Università della Svizzera italiana

Faculty of Informatics

Instrumentation and Performance Analysis of Distributed Systems with Freud

Stefano Taillefert

May 2021

Supervised by
Prof. Antonio Carzaniga

BACHELOR PROJECT REPORT

Abstract

Freud [1] is a software performance analysis tool that derives performance annotations from measurements of running systems. The goal of this project is to extend it to instrument and collect data from a distributed software system. This means augmenting the existing implementation to be able to link the data collected over distributed components using causal relations.

Contents

Chapter 1

Introduction

TODO

Chapter 2

Performance analysis

As the name implies, performance analysis is a field that deals with finding out how efficiently your code runs. It's a very vast realm and everyone is in some way in need of it; after all, especially in large companies, efficiency is key. Imagine if each Google query took one second less because someone found out that they could optimize the database query, or if the loading time of a webpage got halved because the webmaster found a function that was holding for no reason: both of those could be the result of a performance analysis study.

As I mentioned, almost every kind of application can be measured, with a vast variety of what we call metrics: execution time, memory or CPU cycles used, lock holding time, and many more. Those can be used in various ways and will be chosen accordingly to the type of application. For example, a single-threaded app will not be concerned about lock holding time, while a frontend web developer isn't necessarily interested in knowing about pagefaults.

There are a lot of existing tools and solutions out there (some of which discussed in section 3.1) that can instrument the executables: a dedicated tool injects some special code directly in the compiled binary file, thanks to some particular compiler flags. This generated code then measures the various parameters, keeping track of starting and ending times, memory allocations, and so on. At the end, an information dump is produced to summarize the data, which can then be analyzed.

Chapter 3

Project design

3.1 Freud

My thesis follows the steps of Freud, a tool developed by Daniele Rogora, an USI PhD student. This software has multiple components; the first two (`freud-dwarf` and `freud-pin`) are tasked with instrumenting the executable via a special tool called PIN [3], developed by Intel. This library - as explained earlier - inserts the instructions necessary to collect the data during the execution. Basically, coding this part means writing code that will write some code to put into other code.

The other part of Freud, which is the one I will be interacting with, is `freud-statistics`. As the name implies, this module (based on the popular R library[1]) computes some statistics such as regression or clustering given the output of the instrumentation.

3.2 Requirements and analysis

The main idea was to start small and build features incrementally, to ensure that we always had a minimal working prototype. The main objective of Jung - a name chosen after Carl Jung[2], a colleague of Sigmund Freud - is to instrument multiple systems, collect the data and format it. To achieve this, I divided the project in three main components:

- The actual instrumentation library, with an API to track the various metrics

- The merger, which has to collect all the logfiles from the different systems and merge them in a single coherent trace

- The dumper, which is tasked with creating the binary output that will be fed to `freud-statistics`

The main objectives of the developments were the following: first I had to develop a simple distributed application, based on an RPC library, to be used as an initial test environment.
Develop an instrumentation for the client side, server side, and – crucially – the RPC library: the idea was to measure some meaningful statistics on all sides involved. The metrics we settled with are:

- Execution time, divided in total, server, and network time

- Memory usage

- Major and minor pagefaults

- Lock holding and waiting time

- Possible memory leaks (experimental)

---

[1] `https://www.r-project.org/`
[2] `https://en.wikipedia.org/wiki/Carl_Jung`

Except for the first one, all the other are divided in client and server side, to allow for an accurate differentation; futhermore, except for memory leaks, all of them are supported by `freud-statistics`.

Then, i had to devise a method to save and retrieve the measurement logs from all the components. This has been done by dumping the collected information in a text file with some special encoding for data types and more complex values.

The next step was to design an algorithm to merge the logs from all the systems into a single coherent trace. This was tricky because I had to correctly identify all the remote calls to allow to trace back which server execution corresponded to which client function. We don't want to credit someone with the costs of someone else...

Integrating said trace in the existing statistics tool (`freud-statistics`) to derive the performance annotations was the last objective for the coding part. For this I had to rely on the help of its creator, Daniele, to fully understand how to format the data. There were a few hiccups due to some technical issues, but the process concluded perfectly.
Unfortunately, the only abandoned part of the project was to identify some third-party non-trivial distributed applications and analyze them with the created tool. Due to the fact that Jung doesn't work on the binary executable, we need to have access to the source code of the application we want to analyze. This heavily reduces the spectrum of possible target, and coupled with the little time left we decided to abandon it.

Last but not least, I had to write the report, prepare the poster and the presentation. Due to the current situation, we as a class decided to not hold the presentations this year, therefore this point got reduced to the document you're reading.

Have a pizza: this is still on hold but will definitely be completed sooner or later. Every project needs a celebratory pizza at the end.
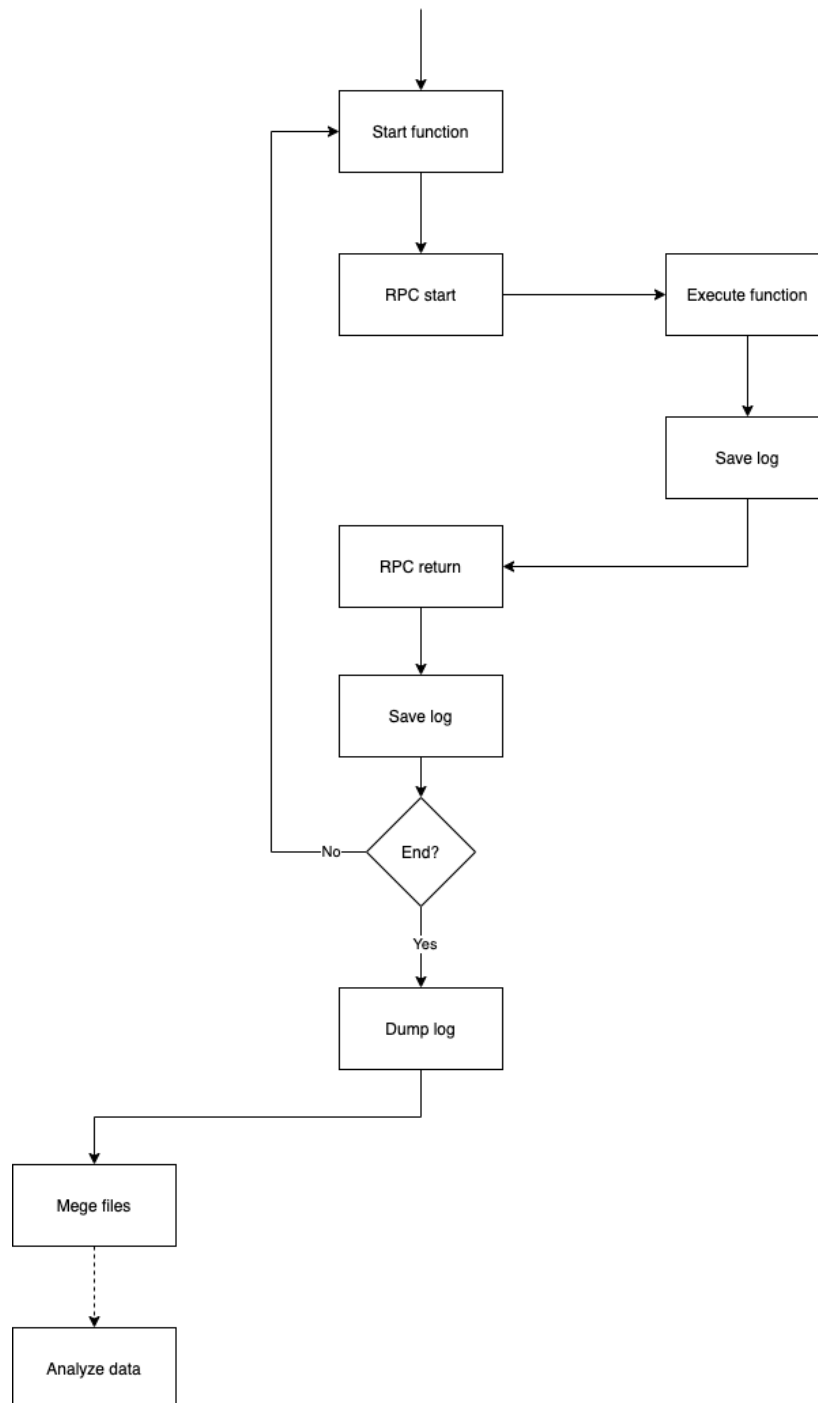
Chapter 4

Implementation

FIGURE 4.1: Jung's general functionment

4.1   Technologies and tools used

I used gRPC [2] as the RPC library, since it seemed pretty straightforward and simple to use. They have plenty of examples in their documentation and support a lot of languages, including C++.

The choice of language was pretty easy as well: Freud is built entirely in C/C++, so using the same language would increase compatibility. The only choice I had to make was regarding the version: I went with C++17 to benefit from the `filesystem::exists()` function, used to check for the existance of previous log-files.

To simplify development and testing, I also packaged an auto-building Docker image (`https://hub.docker.com/repository/docker/steeven9/jung`) with the example server in it. This way I could keep the server running on another machine and, for example, test the network times.
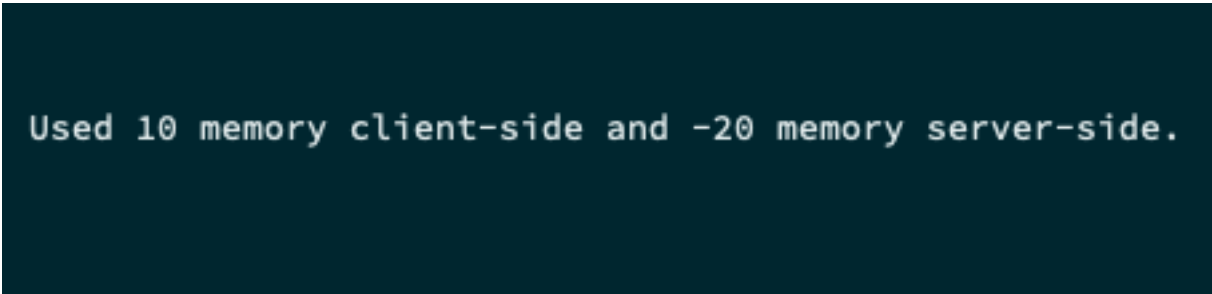
4.2   Issues

The only major problem I had - which I believe to be very common - was that I made some design choices in order to simplify my job initially, but then with the addition of new features or to fix certain problems I had to change approach. The direct consequence was that I had to rewrite some large parts of the code and data structures to adapt to the new needs.

One instance of the aforementioned problem is that I had to develop some particular encodings to represent the parameters, since I had to convert from data (the running code) to text (the logifle) and viceversa. More specifically, I chose to represent the values as `namētype&value` (e.g. `asd̄int&12` for an integer named `asd` of value 12); this added a supplementary layer of encoding/decoding when saving and reading the data to/from the logfile.

When we first started testing the integration with `freud-statistics`, since I was unable to see what was in the binary file, I had troubles checking if I was dumping the metrics correctly, resulting in some weird-looking statistics. With the help of Daniele, we managed to make sense of the output and find the offending code part to patch.

And of course, I had my fair share of miscellaneous issues with pointers, segmentation faults and general weird behaviours. As a positive note, in the process I somehow created a program that frees memory as you run it...

Used 10 memory client-side and -20 memory server-side.

FIGURE 4.2: A very peculiar memory usage

Another problem we ran into was that we were unable to get `freud-statistics` to find a regression despite our data was clearly a good candidate for it. Turns out we simply didn't have a good enough p-value, which was easily fixed by getting more samples, as shown below.

(A) Not enough data point

(B) With enough data points

FIGURE 4.3: The result of our test program

Chapter 5

Evaluation

TODO funcionality, personal experience

5.1 Testing

Due to the complexity and particular nature of the software, we decided to not implement any automated tests. Opposed to test-driven development, I had a more unstructured way of iterating over the code, since the functionality had to be checked by hand comparing the produced logfiles with the program. Therefore, the effective coverage is 0%, but as someone one said:

"Tests don't prove correctness"

5.2 Demo

In this section you can find instruction on how to run a demo program and analyze the obtained data.

To install and compile all the required software, simply run the `install.sh` script from the Jung repository. In case of failure, please refer to the READMEs of each project for more information.

First, start the server:

```
$ ./jung_server
```

While in another shell, run the client:

```
$ ./jung_client
```

Note: if you run the server on another machine, you can pass the `--target=HOSTNAME` argument to the client.

Then, with both the client and server logfiles in the root folder, merge the traces:

```
$ ./trace_merge
```

This will produce the binary data (under `symbols/`) and a summary of the execution. We can then run Freud's analysis tool:

```
$ cd ../freud/freud-statistics
```

```
$ ./freud_statistics 3 0 0 do_stuff ../../Jung/symbols/do_stuff/
```

Note: this is an example, refer to Freud's documentation for details and usage.

This should fine a nice regression and plot it. That's it!

Chapter 6

Conclusions

TODO

6.1   Future work and possible developments

While this project can be considered a success, there are still a number of improvements that could be done. For example a better handling of the data dumping via a separate thread, since now every function writes the buffered metrics when it finished executing. While this is barely noticeable on a small scale, it could add a significative overhead under large loads.

A crucial missing part is an actual instrumentation like Freud's, either with PIN or another library, to allow for direct instrumentation of executables instead of changing the source code.

Another possible improvement would be to support more complex features as parameters, like structs.

Bibliography

[1] GitHub Inc. usi-systems/freud: Freud, a tool to create Performance Annotations for C/C++ programs. `https://github.com/usi-systems/freud`, 2021. Accessed on 15.05.2021.

[2] gRPC Authors. Documentation | gRPC. `https://grpc.io/docs/`, 2021. Accessed on 27.04.2021.

[3] Intel Corporation. Pin - A Dynamic Binary Instrumentation Tool. `https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html`, 2021. Accessed on 22.05.2021.