Turing Arena Light: Enhancing Programming Education Through Competitive Environments

Giorgio Audrito
University of Turin, Italy

Luigi Laura
International Telematic University Uninettuno, Rome, Italy

Alessio Orlandi
Google, Zürich, Switzerland

Dario Ostuni
Università degli Studi di Milano, Italy

Romeo Rizzi
Università di Verona, Italy

Luca Versari
Google, Zürich, Switzerland

Abstract

Turing Arena light, the spiritual successor of Turing Arena, is a contest management system that is designed to be more geared towards the needs of classroom teaching, rather than competitive programming contests. It strives to be as simple as possible, while being very flexible and extensible.

The fundamental idea behind Turing Arena light is to have two programs that talk to each other through the standard input and output channels. One of the two programs is the problem manager, which is a program that interacts with a solution to give it the input and evaluate its output, and eventually give a verdict. The other program is the solution, which is the program written by the contestant that is meant to solve the problem.

In this paper we describe the architecture and the design of Turing Arena light.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Software and its engineering → Development frameworks and environments; Social and professional topics → Computing education

Keywords and phrases Competitive Programming, Contest Management Systems, Online Judges

Digital Object Identifier 10.4230/OASIcs.Grossi.11

Category Education

Supplementary Material *Software (Source Code)*: `https://github.com/romeorizzi/TALight` [32] archived at `swh:1:dir:7e06febf4432bbcf7ce6301cd4de80837fa094d4`

Acknowledgements All the authors of this contribution first met thanks to Roberto Grossi and his dedicated work with the Italian (and International) Olympiads in Informatics. TALight is also a successor of the Fully Integrated Contest Analyzer that Roberto and some of the authors were planning a few years ago that evolved into the CMS [19]. This paper, like the entire volume, is dedicated to Roberto.

1 Introduction

Programming contest management systems are the backbone of competitive programming events, handling everything from problem distribution and solution submission to automated judging and live scoreboarding [28, 19, 20]. Over the years, these systems have evolved from ad-hoc scripts and manual procedures into sophisticated platforms that emphasize

security, scalability, and fairness [18]. Traditional contest systems like the **Programming Contest Control System (PC²)**, used in ACM ICPC since the 1990s, enabled basic contest operations (login, submissions, judging interface) and were reliable for on-site contests. However, many early systems required judges to manually run solutions or provided limited automation.

Turing Arena light (TALight) is a new contest management system that distinguishes itself by focusing on *simplicity, interactivity, and flexibility*. It was conceived as a lightweight platform geared toward educational use and practice environments rather than large-scale contests [31]. TALight's design philosophy is to keep the core system minimal and conceptually simple, delegating most functionality to problem-specific modules. Uniquely, all problems in TALight are treated as interactive by default, meaning a contestant's solution interacts in real-time with a problem manager program that provides inputs and checks outputs.

2 Turing Arena light

In this chapter we introduce *Turing Arena light*, the successor of *Turing Arena* [31]. *Turing Arena light* is a contest management system that is designed to be more geared towards the needs of classroom teaching, rather than competitive programming contests. It strives to be as simple[1] as possible, while being very flexible and extensible.

While we will discuss each point in more detail later, as an overview the design of *Turing Arena light* focuses on the following aspects:

- *Simplicity*: the design of *Turing Arena light* tries to keep things as simple as possible, while achieving the desired functionalities. While a meaningful objective metric for simplicity is hard to define, the current implementation of *Turing Arena light* consists of only 2197 lines of code, with an average of 39 chars per line and an overall of 85666 bytes [32].
- *Interactivity*: in *Turing Arena light* all problems are interactive by default. This means the contestant's solution for a problem always interacts in real-time with the problem. In particular, a problem in *Turing Arena light* is defined by the problem *manager*, which is a program that interacts with the contestant's solution and gives a verdict at the end of the interaction. By being interactive by default, *Turing Arena light* allows a wider range of problems to be implemented with less effort, while not causing much overhead for non-interactive problems.
- *Flexibility*: *Turing Arena light* is designed to be able to run on all major operating systems, and allow solutions and problem managers to be written in any programming language, while still being able to guarantee a certain level of security. To achieve this, *Turing Arena light* only consists of a small core written in Rust [21], whose main purpose is to spawn the process of the problem manager on the server, to spawn the process of the contestant's solution on its own machine, and to connect the standard input and output of the two processes. Thus, the contestants' code is never run on the server, and the problem manager can run without a sandbox, being trusted code written by the problem setter.
- *Extensibility*: as stated in the previous point, *Turing Arena light* only consists of a small core that has the fundamental role of spawning to processes and connecting their standard input and output. All the other functionalities are implemented by the problem manager itself, possibly using a common library of utilities. This allows the problem setter to implement any kind of problem, while still being able to use the same contest management system.

[1] Simple might mean very different things, in this context it is conceptual simplicity.

3 Related Work

Competitive programming systems can be classified into three main categories, i.e. Contest Management Systems (CMS), Online Judges and Classroom Ad-hoc Tools, each serving distinct purposes while sharing some overlapping features.

Contest Management Systems (CMS) are sophisticated platforms specifically designed for formal competitions like the International Olympiad in Informatics (IOI), ACM-ICPC, or national olympiads. Systems like DOMjudge [26], CMS [19, 20], and Kattis [6] provide robust infrastructure for high-stakes, in-person events. They focus on security, reliability, and scalability to handle numerous concurrent submissions while maintaining fair evaluation conditions. These systems typically include features like real-time scoreboards, detailed analytics for judges, and stringent sandboxing mechanisms to ensure solution integrity. CMS platforms prioritize standardized evaluation environments where all participants compete under identical conditions with controlled resource limitations.

Online Judges serve a broader educational purpose by providing continuous access to problem-solving opportunities outside formal competitions. Platforms like Codeforces, LeetCode, and SPOJ host extensive problem libraries that users can attempt at their own pace. Unlike Contest Management Systems, they emphasize learning progression through difficulty-ranked challenges, detailed performance statistics, and community engagement via discussion forums and editorials. While they can host virtual contests, their primary value lies in self-directed practice. Many online judges incorporate gamification elements like ratings, badges, and streaks to motivate continued participation. Furthermore, since these systems have, in some cases, order of thousands of different tasks, there is a vast literature related to the development of recommender systems able to suggest a suitable task depending on the learner's abilities [1, 7, 9, 8]; also the problem of plagiarism is addressed [17]. We refer the interested reader to the surveys of Wasik et al. [37] and Watanobe et al. [38].

Classroom Ad-hoc Tools like Turing Arena Light are specifically tailored for educational settings where pedagogical considerations outweigh competitive rigor. These systems prioritize ease of use, interactive problem types, and flexibility to accommodate diverse learning objectives. Unlike the standardized environments of CMS platforms, classroom tools often allow students to work in familiar development environments on their own machines. They typically feature simplified interfaces, immediate feedback mechanisms, and support for interactive problems that engage students through real-time interactions. While less suited for large-scale competitions, these tools excel at reinforcing classroom concepts and providing instructors with meaningful insights into student progress.

4 Architecture and design

This section explores the technical architecture and design principles that form the foundation of Turing Arena Light. We begin by explaining the core interaction model between problem managers and solutions, which differentiates TALight from traditional contest management systems. Then, we examine each primary component in detail: the problem manager that defines and evaluates tasks, the server that orchestrates communication, the client that runs on contestants' machines, and the user interface that contestants interact with. Throughout this section, we highlight how TALight's design choices support its goals of simplicity, interactivity, flexibility, and extensibility while maintaining a lightweight yet powerful infrastructure for educational programming environments.

The fundamental idea behind *Turing Arena light* is to have two programs that talk to each other through the standard input and output channels. One of the two programs is the problem *manager*, which is a program that interacts with a solution to give it the input and evaluate its output, and eventually give a verdict. The other program is the *solution*, which is the program written by the contestant that is meant to solve the problem.

While this is not too far off from what other contest management systems do, the two main differences are that in *Turing Arena light* these two programs run on different machines, and the interaction between them is done in real-time. This is unlike mainstream contest management systems, where the two programs run on the same machine (like in DOMjudge [5], CMS [19] and Codeforces [3]), or where the interaction is not done in real-time (like in the old Google Code Jam [12] and Meta Hacker Cup [23]).

In the following subsections we will discuss the components of *Turing Arena light* and how they interact with each other. We will start from the problem manager, going through the server and the client, and finally discussing the user interface.

4.1 Problem manager

```
%YAML 1.2
---
public_folder: public
services:
  free_sum:
    evaluator: [python, free_sum_manager.py]
    args:
      numbers:
        regex: ^(onedigit|twodigits|big)$
        default: twodigits
      obj:
        regex: ^(any|max_product)$
        default: any
      num_questions:
        regex: ^([1-9]|[1-2][0-9]|30)$
        default: 10
      lang:
        regex: ^(hardcoded|hardcoded_ext|en|it)$
        default: it
  help:
    evaluator: [python, help.py]
    args:
      page:
        regex: ^(free_sum|help)$
        default: help
      lang:
        regex: ^(en|it)$
        default: it
```

Figure 1 Description file for a problem in *Turing Arena light*.

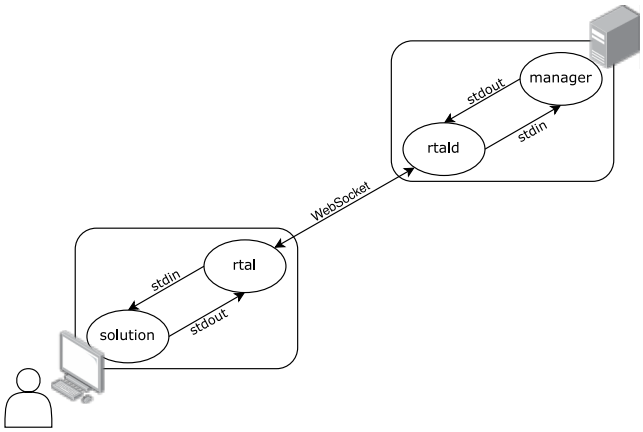

Figure 2 Architecture of *Turing Arena light*.

A problem in *Turing Arena light* is defined as a set of *services* and a set of *attachments*. A service is a program that can be spawned with a set of well-defined parameters, and that will ultimately interact with the solution. An attachment is a generic file that can be attached to the problem and downloaded by the contestant, such as the statement of the problem, or a library that the contestant can use in their solution.

A service defines which parameters it accepts, and the accepted values for each parameter. Parameters can be either strings or files. Each string parameter has a regular expression that defines the set of accepted values and a default value. Furthermore, a service defines which program will be invoked with the given parameters: the problem *manager* (also called the *evaluator*). The attachments are just regular files in a folder on the file system.

The description of a problem is contained in a file called `meta.yaml`, which is a YAML [2] file. The file contains the description of all the services, and their parameters, and the directory of the attachments. An example of a `meta.yaml` file is shown in Figure 1. Thus, a problem in *Turing Arena light* is represented by a folder containing a `meta.yaml` file, and all the files and subdirectories needed for services and attachments.

4.2 Server

After the problem manager, there is the server. The server is the beating heart of *Turing Arena light*: its role is to accept incoming connections from the clients, spawn the problem manager corresponding to the client requested problem and service, passing to it the parameters specified by the client, and finally connect the standard input and output of the problem manager to the client.

Note that up to this point, *Turing Arena light* is merely a specification of how the problem is defined and how the interaction between the problem manager and the solution should happen. This opens up the possibility of having multiple implementations of the *Turing Arena light* framework, since the specification is very simple and does not require any particular technology, such as sandboxing.

Currently, there is only one implementation of the *Turing Arena light* framework, which is `rtal` (*Rust Turing Arena light*). It is written in Rust [21], and it is the reference implementation of *Turing Arena light*. The server component, `rtald`, is a small program that, given a folder containing problems, listens for incoming connections from the clients, and spawns the correct problem manager, and relays the standard input and output of the problem manager to the client via a protocol based on WebSockets [10].

4.3 Client

On the other side of the network[2] there is the client. The client is the program that the contestant runs on their machine to connect to the server and interact with the problem manager. Its role is to connect to the server, send the request for a problem and a service, send the string and file parameters for the service, and finally spawn and attach itself to the standard input and output of the solution running on the local machine of the contestant.

Once everything is up and running, the client will send the standard output of the solution to the server, which will relay it to the problem manager, and forward on the standard input of the solution all the incoming data from the server. Basically, the client is a proxy that connects the standard input and output of the solution to the server.

Like the server, there is also a `rtal` component for the client, also called `rtal`. This client component is a command line program that takes as parameters the address of the server, the problem and the service, and the parameters for the service. It also takes the command to run the solution. The client will then connect to the server, send the request for the problem and service, and spawn the solution with the given command, proxying the data between the solution and the server.

4.4 User interface

As far as the contestant is concerned, what they must do is to write a solution to the problem in their favourite programming language. The only requirement is that it reads from the standard input and writes to the standard output. To read the problem statement, the contestant can download the attachments of the problem using the client. The client will download the attachments and save them on the local machine of the contestant.

Once the solution is ready, the contestant can run the client passing the right parameters, including the command to run their solution. The client will then connect to the server, send the request for the problem and service, and spawn the solution with the given command. Note that the solution is spawned and run on the local machine of the contestant, which means that the contestant has full freedom on which files it can read and write, which resources it can use, and so on. This is unlike other contest management systems that support real-time interaction, where the solution is run on a sandboxed environment on the server.

The ability to run the solution on the local machine opens to many possibilities. For example, the contestant can precompute some large set of data, save it on their machine, and then use it during the interaction with the problem manager to speed up the computation. Another example is the potential to use external libraries, multithreading, or even GPU computation. All of this is possible because the solution is run on the local machine of the contestant, where they have full control, and not on the server.

5 Implementation details

As mentioned in the previous section, *Turing Arena light* currently has only one full implementation, which is *Rust Turing Arena light* (`rtal`). Like the name suggests, it is written in Rust [21]. The choice of language was motivated by the fact that Rust is a systems programming language, and thus it is well suited for writing low-level programs that need

[2] Which might even be on the same machine, if both the server and the client are running on the same machine.

```rust
pub const META: &str = "meta.yaml";

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Problem {
    pub name: String,
    pub root: PathBuf,
    pub meta: Meta,
}

#[derive(Debug, Default, Serialize, Deserialize, Clone)]
pub struct Meta {
    pub public_folder: PathBuf,
    pub services: HashMap<String, Service>,
}

#[derive(Debug, Default, Serialize, Deserialize, Clone)]
pub struct Service {
    pub evaluator: Vec<String>,
    pub args: Option<HashMap<String, Arg>>,
    pub files: Option<Vec<String>>,
}

#[derive(Debug, Serialize, Deserialize, Clone)]
pub struct Arg {
    #[serde(with = "serde_regex")]
    pub regex: Regex,
    pub default: Option<String>,
}
```

Figure 3 Problem description definition in *Rust Turing Arena light*.

to interact with the operating system and other programs. Furthermore, one key factor is portability: Rust is a compiled language whose compiled binaries require only minimal external dependencies to run, which makes it ideal to produce distributable binaries. This is important because *Turing Arena light* is meant to be used by students, which might not have the technical knowledge to install and configure a complex system. Having a single binary that can be downloaded and run without any configuration is a big advantage.

The implementation of *Turing Arena light* is split into three components: the server (`rtald`), the client (`rtal`), and the checker (`rtalc`). All three components share some common parts. The main one is the problem description definition, also known as the `meta.yaml` file. The definition can be found in Figure 3. The definition is written using Rust structures which are then serialized to and deserialized from YAML using *serde* [29], a serialization framework for Rust. `rtalc` is a small independent command-line program that takes as input a directory containing the problem description, and checks that the description is valid and matches the content of the directory. This is useful to check that the problem description is correct before uploading it to the server.

The two main jobs of the client and the server are process spawning and networking. For both of these tasks, `rtal` and `rtald` use the *tokio* [30] library, which is a framework for writing asynchronous programs in Rust. For the process spawning part, there is nothing particularly interesting: the server spawns the problem manager, and the client spawns the

```rust
pub const MAGIC: &str = "rtal";
pub const VERSION: u64 = 4;

#[derive(Serialize, Deserialize, Debug)]
pub enum Request {
    Handshake {
        magic: String,
        version: u64,
    },
    MetaList {},
    Attachment {
        problem: String,
    },
    ConnectBegin {
        problem: String,
        service: String,
        args: HashMap<String, String>,
        tty: bool,
        token: Option<String>,
        files: Vec<String>,
    },
    ConnectStop {},
}

#[derive(Serialize, Deserialize, Debug)]
pub enum Reply {
    Handshake { magic: String, version: u64 },
    MetaList { meta: HashMap<String, Meta> },
    Attachment { status: Result<(), String> },
    ConnectBegin { status: Result<Vec<String>, String> },
    ConnectStart { status: Result<(), String> },
    ConnectStop { status: Result<Vec<String>, String> },
}
```

Figure 4 Network protocol definition in *Rust Turing Arena light*.

solution. They then, through *tokio*, manage the channels of the standard input and output of the spawned processes. All the internal communication within the server and the client is done using the actor threading model [15, 16].

For the networking part, the communication protocol between the server and the client is based on WebSockets [10]. The protocol definition is shown in Figure 4. The protocol is based on JSON [4] messages, which are serialized and deserialized using *serde*. These messages are then exchanged between the server and the client using WebSockets. The interaction between the server and the client is shown in Figure 2. Using WebSockets enables a client of *Turing Arena light* to be implemented as a web application.

Both `rtal` and `rtald` run their spawned processes in an unsandboxed environment. This is done to avoid the complexity of sandboxing, but we argue that it does not pose a major security risk. The reason is that, for the client, the program being run is the contestant's own written solution, which is run on their local machine. Thus, the contestant has full control over the program, and can do whatever they want with it. For the server, the program being run is the problem manager, which is written by the problem setter. Thus, as long as the

```python
class TC:
  def __init__(self, data, time_limit=1):
    self.data = data
    self.tl = time_limit

  def run(self, gen_tc, check_tc):
    output = open(join(environ["TAL_META_OUTPUT_FILES"], "result.txt"
        ), "w")
    total_tc = sum(map(lambda x: x[0], self.data))
    print(total_tc, flush=True)
    tc_ok = 0
    tcn = 1
    for subtask in range(len(self.data)):
      for tc in range(self.data[subtask][0]):
        tc_data = gen_tc(*self.data[subtask][1])
        stdout.flush()
        start = time()
        try:
          ret = check_tc(*tc_data)
          msg = None
          if isinstance(ret, tuple):
            result = ret[0]
            msg = ret[1]
          else:
            result = ret
          if time() - start > self.tl:
            print(f"Case #{tcn:03}: TLE", file=output)
          elif result:
            print(f"Case #{tcn:03}: AC", file=output)
            tc_ok += 1
          else:
            print(f"Case #{tcn:03}: WA", file=output)
          if msg is not None:
            print(file=output)
            print(msg, file=output)
            print(file=output)
        except Exception as e:
          print(f"Case #{tcn:03}: RE", file=output)
          print(file=stderr)
          print("".join(traceback.format_tb(e.__traceback__)), e,
              file=stderr)
        tcn += 1
    print(file=output)
    print(f"Score: {tc_ok}/{total_tc}", file=output)
    output.close()
```

Figure 5 Snippet of the python version of the competitive-programming like problem manager library for *Turing Arena light*.

problem setter is trusted, there is no need to sandbox the problem manager. This is usually the case, as the problem setter is the one who also is responsible for the server where the `rtald` program is running. If this is not the case, then `rtald` can be run in a virtualized environment, such as a Docker container [22], to mitigate the risk of a bug in the problem manager that could cause unauthorized access to the server.

5.1 Problem manager libraries

So far we have discussed the architecture, the design and the implementation of *Turing Arena light*. However, we have not yet discussed how the problem manager is implemented. As mentioned in the previous sections, the problem manager is a program that interacts with the solution, and gives a verdict at the end of the interaction. The problem manager, just like the solution, has to communicate with its counterpart, which is the solution, using the standard input and output channels. Thus, the problem manager has full freedom on how to interact with the solution, as long as it does so using the aforementioned channels.

While this grants the problem maker a great deal of freedom, it also means that the problem maker has to potentially write a lot of boilerplate code each time they want to implement a new problem. To mitigate this problem, a problem maker can create a library of utilities that can be used to implement the problem manager. This library can be based on a particular style of problems, so that the problem maker can offer a consistent experience to the contestants.

In our case, we wrote a library called `tc.py`. A snippet of the library is shown in Figure 5. This library allows to write a *old-Google-Code-Jam* like problem by only writing the code essential to the problem, and leaving all the boilerplate code to the library. What the manager has to implement is a function that generates a test case, and a function that evaluates the solution given by the contestant on a test case. The library will then take care of the rest, including enforcing the time limit, generating the right number of test cases, and assigning and storing the score for the solution. Note that with the *Turing Arena light* there is no way to enforce the memory limit, as the solution is run on the local machine of the contestant. However, the time limit can be enforced by measuring how much time passes between the sending of the input and the receiving of the output. While this is not a very precise measurement, it is good enough for distinguishing between solutions that have very different computational complexities.

```
CREATE TABLE users (
    id TEXT PRIMARY KEY,
    name TEXT NOT NULL,
    other TEXT
);

CREATE TABLE problems (
    name TEXT PRIMARY KEY
);

CREATE TABLE submissions (
    id INTEGER PRIMARY KEY,
    user_id TEXT NOT NULL,
    problem TEXT NOT NULL,
    score INTEGER NOT NULL,
    source BLOB NOT NULL,
    address TEXT,
    FOREIGN KEY (user_id) REFERENCES users(id),
    FOREIGN KEY (problem) REFERENCES problems(name)
);
```

Figure 6 SQLite schema for database used by `tc.py` and `tc.rs`.

As the name suggests, the `tc.py` library is written in Python [36], and it is meant to be used with problem managers written in Python. This works great for problems where the optimal solution plays well with Python, however in problems where the performance of the solution is critical, having the problem manager written in Python may make the evaluation of the contestant's output too slow. To mitigate this problem, we ported the `tc.py` library to Rust, thus creating the `tc.rs` library [34]. By using Rust as the programming language for the problem manager, the whole execution of the problem manager is much faster. The functionality of the two libraries is the same, and they are interoperable with each other. This means that in a single contest, the problem maker can use both Python and Rust problem managers.

Turing Arena light has no built-in support for saving the results of the contest, as this job is left to the problem manager. This is done to allow the problem maker to have full control over how the results are saved. In `tc.py` and `tc.rs` we implemented a simple database that saves the results of the contest in a SQLite [25] database. The schema of the database is shown in Figure 6. The database provides a way to save the results of the contest, and it enables contestants to see their position in the ranking during the contest, using a service defined in *Turing Arena light*.

6 Graphical user interface

The Rust implementation of *Turing Arena light* only comes with a command line interface for the client. While this is enough to run the contest, it is not very user friendly. Contestants have to remember the right parameters to pass to the client, and the less experienced ones might have trouble working with a terminal. To mitigate this problem, a graphical user interface for the client was developed.

A web application was developed as a new client for *Turing Arena light* [33]. A screenshot of the application is shown in Figure 7. It was developed using the *Angular* framework [14], and it is written in TypeScript [35]. The peculiar thing about this application is that aside

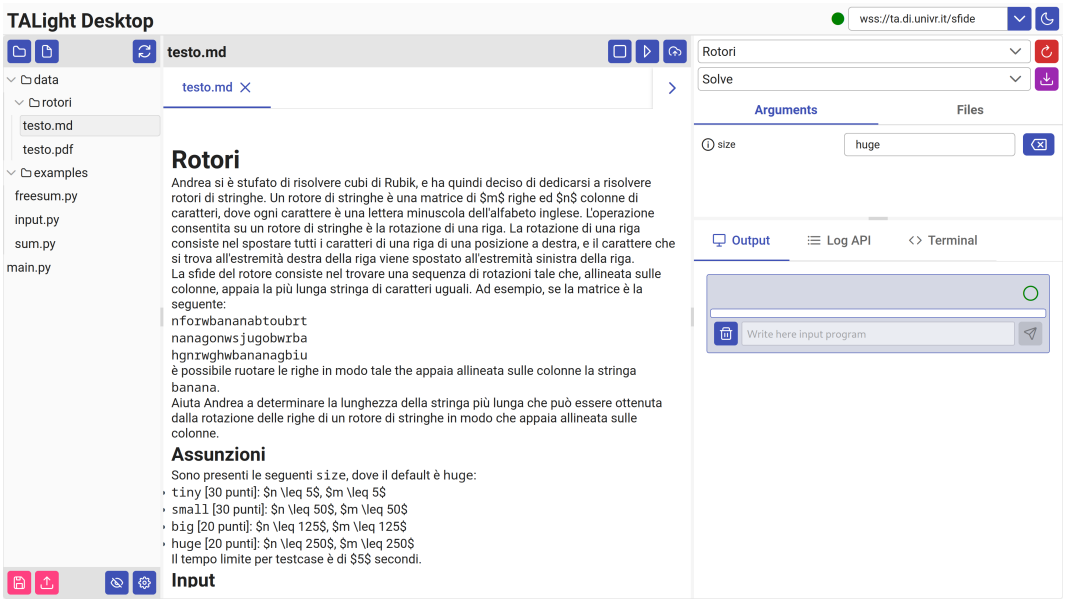

Figure 7 Graphical user interface of *Turing Arena light*.

from offering all the functionalities of the command line client, it also offers a way to write the solution directly in the browser. Not only that, but the solution is run directly in the browser, without the need to install any additional software. This functionality is currently only available for Python solutions, but it could be extended to other languages as well. To do this, the Python interpreter has been compiled to JavaScript, using *Pyodide* [27]. This allows to run Python code directly in the browser. Thus, the contestant can do everything from an integrated environment in its browser.

Aside from running the solution in the browser, the web application also implements an emulated file system within the browser. This allows the contestant to send file parameters and receive file attachments and file outputs, all from the browser. Another useful feature that derives from having a file system is the ability to save and restore the working environment. This is useful for example when the contestant is working on a problem, and they want to save their progress and continue working on it later. Another scenario is when a template is provided to the contestant, and they can start working directly on it. The file system can be exported as a tar archive, or can be stored in the cloud using either GitHub [11], Google Drive [13], or OneDrive [24]. They can be later imported back from a tar archive or from the cloud, specifically from GitHub.

7 Future directions

Turing Arena light has been developed enough to be used in a real-world classroom setting, and it has been used in the course of *Competitive Programming* at the University of Verona. It has been used for both the laboratory lessons and the exams, and it has been well received by the students. However, there is still a debate to be had in which direction *Turing Arena light* should move forward.

While the extreme flexibility of *Turing Arena light* made it possible to experiment a lot with different kinds of problems, it also made it difficult to find a common ground on which to standardize some common features, without having all of the problem manager libraries reimplement them. One such feature is the ability to save the results of the contest. While *Turing Arena light* does not have any built-in support for saving the results of the contest, it is possible to implement it in the problem manager. However, this means that each problem manager has to reimplement the same functionality, which is not ideal.

Moreover, some feature are implementable only by standardizing them at the core of *Turing Arena light*. One such feature is the ability of accurately measuring the time consumed by the solution. Right now, the time used by the solution is measured by measuring the time between the sending of the input and the receiving of the output. However, this is not a very accurate measurement, as it does not take into account the time spent sending and receiving the packets over the network. This is not a problem when the server and the client are on the same local network, as it happened in the course of *Competitive Programming*, but it becomes a problem when the server and the client are on different networks, such as when the server is on the Internet.

There is a solution to mitigate this problem, which is to encrypt the data, send it, then start the clock and send the decryption key. Doing it this way, one can eliminate the time spent sending the data, which can be a significant amount of time when the input is big. However, to implement such a solution, it would require to have some mechanism to make the problem manager and the core communicate on a meta-level to require this functionality from the core. However, such mechanism could cause a narrowing of the flexibility of *Turing Arena light*.

We currently offer client implementations in Rust and a web-based interface, educators and users may prefer clients in other programming languages. Implementing new RTAL clients is relatively straightforward through two approaches:

- Binding Generation: The recommended approach involves generating language bindings from the core Rust library. This method ensures automatic compatibility with future protocol updates and requires minimal maintenance. Our Python implementation already follows this pattern, with the complete binding implementation available in our repository (`py.rs`[3]).
- Protocol Reimplementation: Alternatively, developers can independently implement the WebSocket-based communication protocol used between server and client. This is feasible due to the protocol's simplicity – it consists of only 11 distinct message types as defined in our protocol specification (`proto.rs`[4]). However, this approach requires manual updates to each client implementation whenever the protocol evolves.

Finally, while the command-line interface has worked great for the course of *Competitive Programming*, it is not very probable that it would be fine for other courses with less *programming-focused* students. Thus, the development of the graphical user interface continues, and it is planned to be tested in the next iteration of the course of *Competitive Programming*, and possibly in other courses with more *less specialized* students.

References

1 Giorgio Audrito, Tania Di Mascio, Paolo Fantozzi, Luigi Laura, Gemma Martini, Umberto Nanni, and Marco Temperini. Recommending tasks in online judges. In *Advances in Intelligent Systems and Computing*, pages 129–136. Springer International Publishing, Cham, 2020. doi:10.1007/978-3-030-23990-9_16.

2 Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml™) version 1.1. *Working Draft 2008-05*, 11, 2009.

3 Codeforces. URL: https://codeforces.com/.

4 Douglas Crockford. The application/json media type for javascript object notation (json). Technical report, IETF, 2006. doi:10.17487/RFC4627.

5 Jaap Eldering, Thijs Kinkhorst, and Peter van de Warken. Dom judge–programming contest jury system. 2020, 2010.

6 Emma Enstrom, Gunnar Kreitz, Fredrik Niemela, Pehr Soderman, and Viggo Kann. Five years with kattis — using an automated assessment system in teaching. In *2011 Frontiers in Education Conference (FIE)*, pages T3J–1–T3J–6. IEEE, October 2011. doi:10.1109/FIE.2011.6142931.

7 P Fantozzi and L Laura. Recommending tasks in online judges using autoencoder neural networks. *Olymp. Inform.*, December 2020. doi:10.15388/ioi.2020.05.

8 Paolo Fantozzi and Luigi Laura. Collaborative recommendations in online judges using autoencoder neural networks. In *Advances in Intelligent Systems and Computing*, pages 113–123. Springer International Publishing, Cham, 2021. doi:10.1007/978-3-030-53036-5_12.

9 Paolo Fantozzi and Luigi Laura. A dynamic recommender system for online judges based on autoencoder neural networks. In *Methodologies and Intelligent Systems for Technology Enhanced Learning, 10th International Conference. Workshops*, pages 197–205. Springer International Publishing, Cham, 2021. doi:10.1007/978-3-030-52287-2_20.

10 Ian Fette and Alexey Melnikov. The websocket protocol, 2011. doi:10.17487/RFC6455.

11 Github. URL: https://github.com/.

[3] https://github.com/romeorizzi/TALight/blob/v0.2.5/rtal/src/py.rs
[4] https://github.com/romeorizzi/TALight/blob/v0.2.5/rtal/src/proto.rs

12 Google code jam. URL: https://codingcompetitionsonair.withgoogle.com/#code-jam.
13 Google drive. URL: https://drive.google.com/.
14 Brad Green and Shyam Seshadri. *AngularJS*. O'Reilly Media, Inc., 2013.
15 Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, 1973. URL: http://ijcai.org/Proceedings/73/Papers/027B.pdf.
16 C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. doi:10.1145/359576.359585.
17 Fariha Iffath, A. S. M. Kayes, Md. Tahsin Rahman, Jannatul Ferdows, Mohammad Shamsul Arefin, and Md. Sabir Hossain. Online judging platform utilizing dynamic plagiarism detection facilities. *Comput.*, 10(4):47, April 2021. doi:10.3390/computers10040047.
18 José Paulo Leal and Fernando M. A. Silva. Mooshak: a web-based multi-site programming contest system. *Softw. Pract. Exp.*, 33(6):567–581, 2003. doi:10.1002/spe.522.
19 Stefano Maggiolo and Giovanni Mascellani. Introducing CMS: A contest management system. *Olympiads in Informatics*, 6, 2012.
20 Stefano Maggiolo, Giovanni Mascellani, and Luca Wehrstedt. Cms: a growing grading system. *Olympiads in Informatics*, page 123, 2014.
21 Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014. doi:10.1145/2663171.2663188.
22 Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux j*, 239(2):2, 2014.
23 Meta hacker cup. URL: https://www.facebook.com/codingcompetitions/hacker-cup.
24 Onedrive. URL: https://onedrive.live.com/.
25 Michael Owens. *The definitive guide to SQLite*. Springer, 2006.
26 Minh Tuan Pham and Tan Bao Nguyen. The domjudge based online judge system with plagiarism detection. In *2019 IEEE-RIVF International Conference on Computing and Communication Technologies, RIVF 2019, Danang, Vietnam, March 20-22, 2019*, pages 1–6. IEEE, March 2019. doi:10.1109/RIVF.2019.8713763.
27 Pyodide. URL: https://pyodide.org/.
28 Miguel A Revilla, Shahriar Manzoor, and Rujia Liu. Competitive learning in informatics: The uva online judge experience. *Olympiads in Informatics*, 2:131–148, 2008.
29 Serde. URL: https://serde.rs/.
30 Tokio. URL: https://tokio.rs/.
31 Turing arena. URL: https://github.com/turingarena/turingarena.
32 Turing arena light. URL: https://github.com/romeorizzi/TALight.
33 Turing arena light desktop. URL: https://talco-team.github.io/TALightDesktop/.
34 Turing arena light rust utilities. URL: https://github.com/dariost/tal-utils-rs.
35 Typescript. URL: https://www.typescriptlang.org/.
36 Guido Van Rossum et al. Python programming language. In *USENIX annual technical conference*, volume 41, pages 1–36. Santa Clara, CA, 2007.
37 Szymon Wasik, Maciej Antczak, Jan Badura, Artur Laskowski, and Tomasz Sternal. A survey on online judge systems and their applications. *ACM Comput. Surv.*, 51(1):3:1–3:34, January 2018. doi:10.1145/3143560.
38 Yutaka Watanobe, Md. Mostafizer Rahman, Taku Matsumoto, Rage Uday Kiran, and Penugonda Ravikumar. Online judge system: Requirements, architecture, and experiences. *Int. J. Softw. Eng. Knowl. Eng.*, 32(6):917–946, June 2022. doi:10.1142/S0218194022500346.