Politecnico di Milano

Dipartimento Elettronica, Informazione e Bioingegneria

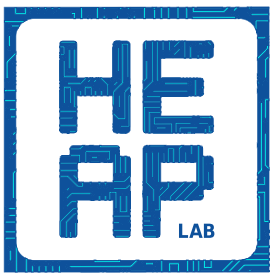
HEAPLab Project Report

TDMH Visualizer

Author:
Francesco Franzini

Supervisor:
Dr. Federico Terraneo

January 8, 2020



Abstract

A Qt viewer for tdmh logs.

1 Introduction

The goal of this project was to develop a Qt based GUI designed to ease reading of the logs generated by *tdmh*. This is done by enabling the user to select a specific line on the log and by displaying a graph showing the active adjacencies at that point in time.

2 Design and Implementation

2.1 Introduction

The code was developed by focusing on the scrolling performance above anything else. The log file is first parsed in its entirety in order to build a fast data structure in which the search of a particular line is logarithmic. This means that, while loading a large file might take some seconds, scrubbing through millions of lines is instantaneous and results in no user-detectable lag.

2.2 Data Structure

Since the lines in the log that have meaning (e.g. *"[U] Topo ..."*) are a very small percentage of the total log, it makes sense to save them in a separate data structure in order to have faster search. The process of building the graphical graph given a line number can be described as the problem of finding for each node the last topology entry which has been written before said line number. The fastest way to implement this is using a vector for each node, each containing the topology entries ordered by increasing line number. If N is the number of nodes and L the number of lines in the log file, N binary searches are needed, so the final result is O(N*log(L)).

2.3 Modes of Operation

The program features three modes of operation: Batch, Real Time and Statistic. Batch loads the given log file and then terminates. This is useful if the

log is not being actively written into.

Real Time instead invokes the same code that Batch uses, but then doesn't close the file and keeps polling for new writes to it.

Statistic mode calculates the adjacency statistics, indicating for each link the percentage of time and/or schedules in which it was up. On the graph each link is a dotted line with dots that are sparser the weaker the link was.

2.4 Qt Implementation

The GUI requirements were that the log should be displayed "text editor style" on the left while the graph is displayed on the right and is updated each time a different line is selected. Since a log file can be very large, the Qt "editor" objects that are provided in the library (*QPlainTextEdit,..*) are not suitable since they behave poorly with millions of lines and store the file as *QString*, which uses UTF16 and increases the memory usage significantly. For these reasons, the Model-View approach provided by Qt has been used. The view extends *QListView* just to add some graphical features, while the model is built to store the log as $std::string_{\mathrm{s}}$, so that the size is kept under control. Since *QListView* still requires $QString_{\mathrm{s}}$ the idea is to build them on demand when the view fetches the visible lines from the model. Since loading can take several seconds for large files, a "fetch on demand" mode has been provided, so that the user can choose to load more lines in the view only when the bottom of the editor is reached rather than to wait for the whole log to be loaded. Loading the lines in the view in background is not possible due to the way in which *QListView* is implemented, as each time a new line is inserted the whole index is scanned, and this results in flickering and stuttering when the file is millions of lines long. For the users convenience, a *jump* hotkey has been added (Ctrl+F), allowing to jump to a specific line in the log.

For the graph, the *QGraphicsView* environment has been used since it provides all the required methods and objects to draw the graph on the map and to modify it as needed in addition to zooming and scrolling. Since the maximum number of connections is $\frac{N*(N-1)}{2}$, performance is not an issue when having to modify the lines on screen.

2.5 Configuration File

The configuration file is a simple text file that contains newline separated lines and is passed as the first argument to the program. Invalid lines or unrecognized commands are ignored. Specifying if the weak mask is present or not is not required. Lines can be commented by leading them with a #. A list of recognized options follows:

- MODE=<value> selects the mode. Possible values are RTIME,BATCH or STAT. Defaults to Batch.
- BATCHFIRST if present makes the program load all the log in the beginning.
- LOGFILE=<path to log> tells where to find the log. Defaults to log.txt.
- IMAGE=<path to image> tells where to find the image to display in background. Defaults to image.jpg.
- NODELIST=(0,0,0)(1,300,300)... tells where the nodes are located on the given image. Format is (index,heightpixel,widthpixel) and (x,0,0) is the top left corner.

2.6 Possible extensions

The code was written by keeping in mind future support for new modes or lines to keep track of. The editor part of the software receives the lines through a consumer/producer pattern, so anything can be put in it without it having knowledge of what it represents. An example of this is the Stat mode, where the same methods are used to insert lines that are not part of a log, without any need to alter the text editor. When a line is selected, the signal/slot framework of Qt is used, so anything can be linked to it. As for the Graph GUI, it takes information from the *LogContainer* object, so to add new types of information it is sufficient to add the relative getter methods to that class.

3 Experimental Results

Testing the software both in Debug and Release mode with big log files (∼600MB - 20M lines) has shown that scrolling doesn't suffer from size at all. Loading times for that file were in the neighborhood of 11 seconds on a laptop.

4 Conclusions

In conclusion the software developed for this project is able to process logs of considerable size without drops in scrolling performance, meaning that the user experience should be smooth and responsive as required. The use of the model/view classes means that in the future the code should be easy to adapt to new types of information or log.