

Eidos Patch System

Client-Side Programming Guide for Microsoft Windows

Version 1.2

Updated April 6, 2005

Prepared by Pinniped Software

Pinniped Software



game and internet programming

Copyright © 2005, Eidos, Inc. All rights reserved.

Table of Contents

Overview.....	1
Restrictions.....	1
Integrating the Patch System.....	1
Asynchronous Operations.....	1
Patch Discovery and Download.....	1
Patcher Creation and Destruction.....	1
Version Check and Patch Discovery.....	2
Determining Whether a Patch is Mandatory.....	2
Displaying a Description of the Patch.....	3
Downloading and Applying a Game Patch.....	3
Downloading and Preparing Asset Patches.....	3
Applying Asset Patches.....	4
Initializing the FilePatcher.....	4
Patching the Files.....	4

Eidos Patch System

Client-Side Programming Guide for Microsoft Windows

Version 1.2

Overview

This document describes how to integrate client-side patching for the Eidos Patch System into your Microsoft Windows game title.

Restrictions

- This version of the patching system only provides C++ interfaces.

Integrating the Patch System

Asynchronous Operations

The patch system is designed to work as part of your main game loop, without the need for your application to explicitly create download threads. As such, the more time-consuming operations, such as network reads, are performed asynchronously using internally managed threads. In order for the patch system to be able to manage this thread, a few requirements are levied on your application:

- Each time through the main loop, you need to call the `Patcher::Update` method.
- If you have an outstanding asynchronous call, you can test for completion of the function using the `Patcher::TestCompletion` method.

Patch Discovery and Download

The **Patcher** class contains all of the code needed for both patch discovery and download. However, for a Windows application, very little of the **Patcher** class is used. This section describes just those parts with which you must be familiar for Windows. For detailed information on all of the **Patcher** methods, see the *Eidos Patch System Client-Side Reference Manual*.

Patcher Creation and Destruction

A **Patcher** object is never created directly by the application. Rather, a single instance of a **Patcher** object is created by the static function `Patcher::Create()`. The priority parameter to `Create` is ignored under Windows, so you can pass anything you like. If your code is shared with other platforms, you can just use the same priority value as for those platforms:

```
Patcher::Status stat = Patcher::Create(prio);
```

When you are finished using the patch downloader, it can be destroyed with the `Destroy` function:

```
Patcher::Destroy();
```

Before the singleton **Patcher** is created, or after it is destroyed, an error will occur if you attempt to make any **Patcher** calls.

Version Check and Patch Discovery

Your game must use a version number to test whether a patch is available. Any mismatch in version number indicates a patch is required, so choice of version number is up to your game. Normally, a sequence of increasing numbers would be used, but that is not required.

Use the version number to query the network as to whether your game needs to be patched:

```
p->BeginVersionCheck(VDF_HOST_NAME, VDF_PATH, version);
```

Where `VDF_HOST_NAME` indicates the http host on which the version description file is stored (e.g., “mygame.patch.eidos.com”), `VDF_PATH` is the full path on that host (e.g., “/updates/v2/version.txt”), and `version` indicates the version number of the running application (or subsystem if multiple patches are being utilized).

When `TestCompletion` indicates the check is done, a size of zero for the required buffer size (`info.reqBufSize`) indicates that there is no patch needed, and a positive size indicates you must download a patch. For Windows, the size is unimportant since you will generally just hand the patching work to the **WinUpdater** application via the `LaunchPatcherApp` call:

```
Patcher::CompletionInfo info;
if (p->TestCompletion(info) == Patcher::PATCH_STATUS_OK)
{
    if (info.reqBufSize)
    {
        // A patch is needed. Get the description, and query the
        // user to see if he wants to update
        [...]
    }
}
```

Determining Whether a Patch is Mandatory

When the `CompletionInfo` structure indicates a patch is available with a positive `info.reqBufSize`, it will also contain a flag indicating whether this patch is mandatory or not:

```
if (info.reqBufSize)
{
    // Display a different user message depending on whether the
    // patch is mandatory
    if (info.mandatory)
    {
        // Let user know the patch is required to continue...
    }
}
```

If a value for whether the patch is mandatory is not specified in the version description file, it is indicated as mandatory by default.

The actual behavior of the game and interpretation of the mandatory flag is left up to the game, but in general a patch flagged as mandatory would trigger the game to indicate to the user that they must download and install the patch to continue.

A list of non-mandatory patches could also be presented to the user to allow the user to select which

ones to install.

Displaying a Description of the Patch

If a patch is available, you can download a description of it from the server if one has been specified in the version description file. Because the description is downloaded from the host serving the patch, it is downloaded asynchronously:

```
p->DownloadDescription(descbuf, bufsize);
```

Use `TestCompletion` as was done after calling `BeginVersionCheck` to wait until the description is downloaded. Because the description is a file, it can be in whatever format your application desires (plain text would be usual). If the entire description does not fit in your buffer, it is truncated at `bufsize-1` bytes.

Once you have a description, you can use that in your user interface to let the user know what is in the patch, and to allow them to select optional patches.

Downloading and Applying a Game Patch

A Windows game will frequently update its own executable, presenting a little problem; the executable cannot be overwritten while it is running. To get around this problem and to help standardize the patching process, a separate application can be launched to handle the patching of your game. It is started with the `LaunchPatcherApp` function call:

```
bool LaunchPatcherApp(const char* host,
                     const char* hostPath,
                     const char* dirToPatch,
                     const char* updateApp = NULL);
```

`LaunchPatcherApp` can take as few as three arguments: a host name on which to get the version information (the same name you passed to `Patcher::BeginVersionCheck`), the full path name of the version information (also the same as given to `Patcher::BeginVersionCheck`), and the full path of the directory that needs to be patched. This directory is normally where your application is installed.

Additionally, a fourth argument can be passed in: `updateApp` is the path of the updater application (including file name). If `updateApp` is `NULL`, `LaunchPatcherApp` looks in the current working directory for **WinUpdater.exe**.

If `LaunchPatcherApp` returns `true`, then you need to immediately exit your application so that it can be patched. Upon successful completion of the patching process, your application will be relaunched.

If `LaunchPatcherApp` returns `false`, that would indicate that there was a problem launching the **WinUpdater** application, which may indicate an installation problem, and you should inform the user.

Note that most of the time using `LaunchPatcherApp` instead of manually downloading and applying your patches (described below) is recommended.

Downloading and Preparing Asset Patches

If your application wishes to download and apply patches to data files and these patches will not be patching your executable(s), you can use the **Patcher** class to download the patches, and the **FilePatcher** class to apply them patch to the data files. This technique involves more work on your part

(and provides more opportunities for bugs and quality assurance issues), but does not require the relaunching of your game for each patch, as would happen if you use `LaunchPatcherApp`.

The patch itself is downloaded into memory much like the patch description, once you have the required buffer of `info.reqBufSize` bytes:

```
p->DownloadPatch(buffer, bufferSize);
```

As with checking the version and downloading the description, you use `TestCompletion` to wait until the patch is fully downloaded.

After a patch has been downloaded to your buffer, you need to prepare it for use (on some platforms this entails decrypting the patch):

```
p->PreparePatch();
```

When `TestCompletion` indicates it is finished, you will have a decrypted patch in memory ready for use by the patch extractor. Remember to save the size of the decrypted patch (`info.clearSize`) to initialize the **FilePatcher** (or **PatchExtractor**) object.

Applying Asset Patches

If you have downloaded asset patches manually using `Patcher::DownloadPatch` and `Patcher::PreparePatch`, you will then need to apply them to your data. The easiest way to apply them under Windows is to use the **FilePatcher** class.

Initializing the FilePatcher

To apply a patch, you need to instantiate a **FilePatcher** for each patch buffer, and verify that it is good:

```
FilePatcher patcher(dirToPatch, buf, bufsize);  
if (!patcher.IsGood()) {  
    // Abort the patching; something is wrong with the data
```

The verification of the patch makes sure that it is a valid format and that the checksums match the correct values.

The constructor for **FilePatcher** (or the `Init` routine if you are reusing a **FilePatcher**) takes three arguments. The `dirToPatch` argument is a string specifying the directory on your system where the patch is to be applied; this is normally the directory in which your game is installed. The `buf` argument is the buffer into which you downloaded the patch, and the `bufsize` argument is the size of the valid data in that buffer as retrieved from `info.clearSize` after preparing the patch.

Note that `buf` must remain a valid pointer as long as you are using the **FilePatcher**, and that `bufsize` must be exact in order to correctly compute the required checksums.

Patching the Files

A **FilePatcher** works very simply by applying a patch one file at a time:


```
int stat;
std::string filename;

do {
    stat = patcher.FindNextFile(filename);
    if (stat > 0)
        stat = patcher.PatchCurrentFile();
} while (stat > 0);
```

Note that the name of each file to be patched is passed back as that file is found in the patch, thus enabling you to display progress information. You can also retrieve the total number of files to be patched ahead of file with the `GetFileCount` method.

Once you have finished patching all the files in a patch (a value of zero is returned from `FindNextFile` or `PatchCurrentFile`), you are finished with the **FilePatcher**. It can be deleted, go out of scope, or be reused (with the `Init` method) for another patch.