Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

A Modular Grader for Competitive Programming

Bachelor Thesis

2019
Jakub Šimo

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

A Modular Grader for Competitive Programming

Bachelor Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Faculty of Informatics
Supervisor:	RNDr. Michal Forišek, PhD.

Bratislava, 2019
Jakub Šimo

59732258

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Jakub Šimo
Študijný program:	informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	informatika
Typ záverečnej práce:	bakalárska
Jazyk záverečnej práce:	slovenský
Sekundárny jazyk:	anglický

Názov: Modulárny testovač pre súťažné programovanie
A Modular Grader for Competitive Programming

Anotácia: Návrh a implementácia modulárneho systému poskytujúceho automatické black-box testovanie programov na efektívnosť a správnosť. Zabezpečenie systému voči útokom formou testovaného programu pokúšajúceho sa o nepovolené akcie.

Cieľ: Práca má nasledujúce ciele:
1. Spraviť prehľad existujúcich systémov s podobným zameraním, vrátane analýzy, či a ktoré ich časti je možné a vhodné prevziať.
2. Spraviť návrh testovacieho systému s dôrazom na jeho modularitu.
3. Implementovať navrhnuté komponenty a v praxi overiť funkčnosť implementácie.
4. Analyzovať otázku bezpečnosti systému v prípade, že sa testovaný program pokúsi systém poškodiť. Navrhnúť sadu testov demonštrujúcich odolnosť systému voči rôznym formám takéhoto útoku.

Vedúci:	RNDr. Michal Forišek, PhD.
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	prof. RNDr. Martin Škoviera, PhD.
Dátum zadania:	06.11.2018

Dátum schválenia: 06.11.2018

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

....

študent vedúci práce

I would like to thank my supervisor Michal Forišek for his advice and help.

I would also like to thank my family and friends, both for their support and understanding.

Abstrakt

V práci je prezentovaný dizajn a ukážková implementácia nového fakultného testovača. Jedná sa o program, ktorý bezpečne testuje programy odovzdávané študentami na predmetoch alebo súťažiacimi na súťažiach ako Olympiáda v informatike. Boli analyzované projekty, ktoré sa snažili vytvoriť podobný systém. Identifikovali sa ich úspechy a nedostatky a na základe týchto zistení bol navrhnutý dizajn a implementácia nového testovača, s dôrazom na jednoduchosť a modularitu. Každý komponent môže byť vymenený za inú implementáciu. Výsledkom je ľahšia adaptácia systému na rôzne druhy požiadaviek, akými sú napr. podpora rôznych programovacích jazykov.

Kľúčové slová: testovač, automatické testovanie

Abstract

In this thesis, a design and an example implementation of a new faculty judge system is presented. It is a piece of software, which safely tests the correctness of programs submitted by university students for their courses or by participants of competitions, such as the Olympiad in Informatics. Projects which have created similar systems have been analysed, pointing out their advantages and disadvantages in the process. Based on these, a new design of a judge system has been proposed and implemented, with the emphasis on its simplicity and modularity. Each component of the system can be freely swapped for a different implementation. This makes it easy to adapt the system to ones particular requirements, such as having to use a new currently unsupported programming language.

Keywords: judge, tester, automatic testing

Contents

Introduction

Programming is slowly starting to be taught at more and more schools. Already, there are elementary schools that teach basic programming skills and thought processes with languages such as Scratch or Imagine. Some schools are even starting to teach Python straight away.

Programs that are written by students need to be thoroughly checked for correctness by their teachers. This is usually done through observation of the outputs of these programs on provided inputs. This kind of testing puts immense strain on the teachers who have little time to do these things. Some teachers are able to create a basic testing script that helps them with this task. However, this is usually not enough due to the sheer number of students and homework there usually is to grade.

Currently, there is a major push in many sectors to adopt more automated processes. Education is no different and the aforementioned testing is inherently automatable. In most cases, it is quite simple to create a set of test cases for a given problem. Creating a script that evaluates a single homework submission on such test cases is also straightforward. However, full automation of this process is not a trivial task. This is mainly due to security concerns that arise from the execution of untrusted pieces code, such as the ones submitted by students.

At my faculty, there are a number of courses which assign homework which is testable in this way. Members also organise programming competitions, such as the International Collegiate Programming Contest (ICPC), Internet Problem Solving Contest (IPSC), Olympiad in Informatics, etc. These competitions require similar services to the ones which are used for the courses. Currently, all of these are handled by a single grader - judge. The judge has been working reliably for more than 15 years. However, it has not been designed to handle workloads of such variety.

Due to the judge's age, there have been many additions and alterations to its code. These have been done by quite a few people, with differing levels of software engineering skills, at wildly differing points in time. There is little to no documentation and some of the decisions which have been made during the development have been lost to time. It is safe to say that the judge can be considered a piece of legacy software.

The purpose of this thesis is to propose and implement a proof of concept modular judging system, which will be able to accommodate current requirements of the faculty.

The design should be able to easily accommodate possible future additions. Another major requirement is that the system should be as simple as possible, so that the members of the faculty with less software engineering experience can still modify the system to their requirements if needed.

This thesis will be divided into 4 chapters. The first chapter will explain the subject matter and terminology. In the second chapter, we will explore projects which try to accomplish similar goals to ours. We will analyse their advantages and their drawbacks. Design of the test system will be proposed in the third chapter. The fourth chapter will contain a description and a possible example implementation of the system.

Chapter 1

Judge

In this chapter, we will take a closer look at what is a judge, in what ways it is usually used and what guarantees it should offer.

1.1 Real life situation

Annual IOI competition is taking place. A few days are designated as programming days. During each of these days, participants are presented with a set of problems called *tasks*. Participants need to create a computer program, commonly referred to as *a submission* or *a submit*), which solves particular instances of these problems. The competition begins. After some time, participants which solve a task submit their solution (hence the name, submission). The submission is usually submitted through a website in the form of source code. The submission goes through and gets passed onto *a judge*.

1.2 Evaluation process

The judge takes the source code, compiles it, if necessary, and executes the resulting program on a pre-defined set of test cases. After each execution, a result code is assigned - AC for accepted, or a correct answer; WA for wrong answer; TLE for time limit exceeded. This code is usually referred to as *a status*. A status can be assigned in various ways. Here are the three most basic ones:

- **output** - the program is given input and it is supposed to create the correct output for the given problem. This output is then compared to the correct output. The submission gets an AC if the two outputs are identical.
- **checker** - the program is given input and it is supposed to create one of the many possible correct outputs. This output is then fed into a *checker*. A checker is a

program that takes an input and validates its correctness. If the checker reports that the output is correct, the submission gets an AC.

- **interactive** - the submitted program A is run side by side with another program B. The output of A is fed into the input of B and vice versa. B is a program created by the task setter and it usually responds to queries which A is allowed to make. The submission gets an AC if program B reports as such.

In most competitions, test cases are grouped into *batches*. Batches are most commonly comprised of test cases, which are similar in some fashion, for example, in the size or in the complexity of the input. A task usually contains one or more batches. There are three major ways in which competitions evaluate the final result status of a submission:

- **IOI style** - used in IOI competitions. A task contains multiple batches of test cases. Each batch gets evaluated individually. If any of the test cases in a batch gets a result status different from AC, the result of the batch is set to this status and subsequent test cases in this batch are ignored. Participants get assigned points for each batch which got an AC, i.e. only for batches, where all of the test cases have gotten an AC.
- **ICPC style** - used in ICPC competitions. Very similar to IOI style, with the only difference being that there is only a single batch.
- **education style** - usually used at university courses and for education purposes. The program is executed on all test cases, regardless of any results. The submission is accepted if all of the test cases are accepted.

However, executing unknown programs poses a security threat. In order for the judge system to keep functioning properly, there need to be safeguards put into place.

1.3 Security

The execution needs to be done in a safe and controlled environment, inside of a *sandbox*. A sandbox refers to a restricted operating system environment. Processes executed inside of a sandbox have limited access to certain resources, such as file descriptors, memory, etc. This is done in order to mitigate the possible harm that could be done by the executed process. However, during competitions, participants are usually not looking to destroy the judging system. They want to take advantage of it, most commonly in order to advance their position in the competition. Here are three illustrative examples:

- One of the solutions to the given task is easily parallelizable. If the program was executed without any constraints, participants could easily start multiple threads/processes to speed up the calculation instead of optimising the algorithm being used.
- In most competitions, you can see whether your submission was correct or not. You can use this information to prioritise which problems are worth solving. Participants could block the judge by, for example, submitting source code which results in an essentially endless compilation. This would block other participants from ever seeing their results as the judge would be stuck on the compilation step of a single submission until the end of the competition.
- Assuming the input is sufficiently short, participants could have their program send the input through the network to a much more powerful computer to solve.

Situations like these are, naturally, unacceptable. Thankfully, existing projects have already successfully tackled this issue, which leads us into the next chapter.

Chapter 2

Current state of judging systems

In this chapter, we will look at the design choices of three judging systems that are currently in use. First, we will take a look at two open-source projects. We will point out their advantages and disadvantages and try to learn from them. Afterwards, we will take a look at the current faculty judge. As this is the one which we are trying to replace, its analysis will be more thorough.

2.1 Current landscape

There is a surprisingly small number of attempts at a generalised judge system. Virtually all takes on this problem have been created as full-on contest systems [7] and it seems to be true to this day. This lead to the majority of them being specific to certain competition types. Needless to say, countless hours have been spent on these projects, with little to no reusability. This is surprising, as while competitions have wildly differing scoring systems, the majority of them evaluate submissions almost identically.

Sadly, many of these systems do not provide their source code. Some out of security concerns, some by the virtue of it going against their business model. For the purposes of this thesis, we will evaluate two major contemporary open-source projects available.

2.2 Contest Management System (CMS)

CMS [2] is the project which has been created for the 2012s International Olympiad in Informatics (IOI) in Italy. Even though it was not meant to, it has been used in subsequent IOIs too.

The project has been tailor-made for the purposes of the IOI style competitions. As mentioned above, it is not only a judge but the whole package - *"CMS is a distributed system for running and (to some extent) organizing a programming contest."* [2]. This naturally influenced the project developers' decisions during the design of their system and their judge part of it.

While CMS developers have created a well-designed system for their use case, its components suffer from high coupling. In our case, taking out the judging part and using it on its own possibly requires much more work than creating a totally new system. This is mostly due to the IOI specific submission scoring being incorporated directly into the judge.

Further complicating things are developers' efforts to document their project. There is a lot of redundant documentation in the code base. While it is mostly well written and pleasant to read, it makes it hard to read the code itself. Names of methods and variables are often nondescript, opting to provide lengthy descriptions of them in the documentation instead of simply using a more descriptive name.

The system of incorporating different languages is done cleverly. Each supported language has a single module, inside of which are things like compilation and execution commands set. Adding a new language is as simple as creating a new language module. This makes it much less error-prone, due to new languages not directly interfering with the core of the existing codebase.

CMS uses *isolate* for its sandboxing requirements, as it is the standard for IOI. Thanks to this, the system is considered to be quite secure, as it has withstood many years worth of submissions by possibly one of the craftiest group of people.

2.3 DMOJ

The other major open source project is DMOJ. Similarly to CMS, DMOJ is a complete, mostly coupled system. It comes with its own website, contest and task management, reporting tool and many more. Probably because of this, developers of DMOJ have made similar design decisions as did the developers of CMS. The system has its own graders, it has language modules and the similarities between the two systems go on.

The major difference between DMOJ and CMS is in terms of execution security. DMOJ does not use *isolate* to sandbox submissions, it uses *seccomp* Linux kernel utility in order to filter system calls done by a process. If implemented correctly, it can reduce the attack surface of the kernel without the complexity of *isolate*. However, according to M. Forišek (personal communication, May 13, 2019), there have already been attempts at this by the developer of *isolate*, M. Mareš. It turned out that filtering, such as when using *seccomp*, causes too many context switches to happen, which leads to a slower and more importantly less consistent execution speed of the program. In addition, there does not seem to be any formal inspection of the system that is implemented here. While this is not strictly necessary, it is not a simple system to implement and there is a very real possibility of critical flaws appearing.

2.4 Faculty judge

As the faculty judge is the one which is supposed to be replaced in this thesis, we will take a closer look at different requirements.

2.4.1 Handling of multiple clients

The judge serves the testing purposes of numerous pages. However, the method of achieving this is less than ideal.

```
if submit['contest'] == 'CLIENT':
  generate_protocol(rootdir,submit_basename)
  ...
  summary = get_summary_xml(submit, result)
  try:
    subprocess.call(
      'echo␣"' + summary +
      '"␣|␣ssh␣client@ksp.sk␣"cat␣>>␣/home/client/EVENTS"',
      shell=True)
  except:
    print('INTERNAL␣ERROR:␣failed␣to␣upload␣submit␣message')
```

Figure 2.1: Result reporting procedure for one of the clients

If we want to add a client, it is necessary to edit the code of the judge. Naturally, this means that the whole system needs to be taken down in order to add a new client, which is less than ideal. Code excerpt in Figure 2.1 shows that the reporting of testing results is done on a case by case basis. In recent years, there have been attempts to standardise this process, but they all failed, mostly due to the amount of work involved. In the end, only a handful of clients have been migrated to newer reporting techniques.

Proposed state

Each client should have a POST API endpoint at which it accepts the results of the evaluation. As mentioned, a similar thing has been in the works, so some clients are already prepared for this.

2.4.2 Submission process

The submission system is one of the most unsecured parts of the system. The submissions are done through the way of writing to an open Internet-facing socket. This has been regarded as an acceptable practice for a long time. However, the Internet turned out to be quite the predatory place, so this practice is now widely accepted to pose a serious security risk. While the judge is not a safety-critical system, its downtime could hinder the faculty staff in a major way, consuming most of their time which could be better spent on research.

Submissions are done by sending the protocol followed by the file directly through *netcat*. This is easy to do if it needs to be done by hand and *netcat* is ubiquitous. However, so is *curl*. When doing things by hand, it does not make much difference, but in code, *curl* wrappers are the superior option.

The judge accepts submissions with a certain protocol (Figure 2.2). This protocol is the one piece of the current judge for which there exists some kind of documentation. It contains metadata which is used during the judging process. However, it has a few shortcomings. The person who submits the file for grading is able to select who is the client, making the impersonation of any client trivial. This is also due to the protocol having no authentication mechanism.

Field	Example value
HEADER	submit1.3
CONTEST_ID	ACM
SUBMIT_ID	1234567890-12345
SENDER	misof
TASKNAME	buoyancy
LANGUAGE	py
PRIORITY	0
FOOTER	magic_footer

Figure 2.2: An example of the submission protocol

The protocol contains a few seemingly redundant pieces of information, such as the username of the submission. This is used for occasional debugging to make the search through all of the submission in the judge easier.

Proposed state

The judge should accept submissions via an HTTPS POST request. The request should contain the protocol with only the required pieces of information and a single file. Due to the client list being static most of the time, simple token authentication can be used to verify the sender. If a new client comes up, it is easy to generate a new token for it.

2.4.3 Task management

The management of tasks in the current judge is done quite well. As can be seen in Fig.3, a single task is contained inside of a single directory. There is a file *metadata.config*, which contains settings for the task - time limit, memory limit, problem type, etc. If the tasks requires it, there can be a script that is used instead of the standard submit execution procedure. This is helpful in tasks in which there are multiple correct answers or in tasks where the submitted program must interact with another program. The rest of the files are input/output files and other various resources required for the task evaluation.

The advantage of this approach is that it is easy to maintain a single task because everything is in that single directory. However, the way this approach is currently executed is far from ideal. The tasks are only stored in the filesystem of the judge system itself. This hinders the possibility of running multiple instances of the judge simultaneously. If we wanted to create another instance, we would need to manually copy the required tasks from the original judge.

Another shortcoming of the current design are task identifiers. Each task has a short string identifier (*rings*, *badsquare*). These identifiers are however shared between all the clients. This has already lead to several name clashes. While this can be solved by remembering to prefix the identifiers with additional data, it is a simple problem that should be avoided in design.

Proposed state

Task management is simple and effective. The single mentioned shortcoming is easily solvable by first creating directories for different clients and storing the task directories in these.

2.4.4 Submission management

Submissions and their respective metadata are all being stored in the filesystem. This has worked for quite a long time and the argument for not using a database was that it was easier to search it this way. However, as time went on, users and submissions have been slowly rising in numbers. Naturally, we are starting to hit filesystem limits of the number of files/directories in a directory. This needs to be solved because the number of users and submissions will only continue to rise.

Proposed state

Use a database for submission metadata. The use of a database allows us to query the metadata without having to resort to combing through makeshift directory structures.

2.4.5 Execution environment and language support

The current judge supports multiple languages. However, each language simply gets an *if* (Figure 2.3) in the execution script, making the addition of more languages prone to breaking other things.

```
if submit['language'] == 'java':
  main='zrubem_sa_lebo_takuto_classu_nemam'
  for classa in sorted(os.listdir(tmpdir)):
       if not (classa.endswith('.class')): continue
       try: shutil.copy( tmpdir+'/'+classa, box+'/box/' )
       except: pass # nemalo by nastat
       if classa.count('$')==0: main = classa[:-6]
  out, err = execute_external( [bindir+'/isolate'] + ... ...
```

Figure 2.3: Code for execution of a Java submission

While the addition of new languages is possible, versions of their compilers and interpreters are quite limited. This is due to the nature of the system the judge is running on. The current judge runs inside of Debian, which is one of the most commonly used stable Linux distribution. With it being stable comes the cost of packages, such as our compilers and interpreters, being quite old. This is not necessarily a functionality problem, however many users are quite upset at not getting the newest features while using their language of choice.

Hand in hand with the previous point comes the problem of replicating the execution environment for testing purposes. For example, the current workflow of testing whether a given task works is to simply upload it to the judge, try to submit and see what happens. This is due to task setters having much newer packages on their systems. It would make for a safer workflow if task setters were able to test their new task directly on their machine with packages equivalent to the ones available on the judge.

The performance differences between languages are also becoming a problem that has not been taken into account. The main problem this causes is that schools are slowly migrating towards Python, which is an interpreted language, hence it is slower than compiled languages. It also has a few quirks, such as certain input and output operations taking significantly longer than in other languages. This is especially bad in educative competitions, in which children are tasked with solving the problem in the language of their choosing. Those, who were taught one of the less efficient languages are at a disadvantage, even though they might be just as capable as their peers who were taught a different language in their school.

Proposed state

Virtual machines or an environment isolation tool should be used for the execution of user submissions. This would automatically have the advantage of task setters being able to try their new tasks on their computers.

There is no obvious silver bullet for solving the language problem. However, certain things such as the memory overhead of the interpreter can be accounted for without any further work on the part of the task setter. Time limits can be made specific to each language for each task. This will, however, require further work by the task setter. Nevertheless, having the option to assign times for each language is strictly better than not having it.

There is the option of designing a test suite of programs that would benchmark certain aspects of languages (such as their I/O speed) and change the limits accordingly. However, this approach is questionable, due to most languages having many quirks. Crafty users could quickly notice how the limits behave under certain conditions and use it to their advantage. As such, this concept requires research which is out of the scope of this thesis.

2.4.6 Parallelisation

By parallelisation in this context, we mean the possibility of starting more than a single instance of the judge, while being transparent to the clients. This means that no further configuration should be needed on the client side. The current judge can indeed be parallelised, however, it has a few disadvantages. The judge has not been designed with this requirement in mind, so the only way to distribute the submissions between multiple instances is to put a TCP load balancer in front of them. However, this does not take into account the load of each worker, simply dealing the submits out in a round-robin fashion. Some paths are also hardcoded in source code of the judge, so running multiple instances on a single machine is quite troublesome.

Proposed state

It should be possible to seamlessly add more workers to the judging system. The system should also be able to deal with asymmetrical work assignment. Both of these problems can be solved with the use of a *message broker*. Workers would connect to their specified broker and simply accept submissions. This keeps the worker code simple, as it does not need to deal with anything apart from the evaluation of single submissions.

2.4.7 Security of the submission evaluation

The security of the current judge environment is considered to be good. Internally, it uses a program called *isolate* [4], which has been developed for the use at the International Olympiad in Informatics (IOI). There have been no security problems with this piece of software, it is created for this exact purpose and as a bonus, it is still being maintained. The only slight disadvantage is that it requires root privileges to work, which poses a security risk if the *isolate* process was to be taken over.

Chapter 3

Proposed Design

In this chapter, we will describe the general ideas for a modular design we are striving to achieve. We will describe each component, focusing on the simplicity of the whole system overall. This is not meant to be a set in stone description of a system, but merely a sketch of what appears to be the general consensus across many projects. We will also discuss some features which should theoretically be unnecessary, however, in the real world, they turn out to be extremely helpful when the need arises.

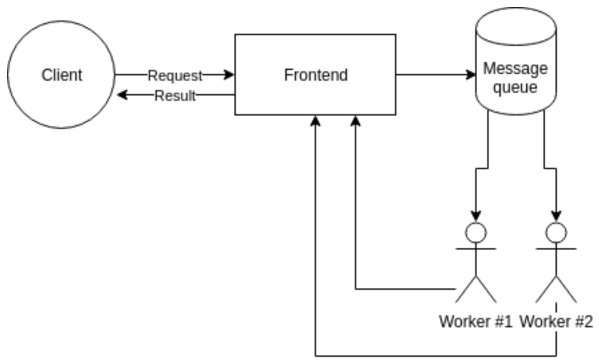

Figure 3.1: Flowchart of the judge components

3.1 Frontend

3.1.1 Description

The frontend is the gateway to the rest of the system. It comprises of a web server which accepts the testing requests and a database for the storage of submissions and relevant metadata.

3.1.2 Responsibilities

Submission processing

The component takes care of the front-facing workflow, i.e. it accepts the submissions, enqueues them for evaluation, stores the results and reports them to clients.

As we have seen in the previous chapter, the addition of grading directly into the judge increases complexity and decreases reusability. Even if the grader is modular and each client can create their own module, it is unnecessary to do so because each client can be given the full testing report and it can grade the submission according to its needs.

One of the optional features is the ability to re-evaluate a submission. This comes up from time to time when a deficiency in the test data is found. In an ideal scenario, such a feature should be taken care of fully by a client by simply creating a new submission with old information. Judge is indifferent to the fact that it is a re-submission and the client can reuse the submission functionality it already has implemented. Putting all of this responsibility on the judge requires adding a whole another layer of complexity in the form of login and access privilege systems, which we are trying to avoid. For example, faculty judge clients do not have any resubmission functionality and it would be time-consuming to fully implement it in each one of them. This is mainly due to the fact that some clients do not store source codes of submissions.

There is a nice middle-of-the-road solution. The judge can implement an API which accepts requests for a re-evaluation of older submissions by ID. This approach keeps the judging part simple, as it does not require a whole other complex page for the selection of submissions to be evaluated.

Client management

Client management should be done in the simplest way possible. As each client presumably needs an approval to be able to use the judge, any valid authentication mechanism can be used and the choice is at the discretion of the developer. The simplest one that could be used is a passphrase or an API token.

Reporting of results to clients

There are two options that can be chosen here: a push or a pull model. In a push model, the judge sends - pushes - the results to the client after the evaluation has completed. This requires work on the end of the client, as it needs to have a separate endpoint for accepting these results. In a pull model, the results are not sent directly after the evaluation. Instead, when the need for the results arises, such as the user opening the submission detail page, the client requests the results and stores them at that point.

The choice here is the push model. While a pull model makes the judge implementation a little simpler, it adds complexity on the end of a client which is disproportionate to the savings we have made at the judge side. For example, if the client re-submits a submission, it needs to delete the cached results and make sure that if a user reloads at the right time, he does not end up with the old results. While theoretically simple to do, implementations of this are certainly prone to error and we can avoid this at a very small cost.

With the push model, we define an API endpoint for receiving results. The judge is given a client URL and the client implements the provided endpoint. After the evaluation, the judge sends a request to the client provided URL. The client accepts the files and stores them appropriately. If implemented correctly, the reporting operation is idempotent. If the client re-submits a submission, the judge reports the updated result just like any other and there is no added logic required, neither on the judge or the client side. Based on these points, the push model seems to be the superior choice under normal circumstances.

Task management

The current faculty judge has simple and straightforward task management and it can be reused here. The directory with the task files can be archived and uploaded through a simple web interface and stored for future redistribution between workers.

3.2 Queue

After arriving at the frontend, the submissions get passed into a queue to be handed out to individual workers. A suitable message broker with a message queue should be chosen here. A message broker provides valuable guarantees, such as automatic retries if any of the workers die or message persistence or the ability to add any number of workers dynamically. It also adds to the modularity aspect of the judge as there are libraries for basically any message queue in all reasonably popular languages. Message brokers can also be used for the optional feature of automatic retries of result reporting with exponential backoff.

3.3 Worker

A worker is the backbone of the judge where the actual evaluation takes place. It accepts messages with individual submissions from the message queue, safely evaluates the submission and sends the result together with the execution metadata back to the frontend.

An optional feature is the possibility to replicate the execution environment on a different machine. This would lead to a better experience for the task setter as he could test his task directly on his machine.

Chapter 4

Implementation

In this chapter, we will discuss the implementation details of the key parts of the judge system designated in the previous chapter. These are the choices that are being made at this point in time, with software that is available today, for the use cases of our faculty. If there ever happens to be a piece of software that makes more sense to use, it is encouraged to do so.

Currently, the language that is easiest to pick up without much previous knowledge is Python. There are libraries available for each of the components, namely web server frameworks, and message broker wrappers. The evaluation that takes place on the worker is required to be serial by nature, so the choice of Python, which is a scripting language, is also fitting.

4.1 Frontend

As the frontend consists of a single API endpoint for submitting and a few simple pages for the task, client and administrator management, any choice of a web framework suffices. The reference implementation uses Flask due to its simplicity, however, Django would also be a valid choice, as its popularity often offsets its added complexity. The database of choice is PostgreSQL as there are no special requirements and Postgres is currently the industry standard.

4.1.1 Submission processing

The submission is done with a POST request. The request consists of the following parameters:

- auth header - API key sent in the header, used to authenticate the client
- submit ID - an ID by which the client wants the submit to be reported

- task ID - the ID of the task on which the submit is to be evaluated
- language - the programming language in which the submission is written
- priority - the priority of the evaluation. It can be used to de-prioritize evaluations if there are a lot of submissions that need to be re-evaluated.
- username - optional, can be used for easier lookup of submissions if necessary
- file - the submitted file

As *curl* [3] and its wrappers are ubiquitous, submitting is as simple as executing the following bit of shell script:

```
curl -X POST \
  --header "Auth-Token: passphrase"
  -d file=@/path/to/submission/file.txt \
  -d submitid=1564789 \
  -d taskid=4 \
  -d language=cpp \
  example.com
```

Afterwards, the server enqueues the submission into the message queue.

4.1.2 Reporting of results to clients

Each client needs to set a URL, where the results can be reported. At this URL, there should be an implementation of a simple API endpoint. This endpoint should accept a submit ID parameter and a single file - an archive, which contains all of the testing protocols.

4.1.3 Client management

Client management is done at the frontend level. When a new client wants to be added, a new auth token is generated and given to the client. This is used as the passphrase in the aforementioned example. Thus, adding a new client does not require a restart of the whole system as only the row with the newly accepted key needs to be inserted into the database. If for some reason a key needs to be revoked, it is simply set as void in the judge's database.

4.1.4 Task management

Task management comprises of a handful of webpages where task setters can upload a single archive with all the test data of a task. Workers can then download and cache it from the server when required.

The issue of the freshness of the test data on a worker arises. This is solved by the worker receiving a timestamp of the last task change together with the submission metadata. The worker can compare this timestamp to the timestamp of the archive it possesses and download fresh test data if necessary.

4.2 Queue

RQ (Redis Queue) [5] has been chosen as our queue manager. It is a library that is backed by the Redis key-store and acts as our message broker. Given our simplistic use case, it allows us to ignore all the complexities of the message delivery process. On one end, the frontend sends the submission data into the queue, on the other end, there is a python module that gets executed. If someone wants to edit this module, he does not need to be bothered by the message delivery, as it is taken care in the background.

4.3 Worker

Each worker is a dedicated machine (or a virtual machine if enough hardware is not available). The worker is divided into two parts: the processing part and execution part. Because of this, each worker needs to have at least two cores assigned in order to make the evaluation as consistent as possible.

The processing part is a Python script, which receives a message from the queue, gathers all the necessary data for the evaluation and then passes it onto the execution part. After the execution is done, it reports the results to the frontend, which subsequently reports them to the client.

The execution part is a Docker container, containing all of the evaluation scripts. Docker is an environment isolation technology that allows us to isolate the execution environment from the rest of the system. This is similar to a virtual machine, the difference being that Docker does not virtualise a whole operating system, only parts of it, reusing the host kernel. This isolation does not inherently add any security, it is used to isolate the different versions of libraries and packages used. This makes it independent from the host system it is running on, in turn making it easy to replicate the execution environment on any PC which supports Docker. It does not replicate the limited hardware capabilities of the production machine, however, for most debugging purposes, the exact environment is more than enough.

Inside of the Docker container are the evaluation scripts. These are simple scripts that take care of the compilation and execution. We have taken cues from CMS and DMOJ and used language modules for the management of different languages. The modules are used in a pseudo-strategy design pattern, greatly simplifying the evaluation scripts. This makes adding a new language as simple as adding the required compiler/interpreter into the Docker image and creating a new language module.

```
class Java:
  source_extension = 'java'
  requires_compilation = True
  compile_command = [
      '/usr/bin/javac',
      'submit.java',
  ]
  run_command = [
      '/usr/bin/java',
      'Submit',
  ]
  memory_overhead = 1912345
  jvm_startup_time = 500

  def normalise_time(self, time_limits):
      return time_limits + self.jvm_startup_time

  def normalise_memory(self, memory_limit):
      return memory_limit + self.memory_overhead
```

Figure 4.1: Language module for the Java programming language

isolate is used as our sandbox based on the points mentioned in the previous chapters. Together with Docker, it creates a nice line of defense against malicious programs, as Docker further complements *isolate's* capabilities by adding more functionality by which the process can be confined.

Conclusion

The aim of this thesis was to design and implement a modular judge system that could replace the current faculty judge. Currently used systems have been analysed, looked at what they did right and what they could have done better. The three systems we have examined are Contest Management System used by the IOI, DM Online Judge used by Canadians and the judge used by the Faculty of Informatics of Comenius University in Bratislava.

Attached is a basic example implementation of the proposed design. In this implementation, we have used technologies which have the prospect of lasting due their popularity. This was done in order to make the system as simple to understand and work with as possible.

This thesis will be used as the documentation for further development of the system.

Future work

The system will be further refactored, polished and tested. There will be a transitionary period, where it will be running alongside the faculty judge, after which all of the testing will be moved from the old judge to the new one. In the future, the final implementation will be available at HERE and alternatively HERE

Bibliography

[1] Cms internals documentation. `https://cms.readthedocs.io/en/v1.4/index.html`. Accessed: 2019-01-24.

[2] Content management system. `https://cms-dev.github.io/`. Accessed: 2019-01-24.

[3] curl. `https://curl.haxx.se/`. Accessed: 2019-05-19.

[4] Isolate execution environment. `http://www.ucw.cz/moe/`. Accessed: 2019-05-15.

[5] Rq. `https://python-rq.org/`. Accessed: 2019-05-19.

[6] Sphere engine. `https://sphere-engine.com/demo`. Accessed: 2019-01-24.

[7] Michal Forišek. Security of programming contest systems. *Information Technologies at School*, pages 553–563, 2006.