location. For each file, a SHA256 (or safer) checksum is kept in the meta-data record of the release. All meta-data together are signed by the publisher, (some of) the authors, and all handling processes[8].

4. Archives behave according to a set of rules. These rules are called the **Constitution** of that archive. In the default Pause6 configuration do authors ‘own’ the project’s name (their place in a name-space). The authors decide who can add new releases with the same project name. By default, only any author can be a publisher for the project. Such configuration rules are part of the constitution.

5. The constitution is written and signed by one or more persons, together forming the **Board** of the archive. Procedures around board changes are as rules part of the constitution as well. An archive will be trusted more when the board contains people you trust.

   You may see the constitution as a project, and the board as the project’s authors. It is implemented that way.

6. Publishers, authors, board members, and (commissioner, deployer, scribe) processes all have an **Identity**. The identity is used to get authentication information of the person or process. Identities are usually stored in special archives, for instance based on the PGP infrastructure or PAUSE-ID.

[8]This way of adding trust to distributed information was developed by Linus Torvalds for his tool GIT: the release mechanism for the Linux kernel.

7. The signatures will be checked on each transport of a release. Signatures, being kept in archives which you trust to a certain level (a value between 0 and 100), are used to calculate the actual level of bf Trust you have in the data.
8. Users will get a **pause** script which enables them to do maintenance on archives which use the Pause6 archiver implementation.

4.2 Projects

In the simple few of past decennia, some hobbyists produced a limited number of lines of code, and posted that via a telephone line on a BBS. Later, it got a little more sophisticated with Internet sites and ftp-servers hosting the code. Those home-brew (Open Source) projects grew larger and larger, up to the size of commercial products. And people grew older, and were replaced by new developers and maintainers.

Over time, what defines the project? Developers change, project names change, licenses change (sometimes), used (supported version of the) programming language change. Everything changes in the long run. And some projects even die....

The current CPAN is not prepared for major changes in the project organization. Only the meta-data of the last version of the module is searched, and there is no way to trace changes over time. You cannot see when the license changed, the author was replaced, or the development status updated. There can only be one author, and passing ownership over is not clean.

Releases

What is a project? On the CPAN6 level of abstraction, there are only releases: sets of files which get distributed. From the Pause6 archiving software angle of view, those releases contain data and meta-data which may need *interpretation*. The meta-data contains the project name (by utf8 string) it belongs to, and a version string (an utf8 string as well). A ‘project’ is a set of releases with the same project-name.

CPAN6 defines releases as very simple things: just any set of files. Therefore, the release structure can be used for many different purposes. Not only normal published information will be kept as projects, but also the constitution, archive index, identities, license descriptions and so on fit in this definition.

We will differentiate projects based on their purpose (their project type) in Pause6, but not on the abstract CPAN6 distribution network. Each project type will use releases which have a state, have access rules, and may require signatures to become available.

Publisher and Authors

The *publisher* is a one-time thing: it is the person who put one single release (for a project) in the archive. The authors of that project are longer lasting entities: one or more persons who are responsible for the project. Usually authors develop a new release together, and finally one of them uploads that release becoming its ‘publisher’.

Authors can permit everyone to publish a release for a project, as long as they realize that it makes DoS attacks at the version numbers very simple. The board can enforce that the publisher is an author.

Versions

Of course, there is more about *a project* than just a set of releases. For instance, from the set of releases you need to know which one has the highest version number to be able to install the latest release. But now, we already reach vague areas: there are many different version numbering schemes. Pick one or be flexible?

Whether Pause6 (or one of its future alternative implementations) fixate the versioning scheme or has a more flexible approach, it cannot be the case that each project defines its own versioning approach: it must be consistent over the whole archive. This makes that a publisher can simply submit a release without the need to tell the archiver what the previous release was; something which is hard to do when multiple publishers can add new versions for a project at the same time.

The meaning of the term 'version' is the main difference between versions of releases in an archive like CPAN, and versions used by version control systems (VCS) like CVS and SVK. In a VCS a new version is defined as the changes to the previous version. In an archive, each release stands for itself and are versions used only to determine order.

States

Releases have a life-cycle. They start-off in state *upload*, when the publisher starts uploading new release content. That the person is permitted to upload is checked beforehand. This state will block other authors for attempts to upload a release with the same version for some time.

When all release data has been received by the commissioner, the release will be flagged *published*. Then –dependent on the rules of the archive's constitution– one or more of the authors have to bless the release as valid, as trustable, by signing it. In the simplest case, the publisher is the only author of the project. The publisher signs automatically, so the published state is reached immediately after the upload is completed.

When enough signatures of project authors have been received, the release will get flagged as *released*. Whether this release is some development version, or ready for end users is beside the point: states of the projects are not about version number interpretations but about data availability.

Most releases will stay 'released' for ever. However, there are ways out. Authors may flag a release as unwanted, for instance because of serious bugs. The state is changed into *deprecated*, which will keep it in the archive but avoids new installation. End-users may get warned when this happens for modules they have installed. In their administration, that release is in *installed* ('used') state.

Authors may provide an expiration date for a release at any time. The state of the release automatically changes to *expired* when that moment is reached. The archive may decide to physically remove all expired items and may provide a way to revive expired releases.

An other special state is called *embargo*: the release is waiting to go to the released status, for some reason. For instance, the public availability has to wait for a certain moment of time or until all deployers have picked-up their copy. The daemon will make the state move when the condition is satisfied, each daemon for itself.

The *rejected* state is reserved as very serious deprecation: when (in most cases by the board) it is decided that the release is either not suitable for the archive because of its content, or not to be trusted at all.

Parallel trails

The creator of a new project allocates a name-space. He (or she) will thereby become the sole author of project. The board may wish to approve this allocation with one or more signatures. Within the bounds determined by the board, the author may add new authors and set rules how this name-space is managed. It may be configured to permit development tracks besides the main track.

When the name-space is created, this will initiate a 'naught' version. The board's configuration default for projects is set as meta-data for this project. Each release which follows this one, following the rules and extending the previous release is part of the main trail. However, it is possible to divert from this trail, to set-up parallel development. These parallel developments may eventually be merged back into the main trail.

Figure 8 shows more details than words can explain. The links are defined on CPAN6 level, but their interpretation is a Pause6 task.
In the figure, the '0' release is the result of the the project creation. It contains no files,

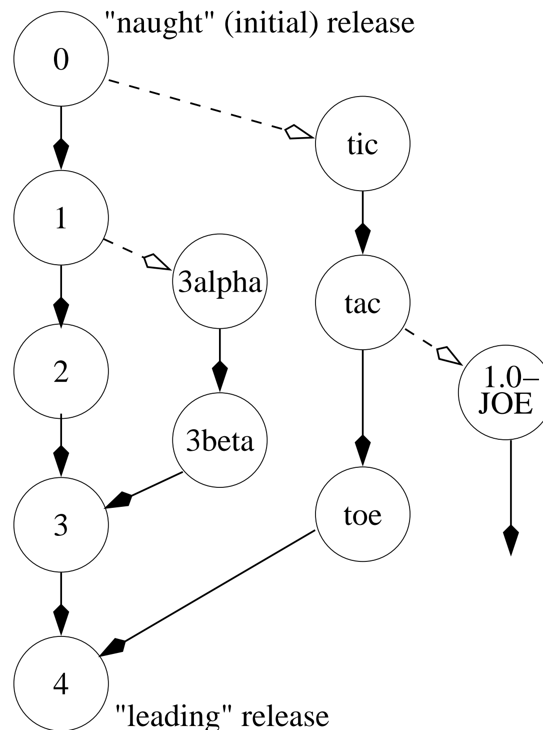

Figure 8: Pause6 release relations.

but only the initial configuration meta-data, where the permissions are the defaults as set by the board. The publisher can modify them with the next release, but is limited

in its freedom by bounds which are also defined by the board. The '1' release is the first one to contain real files.

Some time later, bugfixes and extensions result in a '2' release. The release labels are really unimportant as such (full utf8 freedom by default) but it is good practice to make them sequential containing a number which is incremented in a regular way. Of course, the board can determine which version numbers are accepted. **The actual version label used does not matter as long as it is unique.**

At the same time that release '2' is prepared, someone (or a group of people) decide to start a major redevelopment cycle, in which they produce two intermediate results (3alpha and 3beta) before it is presented to all users by presenting it as release '3'. That lease must extend the interface of both release '2' and release '3beta', and the permission requirements (sufficient signatures) of both trails shall be met before it becomes active.

On the moment that some project is required on the user's system, by default the leading (last) release (which is in `released` state) of the main trail is used.

When an user upgrades its installed release, it will stick on the same trail. So, '1' is upgraded into '2' unless explicitly specified otherwise. In situations where multiple versions of the same project can be used in parallel (for instance Perl6 or shared libraries), this is sufficient: when some other project requires '3alpha', it will be installed in parallel (and eventually upgraded into '3beta' and '3').

When only one release of the project can be used at a time, the user will need to be warned that a trail switch is planned. Probably, this will require a manual decission what release is preferred to be kept.

Within the default limits set by the board (used for the 'naught' release) or the authors (other releases) even more regorous independed releases may get published, using other version label schemes. The order of releases is part of the meta-data (partially automatically filled-in to help the regular publisher), and does not depend on the used version label itself. It may contain a reference to a certain project (like 'SSL') or a Pause-ID (what is planned for the Perl6 versioning scheme).

4.3 Web of Trust

On CPAN, persons are identified by an unique PAUSE-ID identifier managed by the PAUSE system. With a PAUSE-ID, you can claim any distribution name in the archive name-space which is not yet occupied. But who is really hiding behind a PAUSE-ID? You can never be sure. Even well-known IDs can be hijacked, because it is only a simple username/password validation scheme which is used.

On a few spots in the communication protocols of Pause6, people and processes have to establish contact. They have to identify themselves, although that may be an anonymous temporary identity. The board will decide how strict the archive behaves, and which identity archives –which authentication authorities– are accepted.

Companies and communities may want to know whether the publisher and authors of a release are authentic; really the person they expect it to be. This also counts for the

archive itself: who is responsible for it? Therefore, Pause6 implements better authentication schemes.

Trust

There are two views on trust:

- can we trust that we have the untouched, originally published code? This is about trusting the administration in the network: do we trust the board, the transport protocol, the authentication schemes used? Is the received release authentic?
- can we trust the publisher? Is that code really coming from the person we met or contracted?

The current CPAN/Pause gives only little trust in all areas. PAUSE-IDs are sometimes recognized by the users, simply because the owners are well-known figures. The username/password scheme is quite easy to break, but not that easy that it is done on a daily basis. There is some trust in the publisher.

The CPAN releases are distributed to many ftp-servers, and this distribution process can get corrupted. Luckily, ftp-servers are maintained by professional people, which reduces the chance of them having criminal intentions. There is some trust in the distribution process.

Pause6 will trace the trustability of both publisher and transport. You can investigate these trust calculations before installing anything on your system. Some download paths will be more safe than other paths to collect the same data: a sufficiently safe path will be used.

Authentication schemes

Three authentication schemes are considered initially:

1. PAUSE-IDs, which are weak but better than nothing. This support is required to be able to start with the current CPAN user community, without hassle.
2. PGP keys, established at key-signing parties or CAcert. PGP/GPG security seems broken, where the default uses too weak keys.
3. SSL keys, provided by Trusted Third Parties or CAcert. Especially useful in improving the transport trust.

Archives follow the rules of the constitution, which are signed by the archive's board. For each of the board members, the trust is known. For the user, the most trusted board member defines the trust of the commissioner of the archive.

Checking trust

The content of the files is not important as such. The content of the files do not need to be maintained in a safe way: Pause6 distributes the secure checksums of the files which are validated by signatures of the publisher, authors, and processes which handled the data. After a file has been downloaded onto the user's system, the checksums shall be checked automatically to compute the level of trust for the received data.

The checksums on the files do not determine the trust, although the quality of the used checksum type is taken into account when the trust is calculated. The signatures the base for determining the trust: they determine the authenticity of the checksums. It is the trust which is assigned to the user who has put his or her signature on the release what counts most.

An end user has a private collection of known public keys, collected for instance during a conference in face-to-face meetings. Releases signed with one of those keys have a very high trust of authenticity. Public keys which have to be looked-up via internet provide a lower trust, for instance because DNS can be spoofed. And a weaker exchange protocol lowers the trust further.

The commissioner will sign all received releases, just as the publisher and authors of the project do. The deployers will sign the releases as well. Those daemons have their own public identity. The user may choose how to validate the authenticity of the code: take any signature available which can provide a high enough trust.

It is very well possible that the publisher has changed its crypto-keys since the release upload, because keys can expire. In that case, the trust depends on the amount of trust you have in the commissioner, that it has checked the publisher's key during upload. When the data is accessed off-line (distributed on CD), your trust is limited to that in the key of the deployer (the installer process).

One must be aware that keys do change or may not be available. The trust about a release will change with that. It is the user's decision whether the release is trusted sufficiently to be installed or not.

Trust on a release is calculated

- by checking the authenticity of the publisher or (one of) the authors,
- by checking the signature that the commissioner put on release; the commissioner has checked the authenticity of the publisher when the release is uploaded.
- by checking the signature that the deployer put on the it; it trusts the commissioner to work correct for some (high) degree. It may also (try to) check the publishers authenticity.

Identity Archives

User and process identities (public keys and some meta-data) are just a set of files, and therefore can be kept like any other project in Pause6 archives: each identity as one "project". Although any archive can contain any kind of project, it is probably wise to

create separate archives for identities just for clarity.

Pause6 will need to connect to the existing PGP and SSL infrastructures, which require us to implement external connectors. Those extensions are on the level of Pause6, directly on top of the CPAN6 infrastructure. Those connectors need to be implemented at the client (is user) side.

4.4 Archives

The archive administration as maintained by the Pause6 implementation for a commissioner (or deployer) contains the following components:

- a constitution (the configuration)
- list of deployers and scribes
- name-space administration (projects and authors)
- release details (state, signatures)
- archive references (information about other archives)

The precise layout and content of components is included in the additional paper [2]. That paper also contains use-cases. In this chapter, we stick to functional design issues.

Each of the archive administrative components is put in a separate directory in one or more files, just like any release administration is kept in a separate directory as a set of files. This enables us to transport and cache these components as normal project releases. It also provides the same security features to administrative data as normal projects get.

Intentionally we repeat again that the released material (the release content) is kept in the repository –part of a store– and not in the archive administration. The administrative files are kept as projects as well, but not always visible in the repository: only if the board decides to release them. That data can change rapidly, and hence may be seen as continuously in an intermediate development state.

Archive references

When the number of archives grows (from 1 in the CPAN case, to a zillion in the CPAN6 case) it will become harder and harder to find the correct parameters to connect to a specific archive. Especially in a case where public keys need to be distributed, the configuration can become hard to do. To simplify this process, Pause6 archives can include *archive references* to other archives.

Archive references are simply a set of files, again maintained as any other release. Where perl5 distributions have an installation tool, also these reference projects have such a mechanisms. To contact a new archive, simply download the latest release of the referencing data into the correct configuration directory in the user's home, and off you go.

Project classes

Pause6 not only has releases (which are defined on the CPAN6 level) but adds versioning and interpretation. An archive is a name-space, which is under (configuration) control of the board. Each item is kept as a project, which can have releases.

The following project types are defined:

publication any normal author assigned project. Each released Perl5 distribution is a publication;

archive reference contains information how to contact other (related) archives;

identity contains information about a person, especially a public key.

process contains information about commissioners, deployers, and scribes;

constitution describes the board and its rules;

index has the current list of releases, but may not be published.

license contains licensing information; the text in various languages and the tools for automatically detecting the particular license text in source data.

Other organizational project types may be added in the future. In most case, by far most projects will be either a normal publication or an identity. Other types appear only a few times or in dedicated archives.

4.5 Configuration

Configuration is done by the same mechanisms are releases: a change in a configuration may also require signatures. Two special archives are created (at least): one for the system global configuration, and one for the end-user. For more details and examples of the set-up, see the pause6 implementation paper [2].

Daemon configuration

Each daemon can play the role of commissioner, deployer, and scribe for any set of archives. When the number of archives become larger, you prefer to have configuration of the daemon split over multiple components, just like the Apache webserver configuration proposed after some of the site became unmanageably large.

Pause6 makes this task very light: any archive which is under the control of the local daemon is listed as archive reference in a predefined global archive. The board of that archive is formed by the system administrators.

Because of their role as board for the global archive, the system administrators can set the rules. They can decide that anyone can publish a new archive reference in the top-level (automatically starting that archive with the publisher as board), or that it will require a signature from one or more of them. They may also decide that only they can add new archives. The whole way this configuration is handled is exactly the same as how normal archives are treated.

User configuration

An user needs to configure its environment: which archives to access, its identities (a user may have more than one identity, usually from different authorities), administration which releases were downloaded, and so on. This works-out as a set of archives as well, with very relaxed settings. In stead of automatic replication, as used between general archives and between the commissioner and its deployers, here the user will initiate manual replication; specific replication commands.

To avoid the need for manual intervention in case of name-space collisions –which probably will be frequent because of the various views on the same project a user may need– we cannot keep all downloaded material within one local archive. The user's tools will build a set of archives, each representing the user's activities from one remote archive. And, of course, the user has its own "global" archive containing archive references to its local copies.

The primal uses for these personal archives are

- temporary storage for downloaded publications, until the release information is used. For instance, a Perl5 distribution is kept there until the software is installed;
- keeping track on what is installed from which archive;
- personal identity management;
- references to the known archives;
- caching of useful information, for instance identities; and the
- collection of own published material.

The set of archives will usually be a rather sparsely filled directory tree, like gnome configuration trees.

Of course, the administration of personal archives is not done by a daemon: users do not create daemons. Therefore, the commissioner, deployer, and scribe functionalities can also be used in one-shot mode. This mode is also used when projects are taken from CD/DVD, at system installation time: also in that case, there will not be a daemon at hand to keep the general overview over multiple processes.

The one-shot mode will require considerably more processing power per command, per action, requiring configuration data to be processed each time, where a daemon can cache that. On the other hand, there will only be one system using the data; the daemons may get thousands of requests per minute.

5 Epilogue

In this paper, the global design of both CPAN6 and Pause6 were discussed in short. Both designs are relatively fixed, shaping the whole undertaking. Other documents have a more dynamic state, reflecting implementation issues.

Have a look at the project's website at `http://cpan6.net` for the latest versions of all papers and software. This will also be the place for mailing-list and tutorials.

References

[1] Overmeer and Vilain, *CPAN6 Implementation*.

[2] Overmeer and Vilain, *Pause6 Implementation*.

List of Figures

1 Simple ftp-server distribution structure. 3
2 CPAN/Pause set-up. 4
3 CPAN6 simplified general structure. 11
4 CPAN6 general structure. 12
5 CPAN6 allocation of systems. 15
6 CPAN6 archives as web. 16
7 Pause6 extensions to CPAN6 base. 20
8 Pause6 release relations. 23