Yet Another Fast Log (YAFL)

Abstract attached to the poster

Introduction

Background

Outputting messages as a way to know what your code does is an old technique. Developers uses these messages to report about anything they seems to need later when the program runs or as information for post mortem analysis.

The problem

Many programs reached a point that the messaging system itself requires runtime resources such that the program either does its work ***or*** reports about it, but not both. This is also known as the ‘work/report problem’.

Other Solutions

Reduced Messages

Solving this problem is usually done by *reducing* the number of messages being produced by the application by adding “severity level”, “module” and debug/release modes to each message to control those messages that are actually produced. Controlling the output messages is done either at compile time or at runtime or both.

This approach however always results in *less* information than the developer’s initial intent. That, in turn, results in longer time to detect problems. Moreover, the program *behaves* differently as a result of the number of messages being produced. This fact makes it even harder to detect problems by ‘opening’ those messages that help finding the problem. All in all, reducing the number of messages, though it is the most commonly used methodology has its problems.

Binary Messages

Another approach to the messaging problem is to define a priori binary structures that are much faster to produce at run time and can be easily translated to text afterwards. However, these solutions require a lot of work by the developer to prepare the ground for the messaging system itself. The developer ‘works for’ the logging system and not vice versa.

The YAFL Solution

The work presented in YAFL takes the binary solution one step ahead and tries as much as it can to make the work of the developer to be easy as possible. YAFL does include severity

and modules in its messages but not aimed to reduce the number of messages but rather to allow *filtering* and *searching* of messages in the logs.

Relevance

The work is relevance to any application that suffers from the symptoms described in the introduction.

Embedded systems might find this technique relevance as it uses binary outputs, its required space is known at compile time and it is fast.

Intense server applications might find it useful too. The technique was actually developed to solve the work/report issue on a server application.

Another reason that YAFL might be relevance today is its intensive work with CMake to build the entire project. The build system is an integral and crucial part of the project and it wouldn't be possible to ease the developer work without it. The project, even if not used as is, demonstrate CMake's abilities. Disclaimer: the author of YAFL is not a CMake professional, just a user that 'made it happen' to work with CMake. There is a lot that can be done better, faster and clearer with the build system.

Discussion

The poster shows the process of producing a log message: from writing it in the code, via its preprocessing, its compilation, the binary output and into translating the binary log into human text.

The poster tries to emphaseyes the important parts of the process without getting into details, github is for this. ***Every*** aspect of YAFL can be displayed in 3-4 posters and can be long discussed. There is so much to dive into each part.

Here is a list of the issues that are ***not*** presented in the poster but surely be discussed orally by those who are interested:

- M4, preprocessor code. What problems are faced, how to solve, open issues, options, etc.
- Compiled code
  - Can we build even faster log messages? (yes we can!)
  - Thread queue messaging
  - Constexpr of messages, move, etc.
  - More.
- IDE integration
- Post Processing
  - Why python? (not a limitation at all)
  - Searching, filtering, etc.
- Benchmarks

Completion Status

The project was written to solve the work/report problem for a production server. It reached it proof of concept (POC) phase and then the entire server application was abandoned. Not only YAFL, the entire project took another direction and it is effectively a ‘dead’ project.

YAFL has a github repository that as far as we can see it might be used as a starting point, a kick-off for users that need YAFL solution. It is definitely not out-of-the-box library to download and use. For example, a work must be done to embed YAFL building system into the existing project’s build system. Moreover, since any aspect of YAFL might and should be written differently per the specific’s project requirements, it is likely that it is, indeed, be only the starting point, the evidence that this is doable - that YAFL might be considered as a tool in the set of tools already available in the market.

Supporting Material

Github: **https://github.com/regevran/yafl.git**
Cmake: https://cmake.org/

Biography

My name is Ran Regev.
I live in Israel.
I am the sole author of YAFL.
I work in the industry since 1999, for various companies, mainly using c++.
I worked in both servers and embedded systems.
I worked in the telecommunication and cyber security areas.
Except for a small contribution to TACACS authentication discussion and code, and a few talk-reviews as a PC member for CppCon-2019, this is my first public work.

Contact info

Ran Regev
Email: regev.ran@gmail.com
Twitter: @ran_regev
Linked in: linkedin.com/in/ran-regev-8b57386
Phone : +972-50-3891316
Project: https://github.com/regevran/yafl.git