Multi-Threaded File Processor Documentation

June 29th, 2016

David Cohn

Contents

[1 Setting up the MTFP 2](#_Toc454976339)

[1.1 Configuration File, Project Script and Project Jar Location 2](#_Toc454976340)

[1.1.1 Obtaining the Project Jar 2](#_Toc454976341)

[1.1.2 Obtaining the Scripts 2](#_Toc454976342)

[1.2 Configuration File Structure 2](#_Toc454976343)

[1.3 Preparing the Target Directory 3](#_Toc454976344)

[1.4 Files/Folders that the MTFP will Create 3](#_Toc454976345)

[1.4.1 pid.txt files 3](#_Toc454976346)

[1.4.1 Files that Store Metrics 3](#_Toc454976347)

[2 Capabilities of the MTFP 4](#_Toc454976348)

[2.1 What does the MTFP do? 4](#_Toc454976349)

[2.2 Detecting Files 4](#_Toc454976350)

[2.3 Logging 4](#_Toc454976351)

[2.4 Exception Logging 4](#_Toc454976352)

[3 Running the MTFP 4](#_Toc454976353)

[3.1 Starting the MTFP (in processing mode) 4](#_Toc454976354)

[3.2 Terminate the MTFP (during processing mode) 5](#_Toc454976355)

[3.3 Run Performance Measurement Tools 5](#_Toc454976356)

[3.4 Changing Program Settings 5](#_Toc454976357)

[3.5 Optimal Number of Threads 5](#_Toc454976358)

[3.6 Program Speed 6](#_Toc454976359)

[3.7 Deadlock 6](#_Toc454976360)

[4 Frequent Errors and Troubleshooting 6](#_Toc454976361)

[4.1 Errors which Appear in the Console 6](#_Toc454976362)

[4.1.1 “Config file missing.” 6](#_Toc454976363)

[4.1.2 “Config file not properly formatted” or “Config file missing header”. 6](#_Toc454976364)

[4.1.3 “Program initialization failed.” 6](#_Toc454976365)

[4.1.4 “Error re-initializing config file.” 6](#_Toc454976366)

[4.1.5 “Error finding/creating log file.” 6](#_Toc454976367)

[4.1.6 “Processor type (e.g. SABRE/APOLLO/AMADEUS) not recognized.” 7](#_Toc454976368)

[4.2 Errors which Appear in the Log File 7](#_Toc454976369)

[4.2.1 “Processing of file [file name] failed at [timestamp]” 7](#_Toc454976370)

[4.3 Errors which Appear in the Exception Log File 7](#_Toc454976371)

[4.3.1 All System Errors 7](#_Toc454976372)

# 1 Setting up the MTFP

## 1.1 Configuration File, Project Script and Project Jar Location

In order to function properly, the configuration folder (Config, which contains AMADEUS/APOLLO/SABREconfig.txt), the five project scripts (configMTFPSubmit.sh, mpMTFPSubmit.sh, sabreMTFPSubmit.sh, amadeusMTFPSubmit.sh and apolloMTFPSubmit.sh), and the project jar (MultiThreadedFileProcessor.jar) must all be located in the same folder.

### Obtaining the Project Jar

The source code can be downloaded from the repository as two separate projects: JavaApps/GIRRaw Loader, and JavaApps/MultiThreadedFileProcessor. Run a Maven clean on both projects (right-click on the pom.xml file to see this option), and then build the GIRRaw Loader, followed by the MultiThreadedFileProcessor. The project jar can then be found in the target folder of the MultiThreadedFileProcessor project. Note that it will need to be renamed to “MultiThreadedFileProcessor.jar” in order for the scripts to work.

### Obtaining the Scripts

The scripts can be found in the repository at BatchScripts/common/src/David.

## 1.2 Configuration File Structure

The configuration file must be laid out as follows: the first line reads “CONFIG” (no quotes), the next group of lines is introduced by “Source Directories: “ and then lists all the directories which the MTFP will monitor on subsequent lines. The line after this must begin with “Target Directory: “ and indicates where output files will be placed. Following this are the Log Directory and Exception Log Directory lines, and then the number of threads which the program will use. The final two lines detail what server and port the program should try to connect to when it needs to generate a sessionID. An example is provided below (ignore the double-spacing):

*CONFIG*

*Source Directories: /home/la0199/MTFP/Source*

*/home/la0199/MTFP/Source2*

*Target Directory: /home/la0199/MTFP/TestFiles*

*Log Directory: /home/la0199/MTFP/Logs*

*Exception Log Directory: /home/la0199/MTFP/Exception Logs*

*Number of Threads: 8*

*Session ID Server: 10.213.108.21*

*Session ID Port: 11820*

## 1.3 Preparing the Target Directory

Any .processed files which are present in the target directory when the program begins will be considered as already processed by the program, and placing an un-processed version of that file in a source directory will cause the program to delete the un-processed version, not copy it and update the processed version. For example, if the target directory contains the file *example.txt.processed*, which was generated from a previous execution of the program, and during a future execution the file *example.txt* is added to a source directory, *example.txt* will simply be deleted, as the program has already processed it. Thus if it is desired that the program will freshly process every file given to it as input, the target directory should be emptied before the program is run. This can be done either manually, or using the “clean target” command outlined in section 3.4.

## 1.4 Files/Folders that the MTFP will Create

### 1.4.1 pid.txt files

During regular program execution (i.e. as soon as sabre/amadeus/apolloMTFPSubmit.sh is run), a file SABRE/AMADEUS/APOLLO*pid.txt* will be created in the directory from which that script was run. This file contains the process id of the program, so that it can be terminated automatically when the script is next run.

### 1.4.1 Files that Store Metrics

Whenever mpMTFPSubmit.sh is run and metrics are generated, they will be stored inside the Metrics folder, which is located inside the log directory (specified in the config file). Inside this folder are two text files, *AvgProcTimeResults.txt* (to which the results of any “average processing time” command are added), and *NumFilesResults.txt* (to which the results of any “number of files” command are added). Another folder, List of Processed Files, is also located here. Any time that the “list files” command is executed, a file with the user-specified time-frame as its title is generated in this folder, with the body of the file containing the name of every file processed during the period. See section 3.3 for more information on generating performance metrics.

# 2 Capabilities of the MTFP

## 2.1 What does the MTFP do?

The MTFP will continuously monitor a specified directory or directories and detect any and all files that are added to it. It will then copy these files and move them into a specified target directory. The MTFP also contains a currently blank processLogic(File) method, which can be filled with code which will execute on the file before it is copied. The MTFP can support files that need SABER, AMADEUS or APOLLO processing, and thus multiple instances of the program are designed to be run simultaneously, with each instance doing one kind of processing. Files processed into this target directory are given a “.processed” file extension. The MTFP uses a thread pool with configurable size and a LinkedBlockingQueue to process files in parallel. The project accepts input via the active console while it is running, and logs all of its work, using log4j, in a configurable location.

## 2.2 Detecting Files

The MTFP will detect any kind of text file that is placed within the directories that it is monitoring, regardless of how many folders “deep” the file is. When a file is first detected, it will be given an “.input” file extension.

## 2.3 Logging

The MTFP uses log4j, specifically a DailyRollingFileAppender, to create a new log file and a new exception log file every day. Wheneve the program is first started, the main log file will have the date of the initialization logged. Following this, two types of log lines are created for each file which is processed: one for when the processing begins and one for when it finishes. Both lines contain the filepath and a timestamp, and the finished line also contains the total time spent on processing of that file. At the end of the log are listed the time at which the program was terminated, the total number of files processed during this execution of the program, the total time that the program was running, the total amount of active processing time that threads spent on files, and the average amount of time needed to process one file. In the event that a file cannot be processed, this will be logged both in the standard log and the exception log (see section 2.4, below).

## 2.4 Exception Logging

Any application exceptions which occur during runtime will be caught and reported in the exception log. If the issue is with processing a file, the file will be renamed to have a .failed extension. Program execution will continue with the next file. Any system exceptions which occur during runtime will be reported in the exception log and program execution will be terminated after clean-up occurs.

# 3 Running the MTFP

## 3.1 Starting the MTFP (in processing mode)

An instance of the MTFP can be started by running the command “bash [sabre/apollo/amadeus]MTFPsubmit.sh” from the folder which contains all of the program files (see section 1.1).

## 3.2 Terminate the MTFP (during processing mode)

Whenever an instance of the MTFP is started, any existing instance of the same processing type will automatically be terminated. An instance can also be safely manually terminated by simply killing the process – the process id can be found in the SABRE/APOLLO/AMADEUSpid.txt file.

## 3.3 Run Performance Measurement Tools

Start the MTFP in performance measurement mode by running the command “bash mpMTFPSubmit.sh” from the folder which contains all of the program files (see section 1.1). The user will be prompted to enter a start time and an end time, between which the program’s performance will be measured. A time can be entered in the format yyyy mm dd hh:mm:ss, or there are two special options. Entering *now* will record the current time, and entering *–x hours* will record a time exactly x hours ago, where x may be an integer or a double.

Following this, the user will be prompted for a metric. Acceptable options are “list files” (displays the name of every file processed during the time period), “number of files” (displays the total number of files processed during the time period), “average processing time” (displays the average time it took to process one file during the time period), “all” (displays the results of the above three metrics), and “cancel” (ends the program).

## 3.4 Changing Program Settings

While program settings can be changed manually by altering one or more of the config files, in order to avoid difficult-to-detect spacing errors, it is strongly advised to use the built-in feature to change settings.

Start the MTFP in configuration mode by running the command “bash configMTFP.sh Z”, where Z is either “sabre”, “apollo” or “amadeus”, from the folder which contains all of the program files (see section 1.1). Thus only the corresponding config file will be edited. There will be no prompt after this, simply a blinking cursor. The commands “add source X”, “set target X”, “set log X” and “set exception log X”, where X is the path of a valid directory, will, respectively, add a directory to the list of those that the program monitors, set the output folder, set the log folder, and set the exception log folder. The “set threads X” command, where X is a positive integer, will set the number of threads which the program will use when started in processing mode. All of these changes will be reflected in the updated config file which is generated when the program terminates.

Furthermore, the “get sources”, “get target”, “get log”, “get exception log” and “get threads” commands can be used to check the current settings of the program. Finally, the “clean sources” command will delete the contents of all source directories, and the “clean target” command will delete the contents of the target directory. Both of these commands require confirmation, which is given by entering “Y” when prompted. When the user is finished making modifications, enter “done” and the program will automatically terminate.

## 3.5 Optimal Number of Threads

Generally speaking, the optimal number of threads will be roughly one thread per active processor, perhaps slightly more. For instance, on the standard issue HP laptops which this program was created on, the optimal number of threads was 4-5. Increasing the number of threads beyond this tends to slow down the program, as there is overhead involved with creating each thread and assigning tasks to it. However, things seem to be different on the Linux server: there about 8 threads (and possibly more) can be run simultaneously with minimal noticeable slowdown in the processing time of each file. For more information (in general) on the optimal size of a thread pool, see <http://math.hws.edu/javanotes/c12/s3.html>.

## 3.6 Program Speed

Program speed is of course dependent on many factors, including the system that the program is executed on, as well as the size of the files which need to be processed. However, as a baseline measurement, the program requires roughly 4.5 to 5 seconds of processing time to process a single short text file. Since the computer on which this test was run has eight processors, this equates to about 0.5 to 0.6 seconds of absolute time per file, which means that about 1.5 to 2 files can be processed every second, equating to around 155,000 files per 24-hour window.

## 3.7 Deadlock

Deadlock should never occur in this program, due to its use of a LinkedBlockingQueue, but if the program must be terminated, killing the process from the linux terminal is a safe way to do it that should always work. The current process id can be found in the file named [SABRE/AMADEUS/APOLLO]pid.txt. Failing this, running the command “pkill java” should work. If for some reason this fails, then opening cmd and entering “taskkill /f /im java.exe” should end the program and all active threads. Note however that using a forceful taskkill will prevent the program from executing its shutdown hook, which means that the config file may not reinitialize properly (and will need to be fixed manually).

# 4 Frequent Errors and Troubleshooting

## 4.1 Errors which Appear in the Console

### 4.1.1 “Config file missing.”

The config.txt file is not in the proper location. See section 1.1.

### 4.1.2 “Config file not properly formatted” or “Config file missing header”.

The config.txt file has not been properly formatted, and will have to be fixed by hand. See section 1.2

### 4.1.3 “Program initialization failed.”

Will always be accompanied by one of the above two errors. Remedying that error will also fix this one.

### 4.1.4 “Error re-initializing config file.”

Indicates that there was some error in the program’s automatic updating of the config.txt file. The config file will have to be re-formatted and fixed by hand before the program is next run. See section 1.2 for guidance. Note that this error does not mean that the current execution of the program failed, only that future executions will fail unless action is taken.

### 4.1.5 “Error finding/creating log file.”

This is probably due to a config file error: check that the log directory listed there is valid.

### 4.1.6 “Processor type (e.g. SABRE/APOLLO/AMADEUS) not recognized.”

This indicates that an improper command line argument was passed to the program when it was run – check the script that was called to launch the program and make sure that it passes two arguments: first “start”, and then one of “sabre” “apollo” or “amadeus”.

## 4.2 Errors which Appear in the Log File

### 4.2.1 “Processing of file [file name] failed at [timestamp]”

For any one of many possible reasons (the file does not exist, another process had the file open etc), the indicated file could not be processed properly. The exception or error that caused this error will be logged in the corresponding exception log file. This error does not affect the successful processing of all other files.

## 4.3 Errors which Appear in the Exception Log File

### 4.3.1 All System Errors

Any system errors which occur during the program’s execution time (e.g. StackOverflowError, out of memory error, no disk space etc.) will be logged, along with their full stack trace, in the corresponding exception log file. These errors will also cause the program to terminate.

Remedying these errors is dependent on the type of error and the circumstances which caused it to arise: by their nature, system errors have little to do with application logic, and are likely due to some issue with the system the application is run on.