

# **Imagen Cache**

## **for Windows/Linux**

### **Module documentation**

*June 2, 2001. Document Revision 5*

---

Netclime Inc.

P.O.Box 251666, LA, CA-90025  
Tel (310) 571-3135 Fax (646) 514-0412

email: [imagen@netclime.com](mailto:imagen@netclime.com)

# Overview

This document helps understanding the Imagen Cache implementation, and provides instructions how to build its source files.

# Organization

## General Notes

Imagen Cache is an object-oriented program developed in C++. The code can be built under both Windows and Linux.

## Files

Here are the description of the main Imagen Cache modules:

Module	Description
array	Simple self-resizing array.
config	Config file parser.
cache_config	Imagen Cache configuration holder.
cache_mgr	Cache files management.
executable	Module for executing command and getting its output.
lib_executable	Wrapper of the Cache Plugin API.
lock	Exclusive/Shared file-based lock system.
semaphore	Operations with semaphores.
query	Input query parser.
stats	Statistics management.
strip	Smart cleanup.
action	Common actions used in both main and mod_imagen_cache.
mod_imagen_cache	Module for Apache.
extension	ISAPI extension externals
main	Main for the standalone application.

# Developing Cache Plugins

A Cache Plugin is just a dynamic library, which is specified by the *executable* variable in the Cache configuration file.

## Cache Plugin API

The library should conform the following Cache Plugin API in order to be used from Cache:

- ◆ `bool icp_init(const char *arg)`: Called to pass additional text-based arguments to the plugin (configuration filename for example). The arguments are specified through the `lib_arg` command in the Cache configuration file. Not used under Windows.
- ◆ `bool ipc_handler(const char *params, char *&output_data, unsigned &output_size, void *&request_data)`: Called when there is no cache entry for these cgi parameters. Should return the appropriate content and its size. `request_data` parameter is used by the `ipc_free_request` method below. The return value should be true if the output content for these parameters is successfully composed.
- ◆ `void ipc_free_request(void *request_data)`: Called after each successful `ipc_handler` call. Should free the resources allocated for the specified request.
- ◆ `bool ipc_command(const char *cmd, char **message)`: Used for sending commands to the plugin. Returns true if the command is recognized and processed successfully. If succeeded, the function may return a text message to display to the user. By default, `*message` is 0, which force the Cache to display standard message about the command processing. The plugin allocates the memory for the message by operator `new[]`, and the Cache frees it by `delete[]`. The commands are specified through the `lib_cmd` web argument of Cache (please refer to the *Imagen Cache Usage* document for details).

## Library Requirements

There are some requirements and restrictions for the Cache plugins, which should be kept when developing and building the libraries:

- ◆ The library file extension should be `.dll` under Windows, or `.so` under Linux.
- ◆ The library should work correctly under multi-thread environment under Windows, or multi-process environment under Linux.
- ◆ Under Linux, the `ipc_xyz` functions should be declared as *extern "C"*.

# Implementation Details

## Ideas

Imagen Cache calls Imagen to generate the cache data. The query is passed to the Imagen as is.

The Imagen output is saved in a file for each query. The files are located in 38x38 directories, in order to avoid too many files in just one folder. Choosing the directory for a query file is based on two hash functions as described in the *Algorithms* section.

There is a limit for the maximum cache directory size. When the limit is reached, a smart cleanup is started in background to remove files for the cache entries with oldest hit.

Imagen Cache is configurable through an outside text file. Please refer to the *Imagen Cache Usage Guide* for a complete description of the configuration file.

Text trace log file is supported in order to be able to track down Imagen Cache actions.

## **General Flow**

Imagen Cache can be used in 3 different ways:

### **CGI Standalone Application**

First the query is get and checked for special Cache commands ('cleanup' for example). If this is a Cache command, it is executed and an appropriate output is outputted to the browser. Otherwise (this is an Imagen query), the filename for the respective cache entry is composed. If this file exists [cache hit], its content is returned to the browser. If the file does not exist [cache miss], the Imagen executable is called for the same query. The output is collected, stored in the cache entry file, and outputted to the browser.

### **Command-Line Application**

Command-line behaviour is just like the web one, just the query is expected as an argument.

### **Apache Module**

The Cache configuration file is got from the Apache configuration. The file is parsed only once on the Apache startup. The same steps above are performed for each request too.

### **ISAPI Extension**

The Cache configuration is parsed only once when the DLL is loaded. All operations performed for each request are reentrant because the callback can be called in different threads simultaneously.

## **Algorithms**

### **Cache Entry Filename Composition**

The filename format is "cache\_root/dir1/dir2/formatted\_query", where:

- ◆◆ cache\_root is the cache files root folder, as specified in the configuration
- ◆◆ dir1 and dir2 are one-symbol directory names, generated by simple hash functions, based on the query (in the cache\_mgr module)
- ◆◆ formatted\_query is the query itself with all special characters escaped this way: "\_XX", where XX is the hex-code of the char).

Cache files are separated in several directories in order to avoid too many files in just one folder.

### **Smart Cleanup**

When the smart cleanup is started, it browses all cache files, sorts them by access time, and removes the oldest which total size is the amount specified in the Cache configuration. Keeping the whole files list in the memory is not a good idea, so just the oldest files are kept. When a new file is found, it is inserted at the appropriate place in the list, keeping it

sorted. Newest files are removed from the list only if after that the total list size will be still greater than the size scheduled to free. This avoids incorrect files choice [otherwise the algorithm goes as Greedy].

After the files are chosen, they are deleted. The disk space freed should not be greater than the size specified in the configuration.

### Global Synchronizations

Global synchronization is implemented by the lock module. It allows both exclusive & shared lock. Unfortunately non-blocking lock is available under Linux only. The lock module is based on file-locking. Files are named according to a given lock string and are stored in a directory specified in the Cache configuration.

The global synchronizations are used for locking the whole database. Exclusive lock is wanted when deleting multiple cache files. Shared lock is used when add/get cache file.

Synchronization is also used to prevent starting two copies of the smart cleanup. Under Linux the lock module is used. Having no non-blocking locking under Windows, mutex is used there.

### Cache Entry Synchronization

The following algorithm is used when accessing the file for a specified cache entry (in the **get\_entry** method of the Cache Manager module):

1. Try to open the file for reading, locking it shared
2. If failed:
  - a-3. Try to create the file, locking it exclusive
  - b-4. If failed, create the appropriate sub-directories, and try again
  - e-5. If creation failed because the file exists, open the file for reading, locking it shared, and go to 3; otherwise return error
  - d-6. If the creation succeeded, leave the file opened and wait for setting its content (by the **set\_entry** method)
  - 3-7. [Linux] While the filesize is 0, unlock & sleep for a second. This is needed because the file may be opened for reading successfully in the middle of the "create – lock exclusive" sequence in another process. Under Windows, open & lock is performed in just one function, so this problem does not exist.
  - 4-8. The file content is retrieved
  - 5-9. Close and unlock

### Executing Imagen

CreateProcess() is used in the ISAPI extension, popen() otherwise.

Imagen takes many system resources while executing, so max count of simultaneously running Imagen copies can be specified in the configuration. The protection is implemented by semaphores in all Cache versions.

## **Unresolved Issues**

- ◆ Paths convention in the configuration file
- ◆ Per-directory configuration files

## **References**

- [1] Readme
- [2] Install
- [3] Config