Pause6 Implementation

Mark overmeer*

July 28, 2009

Abstract

This document describes implementation details for Pause6, such as file formats and use cases. This results in XML Schema's, which define the implementation requirements.

Contents

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

Introduction

The CPAN6/Pause6 project is described in a few different papers. This paper describes file formats and use cases specific to the Pause6 implementation. This information is useless without the terminology definitions from paper [1] (“CPAN6 and Pause6 Design”), and the base implementation of CPAN6 in paper [2] (“CPAN6 Implementation”).

1 The daemon

The Pause6 interpretation of the CPAN6 network features smart servers, and dump clients. The server will run a daemon, which plays many roles.

One daemon runs as Commissioner and Deployer tasks for many archives at once, as described in paper **??**. The Commissioner accepts new data into an archive, and the Deployer make the archive searchable and release downloadable. The terminology difference between commissioners and deployers is artificial: they have different default access rules. It is also easy to configure one task to be commissioner and deployer at the same time.

Besides archives, there are Stores and scribes. The stores will contain the data (per archive in a separate Repository), and the scribes will distribute the data. Stores are usually directories on local disk or on a remote ftp-server.

1.1 Deamon configuration

The daemon is simply an archive containing local archive references only. Other project types are not permitted on this level.

Initially, the archive will certainly be a Pause6 implementations, with strong update restrictions set. The core archive's board is responsible for the global server set-up. It is king over what role is played for which archive (commissioner, deployer, or both) but will not bother about internal archive configuration.

2 Archive layout

2.1 pause6-constitution

The project named "pause6-constitution" contains the configuration of the archive. The constitution is published as a release (a directory with some files in it) signed by a suffiently large number of board members. It must be visible to all archive accessors.

The constitution contains the following file:

- `constitution.xml`: is the main configuration file

***name-space-layout.xsd

2.2 pause6-references

The board members control this project "pause6-references", which is publicly available information about the configuration with less restrictions than the constitution. Usually one board member can change this settings, where the constitution requires all-but-one signature.

- `archive-list.xml`: list of archive abbreviations
- `deployer-list.xml`: list of deployers

archive-list

The archive list is a single file which maps archive alias names to project names within the archive. These archive reference type of projects are simply directories with some

data in it, especially the location and the public encryption key. Those project names will usually start with `archive-`, although that is just by convention.

Archive references have a trust component in them: that part of the reference data is stored in the archive-list, and has to thereby is signed by the board before it is published to the users.

The archive reference project information can get updates. The board may decide that the owners of the referred archives are permitted to update their referencing information, for instance can update the public key themselves.

deployer-list

Deployers are defined as identities, which on their turn are also maintained as simple projects. The preferred project names for these deployers start with `deployer-`.

The board may decide to open-up that name-space to everyone, so that people can add themselves as deployer without interference. A reviewer process is required to check the correctness of the configuration in such case. The reviewer can update the deployer-list.

2.3 pause6-index

The "pause6-index" project contains the name-space administration of the archive. The index is a directory, like all Pause6 items are kept like project directories. However, in this case the directory will usually not get signatures and versions and therefore stays hidden to normal users.

The board can decide to publish this index on regular intervals as project, or hide it such that people can only query the archive via the regular interface.

The index will contain:

- `release-list.xml`: listing of releases
- `archive-log.xml`: global overview of modifications

release-list

A single file lists all releases. Per release, it contains the humanly understandable project name (in utf8, blanks and other nasty characters permitted), a version indicator (utf8, syntax unspecified), project type, status flag, and a location. The constitution can restrict project-names and version numbers with regular expressions.

In general, it is not possible to use project names and version numbers in actual filenames, because these strings may contain characters which are not permitted in the platform dependent filename syntax. Therefore, releases will get a (random) number assigned during upload, and the path to the release administration (the location) will be derived from that number.

The location may also contain `<archive>:<project>`, in which case the browser will need to check the `archive-list` for additional information and retry the query to the new location.

The project indicator can not contain an explicit version number; these redirections are used for the merging of archives: either to solve naming conflicts or to do a lazy –non-download, abstract– merge. In either case, the version identification does not change.

archive-log

This file gives an overall activity overview on the releases. When the archive contains many releases, inspecting their individual moves will get expensive. This log will show which releases changed their state. The archive-log can be used to inform deployers and derived archives which releases they need to inspect to become up-to-date.

The archive-log will grow with one line per release change. Happily, the number of state changes per release has a relatively modest maximum. However, in case this file grows too large it may get condensed, leaving only the last state change per release.

2.4 Project releases

In the next chapter we will detail more on the expected content of projects, and how to work with them. On the administrative side of projects, we see the following files (again: together in one directory per release)

- `meta-data.xml`: contains the meta-data of the release, which will not change over time;
- `release-log.xml`: detailed trace on all actions involving this release;
- `status-current.xml`: extract of the log, listing what has sufficient signatures. (The current state of the release)
- `status-next.xml`: extract of the log, listing what may not have sufficient signatures yet. (The next state of the release)
- `repository.xml`: contains the actual location (or locations) of each of the non-inlined release components.

The reasoning behind this becomes more clear with the examples shown in paper [2].

meta-data

The meta-data file contains meta-data which is provided by the publisher on the moment the release was uploaded: project name, version number, publisher's identity, list of filenames, description, licence, the current list of project authors, and some project maintenance rules (a kind of project private constitution).

release-log

The release-log contains all changes and all changeable meta-data: signatures of authors, explicit project state changes (for instance deprecation), release expiration.

In Perl5 distribution terms: the `Makefile.PL` and `META.yml` files are used to specify some meta-data information. Pause6 compliant user tools can extract this information to provide Pause6 with sufficient data in a standardized format.

status-current

The status file is a combination of all static information from the meta-data file and an extract of the dynamic release-log information. This information is completed with sufficient the signatures of authors and CPAN6 commissioners, deployers, and so on on that data section of the file.

Normal users will work with this status-current file: before they download actual files, they need to inspect this meta-data.

status-next

This file is for the authors of the project: it contains the same static meta information but a slightly different dynamic release-log extract: showing the next state of the release. The number attached signatures is not yet sufficient to tell the users.

2.5 Archive references

In a large mesh of archives, it is hard to collect contact information. Therefore, information of remote archives can be kept as "projects" inside an archive.

In most cases, these archives will have an alias defined in the constitutional archive-list. In any case the reverse is true: each alias declared in the archive list requires an archive reference project.

The archive directory contains the following files:

- `definition.xml`: describing the remote archive
- public keys and other info required to trust the initial contact to the archive.

The board may decide that other external people own (and maintain) these references, or can keep this in their own hands.

definition

This file lists the url of the archive, information about the archive's type, and contact protocol information. Amongst others, the name of the file which contains the public key is listed. That key can be used if the public key infrastructure cannot be reached (i.e. no network available or limited system capabilities).

3 File formats

3.1 General project files

The concept of 'projects' is mainly for the users, but not that much for the Pause6 internals. Projects (release names) are used for name-space allocation, and the version

string is used to define (optional and artificial) ideas about a sequence: which release was last?

The Pause6 archiver uses a unique code to store a release, because the (project) name a version string are free-format utf8 and therefore can conflict with the file-system capabilities. For the same reason, filenames as used by the project will get mutulated by the Pause6 administration.

Each of the files is readible in informal YAML or formal validatable XML. See the XML specification is section **??**.

3.1.1 The meta-data file

The meta data is kept per release: over time, any aspect of a project can change: authors, description, rules, the name, and so on. Changes in next releases of a project should not destroy information from the past.

The meta-data has to be provided by the client application. In some enviroments –like perl5–, there is already an infrastructure present to collect these facts: Pause6 requires a uniform representation of that data. During upload, the client-side tool will convert the meta-data into a Pause6 structure like:

```
1  project: Mail::Box
2  version: 2.065
3  authors: pause-id:markov pause-id:sam
4  licence: GPLv3
5  files:
6     filename: Mail::Box-v2.065.tar.gz
7  archives:
8     -
9       name: pause-id
10      type: pause6
11      address: http://pause.cpan.org
```

3.1.2 The release-log file

Each release directory contains a `release-log` file, which contains an incremental trace on the changes. The format is (globally):

```
1  -
2   type: request
3   at: Mon May 1 08:52:23 CEST 2006
4   by: pause-id:markov
5   state: upload
6  -
7   type: trace
8   at: Mon May 1 08:52:23 CEST 2006
9   log: publisher validated, trust connection 10
10 -
```

```
  type: trace
  at: Mon May 1 08:52:23 CEST 2006
  log: upload file
  by: pause-id:markov
  filename: Mail::Box-v2.065.tar.gz
  size: 23123
  checksum: sha1 SDFIWUOWHCJKHGWIUHWKIC2384729f3scewjhk
-
  type: trace
  at: Mon May 1 08:52:28 CEST 2006
  change: store file
  checksum: sha1 SDFIWUOWHCJKHGWIUHWKIC2384729f3scewjhk
  filename: SDF/WUO/SDFIWUOWHCJ
-
  type: signature
  at: Mon May 1 08:55:00 CEST 2006
  by: pause-id:markov
  signer: pause-id:markov
  type: sha1
  authograph: sha1 DCIOWUFW#$&*FWFHKJ@#YKJ
-
  type: confirm
  at: Mon May 1 08:55:00 CEST 2006
  change:
      state: published
-
  type: confirm
  at: Mon May 1 08:57:00 CEST 2006
  change:
      state: released
```

3.1.3 The status files

The `status-current` and `status-next` files are extractions of the `release-log`.

Whenever needed, these status files can be regenerated from the log. The 'next' collects the latest information, but does not require sufficient signatures: it is used to sign release changes. The 'current' collects the latest information with sufficient signatures. If the board decides to change the constitution, that may have implications for the status files: they need to be regenerated.

```
release:
  name: Mail::Box
  version: v2.065
  publisher: pause-id:markov
  authors: pause-id:markov pause-id:sam
  licence: GPLv3
  state: released
```

```
   files:
     -
       filename: Mail::Box-v2.065.tar.gz
       checksum: sha1 GoJWlg9avwW4MzSXbtRZrZPygAw
       size: 592389
archives:
   -
     name: pause-id
     type: pause6
     location: http://pause.cpan.org
   -
     name: pgp
     type: pgp
signatures:
 -
  signer: pause-id:markov
  type: pgp
  autograph: "@#DFKJO@UWUFH@YC"
 -
  archiver:
     role: commissioner
     type: pause6
     address: http://cpan.org/cpan5/
     trust: connection 10, publisher 90
  signer: gpg:cpan5@cpan.org
  type: sha1
  autograph: sha1 DCIOWUFW#$&*FWFHKJ@#YKJC
 -
  archiver:
     role: deployer
     type: pause6
     address: http://perl.example.com/cpan
     trust: connection 100, commissioner 80
  signer: gpg:mark@example.com
  autograph: ASJFIOWURWOHCWjh

```

The authors and publisher all sign the `release-next` file. Each person signs the part upto the dashes, and sends that personal signature to the commissioner, which will add it to the release-log if the signature is correct. Then a new release-next is generated.

When enough signatures where received for the next release state, the commissioner will replace the `release-current` with the next information. Still, more signatures may come in: from other authors who thereby add trust to the release. It is thinkable that signatures are received for project states which already have passed: a warning will be issued in that case, and the signature ignored.

When the release has moved to the next state on the commissioner, that change will be reflected in the global `archive-log`. That log-file is used to update the deployers. Deployers only require a signature from the commissioner to trust the data their receive: they make check the publisher or authors as well, but that data may be expired.

When the release is copied from archive to archive, the commissioner and deployer parts get repeated. That way, the user can trace how a release came to his system. The trace is required to establish the trust of the release.

If possible, the user will check the signature of the release from his most trusted listed author. Otherwise, the publisher, commissioner, or deployer. The last option is needed, for instance, when everything is delivered off-line on CD/DVD. For instance, during system installation or in a secure environment or when installing a new Linux distribution from scratch. The calculated trust on authenticity must come above an user adaptable threshold or ask for human intervention.

3.2 Constitution specific files

The Constitution describes rules of the archive, as degreed by archive's board. Notably, it does not contain any release or owner information.

This is an example how "xml2yaml constitution.xml" could look like:

```
archive:
   alias: cpan-traditional
   address: http://perl.overmeer.net/cpan
board:
   member: pause-id:people/SAMV
   member: cacert:john@nlnet.nl
permissions:
  class: author-defaults
  required-votes:
    minimum: all
    permit: release-add release-accept
```

4 Use cases

In this section, a few use cases are described. The order is rather random, expressing various ideas which got a place in the design.

The list of use-cases will certainly be extended in future releases of this paper.

4.1 Manage a project

If someone wants to claim a name-space, he simply uploads a first release. Of course, the publisher must authenticate itself to a level which satisfies the board. For instance, the publisher must authenticate as part of a certain authority or be listed on a manually picked list of acceptable authentities.

Not only the identity is required to start a project: it must also comply to name-space rules, for instance in the structure of the name and used characters. Besides, it may require one or more signatures of the board to start-off.

If you fulfilled the requirements, you can upload your first release. That release will produce the initial meta-data as required for searching, configuration and such.

The current set of authors is kept in the meta-data of the release. The information provided for the latest 'released' release contains the current set of authors. Until the first release, the publisher the the author.

When no versioning scheme is used, the release of the project which became 'released' last (and still is in that state) is providing that information. The concepts of authors and release orderning are configurable.

When a project wants to change name or to split-up, those new names will be considered new projects. Simply submit a final release of under the original name which depends on its follow-ups. Dependencies are defined in the release meta-data.

4.2 Submitting a releases

When a publisher sends a new release to the commissioner, it will check that the user provided prove of identity is sincere (enough) and acceptible for the archive. When the project name of the release was not seen before, a new project is seeded, with the publisher as author. Otherwise, the permission rules relating to the last release for this project are checked to see whether the publisher is allowed to upload a new version.

The commissioner transports the uploaded data files to the archive related repository within its store. Those files are checked to have the correct checksum before they are written to the store. The size, the checksum, and the original filename of all the files in the release are kept in the meta-data of the release, and will be signed.

When all files have been received, the publisher will download the meta-data file as constructed by the commissioner. It will sign this data with his private key, sending his signature on the data to the commissioner. Then the release state will become 'published'.

On the moment that the release is published, all authors of the project will be able to see (and download) the release meta-data. They may even get an e-mail to inform them that a signature on the release is needed.

4.3 FTP-server

Can we offer additional services to existing ftp-servers? Quite easily: they already have their upload and distribution network in place. The additional service that CPAN6 can offer are:

- checking that the data on the ftp-servers is the same on all servers
- offer package search facilities to users (no detailed search)
- automating the search for mirror servers to the users,

- provide the user with install tools which also do the download,
- administration of installed software.

To set this up is quite simple: take one of the ftp-servers and flag that one as “the store of the commissioner”. Scan the directories of that server every hour (or less) for new files.

The hardest thing is to translate the newly found files into releases. In many cases, ftp-releases are just single file, which makes this a none-operation. Sometimes, the archive adds checks files. Some releases are more complex. Then, a version number needs to be derived (otherwise the files time-stamp is used).

It may be possible to extract more meta-data from the item found on the ftp-server, for instance if it is an rpm-package. It may be possible to do some extra checks on the item, for instance because it contains MD5 checksum (like in rpm-packages). In general, the person who is creating the CPAN6 wrapper will need to spend some time collecting meta-data which is otherwise provided by the publisher.

Once the meta-data is found, and checksum are generated from the files, the release can be added and directly move into the ‘released’ state. The process which collected the meta-data will probably sign the data it has collected, so the source can be traced. All discovered releases in the archive will have the dicovering process as author.

The deployer’s have a much simpler task. As usual, they ask the commissioner for detected releases. They do not have to use a scribe to copy the data, because that is already done by the existing ftp-mirror infrastructure. The only thing they have to do is to check whether their archive contains the same files: check the checksums on the files found.

The default deployer functionality is very close to that of the rpmfind websites: they help locating released material within a set, within the archive. As limitation, deployers will (in the first implementation of Pause6 at least) not know about archive content so not provide for searches inside the released material.

4.4 Daemon global configuration

The examples in figures 1 and 2 show the top-level system global archive and one of the contained archives. Each box is a directory with some data-files, together in a repository on an accessible store. The system administrators found the top-level archive, so are by default the administrators of the top-level archive. They may decide that (a selected set of) users can start their own archive (with or without signature of the board) or that they have to initiate that themselves.

An archive reference has a (project-)name, and can thereby be downloaded and installed in the user’s environment. The content will lead the user to the url of the referred archive, which may be anything. By default –and common practice– the `xyz` archive-ref in archive `http://abc` will lead you to `http://abc/xyz`.

The constitution of the global archive limits the configuration possibilities of the lower level archives, as those constitutions will reduce the permissions that the projects can set. For instance, the sysadmins can forbid any archive to accept publications from non-authors. For this reason, treat archive references as symbolic links.

“global”	address: http://archives.example.com
Constitution	project: pause6-constitution authors: board = sysadmins daemon configuration rules.
Archive-ref	project: user authors: sysadmins, publisher: user:root connects to local username table.
Archive-ref	project: tutorials authors: user:root, publisher: pgp:john published conference material.
Archive-ref	project: pgp connects to the pgp/gpg infrastructure
etc...	
Identity	project: pause6-daemon authors: sysadmins commissioner, deployer, scribe public key.

Figure 1: Example: global archive

“tutorials”	address: http://archives.example.com/tutorials
Constitution	project: pause6-constitution authors: board = pgp:john archive configuration rules.
Publication	project: yapc-eu-cpan6.odt publisher: pgp:markov OpenOffice presentation for YAPC::EU.
etc...	

Figure 2: Example: archive “tutorial”

4.5 User local configuration

4.6 Optimizing storage

The amount of files and storage can be improved by taking into account that various projects use the same files: parallel installed releases from one project, and releases from different projects may have overlapping code. For instance, many projects will distribute the same license file.

As more complex example, the ImageMagick Perl module requires the ImageMagick C libraries to be installed. The Python version of the interface requires the same external library. To be certain that you have the right version of all libraries, and header-files, you have to be very careful about project dependencies.

The solution of many Windows applications is to ship an application with all its requirements as one big integrated block. All dependency configuration is avoided this way. When a user runs multiple related applications on the system at the same time,

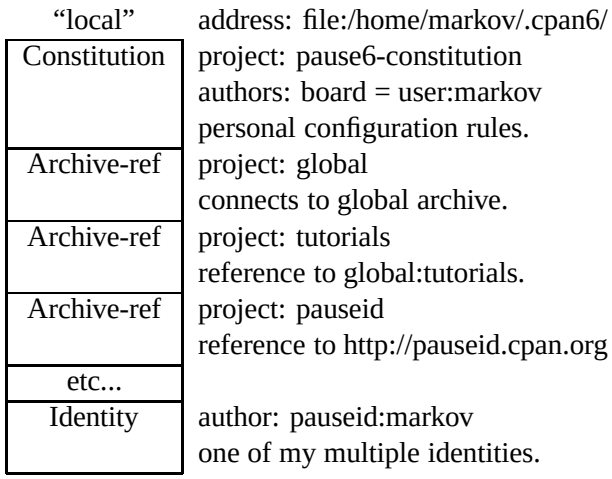

Figure 3: Example: user archive

multiple copies of exactly the same binary may have to be loaded into memory. This spoils disk-space, memory, and therewith system performance.

The GIT version control system introduces a nice concept. Based on the idea that you can determine a simple unique key per file (using MD5), you can easily check whether the file you need is already on the system. The concept of a *product release* did change from *a collection of versioned files* to simply *a collection of files*: that two following project releases both contain a file with the same name but with a different content is more a 'coincident' than a relation. Of course, applications running outside this VCS may interpret it as more than that.

Seeing a release simply as a set of related files which are selected by they checksum, as GIT introduced, has a few considerations:

- multiple archives may build on the same set of files, to represent different *views*. For instance, one archive could show a sub-set of the files which can be installed, an other view selects only the documentation files, a third includes the core files of the project plus extra files to build an rpm, a fourth view only shows that rpm.

  Users can decide which archive they want to address, for instance to copy to local disk. A server may decide to only deploy the pure documentation archive (to create a search facility). To create a (Linux or FreeBSD) distribution CD, one may take a clone of the rpm archive.

- you do not need to apply a sequence of patches to go from a file(name) in one release to the same file(name) in the next one, so both the VCS as the client side can be dumb in this respect. The data can be stored on any general ftp-server or directory structure.

- the release index (part of the releases meta-data, provided at upload) defines that is most important: which set of files will work together. That is more important than tracing the changes between two releases. The precise changes are not important for end-users, in general.

- some (graphical) tool can help you maintaining a project by displaying the related views in different archives. For instance, the tool can detect that the rpm is out-dated because one of its building-bricks has a new release.

Application

The tric of unifying equivalent files from various sources based on their checksum can be applied on many places. On the stores, the files will need to be kept some form of encoded, because CPAN6 permits unicode file-labels and version numbers without restrictions: many file-systems, operating-systems, and transport protocols can not handle those names. Therefore, the release index needs to connect the abstract transport filename with an original filename. Only on the user's system, mutulations may be needed to get the data to work. In the ideal world, the user's system understand unicode filenames.

Stores may consider to use the checksum of the file as abstract filename. Client-applications may be smart enough to keep track on the checksums of downloaded and installed files to avoid dupplicate downloading. Client applications may install a file which is already available elsewhere on the system by creating a hard-link.

MD5 checksums are quite weak: it is possible to produce a different file with the same MD5 within minutes. Accidental collisions are close to impossible, but the are unsufficient to proof trust. Therefore, a better checksum is adviced, either additional in the index or as filename. Which type of checksum is used is part in the trust computation.

Perl5

The Perl community distributes code as tar-balls (files packed into one `tar` archive which then is compressed using `gzip`) This has the advantaged that all files are loaded at once, but postpones the above optimization of file-folding to after the downloading. Besides, how to handle unicode filenames within the tar archive (conversions not supported)?

It may be useful to change the current tar-ball practice for these reasons.

A Schema: Pause6 basic types

The schema defined in this section defines basic types used for Pause6. They extend the basic types of CPAN6.

A.1 Schema wrapper

This schema uses the Dublin Core (`http://dublincore.org`) definitions for document meta-data.

```
1  <schema
2    xmlns="http://www.w3.org/2001/XMLSchema"
3    elementFormDefault="qualified"
4
5    targetNamespace="http://cpan6.net/2008/pause6-basic"
6    schemaLocation="https://xml.cpan6.net/schema/2008/pause6-basic.xsd"
```

```
    version="1.0"

    xmlns:p6="http://cpan6.net/2008/pause6-basic"
    xmlns:c6="http://cpan6.net/2008/cpan6-basic"
  >

    <import
      namespace="http://cpan6.net/2008/cpan6-basic"
      location="https://xml.cpan6.net/schema/2008/cpan6-basic.xsd" />
```

A.2 Releases

pause6-release

Pause6 adds security and trust to CPAN6 releases. The `creator` is only used to honor people who contributed to the package, but has no additional value. As `author`, the content responsible idenities are listed. The `publisher` is responsible for the distribution.

Each release can specify a new set of authors and permissions. Both authors and permissions are only effective when the release is accepted. So, whether the publisher is permitted to upload, change authors or permissions is defined by the previous release. The acceptable changes also depend on the rules in the constitution.

Pause6 defines some additional release types:
`x-cpan6/pause6-constitution`,
`x-cpan6/pause6-index`,
`x-cpan6/pause6-archive-ref`,
`x-cpan6/pause6-identity`, `x-cpan6/pause6-license`, and
`x-cpan6/pause6-process`. The latter defines a daemon or scribe.

```
  <element name="pause6-release" type="p6:pause6-release"
    substitutionGroup="c6:release-extension"/>

  <complexType name="pause6-release">
    <complexContent>
      <extension base="c6:release-extension">
        <sequence>
          <element name="creator"   type="c6:release-id"
             minOccurs="0" maxOccurs="unbounded" />
          <element name="author"    type="c6:release-id"
             minOccurs="0" maxOccurs="unbounded" />
          <element name="publisher" type="c6:release-id" />
          <element name="permit"    type="p6:permission-set"
             minOccurs="0" maxOccurs="unbounded" />
          <element name="depends"   type="p6:dependencies" minOccurs="0" />
          <element name="license"   type="c6:release-id"
             minOccurs="0" maxOccurs="unbounded" />
          <element name="plan"      type="p6:release-plan" minOccurs="0" />
        </sequence>
      </extension>
    </complexContent>
  </complexType>
```

release-component

Security for files is implemented by protected complex checksum for each of the published files. When more than one checksum is provided, they will use different formats. Only the 'best' format which is supported by the client will need to be checked.

```
1  <complexType name="release-component">
2    <complexContent>
3      <extension base="c6:release-component">
4        <sequence>
5          <element name="checksum" type="p6:checksum"
6            minOccurs="0" maxOccurs="unbounded" />
7        </sequence>
8      </extension>
9    </complexContent>
10 </complexType>
```

meta-data

Commissioners probably require some of these fields to be present, especially a title and a description. However, This data cannot be processed automatically and is not related to Pause6 or CPAN6's behavior.

As examples, an `creator` is added here if he or she has no provable identity. The `publisher` may represent a company or such in the traditional meaning of the word, responsible for the publication of the material, but not the actual uploading the data into the archive. So be careful what to put here.

```
1  <complexType name="meta-data">
2    <sequence />
3  </complexType>
```

A.2.1 Release states

release-state

A release will move from one state to the other. When a release is in `uploading` state, it will not be available to deployers. When in `published` or `embargo` state, it will only be available to authors. The `embargo` means that the project has already suffient signatures to go into the `released` state, but will be kept back by the daemon until the embargo is lifted (see item A.2.2)

Releases which are `expired` may still be available, although a clean-up may happen during download of the release (which will result in an error). The `deprecated` state of a release means that their are known problems with the release. The client may warn users to upgrade.

Usually, the project was in `released` state when it got installed. The user will be warned when the scribe (downloader) detects that release gets flagged as `expired`, `rejected`, or `deprecated` on the source archive.

```
1  <simpleType name="release-state">
2    <restriction base="token">
3      <enumeration value="uploading"  />
4      <enumeration value="published"  />
5      <enumeration value="embargo"    />
6      <enumeration value="released"   />
7      <enumeration value="deprecated" />
8      <enumeration value="expired"    />
```

```
      <enumeration value="rejected"   />
    </restriction>
  </simpleType>
```

release-state-change

Each state change is issued by someone or the cpan6 daemon; in any case, the identity needs to be included. Then, zero or more signatures are required to get this state change accepted.

```
<complexType name="release-state-change">
  <complexContent>
    <extension base="p6:initiated">
      <sequence>
        <element name="state" type="p6:release-state" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

A.2.2 Release Plan

release-plan

To change the release plan, you need to have the `release-plan-change` rights.

```
<complexType name="release-plan">
  <sequence>
     <element ref="p6:release-plan-control"
       minOccurs="0" maxOccurs="unbounded" />
  </sequence>
</complexType>
```

release-plan-control

```
<element name="release-plan-control" type="p6:release-plan-control"
   abstract="true" />

<complexType name="release-plan-control">
  <complexContent>
    <extension base="p6:initiated">
      <sequence>
        <element name="cancelled" type="p6:initiated" minOccurs="0" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

initiated

Used as base-class to anything which needs to be logged, triggered by a human or a process. The identity usually lists the process which produced the log entry or the user who request something.

```
<complexType name="initiated">
  <sequence>
```

```
      <element name="at"     type="dateTime"           />
      <element name="by"     type="c6:release-id"      />
      <element name="reason" type="c6:language-string"
        minOccurs="0" maxOccurs="unbounded" />
    </sequence>
  </complexType>
```

embargo-time

Releases can be uploaded under embargo, which will block the transition from `published` until `released` state until the condition is reached. Meanwhile, the release will be stuck in the `embargo` state, it will not be distributed to the deployers, and only available to authors.

When the release was already `released` before the embargo was set, this probably is done to correct a mistake. In this case, it is important not to allert the users who already got the data. The release will still move into `embargo` state, but users who have it installed or downloaded must not be warned by the client software.

```
  <element name="embargo-time" type="p6:embargo-time"
    substitutionGroup="p6:release-plan-control" />

  <complexType name="embargo-time">
    <complexContent>
      <extension base="p6:release-plan-control">
        <sequence>
          <element name="after" type="dateTime" />
        </sequence>
      </extension>
    </complexContent>
  </complexType>
```

embargo-release

The release of this version is delayed until one or more other releases from other projects are available in the same archive.

```
  <element name="embargo-release" type="p6:embargo-release"
    substitutionGroup="p6:release-plan-control" />

  <complexType name="embargo-release">
    <complexContent>
      <extension base="p6:release-plan-control">
        <sequence>
          <element name="need-release" type="c6:release-id"
            minOccurs="1" maxOccurs="unbounded" />
        </sequence>
      </extension>
    </complexContent>
  </complexType>
```

is-final

New releases in the same strand are not accepted.

```
  <element name="is-final" type="p6:is-final"
    substitutionGroup="p6:release-plan-control"/>

```

```
<complexType name="is-final">
  <complexContent>
    <extension base="p6:release-plan-control" />
  </complexContent>
</complexType>
```

expiration

Move the release to the `expired` state after this moment. When `keep-last` is set, the release will not expire if it does not have a follow-up.

```
<element name="expiration" type="p6:expiration"
  substitutionGroup="p6:expiration" />

<complexType name="expiration">
  <complexContent>
    <extension base="p6:release-plan-control">
      <sequence>
        <element name="after"     type="dateTime" />
        <element name="keep-last" type="boolean" default="true" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

deprecate

The ‘deprecated’ state is a special form of ‘released’: the release can still be the last with respect to permissions, but will not be used on clients except when explicitly asked for. Clients will also report the deprecation of installed software releases to users.

```
<element name="deprecate" type="p6:deprecate"
  substitutionGroup="p6:release-plan-control" />

<complexType name="deprecate">
  <complexContent>
    <extension base="p6:release-plan-control">
      <sequence>
        <element name="after" type="dateTime" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

has-followup

When a release (independent on its current state) is followed by a newer release (defined by the release strand definitions, not by the moment of release), it may automatically be phased-out.

When the `minimal-newer` number of releases for this project in ‘released’ state are sorted after this release, then the changes will get scheduled (probably irreversably).

When the condition is met, the specified `deprecate-after` duration is waited until the release is flagged to be ‘deprecated’. Then, the `expire-after` duration is waited, before the release state changes into ‘expired’. Finally, another `reject-after` time-span is waited before the release is prepared to be removed.

At least one of the fields must be specified, otherwise the whole plan is lost (removed). The intermediate states may take longer then specified. It may also be possible that

the state changes cannot be implemented on-time, for instance when the whole archive administration is distributed on a disk. In that case, the one-shot deployer must handle these state changes and merge them dynamically into the static archive index.

```
<element name="has-followup" type="p6:has-followup"
  substitutionGroup="p6:release-plan-control" />

<complexType name="has-followup">
  <complexContent>
    <extension base="p6:release-plan-control">
      <sequence>
        <element name="minimal-newer"   type="int"      default="1" />
        <element name="deprecate-after" type="duration" default="0D" />
        <element name="expire-after"    type="duration" default="0D" />
        <element name="reject-after"    type="duration" default="0D" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

A.2.3 Release relations

A set of releases occupy one name-space location. These relations specify the order of succession.

The whole set of related releases contains one or more **strand**s, each containing one or more versions of the project. A trail of releases is a logical follow-up, where each next release in the list has a super-set of functionality or content from the previous. A new software release MUST not break the described (the official) interface. When the release contains documentation, the next release will include all the previous documents adding new text, removing mistakes, and typos fixed.

Parallel strands are used for diverting developments, like alpha-releases, which may break compatibility or have limited applicability. Usually, people should avoid these releases, by default stay away from these less certain paths. These diverted strands can be merged with main-line later.

Release relations may need to be secured with signatures, and can therefore not be changed after a release is been distributed. For this reason, relations can only refer back. The `version` of the release itself is insufficient to represent the needs for parallel development, and thereby only used for uniqueness.

release-parent
: Back-reference to the release which was used to base this version on.

```
<complexType name="release-parent">
  <simpleContent>
    <extension base="c6:release-id">
      <element name="relation" type="c6:release-parent-relation"
        default="succeeds" maxOccurs="unbounded" />
    </extension>
  </simpleContent>
</complexType>
```

release-parent-relation
: When this release `succeeds` the other, it is prefered to become used or installed by the end-user. Preferred over the use of other releases within this strand, but not over

releases within other strands. You may succeed on multiple strands at the same time, which means that those strands will be merged into one new.

You will need to fulfil the superset of the requirements for all merged releases, which may imply a serious amount of signatures. For simplify this process, it may be useful not to attempt to merge more than two strands with one new release.

When `diverts` is used, you start a new strand. You can only have one such relationship for your release.

```
<simpleType name="release-parent-relation">
  <restriction base="token">
    <enumeration value="succeeds" />
    <enumeration value="diverts" />
  </restriction>
</simpleType>
```

A.2.4 Release dependencies

Various dependency relationships may exist between releases of different projects, maybe originating from different archives. Don't forget that these relations are quite simple on the level of CPAN6, and applications may want do implement a more complex schema.

The archive maintainer (Pause6) keeps track on relations between releases which are in use. When some release is upgraded, this may break other applications which can not work with that release. The user must get an options to bail-out before these horrors happen.

dependencies

The order of dependencies is important in the case where projects are listed multiple times. In this case, the dependency resolver must know how the releases are structured within the project's name-space. The release specified is used as starting-point to flag all releases which follow this one in the same release **trail**. Say, mentioning version 2.08 as minimal requirement of a software product will flag all other 2.xx releases which are in the same trail, but not the 3.xx versions, because they have an incompatible usage interface. When version 2.16 is specified as `breaking` this will leave only 2.08 upto 2.15 as acceptable.

```
<complexType name="dependencies">
  <sequence>
    <element name="link" type="p6:release-link"
      minOccurs="0" maxOccurs="unbounded" />
  </sequence>
</complexType>
```

release-link

A relation between two projects is usually expressed with releases, but it can be defined as a project as well, like usually meaning the last release of the main track. Use that only when there is no knowledge about interoperatebility of versions.

```
<complexType name="release-link">
  <complexContent>
    <extension base="c6:release-id">
      <attribute name="for"  type="p6:release-link-purpose" use="required"
      <attribute name="need" type="p6:release-link-need" default="required"
    </extension>
  </complexContent>
</complexType>
```

release-link-purpose

Specifies why this release is referenced to. The `build` means that the release is needed during the download, installation and test process, but after that not anymore. The system may decide to remove that referenced release from the system after the installation was successful, for instance by only downloading it into a temporary location.

When data from a release is run as a program or displayed as data, it may require assistance of some releases: it will `use` those releases.

The `support` relation means that the release at hand contains information which is based on some other release, usually with the same name in an other archive (an other View on the project).

The contents of the release at hand contributes to the cloud of knowledge around that other release: in contains tests, patches, documentation, and such, supporting the use of that other release. The author is probably someone else. You may also see this as additional services to end-users.

```
1  <simpleType name="release-link-purpose">
2    <restriction base="token">
3      <enumeration value="build"   />
4      <enumeration value="use"     />
5      <enumeration value="support" />
6    </restriction>
7  </simpleType>
```

release-link-need

When the referenced release is specified to be `required`, it must be present before this release can be handled. For optional links, the user may get a choice whether to get them beforehand or not. The client-side tool may also decide to take all optional modules (and maybe ignore them when there are errors while building them) or default to ignore all optional relations.

```
1  <simpleType name="release-link-need">
2    <restriction base="token">
3      <enumeration value="required" />
4      <enumeration value="optional" />
5      <enumeration value="conflicts" />
6    </restriction>
7  </simpleType>
```

A.3 Archives

archive

Defines the archive.

```
1  <element name="archive" type="p6:archive"
2    substitutionGroup="p6:role-player" />
3
4  <complexType name="archive">
5    <complexContent>
6      <extension base="p6:role-player">
7        <sequence>
8         <element name="constitution" type="p6:constitution" />
9        </sequence>
```

```
      </extension>
    </complexContent>
  </complexType>
```

constitution

```
<complexType name="constitution">
  <sequence>
    <element name="board"                type="p6:board" />
    <element name="permissions"          type="p6:permission-set"
      maxOccurs="unbounded"       />
    <element name="name-space-layout"    type="p6:name-space-layout" />
    <element name="name-restrictions"    type="p6:label-restrictions"
      minOccurs="0" />
    <element name="version-restrictions" type="p6:label-restrictions"
      minOccurs="0" />
  </sequence>
</complexType>
```

name-space-layout

The name-space layout limits the project and version labels, and the mime-types accepted by the archive.

```
<complexType name="name-space-layout">
  <sequence>

    <element name="accept-releases" minOccurs="0" maxOccurs="unbounded">
      <sequence>
         <element name="names"   type="p6:label-restrictions" />
         <element name="version" type="p6:label-restrictions" />
      </sequence>
      <attributeGroup ref="ff:mime-type-set" default="*/*" />
    </element>

    <element name="used-names" minOccurs="0">
      <sequence>
        <element name="constitution" type="c6:label"
          default="pause6-constitution" />
        <element name="index" type="c6:label" default="pause6-index" />
      </sequence>
    </element>

  </sequence>
</complexType>
```

board

```
<complexType name="board">
  <sequence>
    <element name="member"  type="p6:identity"
      minOccurs="1" maxOccurs="unbounded" />
    <element name="minimum-members" type="int"
```

```
6         minExclusive="0" default="1"        />
7     </sequence>
8   </complexType>
```

label-restrictions

As `xml-pattern`, you specify an XML compliant pattern: when more than one pattern is provided, any of them must match. Added to this list are the exactly specified labels (`equals`). You may also use a (non-capturing) perl5 regular expression. By default, all labels are selected.

The optional regular expression will reduce the set of selected labels, for instance in the used character-set. Because of the power of regular expressions, you may be able to avoid the use of `xml-patterns` and `equals`.

```
1   <complexType name="label-restrictions">
2     <sequence>
3       <choice minOccurs="0" maxOccurs="unbounded">
4          <element name="equals"      type="string"  />
5          <element name="xml-pattern" type="pattern" />
6       </choice>
7       <element name="regex-perl5"    type="string"
8         minOccurs="0" />
9       <element name="min-length"     type="unsignedInt"
10        default="0" />
11      <element name="max-length"     type="allNNI"
12        default="unbounded" />
13    </sequence>
14  </complexType>
```

mime-types-set

In the future, a more explicit definition of the mime-types may be given; therefore: follow the rules of the RFCs! Both attributes can be a comma-seperated list. Both are case-insensitive and ignore leading `x-`. The `mime-types` components may contain `*` to indicate a whole class, like `video/*`.

```
1   <attributeGroup name="mime-types-set">
2     <attribute name="mime-types" type="string" />
3     <attribute name="mime-type"  type="string" />
4   </attributeGroup>
```

protocol

Specifies the Pause6 level.

```
1   <element name="protocol" type="anyURI" />
```

A.4 Processes

role-player

Roles are played by components which can live in separate processes. In the implementation, they may get mapped on different threads or system processes. Although, they may also be run by one user program together.

The `identity` is a private identity description, which also contains the information about the location of the public identity.

The `algorithms` specify which choices were made by the role-player, i.e. which requirements are set to understand how the role-players meta-data.

```
<complexType name="role-player">
  <sequence>
    <element name="address"    type="anyURI"        />
    <element name="identity"   type="c6:release-id" />
    <element name="algorithms" type="c6:algorithms" />
  </sequence>
</complexType>
```

A.5 Store

The basic store components are implemented in name-space `Pause6::Store`.

The **store** is a process and the administration which manages one copy of the data of any number of archives. Each archive has its own **repository**.

A store is maintained by the daemon. The store administration is always on the local system, for performance reasons. Either as files on disk or in a database. Some stores copy the data to external systems, for instance the FTP implementation of a store.

Stores will be defined on system and/or user level. Before an archive can use a store, it will need to register itself in the store, because the administrator of the store may impose all kinds of restrictions.

store

A `store` is a process which administers a storage space, and can receive request for load and save via a socket.

```
<element name="store" type="p6:store"
  substitutionGroup="p6:role-player" />

<complexType name="store">
  <complexContent>
    <extension base="p6:role-player">
      <sequence>
        <element name="remote" type="p6:scribe-interface"
           minOccurs="0" maxOccurs="unbounded" />
        <element name="keeper" type="p6:store-keeper" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

store-keeper

The part of the store which is responsible for maintaining the data. The meta-data and files can be kept on one spot, or on different locations.

Access rights are determined by the environment. In case this keeper is used by a Store, that will determine the authorization to write. In other cases, this may be part of a user process, which uses platform native access restrictions.

```
<element name="store-keeper" type="p6:store-keeper" />
<complexType name="store-keeper">
  <choice>
    <element name="all" type="p6:scribe-interface" />
    <sequence>
      <element name="meta" type="p6:scribe-interface" />
      <element name="data" type="p6:scribe-interface" />
    </sequence>
```

```
    </choice>
  </complexType>
```

repository

```
<element name="repository">
  <complexType>
    <sequence>
      <element name="store"   type="c6:release-id" />
      <element name="archive" type="c6:release-id" />
      <element name="created" type="p6:initiated"  />
    </sequence>
  </complexType>
</element>
```

file-system

Probably most stores will be file-system based, sometimes in a way that you can see the names of archives, projects and version, sometimes with abstract names. In any case, you can bump into platform specific limits.

When `max-name-length` is not defined, The `file-system` details may refer to details of a remote store, like ftp-server. Access to a case-insensitive file-system via UNIX may be slow.

```
<element name="file-system" type="p6:file-system" />
<complexType name="file-system">
  <sequence>
    <element name="name-restrictions" type="p6:name-restrictions" />
  </sequence>
  <attribute name="type"     type="string" />
  <attribute name="revision" type="string" />
  <attribute name="platform" type="string" />
</complexType>
```

A.6 Scribe

The basic store components are implemented in name-space `Pause6::Scribe`.

scribe

A `scribe` is a process which handles communication between processes and between processes and their physical world.

```
<element name="scribe" type="p6:scribe"
  substitutionGroup="p6:role-player" />

<complexType name="scribe">
  <complexContent>
    <extension base="p6:role-player">
      <sequence>
        <element name="interface" type="p6:scribe-interface"
           minOccurs="1" maxOccurs="unbounded" />
        <any processContent="#lax" />
      </sequence>
    </extension>
```

```
    </complexContent>
  </complexType>
```

scribe-capabilities

The scribe will write to disk or send the data to a remote server. This server has real physical limits, which must be known to safely administer the archive. Those limits can sometimes be autodetected using `Pause6::Scribe::Autodetect` function `medium-limits`.

The length values are in bytes, which may differ from the characters when unicode encoding is available.

This data-structure is also used to list additional user limits on files for the medium. For instance, because the user knows that other applications may break when long filenames are used. Both user as system limits are to be obeyed.

```
<complexType name="scribe-capabilities">
  <sequence>
    <element name="name"            type="c6:label" />
    <element name="character-set"   type="token"    default="utf-8" />
    <element name="case-sensitive"  type="boolean"  default="true"  />
    <element name="accept-chars"    type="string"   minOccurs="0"   />
    <element name="encode-chars"    type="string"   minOccurs="0"   />
    <element name="max-name-length" type="int"      minOccurs="0"   />
    <element name="max-ext-length"  type="int"      minOccurs="0"   />
    <element name="max-path-length" type="int"      minOccurs="0"   />
    <element name="read-only"       type="boolean"  default="false" />
    <element name="max-file-size"   type="integer"  minOccurs="0"   />
    <element name="symbolic-links"  type="boolean"  default="true"  />
    <choice>
      <element name="legal-name"    type="pattern"
         minOccurs="0" maxOccurs="unbounded" />
      <element name="illegal-name"  type="pattern"
         minOccurs="0" maxOccurs="unbounded" />
    </choice>
  </sequence>
</complexType>
```

A.7 Authentication

The public and private identities are created together. The public identity shall be made available to everyone, where the private part is only accessible to the authors and publishers of the identity release.

identity-public

The public identity relates to a private identity. It contains information which everyone is permitted to see, for instance a personal description for a blog and public-keys for an asymetric encrypted connection.

```
<element name="identity-public" type="p6:identity-public" />

<complexType name="identity-public">
  <sequence>
    <choice>
      <element name="user"    type="p6:person"  />
      <element name="system"  type="p6:process" />
```

```
        <element name="process" type="p6:process" />
      </choice>
      <element name="authorization" type="p6:authorization" />
      <any processContents="lax" minOccurs="0" maxOccurs="unbounded" />
    </sequence>
  </complexType>
```

identity-private

Collects the identity information which is not to be accessible by other people or processes.

```
<element name="identity-private" type="p6:identity-private" />

<complexType name="identity-private">
  <sequence>
    <element name="identity-public" type="c6:release-id"   />
    <element name="authorization"   type="p6:authorization" />
  </sequence>
</complexType>
```

destination

The data which relates with this destination structure can be used when both the *host* and the *service* restrictions are fulfilled.

The `host` fields can each contain a list of host or domain names, and CIDR notations of IP-addresses. The related data can be used when no `host` fields are present or when the connection matches at least one item in the specified list.

As `services`, you can (case-insensively) use any IANA defined port name or port number[1]. For instance, a valid value would be “ftp 21 HTTP”. Port names and numbers which are not understood are silently ignored.

```
<complexType name="destination">
  <sequence>
    <element name="host"    type="NMTOKENS"
        minOccurs="0" maxOccurs="unbounded" />
    <element name="service" type="NMTOKENS" />
  </sequence>
</complexType>
```

A.8 Rights

admin-rights

The list of permissions wil certainly grow. A list of this rights is used for various purposes: to express capabilities, limits, and vote requirements.

```
<simpleType name="admin-rights">
  <restriction base="token">
    <enumeration value="constitution-change" />
    <enumeration value="archive-list-change" />
    <enumeration value="project-delete"      />
    <enumeration value="project-start"       />
    <enumeration value="release-publish"     />
    <enumeration value="release-embargo"     />
```

[1] On UNIX, have a look at `/etc/services`.

```
      <enumeration value="release-accept"      /> <!--released-->
      <enumeration value="release-reject"      />
      <enumeration value="release-plan-change" />
    </restriction>
  </simpleType>
```

permission-class

Used to indicate the community to which a set of permission applies. The `autors-defaults` and `authors-maximum` are set by the board to limit the freedom of the project authors. The `authors` and `non-authors` are used by project authors, as meta-data item inside a release.

```
<simpleType name="permission-class">
  <restriction base="token">
    <enumeration value="board-members"   />
    <enumeration value="authors"         />
    <enumeration value="author-defaults" />
    <enumeration value="author-maximum"  />
    <enumeration value="non-authors"     />
  </restriction>
</simpleType>
```

permission-set

Groups sets of rights for one target community. All permission flags should only be used once within this container. Unspecified permission flags will have a default.

```
<complexType name="permission-set">
  <sequence>
    <element name="permissions" type="p6:signatures-required"
      minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <attribute name="class"       type="p6:permission-class"
    use="required" />
</complexType>
```

signature

One signature package, which can be checked. When the signer (attribute `by`) information (is a versioned release) is not available, or its version got deprecated, or the infrastructure to check this identity is not available, then the signature should be ignored. In other cases, an failing check should block use of the signed release.

```
<complexType name="signature">
  <complexContent>
    <extension base="p6:initiated">
      <sequence>
        <element name="type"      type="p6:signature-type" />
        <element name="encoding"  type="c6:encoding-type"  />
        <element name="autograph" type="string"            />
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

signature-type

```
<simpleType name="signature-type">
  <restriction base="token">
    <enumeration value="PGP" />
    <enumeration value="SSL2" />
    <enumeration value="SSL3" />
  </restriction>
</simpleType>
```

signatures-required

The minimum number of signatures required for a set of permissions.

```
<complexType name="signatures-required">
  <sequence>
    <element name="permit">
      <simpleType>
        <list itemType="p6:admin-rights" />
      </simpleType>
    </element>
  </sequence>
  <attribute name="minimum" type="p6:signature-count"
    use="required" />
</complexType>
```

signature-count

The named situations are especially useful for the board, although it can be used everywhere. Symbolic names have the advantage when the number of authors or board changes: the count probably does not need to change in that case.

```
<simpleType name="signature-count">
  <union>
    <simpleType>
      <restriction base="int">
        <minInclusive value="0" />
      </restriction>
    </simpleType>
    <simpleType>
      <restriction base="token">
        <enumeration value="all"         />
        <enumeration value="all-but-one" />
        <enumeration value="majority"    />
        <enumeration value="never"       />
        <enumeration value="none"        /> <!-- 0 -->
      </restriction>
    </simpleType>
  </union>
</simpleType>
```

A.9 Trust

trust

Trust is a calculated value, based on the way data is distributed and signed. A higher value means a higher trust. The more processes and steps are involved in the process, the lower the trust will be.

The trust value is dynamic: when a release is signed with a very trusted public key (for instance exchanged during some meeting in person), and that key expires, all releases which where downloaded using that key will be lowered in trust immediately. This may cause warning messages to system administrators which use these releases.

```
<simpleType name="trust">
  <restriction base="float">
    <minInclusive value="0.0"   />
    <maxInclusive value="100.0" />
  </restriction>
</simpleType>
```

B Schema: Pause6 file formats

The `pause6-basic` schema defines content which is useful for Pause6 organization. This schema defines wrappers around the data elements, to be able to store them in files on disk or a database.

B.1 Schema wrapper

```
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"

  targetNamespace="http://cpan6.net/2008/pause6-files"
  schemaLocation="https://xml.cpan6.net/schema/2008/pause6-files.xsd"
  version="1.0"

  xmlns:ff="http://cpan6.net/2008/pause6-files"
  xmlns:p6="http://cpan6.net/2008/pause6-basic"
  xmlns:c6="http://cpan6.net/2008/cpan6-basic"
>

  <import
    namespace="http://cpan6.net/2008/pause6-basic"
    location="https://xml.cpan6.net/schema/2008/pause6-basic.xsd" />

  <import
    namespace="http://cpan6.net/2008/cpan6-basic"
    location="https://xml.cpan6.net/schema/2008/cpan6-basic.xsd" />
```

B.2 File `release-log.xml`

release-log-entry

```
<simpleType name="release-log-entry">
  <choice>
    <element name="request"   type="state-change-request"   />
    <element name="confirmed" type="state-change-confirmed" />
    <element name="trace"     type="daemon-trace"           />
    <element name="signature" type="state-approved"         />
  </choice>
```

```
8   </simpleType>
```

state-change-request

```
1   <complexType name="state-change-request">
2     <complexContent>
3       <extension base="p6:initiator">
4         <sequence>
5           <element name="state" type="p6:release-state" />
6         </sequence>
7       </extension>
8     </complexContent>
9   </complexType>
```

state-change-confirmed

```
1   <complexType name="state-change-confirmed">
2     <complexContent>
3       <extension base="p6:initiator">
4         <sequence>
5           <element name="state" type="p6:release-state"   />
6           <element name="votes" type="p6:signatures-count" />
7         </sequence>
8       </extension>
9     </complexContent>
10  </complexType>
```

B.3 File `archive-log.xml`

archive-log-entry

```
1   <complexType name="archive-log-entry">
2     <complexContent>
3       <extension base="initiated">
4         <sequence>
5           <element name="release" type="ff:visible-release-change" />
6         </sequence>
7       </extension>
8     </complexContent>
9   </complexType>
```

visible-release-change

```
1   <complexType name="visible-release-change">
2     <sequence>
3       <element name="state"      type="p6:release-state" />
4       <element name="total-size" type="c6:size"          />
5     </sequence>
6     <attributeGroup              ref="p6:release-label"  />
7   </complexType>
```

B.4 File `archive-list.xml`

archive-list-entry

```
<complexType name="archive-list-entry">
  <complexContent>
    <extension base="p6:initiated">
      <sequence>
        <element name="archive" type="ff:archive-definition" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

archive-definition

```
<complexType name="archive-definition">
  <sequence>
    <element name="id"    type="c6:release-id" />
    <element name="trust" type="p6:trust"      />
    <element name="alias" type="c6:label"
      minOccurs="0" maxOccurs="unbounded" />
  </sequence>
</complexType>
```

B.5 Files `status-current.xml` and `status-next.xml`

release-state-which

```
<simpleType name="release-state-which">
  <restriction base="token">
    <enumeration value="CURRENT" />
    <enumeration value="NEXT"    />
  </restriction>
</simpleType>
```

status-to-sign

```
<complexType name="status-to-sign">
  <sequence>
    <element name="status-seqnr" type="int"              />
    <element name="release"      type="p6:release"       />
    <element name="state"        type="p6:release-state" />
  </sequence>
</complexType>

```

release-status-description

```
<complexType name="release-status-description">
  <sequence>
    <element name="to-sign"   type="ff:status-to-sign" />
    <element name="signature" type="p6:signature"
        minOccurs="0" maxOccurs="unbounded" />
  </sequence>
</complexType>
```

state-approved

```
<complexType name="state-approved">
  <complexContent>
    <extension base="p6:initiated">
      <sequence>
        <element name="signature" type="p6:signature" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

B.6 File `constitution.xml`

B.7 File `repository.xml`

repository-index

The index is needed for situations where there is no fast indexing support for the meta-data, i.e, when the data is not stored in a database. The index will need to be regenerated on regular basis.

```
<element name="repository-index">
  <complexType>
    <element name="generated" type="dateTime" />
    <sequence>
      <element name="project-summary" type="ff:project-summary"
        minOccurs="0" maxOccurs="0" />
    </sequence>
  </complexType>
</element>
```

name-summary

Summary about one name.

```
<element name="name-summary">
  <complexType>
    <sequence>
      <element name="sum-size"    type="nonNegativeInteger"  />
      <element name="nr-releases" type="nonNegativeInteger"  />
      <element name="last-update" type="dateTime"            />
    </sequence>
    <attribute name="source" type="c6:address" use="required" />
    <attribute name="name"   type="c6:label"   use="required" />
  </complexType>
</element>
```

repository-release

Each released file may appear more than once, but at least once. It may be so that the file's original name (in `path`) can be recognized from the location URI, but it may not be.

```
<element name="repository-release">
  <complexType>
    <sequence>
      <element name="release" type="c6:release-label" />
      <element name="item"    type="c6:encapsulated-item"
         minOccurs="0" maxOccurs="unbounded" />
    </sequence>
    <attribute name="store"   type="c6:address" />
    <attribute name="archive" type="c6:address" />
  </complexType>
</element>
```

B.8 Files *filesystem*`.xml`

file-system-descriptions

```
<element name="file-system-descriptions">
  <complexType>
    <sequence minOccurs="0" maxOccurs="unbounded">
      <element ref="p6:file-system" />
    </sequence>
  </complexType>
</element>
```

B.9 User configuration, `pause6-config.xml`

Objects of this type collect information to simplify the life of humans, using Pause6/CPAN6: the source/name/version triplet for items it too long to type.

user-config

Keeps pause6 specific preferences of a human.

```
<element name="user-config" type="p6f:user-config" />

<complexType name="user-config">
  <sequence>
    <element name="aliases" type="p6f:user-aliases"
      minOccurs="0" maxOccurs="unbounded" />
  </sequence>
</complexType>
```

user-aliases

Each block of `user-aliases` relates to a type of releases, which may be `publication` for some kind of content, or for instance `archive` or `identity`.

Alias names can be reused in different user-aliases lists, as long as their definitions are in different classes. Each class should only appear once, for clarity.

```
<complexType name="user-aliases">
  <sequence>
    <element name="alias" type="p6f:user-alias"
      minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <attribute name="class" type="token" />
</complexType>
```

user-alias

An alias is solely meant to simplify the interaction with humans. It is not simple to remember release ids for seperate publications and remember in which archives they reside. Aliases shall **never** appear in the pause6 or cpan6 protocol.

For instance, the alias “me”, could refer to my private identity description, as stored in my own private archive. The `users-aliases` class in which this alias is listed should be “identity”.

When no archive is specified (or when non of the specified archives can be contacted), the source address of the release must be used to collect that information.

```
<complexType name="user-alias">
  <sequence>
    <element name="release" type="c6:release-id" />
    <element name="archive" type="c6:release-id" />
  </sequence>
  <attribute name="alias" type="NMTOKENS" />
</complexType>
```

C Schema: Pause6 messages

The schema defined in this section defines messages types used for Pause6. They extend the basic types of CPAN6.

C.1 Schema wrapper

```
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"

  targetNamespace="http://cpan6.net/2008/pause6-messages"
  schemaLocation="https://xml.cpan6.net/schema/2008/pause6-messages.xsd"
  version="1.0"

  xmlns:p6="http://cpan6.net/2008/pause6-basic"
  xmlns:c6="http://cpan6.net/2008/cpan6-basic"
  xmlns:c6m="http://cpan6.net/2008/cpan6-messages"
  xmlns:c6s="http://cpan6.net/2008/cpan6-stable"
>

  <import
    namespace="http://cpan6.net/2008/pause6-basic"
    location="https://xml.cpan6.net/schema/2008/pause6-basic.xsd" />

  <import
```

```
      namespace="http://cpan6.net/2008/cpan6-basic"
      location="https://xml.cpan6.net/schema/2008/cpan6-basic.xsd" />

  <import
      namespace="http://cpan6.net/2008/cpan6-messages"
      location="https://xml.cpan6.net/schema/2008/cpan6-messages.xsd" />

  <import
      namespace="http://cpan6.net/2008/cpan6-stable"
      location="https://xml.cpan6.net/schema/2008/cpan6-stable.xsd" />
```

This messages asks a list all releases, only supporting simple filtering. It is used by scribes to make an inventory of all releases to be copied. The data is directly taken from the archive-log.

This message is not meant for normal users. They should use the search interface to get a much smaller set of answers, probably optimized with database searches.

list-releases-request

The scribe can request only releases with certain states. By default, only released and deprecated releases are returned. With the `only-last` flag set, only the last version of each project is returned.

```
<element name="list-releases-request"
  substitutionGroup="c6m:request-message">
  <complexType>
    <complexContent>
      <extension base="c6m:message-lrr-type">
        <sequence>
          <element name="states"     type="p6:release-state"
            minOccurs="0" maxOccurs="unbounded" />
          <element name="types"      type="p6:project-type"
            minOccurs="0" maxOccurs="unbounded" />
        </sequence>
      </extension>
    </complexContent>
  </complexType>
</element>
```

list-releases-answer

```
<element name="list-releases-answer"
  substitutionGroup="c6m:answer-message">
  <complexType>
    <complexContent>
      <extension base="c6m:message-lra-type">
        <sequence>
          <element name="release" type="p6:visible-release-change"
            minOccurs="0" maxOccurs="unbounded" />
        </sequence>
      </extension>
    </complexContent>
  </complexType>
</element>
```

C.2 Accept signature

accept-signature-request

```
<element name="accept-signature-request"
  substitutionGroup="c6m:request-message">
  <complexType>
    <complexContent>
      <extension base="c6m:request-message-type">
        <sequence>
          <element name="release"      type="p6:release-label" />
          <element name="signature"    type="p6:signature"     />
          <element name="status-seqnr" type="positiveInteger"  />
        </sequence>
      </extension>
    </complexContent>
  </complexType>
</element>
```

accept-signature-answer

```
<element name="accept-signature-answer"
  substitutionGroup="c6m:answer-message">
  <complexType>
    <complexContent>
      <extension base="c6m:answer-message-type">
        <sequence>
          <element name="votes-received" type="p6:vote-count" />
          <element name="votes-required" type="p6:vote-count" />
        </sequence>
      </extension>
    </complexContent>
  </complexType>
</element>
```

C.3 Get release status

get-release-status-request

```
<element name="get-release-status-request"
  substitutionGroup="c6m:request-message">
  <complexType>
    <complexContent>
      <extension base="c6m:request-message-type">
        <sequence>
          <element name="release" type="p6:release-label" />
          <element name="which"   type="ff:release-state-which" />
        </sequence>
      </extension>
    </complexContent>
  </complexType>
```

```
  </element>
```

get-release-status-answer

```
<element name="get-release-status-answer"
  substitutionGroup="c6m:answer-message">
  <complexType>
    <complexContent>
      <extension base="c6m:answer-message-type">
        <sequence>
          <element name="status"
             type="p6:release-status-description" />
          <element name="votes-received" type="p6:vote-count" />
          <element name="votes-required" type="p6:vote-count" />
        </sequence>
      </extension>
    </complexContent>
  </complexType>
</element>
```

References

[1] Overmeer and Vilain, *CPAN6 and Pause6 Design*.

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

Index