Repository Working Group	J. Kunze
	M. Haye
	E. Hetzner
	M. Reyes
	California Digital Library
	C. Snavely
	University of Michigan Library IT
	Core Services
	December 12, 2008

Pairtrees for Collection Storage (V0.1)

Abstract

This document specifies Pairtree, a filesystem hierarchy for holding objects that are located within that hierarchy by mapping identifier strings to object directory (or folder) paths two characters at a time. If an object directory (folder) holds all the files, and nothing but the files, that comprise the object, a "pairtree" can be imported by a system that knows nothing about the nature or structure of the objects but can still deliver any object's files by requested identifier. The mapping is reversible, so the importing system can also walk the pairtree and reliably enumerate all the contained object identifiers. To the extent that object dependencies are stored inside the pairtree (e.g., fast indexes stored outside contain only derivative data), simple or complex collections built on top of pairtrees can recover from index failures and reconstruct a collection view simply by walking the trees. Pairtrees have the advantage that many object operations, including backup and restore, can be performed with native operating system tools.

---

1. The basic pairtree algorithm

The pairtree algorithm maps an arbitrary UTF-8 **[RFC3629]** encoded identifier string into a filesystem directory path based on successive pairs of characters, and also defines the reverse mapping (from pathname to identifier).

In this document the word "directory" is used interchangeably with the word "folder" and all examples conform to Unix-based filesystem conventions which should tranlate easily to Windows conventions after substituting the path separator ('\' instead of '/'). Pairtree places no limitations on file and path lengths, so implementors thinking about maximal interoperation may wish to consider the issues listed in the Interoperability section of this document.

The mapping from identifier string to path has two parts. First, the string is cleaned by converting characters that would be illegal or especially problemmatic in Unix or Windows filesystems. The cleaned string is then split into pairs of characters, each of which becomes a directory name in a filesystem path: successive pairs map to successive path components until there are no characters left, with the last component being either a 1- or 2-character directory name. The resulting path is known as a *pairpath*, or *ppath*.

```
abcd       -> ab/cd/
abcdefg    -> ab/cd/ef/g/
12-986xy4 -> 12/-9/86/xy/4/
```

Armed with specific knowledge of a given namespace's identifier distribution, one might achieve more balanced or efficient trees by mapping to paths from character groupings other than successive pairs. Pairtree assumes that this sort of optimization, however, being tailored to individual and transient namespace conditions, is often less important than having a single generalized and shareable mapping. It uses pairs of characters to achieve hierarchies that exhibit a reasonable balance of path length and fanout (number of probable entries in any component directory).

2. Pairpath termination

The inverse mapping, from a ppath (pairpath) to its identifier string, requires recognizing the different ways that a ppath can end. Usually a ppath ends when an object is reached. Any directory name of three characters or more or any file, excluding file and directory names beginning with "pairtree", is considered to start an object; any such name is called *non-extending* because a ppath cannot extend through it. Non-extending names are critical not only for recognizing the start of an object but also to permit a ppath for one object to extend "beyond" another object that has a shorter ppath. In this way non-extending names permit a pairtree to accommodate variable-length identifiers, including one identifier that is a substring of another. The pairtree below contains two objects with identifiers "abcd" and "abcde".

```
ab/
|
\--- cd/
      |
      |--- foo/
      |     |   README.txt
      |     |   thumbnail.gif
      |     |
      |     |--- master_images/
      |     |     |   ...
      |     |     ...
      |     |
      |     \--- gh/
      |
      \--- e/
            |
            \--- bar/
                  |   metadata
                  |   54321.wav
                  |   index.html
```

A little jargon helps explain the other three ways that a ppath can end. A *shorty* is a 2-character directory name and a *morty* is a 1-character directory name. A ppath is therefore a sequence of zero or more "shorties" ending in a shorty or a morty. If a morty is encountered it always terminates the ppath in the sense that no ppath can extend through a morty (to the right of it).

A ppath also terminates at an empty shorty or morty, either of which means an empty ppath. An empty ppath can be a side-effect of optimization (lazy path removal) and is not to be counted as an object in the pairtree. Finally, a ppath also terminates at any file or directory name beginning with "pairtree". Such ppaths are reserved for future use and do not (as yet) alter the count of objects considered to be present in the pairtree.

The following paths each contains a pairpath that ends at the letter `z'. Only the last two ppaths have objects associated with them.

```
   /mn/op/qz/<empty_dir>        -> mnopqz
   /mn/op/qz/pairtree_bar/tu/   -> mnopqz
   /po/nm/z/qs/tu/              -> ponmz
   /mn/op/qz/bar.txt            -> mnopqz
```

3. Object encapsulation

Pairtree does not dictate how an object must be encapsulated in the final directory (shorty or morty) that ends a ppath, but it does define proper and improper encapsulation. To avoid ambiguity there are rules. First, so that a ppath can be associated with at most one object, any non-empty set of non-extending file and directory names contained in that final directory, as a set, comprises one object. Second, no "hidden riders" are allowed, which means that every datum in the filesystem tree representing a pairtree must be accounted for and have a defined role in the pairtree.

An object is *properly encapsulated* if it is entirely contained in a directory that has three or more characters in its name and that is the immediate child of the shorty or morty ending the object's ppath. The two objects "abcd" and "abcde" above are properly encapsulated. The "foo/" directory above terminates the ppath for "abcd" and, by encapsulation, prevents the "gh/" from extending it. Meanwhile, the "e/" directory at the same level as "foo/" permits a ppath for "abcde" to extend beyond that for "abcd".

Pairtree is silent on other aspects of choosing a properly encapsulating directory name. The name need not depend on the ppath at all, and could be arbitrary or even uniform; for example, every object could be encapsulated in a directory called "thingy" at the end of a ppath. It is often convenient, however, to choose a name that is either the identifier itself or a cleaned (see next section), abbreviated form of it. Local practices vary, but full object directory pathnames could plausibly be chosen along these lines:

```
abcd -> ab/cd/abcd/
abcde -> ab/cd/e/abcde/
45xqv_793842495 -> 45/xq/v_/79/38/42/49/5/793842495/
```

An object that is not properly encapsulated might possibly be well-defined and healthy, but it is generally a sign of dysfunction. The practice of improper encapsulation is discouraged to the extent that a *standard encapsulation patch* is defined as an automated intervention in which all non-extending names, except those beginning with "pairtree", are pushed into a newly created subdirectory called "obj" that descends from the end of the ppath. Here is an example of two improperly encapsulated objects, "bent" and "bento", the first containing two files, the second containing a single directory (that is not properly encapsulated because its name has fewer than three characters).

```
be/
|
\--- nt/              [ two files, no encapsulation ]
     |   README.txt
     |   report.pdf
     |
     \--- o/
          |
          \--- r/
               |   ...
```

Next is the same object after the recommended standard encapsulation patch is applied by an importing system, which normalizes each object by moving its two files under a new, properly encapsulated object directory called "obj".

```
be/
|
\--- nt/
     |
     |--- obj/        [ standard encapsulation patch with "obj" ]
     |    |   README.txt
     |    |   report.pdf
     |
     \--- o
          |
          \--- obj/
               |
               \--- r/
                    |   ...
```

4. Identifier string cleaning

Prior to being separated into character pairs, identifier strings are cleaned in two distinct steps. One step would be simpler, but pairtree is designed so that commonly used characters in reasonably opaque identifiers (e.g., not containing natural language words, phrases, or hints) result in reasonably short and familiar-looking paths. For completeness, the pairtree algorithm specifies what to do with all possible XXX UTF-8 characters, and relies for this on a kind of URL hex-encoding. To avoid conflict with URLs, pairtree hex-encoding is introduced with the '^' character instead of '%'.

First, the identifier string is cleaned of characters that are expected to occur rarely in object identifiers but that would cause certain known problems for file systems. In this step, every UTF-8 octet outside the range of visible ASCII (94 characters with hexadecimal codes 21-7e) **[ASCII]**, as well as the following visible ASCII characters,

```
   "   hex 22          <   hex 3c          \   hex 5c
   *   hex 2a          =   hex 3d          ^   hex 5e
   +   hex 2b          >   hex 3e          |   hex 7c
   ,   hex 2c          ?   hex 3f
```

must be converted to their corresponding 3-character hexadecimal encoding, ^hh, where ^ is a circumflex and hh is two hex digits. For example, ' ' (space) is converted to ^20 and '*' to ^2a.

In the second step, the following single-character to single-character conversions must be done.

```
   / -> =
   : -> +
   . -> ,
```

These are characters that occur quite commonly in opaque identifiers but present special problems for filesystems. This step avoids requiring them to be hex encoded (hence expanded to three characters), which keeps the typical ppath reasonably short. Here are examples of identifier strings after cleaning and after ppath mapping.

```
id:  ark:/13030/xt12t3
 ->  ark+=13030=xt12t3
 ->  ar/k+/=1/30/30/=x/t1/2t/3/
id:  http://n2t.info/urn:nbn:se:kb:repos-1
 ->  http+==n2t,info=urn+nbn+se+kb+repos-1
 ->  ht/tp/+=/=n/2t/,i/nf/o=/ur/n+/nb/n+/se/+k/b+/re/po/s-/1/
id:  what-the-*@?#!^!?
 ->  what-the-^2a@^3f#!^5e!^3f
 ->  wh/at/-t/he/-^/2a/@^/3f/#!/^5/e!/^3/f/
```

After this character cleaning procedure, directory names resulting from splitting the string into character pairs will be legal and not terribly inconvenient for mainstream Unix and Windows systems, for their command interpreters, and as web-exposed URL paths.

---

5. Pairpath initiation

The top of a pairtree hierarchy is signaled by the presence of a directory called "pairtree_root". There may be other filenames beginning with "pairtree" accompanying it, as in the example below. Lines of file content, when shown, appear in parentheses beneath the file name.

```
current_directory/
|   pairtree_version0_1        [which version of pairtree]
|    ( This directory conforms to Pairtree Version 0.1. Updated spec: )
|    ( http://www.cdlib.org/inside/diglib/pairtree/pairtreespec.html  )
|
|   pairtree_prefix
|    ( http://n2t.info/ark:/13030/xt2                                 )
|
\--- pairtree_root/
     |--- aa/
     |    |--- cd/
     |    |    |--- foo/
     |    |    |    |   README.txt
     |    |    |    |   thumbnail.gif
     |    |    ...
     |    |--- ab/ ...
     |    |--- af/ ...
     |    |--- ag/ ...
     |    ...
     |--- ab/ ...
     ...
     \--- zz/ ...
          | ...
```

The "pairtree_prefix" contains a string that should be prepended to every identifier inferred from the pairtree rooted at "pairtree_root". This may be used to reduce path lengths when every identifier in a given pairtree shares the same initial substring. In the example above, the pairpath "/aa/cd/" would thus correspond to the identifier "http://n2t.info/ark:/13030/xt2aacd".

---

6. Pairtree benefits

Pairtree can be used with any object identifier, but its real strength comes when two main assumptions are also in effect. The first assumption is that every component on which an object depends will be stored in the filesystem. Increasingly, digital library systems recognize that the risk of scattering components among databases and files can be reduced when all primary data is kept in non-volatile storage that can be backed up and manipulated using core, ubiquitous operating system tools. While database indexes are important for supporting fast or complex query execution, this pre-condition merely requires that those indexes hold secondary copies of object components (e.g., metadata).

The second assumption is that all the components of an object, and only the components of that object, are stored in an object's directory. Thus an object's directory contains no components belonging to another object. Of course complex objects will still contain other objects, and possibly other pairtrees, but such object containment is not visible to the pairtree algorithm except with a recursive pass.

With these two pre-conditions met, a pairtree can be imported by a system that knows nothing about the nature or structure of the objects but can still deliver any object's files by requested identifier. The mapping is reversible, so the importing system can also walk the pairtree and reliably enumerate all the contained object identifiers. To the extent that object dependencies are stored inside the pairtree, simple or complex collections built on top of pairtrees can recover from index failures and reconstruct a collection catalog simply by walking the trees. Finally, pairtrees have the advantage that many object operations, including backup and restore, can be performed with native operating system tools.

---

7. Interoperability: Windows and Unix File Naming

Besides the fundamental difference between path separators ('\' and '/'), generally, Windows filesystems have more limitations than Unix filesystems. Windows path names have a maximum of 255 characters, and none of these characters may be used in a path component:

```
    < > : " / | ? *
```

Windows also reserves the following names: CON, PRN, AUX, NUL, COM1, COM2, COM3, COM4, COM5, COM6, COM7, COM8, COM9, LPT1, LPT2, LPT3, LPT4, LPT5, LPT6, LPT7, LPT8, and LPT9. See **[MSFNAM]** for more information.

---

8. Security Considerations

Pairtree poses no direct risk to computers and networks. As a filesystem format, pairtree is capable of holding files that might contain malicious executable content, but it is no more vulnerable in this regard than formats such as TAR and ZIP.

Appendix A. Sample Implementation

There is a **[PAIRTREE]** Perl module at CPAN that implements two mappings. The routine, id2ppath, maps an identifier to a pairpath, and another routine, ppath2id, performs the inverse mapping. The usage synopsis follows.

```
 use File::Pairtree;            # imports routines into a Perl script

 id2ppath($id);                 # returns pairpath corresponding to $id
 ppath2id($path);               # returns id corresponding to $path
```

---

9. References

[ASCII]	"Coded Character Set -- 7-bit American Standard Code for Information Interchange, ANSI X3.4," 1986.
[MSFNAM]	Microsoft, "**Naming a File**," 2008 (**HTML**).
[PAIRTREE]	Kunze, "**File::Pairtree Perl Module**," November 2008 (**HTML**).
[RFC3629]	Yergeau, F., "**UTF-8, a transformation format of ISO 10646**," STD 63, RFC 3629, November 2003 (**TXT**).

Authors' Addresses

John A. Kunze
California Digital Library
415 20th St, 4th Floor
Oakland, CA 94612
US
Fax: +1 510-893-5212
Email: **jak@ucop.edu**

Martin Haye
California Digital Library
415 20th St, 4th Floor
Oakland, CA 94612
US
Fax: +1 503-234-3581
Email: **martin.haye@ucop.edu**

Erik Hetzner
California Digital Library
415 20th St, 4th Floor
Oakland, CA 94612
US
Fax: +1 503-234-3581
Email: **erik.hetzner@ucop.edu**

Mark Reyes
California Digital Library
415 20th St, 4th Floor
Oakland, CA 94612
US
Fax: +1 503-234-3581
Email: **mark.reyes@ucop.edu**

Cory Snavely
University of Michigan Library IT Core Services
920 N University Ave, 300D Hatcher Library N
Ann Arbor, MI 48109
US
Fax: +1 734-647-6897
Email: **csnavely@umich.edu**