Software Performance and Scalability

Project Documentation

Fabio Dainese 857661, Matteo Facci 875377
Alessandro Salvagnin 864333, Brahmashwini Regonda 887689

Ca' Foscari, University of Venice

May, 2022

Keywords: *Performance, Compilation, C++, Tsung*

Contents

1 Introduction

The goal of the project discussed in this paper was to implement a web application that allows users to upload a C++ program and get in return the output of the compilation and, if this last one succeeds, then also the output of the execution run in a sandbox environment - for obvious security reasons - and carry out a performance analysis by utilizing different tools to identify the possible problems and/or bottlenecks of the real system and to eventually propose further improvements to the design of the application.

In the following chapters will be presented the design and the implementation of the web application, meanwhile the results of its analysis will be described later on.

2 Design and Implementation

The web application was designed to be deployed as a micro-services architecture by logically separating all the sub-components in distributable and replicable partitions that will be briefly analyzed in the following subsections.

Moreover, it's worth mentioning that, during the initial development phases, some choices regarding the introduction of several practical constraints were made, such as:

- The user can only upload *.CPP* and *.CC* files and they cannot exceed 2MB of file size (these checks are implemented both in the frontend and backend sides);
- The execution time is limited up to 5 seconds. If the program doesn't terminate on time, it will be halted automatically;
- The program cannot exceed 200KB of standard output buffer size - i.e. it cannot print unlimited output;

Keep in mind that these additional limitations have been introduced with the only aim of fairly sharing the available server resources among the users without sacrificing excessively their user experience.

2.1 Frontend

The frontend part of the web application relies on two major libraries, the first one is *ReactJS*, which was used to create custom interactive UI components and logics, and the other one is *Ant Design*, which provides pre-built *React* components straight out of the box.
This minimal setup ensured us a simple and quick development process minimizing all the possible overheads.

Before concluding this subsection, here's a couple of screenshots of the developed *SPA* just to give you a little bit more context:

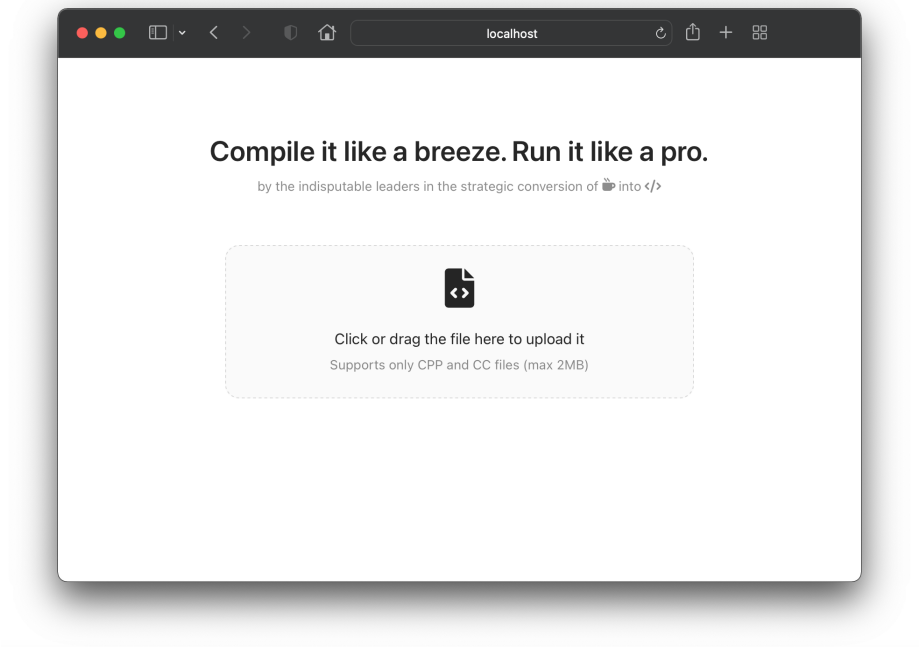


Figure 1: Homepage

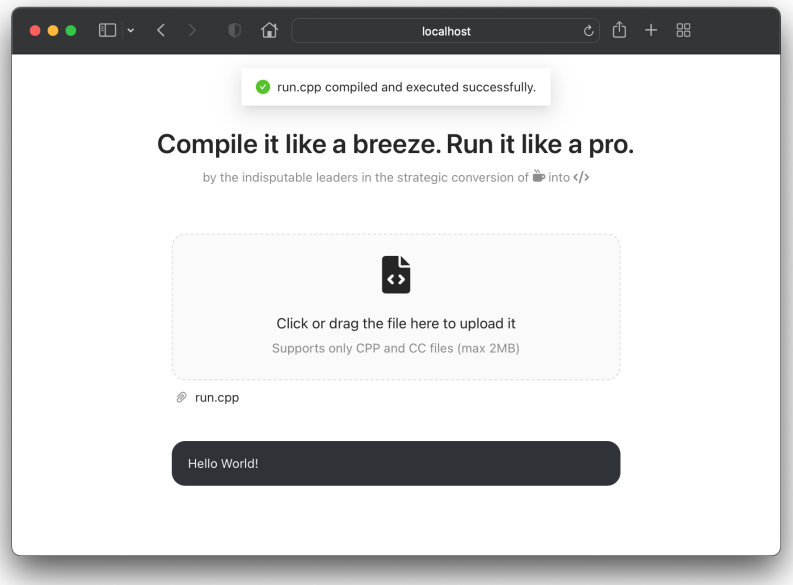


Figure 2: Success Execution Message

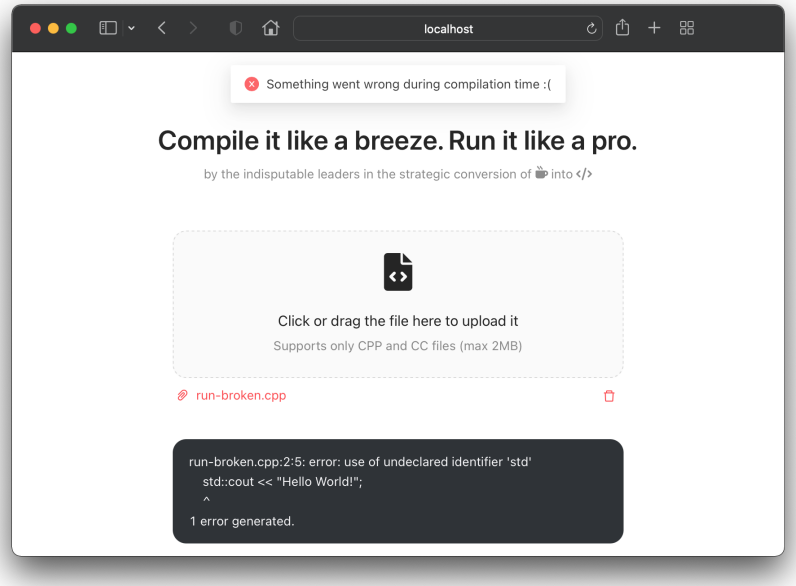


Figure 3: Error Message

2.2 Backend

For what concerns the backend side we decided to use *ExpressJS* - it's a minimal and flexible *NodeJS* web application framework - in conjunction with additional libraries in order to expand and adapt its functionalities to meet our specific project needs.

The ones worth mentioning are *Multer*, an efficient *NodeJS* middleware used for handling multipart/form-data, which in our case manages the upload and the temporary storing of the provided *.CPP* and/or *.CC* file in the server.
Furthermore, since it was required to design a queue for the incoming requests so that, at any given time, no more than N concurrent compiling jobs are active - and one of our goals was to determine the optimal value of N - we opted to use a library called *Bull* which is a *Redis*-based queue for handling distributed jobs and messages in *NodeJS*. This particular package took care of both managing the queue and dispatching to workload to N different separate processes. Finally, to fully utilize this library we decided to spin-up a *Redis* instance in a *Docker* container.
Lastly, to achieve the sandbox execution of the compiled uploaded file we have chosen to stick to the native *macOS* sandbox environment called *sandbox-exec*. This particular choice was mainly made by knowing in advance that the final server that was going to host our *Single Page Application* was a *Mac Mini*, plus all the team members had a *macOS* based device. In particular, we have considered *Sandboxtron*, a wrapper around Mac's *sandbox-exec* that simplifies its setting and execution.

A further optimization was introduced later in the project in order to achieve some sort of naive caching technique regarding the uploaded files (i.e. if the given file is the same it's a waste of resources recompiling and executing it once again). So, we decided to utilize the same instance of *Redis* that we already had up and running to additionally store a tuple containing the *hash value* of the whole *.CPP* or *.CC* source file together with its output in such a way that if the *hash* of the provided file is found in the cache, the program won't launch its compilation and execution but rather it simply recovers and returns its previously stored output.

2.3 Networked Queuing System

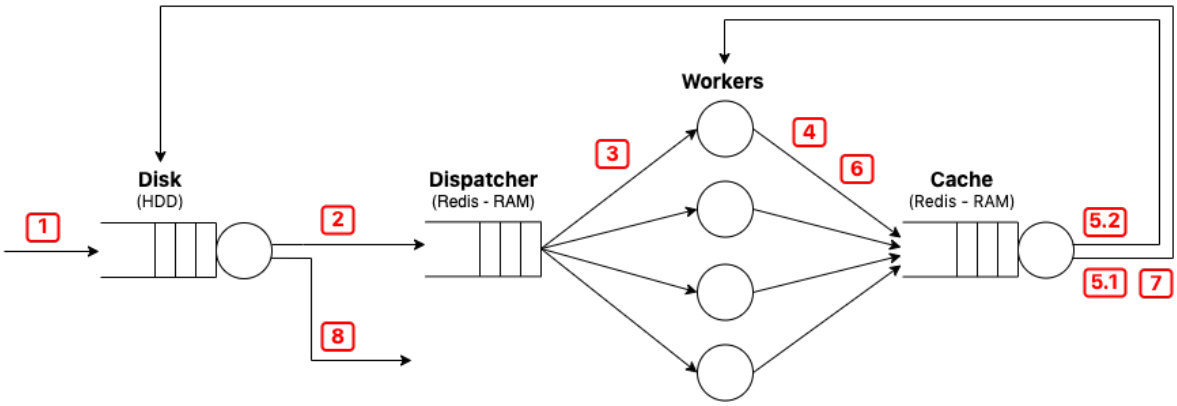


Figure 4: Web application queuing network

In Figure 4 you can appreciate a visual representation of our web application theoretical queuing network. Logically, our system (i.e. the *server*) is constituted by a series of queuing nodes inter-operating with each other.

The subsystems involved during the serving of the jobs are:

- The *hard disk drive*;
- The *dispatcher* with centralized queue with pulling technique (with 4 workers - explanation in Section 3.3.1);
- The *caching* system;

Just to recall that both the *dispatcher* and the *caching* subsystems rely on *Redis*, which is an in-memory data store (i.e. it stores the data in the *RAM*). Moreover, it's worth mentioning that we didn't configure any load balancer or other traffic controller on our web server, meaning that the web application can theoretically accept an infinite number of compilation requests (until possible, on-paper $n = \infty$) from the users.

For simplicity, we also marked out - identified by red numbers in Figure 4 - the possible paths that a new job might take during its lifetime inside our web app. Here's a quick overview of it:

- **1**: Temporary storing the uploaded file in the HDD;
- **2**, **3**: Adding the new job into the waiting queue and dispatch it to an available worker;

- **4**: Checking if the hashed version of the given file is already present in the cache;
- **5.1**: If there was a *cache hit*, the web app doesn't need to compile and execute the program;
- **5.2**: Otherwise, it means that there was a *cache miss* and the *.CPP/.CC* file needs to be compiled first and then executed;
- **6**: Creating a new entry in the cache for the new file containing its hash signature and the output of its execution;
- **5.1**, **7**: Deleting the file from the HDD;
- **8**: Returning the output and exiting from the system;

Finally, we would like to bring the focus on the *dispatcher* subsystem since we think it's the most interesting one. In fact, this type of resource can be seen as an **M/G/4/PS** queue, meaning:

- **M**, *Poisson* arrival process distribution;
- **G**, *General* service time distribution;
- **4**, number of workers available in the subsystem;
- **PS**, cores are serving the jobs with a processor-sharing service discipline;

Recalling also that at each compilation a single virtual core per file is used. Furthermore, it's worth emphasize that since the insensitivity property is still valid, our queue can also be treated as an M/M/4.

3 Benchmarking the Web Application

3.1 Hardware Configuration

The main parts for what concern the hardware configuration used during the several benchmarking tests were:

- **Server**: As briefly mentioned before, the web application during the tests was hosted on an *Apple* desktop *Mac Mini* with the following specifications:
  - CPU: Intel Core i5 dual-core (i5-4260U) with hyper-threading, base clock 1.4 GHz (2.70 GHz in *Turbo Boost*), Cache L1/L2/L3 of respectively 32KB/256KB/3MB (L1 and L2 dedicated for each core and L3 shared), 22nm lithography, 64-bit architecture;
  - RAM: 4GB LPDDR3 1600 MHz non-ECC;
  - Storage: HDD 500GB by Hitachi Global Storage Technologies, SATA III (Transfer rate: 600MB/seconds), 5400 RPM;

  - OS: macOS Big Sur (v11.2.3).

- **Client**: This was the testing machine used to launch the various *Tsung* benchmarking tests. It was an *Apple* laptop *MacBook Pro* with the following specifications:
  - CPU: Intel Core i7 quad-core (i7-4850HQ) with hyper-threading, base clock 2.3 GHz (3.5 GHz in *Turbo Boost*), 64-bit architecture;
  - RAM: 16GB DDR3 1600 MHz non-ECC;
  - OS: macOS Big Sur (v11.6.4).
- **Infrastructure**: In order to minimize the delay and/or possibles interferences we decided to interconnect the *client* to the *server* through a gigabit switch with RJ45 gigabit Ethernet cables.

3.2 Open Loop Test

3.2.1 Assumptions

Before diving into the analysis of the *open loop* tests that were made, we would like to stress about a few important aspects of the used testing setup.

Firstly, as you may have noticed in the previous section, the *tested* (aka the *Mac Mini*) and *testing* machine used to carry out the benchmarking tests were two different and independent machines. This peculiar fact of separating the roles into two distinct machines ensured us to avoid running the tests in a system already stressed up.
Secondly, the testing machine was more powerful than the tested one. This additional fact guaranteed us to actually test the *server* in which was hosted the web application rather than the testing machine.
Finally, the statistics that we were more interested in capturing and further analyzing were the various *response times* (e.g. session, page, ...) and the server physical performances (e.g. CPU load, RAM usage, ...), as we want to track both network and server statistics.

3.2.2 Testing Approach

To test out the web application we used *Tsung* - a distributed load testing tool - that thanks to its simple and easy setup through the use of an *XML* configuration file helps us to capture and then produce multiple benchmarking reports.

In the building process of creating a valid Tsung XML config file, as mentioned in the previous paragraph, we utilized one of its build-in tools (called *tsung-recorder*) to capture various user sessions.
To simulate some sort of authentic and diversified use of our web application we decided to record different scenarios (e.g. sandbox execution timeout, max

stdout buffer size, incorrect source file, and correct source file) in order to cover all the major use cases.

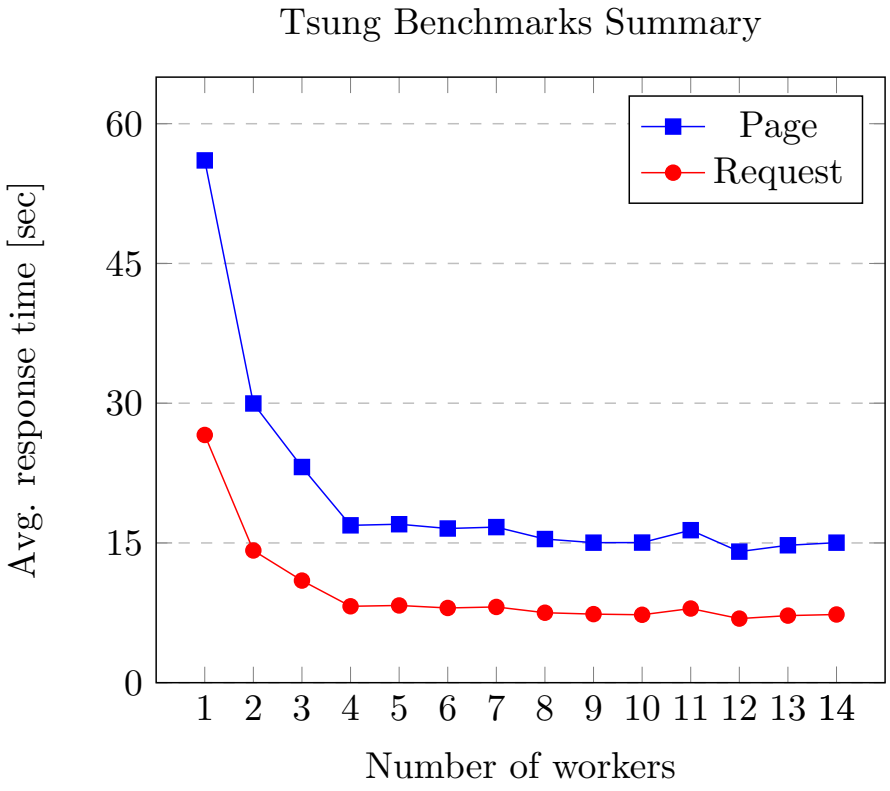
Moreover, the type of queuing system that we were more interested in analyzing was an open loop one, due to the intrinsic nature of the use of the web app (i.e. a user arrives, it uploads a file and then it leaves the system) and that's why we decided to study this specific scenario.

3.3 Tsung Test Analysis

3.3.1 Choosing the Optimal *N*

As previously mentioned, one of our main focus was to determine the optimal number N of parallel workers so that, at any given time, no more than N concurrent compiling jobs were active.

Our first theoretical estimation was that N would be somehow close to the number of virtual cores present in the *server*. The reasoning behind that is due to the fact that going beyond the numbers of available cores wouldn't bring any significant advantages, since it would implies the introduction of more context switching (i.e. more possible overhead), meanwhile considering using less cores than the available ones would certainly means increasing the overall response time (i.e. few resources, long waiting time).
To empirically prove our hypothesis we run a series of *Tsung* performance tests by changing each time the value assigned to N - this parameter can be easily adjusted in the backend configuration - and finally compared the obtained response times of each benchmark. The final analysis is summarized in the following graph:



Without any surprise we can easily confirm that our reasoning was indeed correct, meaning that N should be set equals to the number of virtual cores available in the hosting machine - recalling that in our case the *server* was a dual-core with *hyper-threading*, so in total we had 4 virtual cores.

3.4 Bottlenecks

During the benchmarking tests, we also decided to monitor the behaviour of the *HDD* (i.e. its I/O operations) since we highly thought it was the slowest component in the system. Indeed, our reasoning was shortly proven right and the reasons behind that can be traced back to the fact that for each job, we interact with the hard disk two times (in writing and deleting operations) and, due to its physical nature, its access time is at least one to two orders of magnitude slower than the nearest slowest component involved - in this case, the *RAM* due to *Redis*.
Instead of what concerns the other components, we generally saw that the *CPU* usage was around 50% in its highest peaks, meanwhile, the *RAM* never reached the 70% usage.

4 Conclusion

Finally, to briefly sum up, we have seen that the optimal number of compiling jobs that can run in parallel on the tested hosting machine (i.e. the *Mac Mini*) was $N = 4$, as explained in *Section* 3.3.1.

4.1 Further Improvements

To achieve full scalability of our web application, the next thing that can be done in the system would be to create a containerized version of the web application and set up the application in a container orchestration system like Kubernetes, Google container engine, or Amazon ECS. Doing this would allow us to scale the system at request so we would not have to care about the bottleneck part (i.e. the Mac mini's HDD) since the application would run in a cluster with very performant resources and based on how many cores our machines have, we could handle more compilation at the same time. If we assume machines with the same number of cores, defining the maximum number of compilations in the same time t as $X(t)$, the number of virtual cores of the machines as N and the number of machines as M, we can define $X(t)$ as follows in *Equation 1.*

$$X(t) = N \times M \tag{1}$$