Offline Patching

BigWorld Technology 2.1. Released 2012.

Software designed and built in Australia by BigWorld.

Level 2, Wentworth Park Grandstand, Wattle St
Glebe NSW 2037, Australia
www.bigworldtech.com

Table of Contents

Chapter 1. Introduction

1.1. Overview

This document describes the offline patching library, which allows for difference-generation and patching of resources and game binaries in a systematic manner.

In development, much of the game resources exist as text XML files, for example, `.chunk` files. These are generally packed into a binary format for shipping at release time. Thus, at release time, the game resources consist of directory hierarchies of mostly binary resources and executables, and this tree of resources can be further packed into a single (or multiple) ZIP archive and used directly by the BigWorld game engine. Packing binary resources is done via the ResPacker tool[1].

After the game has been shipped, game logic may be modified to fix bugs or tune behaviour, or new content may be added in the form of new space geometries, new entity types, character models, and so on. Distributing these to end-users involves sending them only the changes, in the form of the new files and patches to existing files that have been modified. These changes are bundled into patch archives, and can be targeted against specific resource archives or directory trees alike.

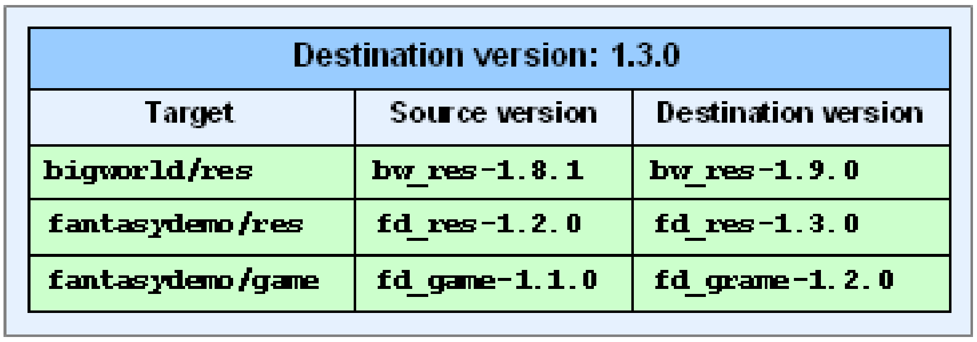
The offline patcher tools are located in `bigworld/tools/misc/offline_patcher`.

1.1.1. Preparing release patch archives

For each released version, patch archives against previously released versions need to be created. For example, if you have released version *A*, *B* and *C*, and have a new version to release (say, *D*), then for each of *A*, *B* and *C*, an upgrade path will need to be available for end users with those versions (*e.g.*, *A* → *D*, *B* → *D*, *C* → *D*).

The `MakePatch` tool creates a patch archive between two directory trees or two ZIP file archives[2].

A patch archive file may contain many such sets of changes between two directory trees. Diagrammatically, a patch archive file consists of the following structure:



Destination version: 1.3.0		
Target	Source version	Destination version
bigworld/res	bw_res-1.8.1	bw_res-1.9.0
fantasydemo/res	fd_res-1.2.0	fd_res-1.3.0
fantasydemo/game	fd_game-1.1.0	fd_grame-1.2.0

Patch archive file layout

The above diagram illustrates a patch file intended for patching against the BigWorld demonstration package FantasyDemo. Firstly, the BigWorld resources in the intended target path `bigworld/res` have been upgraded from the 1.8.1 release to the 1.9.0 release. Secondly, the game-specific resources, present in intended target path `fantasydemo/res`, are also being patched, with their versions incremented to 1.3.0 from 1.2.0. Finally, the game client itself is being upgraded, going to version 1.2.0 from 1.1.0 for the intended target path `fantasydemo/game`.

This example demonstrates that different parts of the end-user's distribution can be targeted for patching. The above intended targets (*i.e.*, `bigworld/res`, `fantasydemo/res`, `fantasydemo/game`) are not de-

[1]For details on ResPacker, see the Client Programming Guide's section *Releasing The Game* → "Prepare the assets" → "`res_packer`".
[2]For details, see "MakePatch" on page 9 .

fined anywhere in the end-user's distribution, and are defined in the version file that is downloaded by the patching client.

As another example, small script changes can be sent to end-users where there are only a few changes scattered around the client scripts in `fantasydemo/res/scripts/client`. Thus, this path could be a target path being patched against contained in a patch archive file, and the changeset can be added to a patch file by running MakePatch against the old and new versions of the contents of `fantasydemo/res/scripts/client`.

1.1.2. Client-side patching process

The Python-based patching libraries are designed to be used by a game launcher application to check for updates against a *versions file* from a HTTP server, which specifies how to update to the current version. As part of the offline patcher feature, there is a minimal update client implemented as part of the CheckVersion tool[3].

1.1.2.1. Targets and the state file

A *target* is either a directory tree or a resource ZIP archive, and is version-controlled locally on the end-user's distribution via a *state file*. This file contains a list of defined targets (identified by a name string), and for each of these targets, the version of that target that is currently installed. For example, a target may be the game resource ZIP archive (*e.g.*, `/res.zip`), or it may be a directory tree (*e.g.*, `./game`).

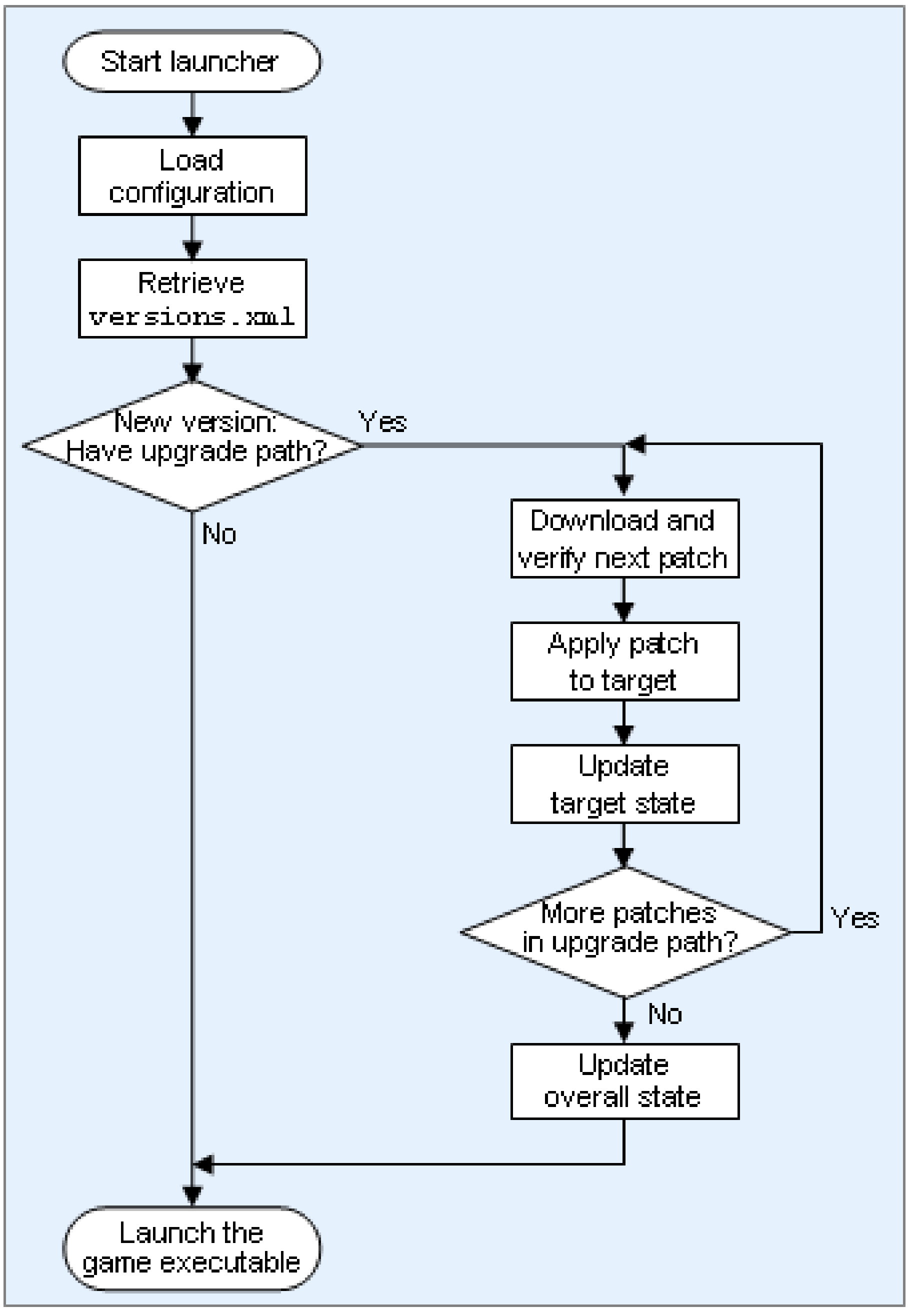
1.1.2.2. Versions file and upgrade paths

The versions file contains *upgrade paths* to the current release version for all previous release versions. Each upgrade path specifies a set of *targets* and patch retrieval information for those targets. Versions are identified by version strings[4].

1.1.2.3. Patching

The patching process follows this flowchart:

[3]For details, see “CheckVersion” on page 12 .
[4]For details, see “Versions file” on page 13 .



Process for starting a game, possibly updating before launch

Typically, when the user launches the game, they actually launch the update client, which checks whether it has the current version from the versions file. The game is launched from within the update client only if the current version matches the current version on the server's versions file.

Note that there are potentially many patch files as part of an upgrade path targeting multiple paths. This allows you to divide up your resources into those that change often, to those that change infrequently (*e.g.*, the BigWorld resources would change infrequently, whereas space geometry data and script would change more frequently), and have independent versioning for each target.

1.2. Requirements

The offline patcher has been tested against Python version 2.5. It also makes use of a Python extension module called `bsdiff`, which is located in `bigworld/tools/misc/offline_patcher/patcherlib`. It is supplied in the form of a `_bsdiff.so` for use under Linux, and `_bsdiff.pyd` for use under Windows. Please note that you can recompile against another version of Python[5].

1.3. Layout of the offline patcher tools and libraries

The layout of the tools and library in the BigWorld distribution at `bigworld/tools/misc/offline_patcher` is described below:

- `make_patch.py`

  Top-level script that implements the MakePatch tool - it creates a patch file archive between two directory trees. For details on MakePatch, see “MakePatch” on page 9 .

- `apply_patch.py`

  Top-level script that implements the ApplyPatch tool - it applies a patch file to either a directory hierarchy or to the contents of a ZIP file archive. For details on ApplyPatch, see “ApplyPatch” on page 11 .

- `check_version.py`

  Top-level script that implements the CheckVersion tool - it checks for an upgrade path on a remote server, then downloads and applies patch files against target paths defined in the upgrade path. For details on CheckVersion, see “CheckVersion” on page 12 .

- `patcherlib`

  Python module directory containing modules with functionality useful for creating and applying patch files. For details on this library, see “The `patcherlib` package” on page 17 .

- `versionslib`

  Python module directory containing modules with functionality useful for checking versions and applying an upgrade path. For details on this library, see “The `versionslib` package” on page 21 .

[5]For details, see “Compiling the `bsdiff` extension module” on page 19 .

Chapter 2. Operation

2.1. MakePatch

This tool is located at `bigworld/tools/misc/offline_patcher/make_patch.py`.

MakePatch builds patches which can be used for ApplyPatch to patch *targets*, and supports multi-versioned patch files (for details on ApplyPatch, see “ApplyPatch” on page 11).

Both MakePatch and ApplyPatch have their implementations as part of the `patcherlib.command_line` module.

2.1.1. Command-line usage

The syntax for invoking MakePatch via the command line is described below:

```
make_patch.py [options] <source-path> <dest-path> <patch-output-path>
```

The three required arguments (one source path for each source version that the patch will support) are described below:

- `<source-path>`

  The source tree path. This can be either a directory or a ZIP file archive.

- `<dest-path>`

  The destination tree path. This can be either a directory or a ZIP file archive.

- `<patch-output-path>`

  The patch output path.

The available *`options`* are described below:

- `-h`, `--help`

  Displays the help page.

- `--debug-level=LOGLEVEL`

  Specifies the debug level of MakePatch - *`LOGLEVEL`* must have one of the following values: `DEBUG`, `INFO`, `WARNING`, `CRITICAL`, or `ERROR`.

- `--source-version=SOURCEVERSION`

  Specifies the label for the source version - the default value is the basename of the `<source_path>` argument.

- `--use-checksums-if-unmodified`

  Specifies that MakePatch should use checksums for unmodified files as well as modified files - the default is to not use checksum for unmodified files.

- `--ignore-patterns=IGNOREPATTERNS`

Specifies the patterns to ignore when traversing the directory hierarchy - the default is `/CVS$|/\.svn$|/\.#[^/]+$`.

Uses commas to delimit the patterns, and use the backlash character to escape the comma character in a pattern (*i.e.*, `'\,'`).

By default, certain file patterns are not considered when traversing the hierarchies looking for changes. Files matching the ignored patterns are not considered when computing the difference between the two trees. If directories match the ignored file patterns, then the entire tree is culled from the search space. These file patterns (as regular expressions) are:

- `/CVS$`

  This pattern matches CVS control directories.

- `/\.svn$`

  This pattern matches Subversion control directories.

- `/\.#[^/]+$`

  This pattern matches `.#` files that are a result of CVS updates.

2.1.2. Patch archives

The MakePatch tool generates a patch archive, which is a tarfile with the following:

- The changes between the source directory tree and a destination directory tree.
- A manifest file.

The directory tree versions are given names indicating their version (*e.g.*, `bw_res-1.9.0`). The only requirement for a version is that they be a string - they do not have to conform to any convention, but it may be wise to have some scheme where the target trees are identified by name and a version string appended to the target name.

Currently, you can specify more than one source version to provide changes against the destination version. This could be used to make a patch multi-versioned, that is, supporting multiple released versions by combining their changes from each supported source version into a single patch archive. However, this is discouraged, as it is much better for the CheckVersion script or equivalent to handle selection of individual patches to download from the *versions file* (for details on CheckVersion, see “CheckVersion” on page 12).

The changes for a particular version consist of the following:

- New files added in the destination version.
- Binary delta patches to existing files.

The *manifest file* contains instructions on which files to add, modify or delete, as well as checksum information. By default, checksums are generated for modified files only. However, you can use the `--use-checksums-if-unmodified` option to have checksums generated for every file encountered in the source directory tree. If checksums are present in the manifest file, then they will each be checked.

Note

ZIP file archives inside ZIP archives have their files extracted and the deltas generated based on the contents of the files contained inside the ZIP archive, rather than on the ZIP archive files themselves.

2.1.3. Multi-source support

MakePatch supports patch files that contain changes for multiple sources, intended against either multiple targets and/or versions.

This means that for a group of targets, changes for each target from one version to another can be bundled into the same patch archive.

This also means that for a particular target the changes for multiple versions of that target can be contained within the same patch archive. Typically, however, it is more advantageous to have a single-version patch files for each released version, and rely on VersionCheck and the versions file to drive the patching client to download a single-version patch. This is more flexible and saves on download bandwidth.

Changes for multiple sources can be added at the archive creation, or they can be added to an existing patch archive by re-running MakePatch with the same patch archive as the patch output path.

2.1.4. Example usage

2.1.4.1. Creating a single-source patch archive

The example below creates a patch archive between the source tree with path `1.8.0/bigworld/res` (labelled `bw_res-1.8.0`) and `1.9.0/bigworld/res` (labelled `bw_res-1.9.0`), and outputs it to a patch archive file called `bw_res-1.8.0-1.9.0.patch`.

```
$ python make_patch.py --source-version=bw_res-1.8.0 \
  1.8.0/bigworld/res     1.9.0/bigworld/res     bw_res-1.8.0-1.9.0.patch
```

2.1.4.2. Creating a multiple-source patch archive

The example below creates a patch archive between source trees:

- Path `1.8.0/fantasydemo/res` (labelled `fantasydemo_res-1.8.0`) and path `1.9.0/fantasydemo/res`
- Path `1.8.0/fantasydemo/game` (labelled `fantasydemo_game-1.8.0`) and path `1.9.0/fantasydemo/game`

These two sets of changes are packaged into a patch archive file called `fantasydemo-1.8.0-1.9.0.patch`.

```
$ python make_patch.py --source-version=fantasydemo_res-1.8.0
  1.8.0/fantasydemo/res     1.9.0/fantasydemo/res
 fantasydemo-1.8.0-1.9.0.patch
...

$ python make_patch.py --source-version=fantasydemo_game-1.8.0
  1.8.0/fantasydemo/game    1.9.0/fantasydemo/game
 fantasydemo-1.8.0-1.9.0.patch
```

2.2. ApplyPatch

This tool is located at `bigworld/tools/misc/offline_patcher/apply_patch.py`. It complements the MakePatch tool by applying patches made by it to a directory tree or ZIP archive. The ApplyPatch tool relies on functions exposed by the `patcherlib.patch_apply` module (for details, see “The `patch_apply` module” on page 18).

Both MakePatch and ApplyPatch have their implementations as part of the `patcherlib.command_line` module.

2.2.1. Command line usage

The syntax for invoking ApplyPatch via the command line is described below:

```
apply_patch.py [options] <patch-path> <target-path> <target-version>

-h, --help            show this help message and exit
--debug-level=LOGLEVEL one of DEBUG, INFO, WARNING, CRITICAL, ERROR
```

The three required arguments are described below:

- `<patch-path>`

  The path to the patch archive.

- `<target-path>`

  The destination tree path.

- `<target_version>`

  The target version before patching.

The available *`options`* are described below:

- `-h`, `--help`

  Displays the help page.

- `--debug-level=`*`LOGLEVEL`*

  Specifies the debug level of ApplyPatch - *`LOGLEVEL`* must have one of the following values: `DEBUG`, `INFO`, `WARNING`, `CRITICAL`, or `ERROR`.

2.2.2. Example usage

2.2.2.1. Patching against a directory tree with a patch file

The example below applies a patch archive on to a source tree with path `./res` and version `bw_res-1.8.0`, and outputs it to a patch archive file at `downloads/bw_res-1.8.0-1.9.0.patch`.

```
$ python apply_patch.py downloads/bw_res-1.8.0-1.9.0.patch bigworld/res
 bw_res-1.8.0
```

2.3. CheckVersion

CheckVersion is a command line tool that implements a minimal patching client. As input, it takes the following configuration files:

- A *state file*.

  Contains the state of targets for that end-user's distribution. The default is `state.xml`.

- A *versions file*.

  Contains upgrade paths for possible release versions. This is retrieved from a remote server via HTTP. An example is supplied in `bigworld/tools/misc/offline_patcher/examples/versions.xml`.

- A general game configuration file.

  Contains the URL to retrieve the versions file from. The default is `patcher_config.xml`, and an example is supplied in `bigworld/tools/misc/offline_patcher/examples/patcher_config.xml`.

CheckVersion gets from the general game configuration the URL for the versions file, then retrieves its contents and parses it. It then compares its local version from the local state with this remote version specified in the versions file, and determines an upgrade path if necessary from the versions file. It then downloads each patch archive, and applies them using the functionality provided in the `patcherlib.patch_apply` module (for details on this module, see “The `patch_apply` module” on page 18).

This script is intended as a bare-bones update client, and it is expected that game developers implement their own custom launcher application incorporating an update client that may reuse the functionality provided in the CheckVersion script. For example, a UI update client could be implemented which utilises progress bar widgets, HTML display of release notes, and so on.

The CheckVersion script is driven by the retrieved versions file (for details on this file, see “Versions file” on page 13).

2.3.1. Command line usage

The syntax for invoking CheckVersion via the command line is described below:

```
usage: check_version.py [options]

options:
-h, --help            show this help message and exit
-c CONFIGPATH, --config=CONFIGPATH
                      The patcher configuration file that contains the
                      versions.xml URL. Defaults to "patcher_config.xml".
-s STATEPATH, --state=STATEPATH
                      The state file. Defaults to "state.xml".
--debug-level=DEBUGLEVEL
                      The logging debug level. Defaults to "INFO".
```

2.3.2. Versions file

This file contains upgrade paths for past released versions to upgrade to the current version. Functionality for parsing this XML document is supplied via the `VersionsXML` class (for details, see “The `versions` module” on page 21).

The versions file is downloaded from a URL contained within the configuration file, defaulting to a `patcher_config.xml` in the current working directory, and is parsed using the simple `Config` class in `check_versions.py`. Developers may wish to change this scheme when developing their own update client.

An example versions file is supplied in `bigworld/tools/misc/offline_patcher/examples/versions.xml`.

The example file below describes an upgrade path from a past released version of a theoretical game from an initial version 1.0 to a new current version 1.1. In the example, there are three targets for which there are patches for:

- `your_game_res`

  Refers to a ZIP archive of the packed game resources (*e.g.*., res-packed tree of the contents of `your_game/res`).

- `your_game_bin`

The game executable directory, designed for patching against the game executable and any other binaries, such as DLLs needed by the executable, as well as configuration files needed by the game, such as `paths.xml`.

- `bw_res`

  Refers to a ZIP archive of the packed BigWorld base resources (*i.e.*, res-packed tree of the contents of `bigworld/res`).

```
<?xml version="1.0"?>
<root>
  <currentVersion>1.1</currentVersion>
  <property name="baseURL">http://update.yourgame.com/downloads</property>
  <supportedVersions>
      <version>
          <name>1.0</name>
          <property name="updateInfoURL">info/test.php</property>
          <targets>
              <target>
                  <name>your_game_res</name>
                  <path>../your_game_res.zip</path>
                  <sourceVersion>1.0</sourceVersion>
                  <destVersion>1.1</destVersion>
                  <patch>
                      <transferType>http</transferType>
                      <name>your_game_res.1.0-1.1.patch</name>
                      <property name="url">patches/
your_game_res.1.0-1.1.patch</property>
                      <property
 name="md5sum">00112233445566778899aabbccddeeff</property>
                  </patch>
              </target>
              <target>
                  <name>your_game_bin</name>
                  <path>.</path>
                  <sourceVersion>1.0</sourceVersion>
                  <destVersion>1.1</destVersion>
                  <patch>
                      <transferType>http</transferType>
                      <name>your_game_bin.1.0-1.1.patch</name>
                      <property name="url">patches/
your_game_bin.1.0-1.1.patch</property>
                      <property
 name="md5sum">00112233445566778899aabbccddeeff</property>
                  </patch>
              </target>
              <target>
                  <name>bigworld_res</name>
                  <path>../bw_res.zip</path>
                  <sourceVersion>1.9.3.0</sourceVersion>
                  <destVersion>1.9.4</destVersion>
                  <patch>
                      <transferType>http</transferType>
                      <name>bw_res.1.9.3.0-1.9.4.patch</name>
                      <property name="url">patches/
bw_res.1.9.3.0-1.9.4.patch</property>
                      <property
 name="md5sum">00112233445566778899aabbccddeeff</property>
                  </patch>
              </target>
          </targets>
```

```
        </version>
    </supportedVersions>
</root>
```

Example versions file

Note that version 1.1 of the overall game uses 1.9.4 BigWorld resources over version 1.0's usage of 1.9.3 BigWorld resources. This illustrates that the versioning of the individual targets is independent from the overall game versioning.

This is only an example layout for a theoretical game. Game developers must take into account the requirements of their game when partitioning their resources into suitable resource trees.

2.3.2.1. Properties

Properties are used to store specific details about some part of the patching process. For example, you might have a URL to a web page for each upgrade path containing the release notes that you wish to have displayed while the patching process is running. Custom transfer handlers can utilise the property mechanism to get details about how to retrieve a patch. Properties can be defined at the root level, at the upgrade path level (that is, under the `version` element), or at the patch level (under the `patch` element).

Properties are key-value pairs, described as in the following:

```
<property name="key_name">value</property>
```

The combined properties from root-level, upgrade path-level to patch-level are passed to the constructor of any transfer handler as keyword arguments. Properties at lower levels override properties at higher levels.

For example, the versionslib.transfer module defines the HTTP transfer method in the class HTTPTransferHandler. The constructor to HTTPTransferHandler has a baseURL property which, in the example above, is described at the root level. However, it could also be described per-upgrade path, so that each upgrade path has a different base URL.

2.3.3. State file

The state file specifies for each target which version is currently installed on the end-user's distribution. This is only a simple implementation - developers may choose to implement persistence of target state in another way as part of their launcher process (*e.g.*, Windows Registry keys). Other implementations should implement the Python State interface for reusing the rest of the CheckVersions script and the `versionslib` library.

An example state file is supplied in bigworld/tools/misc/offline_patcher/examples/state.xml. It corresponds to the supplied example versions.xml versions file.

Functionality for parsing the state XML file is supplied in the form of the `StateXML` class defined in the `versionslib.state` module (for details on this class, see “The `state` module” on page 23).

2.3.3.1. Example state file

```
<?xml version="1.0"?>
<root>
    <currentVersion> 1.0 </currentVersion>
    <targets>
        <target name="your_game_res"> 1.0 </target>
        <target name="your_game_bin"> 1.0 </target>
        <target name="bigworld_res"> 1.9.4.0 </target>
    </targets>
```

```
</root>
```

Example state file

2.3.4. Deployment notes

The CheckVersion script relies on `patcherlib` for applying patches to targets. Deploying this may involve compiling all the `.py` files to `.pyc` or `.pyo` files, then adding them in a ZIP archive and placing in the Python library path.

The game launcher must run script methods for checking against a versions file to verify that its version is up-to-date. This requires that a Python interpreter be involved at some point in the game launch process, which implies that the launcher is itself a Python script or an executable that is linked against the Python runtime library.

2.3.5. Patch transport schemes

There is currently only one method of transport for patches: HTTP. However, the versions file syntax can allow for more transport schemes (such as BitTorrent, for example). Transport schemes are specified as part of the delivery element in the versions document - *e.g.*, consider the following example versions file fragment:

```
<patch>
 <transferType>           http                              </transferType>
 <name>                   bw_res-1.9.4.0-1.9.x.ypatch          </name>
 <property name="url">    patches/bw_res-1.9.4.0-1.9.x.y.patch </property>
 <property name="md5sum"> c166772f145bcc59ac714b7995e52c2a </property>
</patch>
```

The transferType tags describes how to download a patch file from a URL via HTTP. This transfer handler supports resuming of partial downloads with a small amount of rollback, that is, resuming will occur at a point (by default 4K) before the end of the received file fragment. In order to detect possible file corruption, the newly downloaded portion corresponding to the rollback amount is checked against what was downloaded previously, and the download will restart from the beginning if corruption is detected.

Properties for a patch transfer are specified using the `property` tags. The HTTP transfer type defines the following properties:

- `url`

  The URL from which to retrieve the patch file. If a relative URL is given, then it is relative to the base of the version file URL.

- `md5sum`

  The MD5 sum of the retrieved patch file.

You can add your own transfer type by specifying properties that describe how to retrieve a patch file. You will need to modify the CheckVersions script and create and register a new `PatchTransferHandler` class.

Chapter 3. Application Programmers Interface

This chapter describes the API that the offline patcher tool is constructed from. This part is of interest to developers wishing to automate some part of the patch packaging process, write extensions to the offline patcher, or write their own patching client.

3.1. The `patcherlib` package

This section describes the `patcherlib` package, which contains functionality to compute differences between two directory trees, generate patch file archives for those changes, and apply patch file archives to directory trees.

3.1.1. The `patch_file_builder` module

This section pertains to the `patcherlib.patch_file_builder module`, which is used by MakePatch to generate patch files. Developers wishing to integrate automated release tools can reuse functionality in this module to generate patch files readable by the patcherlib.apply_patch library module and the ApplyPatch command line program (for details on ApplyPatch, see “ApplyPatch” on page 11).

This module implements the `PatchFileBuilder` class, which is used to build a patch file archive containing changes between two directory hierarchies. It has the following methods:

- **`PatchFileBuilder( sourcePath, destPath, outFilePath, sourceVersion, useChecksumsIfUnmod=False, ignorePatterns=DEFAULT_IGNORE_PATTERNS )`**

  The arguments of `PatchFileBuilder`'s constructor are described below:

  - `sourcePath`

    One of the directory trees (the other is specified by `destPath`) to take changes between for storing in the patch archive that will be written out to `outFilePath`.

  - `destPath`

    See `sourcePath`.

  - `outFilePath`

    Patch archive to which the changes will be written.

  - `sourceVersion`

    The source version name label under which these sets of changes will be added to the patch file archive.

  - `useChecksumsIfUnmod=False`

    Determines whether files that are not modified will also have their checksums computed and stored in the manifest file for checking against when the patch file is applied.

    By default this is `False`, which means that only checksums for files that have been modified will have their checksums computed and checked.

  - `ignorePatterns=DEFAULT_IGNORE_PATTERNS`

    A list of patterns for which files which have a successfully matching relative path will be ignored. This defaults to `DEFAULT_IGNORE_PATTERNS` which is defined in `offline_patcher/patcherlib/treediff.py`.

- **`run()`**

Creates the patch file archive.

3.1.2. The `patch_apply` module

The functions defined in the `patcherlib.patch_apply` module are used in the ApplyPatch script to apply a patch against a directory hierarchy, and are described below.

- **`PatchApply` class**

  This class implements the patching process, and has the following methods:

  - **`PatchApply( sourceVersion, sourcePath, patchFilePath, callback=PatchApplyCallbackInterface() )`**

    The arguments are described below:

    - `sourceVersion`

      Name of the version.

    - `sourcePath`

      Directory against which the patch will be applied.

    - `patchFilePath`

      The patch archive file.

    - `callback=PatchApplyCallbackInterface()`

      Object that implements the `PatchApplyCallbackInterface`, and that will be called back with the progress of the patching process.

  - **`run()`**

    Applies the patch.

- **`PatchApplyCallbackInterface` interface**

  The patching process implemented by the `PatchApply` class can be monitored for progress by means of a callback interface that calls back on certain events, which are described below. This is useful for updating GUIs on what exactly is being patched, and the progress of the overall patching effort.

  - **`onStart( numOperations )`**

    Called when the patching process has started, it receives as argument the number of operations taken from the patch archive file. This gives the number of additions, modifications and deletion operations in this patch

  - **`onDirAddStart( path )`**

    Called when the patching process has to add a directory - it calls `onDirAddStart` before executing the addition, then `onDirAddFinish` when it has finished.

  - **`onDirAddFinish( path )`**

    See **`onDirAddStart( path )`**.

  - **`onDirDelStart( path )`**

Called when the patching process has to delete a directory - it calls `onDirDelStart` before executing the addition, then `onDirDelFinish` when it has finished.

- **`onDirDelFinish( path )`**

  See **`onDirDelStart( path )`**.

- **`onFileAddStart( path )`**

  Called when the patching process has to add a file - it calls `onFileAddStart` before executing the addition, then `onFileAddFinish` when it has finished.

- **`onFileAddFinish( path )`**

  See **`onFileAddStart( path )`**.

- **`onFileDelStart( path )`**

  Called when the patching process has to delete a file - it calls `onFileDelStart` before executing the addition, then `onFileDelFinish` when it has finished.

- **`onFileDelFinish( path )`**

  See **`onFileDelStart( path )`**.

- **`onFileModStart( path )`**

  Called when the patching process has to patch an individual file - it calls `onFileModStart` before executing the patch, then `onFileModFinish` when it has finished.

- **`onFileModFinish( path )`**

  See **`onFileModStart( path )`**.

- **`onFinish()`**

  Called when the patching process has finished.

3.1.3. The `bsdiff` module

This section describes the functions defined in the `patcherlib.bsdiff` module, which provides the functionality to compute binary differences between two individual files.

This module is implemented as a Python extension module, written in C++, and is derived from Colin Percival's `bsdiff` utility (for details, see "`bsdiff` - Binary diff/patch utility"on page 29). The actual library is loaded from an architecture-specific package directory under the `bigworld/tools/misc/offline_patcher/patcherlib/bin` directory containing a version of `_bsdiff.so` under Linux, and `_bsdiff.pyd` under Windows. The module has been compiled against Python 2.5 - the source is provided so that it can be compiled against another version of Python.

3.1.3.1. Compiling the `bsdiff` extension module

The source to the `bsdiff` extension module can be found in `src/tools/offline_patcher/bsdiff`. To compile, follow the instructions below:

- **Under Linux**

  Under Linux, the Python development package needs to be installed to recompile `bsdiff` - under Fedora and CentOS, this is called `python-devel`, and in Debian, this is called `python-dev`.

Compile the module using the following commands:

```
$ make clean all
```

This will build the Python extension module into an architecture-specific directory in `bigworld/tools/misc/offline_patcher/patcherlib/bin`.

- **Under Windows**

  The Visual Studio solution file `bsdiff_2005.sln` has been tested for use with Visual Studio 2005. Developers should ensure that they have a copy of the Python sources, and that they link against the compiled library file (*e.g.*, `python25.lib`), which needs to be built from the sources (usually using the `python-core` project). Please refer to the Python documentation for how to build under VS 2005.

  Developers need to set the environment variable `PYTHONHOME` to point to the Python source distribution that they wish to link against, or edit the relevant properties where the variable `$(PYTHONHOME)` is used to point to the built Python distribution.

3.1.3.2. The `bsdiff` module interface

The `bsdiff` module uses the `bslib` C++ library to do its work. The `bslib` sources are located in `src/tools/offline_patcher/bsdiff`, and implemented in the files `bslib.hpp` and `bslib.cpp`. Those functions are exposed to Python by the following Python methods:

- **`haveDiffs( path1, path2, bufSize = 4096 )`**

  Returns `True` if files at *path1* and *path2* have differences, and `False` otherwise. The `bufSize` parameter specifies how much data to read and compare at a time. It raises `IOError` if either of the files at those paths cannot be read, or `TypeError` if the files at those paths are not regular files.

- **`diffFilesToFile( srcPath, dstPath, patchPath )`**

  Computes the binary differences between the files at `srcPath` and `dstPath`, and writes them to a file at `patchPath` in a format recognised by `patchFromFile`, as well as the original `bspatch` command line utility.

- **`patchFilesFromFile( srcPath, dstPath, patchPath )`**

  Patches a file created from the `diffToFile` function at `srcPath` from a patch file at `patchPath` (created by `diffFilesToFile`), and writes it to `dstPath` (which can be the same as `srcPath` to overwrite the original source file).

3.1.4. Other modules

A brief description of the other utility modules in `patcherlib` are given here.

- `command_line`: Used for implementing the commands used by the top level MakePatch and ApplyPatch command line tools.
- `manifest`: Used for parsing and creating patch manifests.
- `md5_interface`: Module abstracting different MD5 libraries across different versions of Python.
- `temp_dir_list`: A module for creating temporary directories and deleting them when no longer needed.
- `treediff`: A module for traversing two directory trees and determining the differences.
- `utils`: A general utility module.

3.2. The versionslib package

This package is used to query a remote server and download patches to be applied against a local distribution using the patcherlib package.

It is composed of the following modules:

- `versions`
- `state`
- `transfer`
- `simple_config`

The `version`, `state`, and `transfer` modules contain the main classes dealing with how to check against a remote server and download patches, and are described in the following sections.

The `simple_config` module contains a simple configuration file parser, which CheckVersion uses to read a configuration containing a URL to the versions file.

3.2.1. The versions module

The interfaces and classes implemented by this module are described below:

- **`VersionsInterface` interface**

  The Python `Versions` interface is used by `CheckVersion` to extract the patching instructions contained within a versions file. It is defined in the `versionslib.versions` module. For alternate implementations of a versions file, developers can implement this interface and pass it to `CheckVersion` in the same way that it currently passes the `VersionsXML` class instances around.

  `VersionInterface` implements the following methods:

  - **`getUpgradePath( versionName )`**

    Returns the version upgrade path for a named version in the form of an `UpgradePath` object, or `None` if no such path exists.

  - **`getCurrentVersion()`**

    Returns the remote current version name.

- **`VersionsXML` class**

  This is the default implementation of `VersionsInterface` - it uses XML as the format, and is also defined in the `versionslib.versions` module, and implements the following methods:

  - **`VersionsXML( versionsDocument )`**

    Constructs a `VersionsXML` instance. The `versionsDocument` parameter is an XML DOM object of the versions file.

  An example on how to use the `VersionXML` class is displayed below:

```
from versionslib import versions
from xml.dom import minidom
import urllib

conn = urllib.urlopen( "http://example.com/versions.xml" )
```

```
versionsContent = conn.read()
conn.close()

versionsDoc = minidom.parseString( versionsContent )
versions = versions.VersionsXML( versionsDoc )

if localVersion != versions.getCurrentVersion():
  upgradePath = versions.getUpgradePath( localVersion )
```

Using the `VersionsXML` class

- **`UpgradePath` class**

  This class is defined in the `versionslib.versions` module - it is an upgrade path for a particular older release version, and holds target patches (in the form of `Target` objects required to patch to the new release version. For details, see `Target` class defined below.

  This class has the following attributes:

  - **`name`**

    The name of the upgrade path's source version.

  - **`targets`**

    A Python list of `Target` instances.

  This class has the following methods:

  - **`getProperties()`**

    Return the accumulated properties from the root-level to the upgrade-path level.

  - **`setUpgradePathProperties()`**

    This methods is called as part of the parsing process, and is not intended to be used by applications.

- **`Target` class**

  A `Target` object specifies a method of transferring the patch file using a transfer handler, and which target path to apply the patch against.

  This class has the following attributes:

  - **`name`**

    The name of the target.

  - **`path`**

    The path to the target to patch.

  - **`sourceVersion`**

    The source version name of the target (what its current version should be).

  - **`destVersion`**

    The destination version name of the target (what its new version will be after successful patching).

  - **`transferType`**

The transfer type string, which specifies what transfer handler to use. For details, see “The `transfer` module” on page 24 .

- **`patchName`**

  The file name of the patch archive file.

Additionally, it has the following method:

- **`getProperties()`**

  Return a Python dictionary containing the accumulated root-level, upgrade path-level to partch-level properties.

3.2.2. The `state` module

This module contains definitions for `StateInterface` and the default implementing interface `StateXML`, both of which are used to read the local game version state.

The interfaces and classes implemented by this module are described below:

- **`StateInterface` interface**

  Objects implementing this interface must support the following methods:

  - **`getCurrentVersion()`**

    Returns the current local version of this distribution.

  - **`getTargetVersion( targetName )`**

    Returns the current local version of the given target.

  - **`setCurrentVersion( versionName )`**

    Changes the current local version of this distribution to `versionName`.

  - **`setTargetVersion( targetName, versionName )`**

    Changes the current local version of the given target.

- **`StateXML` class**

  Like the `VersionsXML` class, the default implementation of the `State` interface uses XML as the format, and is defined in the `versionslib.state` module.

  The classes implemented by this module are described below:

  - **`StateXML( path )`**

    Constructs an instance of the `StateXML` class, where `path` is a path to the XML document.

  An example on how to use the `StateXML` class is displayed below:

```
state = StateXML( "state.xml" )
if state.getCurrentVersion() != versions.getCurrentVersion():
 upgradePath = versions.getUpgradePath( state.getCurrentVersion() )
 ...
 for target in upgradePath.targets:
    localTargetVersion = state.getTargetVersion( target.name )
    ...
    # finished target upgrade, update state
    state.setTargetVersion( target.name, target.destVersion )

 # finished upgrade, set state version
 state.setCurrentVersion( upgradePath.destVersion )
```

Using the StateXML class

3.2.3. The `transfer` module

This module contains the interface for a `PatchTransferHandler`, which defines a method of retrieving a patch from a some location, for example via HTTP.

This section has some guidance on how to add a sub-class of a `PatchTransferHandler` when creating new transfer methods.

The interfaces and classes implemented by this module are described below:

- **`PatchTransferHandler` interface**

  Implementors of this interface define a method of retrieving a patch file from a remote source (*e.g.*, through HTTP or BitTorrent). It implements the following static methods:

  - **`PatchTransferHandler.create( transferType, destPath, **properties) )`**

    Creates an appropriate instance of a `PatchTransferHandler` object that can handle the given `transferType`. The properties parameter is a keyword argument dictionary which is used when constructing the appropriate instance of the transfer handler, and is passed to the appropriate subclass' constructor as keyword arguments.

  - **`PatchTransferHandler.register( transferType, klass )`**

    Registers a `PatchTransferHandler` sub-class (given as the `klass` parameter) with the given `transferType` string.

  Implementors of this interface should implement the following methods:

  - **`retrieve`()**

    Retrieve the patch now.

  - **addProgressListener()**

    Attach a progress listener object. See “Progress listeners” on page 26

  - **removeProgressLIstener()**

    Detach a progress listener object.

- **`HTTPTransferHandler` class**

  This class implements a transfer handler for transferring a patch from a HTTP server, and implements the following methods:

  - **`HTTPTransferHandler( destPath, baseURL, url, md5sum, **extraProps )`**

    The HTTP transfer handler downloads the file located at `url` (which may be a relative from `baseURL`, or an absolute HTTP URL), and the file should have a MD5 sum equal to that in the `md5sum` parameter.

The baseURL, url and md5sum parameters are passed in as a keyword arguments dictionary from PatchTransferHandler.create().

3.2.3.1. Creating the appropriate instance of transfer handler

Recall from the definition of the versions file that the transfer type and transfer properties are defined as part of the targets in the upgrade path's target list (for details, see "Versions file" on page 13). An example is displayed below.

```
<root>
 ...
 <property name="gameInfoURL">     http://www.yourgame.com/game_info.php
 ...
 <supportedVersions>
    <supportedVersion>
       <name> 1.0 </name>
       ...
       <property name="baseURL">   http://update.yourgame.com/patches/1.0
       ...
       <target>
       <patch>
          <transferType>           http                                </
transferType>
          <patchName>              bw_res-1.9.4.0-1.9.x.y.patch         </
patchName>
          <property name="url">    patches/bw_res-1.9.4.0-1.9.x.y.patch </
property>
          <property name="md5sum"> aabbccddeeff00112233445566778899     </
property>
       </patch>
    </supportedVersion>
    ...
 </supportedVersions>
</root>
```

Example versions file excerpt

As seen above, a patch XML definition consists of a `transferType` tag, which is mapped to a transfer handler class. Transfer handlers are registered with the base class `PatchTransferHandler`, so that instances are created using the `PatchTransferHandler.create` static method.

The `property` tags are parsed and placed into a Python dictionary, which can be passed to the transfer handler as part of the keyword arguments in the `PatchTransferHandler.create` method. The constructor for the appropriate `PatchTransferHandler` sub-class must accept these arguments or allow for general keyword parameters.

Each transfer handler class is registered with the `PatchTransferHandler` static method `PatchTransferHandler.register`.

3.2.3.2. Adding another `PatchTransferHandler` sub-class

New methods of transferring patches are implemented by adding another PatchTransferHandler object that subclasses the `PatchTransferHandler` abstract class, and it must be registered with the `PatchTransferHandler` class using its `register()` static method with an appropriate string tag, which is specified as part of the `transferType` element.

PatchTransferHandler sub-classes are expected to periodically notify of their progress using `PatchTransferHandler`'s `_notifyProgress()` method:

```
def _notifyProgress( self, progress, description ):
```

The progress parameter is an integer indicating percentage progress from 0 to 100. The description parameter is a dictionary containing key-value pairs that provide progress reports that are specific to the type of transfer handler used.

3.2.3.3. Progress listeners

Progress listeners are objects that listen to the progress of a patch transfer. They can be used to notify the end user of the progress of a particular transfer. They should be callable and they are passed the following parameters:

- **`progress`** (float)

  A floating point number from 0 to 100 indicating the percentage progress of the transfer. A special value of -1.0 indicates that the transfer failed.

- **`description`** (dict)

  A dictionary containing key-value pairs describing the context of the transfer. Different transfer methods will define different keys. The `HTTPPatchTransferHandler` defines the following keys:

3.2.3.3.1. HTTPPatchTransferHandler progress listener description keys

The `HTTPPatchTransferHandler` defines the following keys:

- `url`

  The URL being retrieved.

- `bytesReceived`

  The number of bytes downloaded.

- `bytesTotal`

  The total number of bytes to download.

- `status`

  The HTTP status code of the most recent attempt to download this patch.

- `errorMsg`

  If progress is set to -1, then this is an error message indicating what the failure was.

Chapter 4. Offline Patcher Terminology

The following terms are used in this document:

- *state file*

  An XML file which patcher clients maintain for each target.

- *upgrade path*

  A specification for upgrading to the current version. Each released version should have an entry in the versions file.

- *versions file*

  An XML file that patcher clients download via HTTP. The current version name is listed, and for each previously released version, it contains an upgrade path to the current version.

  The idea is for clients to check this file before launching the game, and only launch the game executable if the current version exists, otherwise it applies the Upgrade Path specified in the versions file.

Appendix A. Acknowledgements

Table of Contents

A.1. `bsdiff` - Binary diff/patch utility

Some parts of the BigWorld Offline Patcher feature are derived from the `bsdiff` binary `diff/patch` utility developed by Colin Percival. As per the licensing requirements, the copyright notice and licensing conditions are reproduced below.