CMS Documentation

Release 1.4.dev0

The CMS development team

Aug 17, 2018

Contents

CHAPTER 1

Introduction

CMS (Contest Management System) is a software for organizing programming contests similar to well-known international contests like the IOI (International Olympiad in Informatics). It was written by and it received contributions from people involved in the organization of similar contests on a local, national and international level, and it is regularly used for such contests in many different countries. It is meant to be secure, extendable, adaptable to different situations and easy to use.

CMS is a complete, tested and well proved solution for managing a contest. However, it only provides limited tools for the development of the task data belonging to the contest (task statements, solutions, testcases, etc.). Also, the configuration of machines and network resources that host the contest is a responsibility of the contest administrators.

1.1 General structure

The system is organized in a modular way, with different services running (potentially) on different machines, and providing extendability via service replications on several machines.

The state of the contest is wholly kept on a PostgreSQL database (other DBMSs are not supported, as CMS relies on the Large Object (LO) feature of PostgreSQL). It is unlikely that in the future we will target different databases.

As long as the database is operating correctly, all other services can be started and stopped independently. For example, the administrator can quickly replace a broken machine with an identical one, which will take its roles (without having to move information from the broken machine). Of course, this also means that CMS is completely dependent on the database to run. In critical contexts, it is necessary to configure the database redundantly and be prepared to rapidly do a fail-over in case something bad happens. The choice of PostgreSQL as the database to use should ease this part, since there are many different, mature and well-known solutions to provide such redundance and fail-over procedures.

1.2 Services

CMS is composed of several services, that can be run on a single or on many servers. The core services are:

- LogService: collects all log messages in a single place;
- ResourceService: collects data about the services running on the same server, and takes care of starting all of them with a single command;

- Checker: simple heartbeat monitor for all services;
- EvaluationService: organizes the queue of the submissions to compile or evaluate on the testcases, and dispatches these jobs to the workers;
- Worker: actually runs the jobs in a sandboxed environment;
- ScoringService: collects the outcomes of the submissions and computes the score;
- ProxyService: sends the computed scores to the rankings;
- PrintingService: processes files submitted for printing and sends them to a printer;
- ContestWebServer: the webserver that the contestants will be interacting with;
- AdminWebServer: the webserver to control and modify the parameters of the contests.

Finally, RankingWebServer, whose duty is of course to show the ranking. This webserver is - on purpose - separated from the inner core of CMS in order to ease the creation of mirrors and restrict the number of people that can access services that are directly connected to the database.

Each of the core services is designed to be able to be killed and reactivated in a way that keeps the consistency of data, and does not block the functionalities provided by the other services.

Some of the services can be replicated on several machine: these are ResourceService (designed to be run on every machine), ContestWebServer and Worker.

In addition to services, CMS includes many command line tools. They help with importing, exporting and managing of contest data, and with testing.

1.3 Security considerations

With the exception of RWS, there are no cryptographic or authentication schemes between the various services or between the services and the database. Thus, it is mandatory to keep the services on a dedicated network, properly isolating it via firewalls from contestants or other people's computers. This sort of operations, like also preventing contestants from communicating and cheating, is responsibility of the administrator and is not managed by CMS itself.

CHAPTER 2

Installation

2.1 Dependencies and available compilers

These are our requirements (in particular we highlight those that are not usually installed by default) - previous versions may or may not work:

- build environment for the programming languages allowed in the competition;
- PostgreSQL >= 9.4;
- GNU compiler collection (in particular the C compiler `gcc`);
- Python 2.7 or 3.6;
- libcg;
- TeX Live (only for printing);
- a2ps (only for printing).

You will also require a Linux kernel with support for control groups and namespaces. Support has been in the Linux kernel since 2.6.32. Other distributions, or systems with custom kernels, may not have support enabled. At a minimum, you will need to enable the following Linux kernel options: `CONFIG_CGROUPS`, `CONFIG_CGROUP_CPUACCT`, `CONFIG_MEMCG` (previously called as `CONFIG_CGROUP_MEM_RES_CTLR`), `CONFIG_CPUSETS`, `CONFIG_PID_NS`, `CONFIG_IPC_NS`, `CONFIG_NET_NS`. It is anyway suggested to use Linux kernel version at least 3.8.

Then you require the compilation and execution environments for the languages you will use in your contest:

- GNU compiler collection (for C, C++ and Java, respectively with executables `gcc`, `g++` and `gcj`);
- alternatively, for Java, your choice of a JDK, for example OpenJDK (but any other JDK behaving similarly is fine, for example Oracle's);
- Free Pascal (for Pascal, with executable `fpc`);
- Python >= 2.7, < 3.0 (for Python, with executable `python2`; note though that this must be installed anyway because it is required by CMS itself);
- PHP >= 5 (for PHP, with executable `php`);
- Glasgow Haskell Compiler (for Haskell, with executable `ghc`);
- Rust (for Rust, with executable `rustc`);

- C# (for C#, with executable `mcs`).

All dependencies can be installed automatically on most Linux distributions.

2.1.1 Ubuntu

On Ubuntu 18.04, one will need to run the following script to satisfy all dependencies:

```
# Feel free to change OpenJDK packages with your preferred JDK.
sudo apt-get install build-essential openjdk-8-jdk-headless fp-compiler \
    postgresql postgresql-client python3.6 cppreference-doc-en-html \
    cgroup-lite libcap-dev

# Only if you are going to use pip/venv to install python dependencies
sudo apt-get install python3.6-dev libpq-dev libcups2-dev libyaml-dev \
    libffi-dev python3-pip

# Optional
sudo apt-get install nginx-full python2.7 php7.2-cli php7.2-fpm \
    phppgadmin texlive-latex-base a2ps gcj-jdk haskell-platform rustc \
    mono-mcs
```

The above commands provide a very essential Pascal environment. Consider installing the following packages for additional units: *fp-units-base*, *fp-units-fcl*, *fp-units-misc*, *fp-units-math* and *fp-units-rtl*.

2.1.2 Arch Linux

On Arch Linux, unofficial AUR packages can be found: cms or cms-git. However, if you do not want to use them, the following command will install almost all dependencies (some of them can be found in the AUR):

```
sudo pacman -S base-devel jdk8-openjdk fpc postgresql postgresql-client \
    python libcap

# Install the following from AUR.
# https://aur.archlinux.org/packages/libcgroup/
# https://aur.archlinux.org/packages/cppreference/

# Only if you are going to use pip/venv to install python dependencies
sudo pacman -S --needed postgresql-libs libcups libyaml python-pip

# Optional
sudo pacman -S --needed nginx python2 php php-fpm phppgadmin texlive-core \
    a2ps ghc rust mono
```

2.2 Preparation steps

Download CMS 1.4.dev0 from GitHub as an archive, then extract it on your filesystem. You should then access the `cms` folder using a terminal.

Warning: If you decided to `git clone` the repository instead of downloading the archive, and you didn't use the `--recursive` option when cloning, then **you need** to issue the following command to fetch the source code of the sandbox:

```
git submodule update --init
```

In order to run CMS there are some preparation steps to run (like installing the sandbox, compiling localization files, creating the `cmsuser`, and so on). You can either do all these steps by hand or you can run the following command:

```
sudo python3 prerequisites.py install
```

This script will add you to the `cmsuser` group if you answer `Y` when asked. If you want to handle your groups by yourself, answer `N` and then run:

```
sudo usermod -a -G cmsuser <your user>
```

You can verify to be in the group by issuing the command:

```
groups
```

Remember to logout, to make the change effective.

Warning: Users in the group `cmsuser` will be able to launch the `isolate` program with root permission. They may exploit this to gain root privileges. It is then imperative that no untrusted user is allowed in the group `cmsuser`.

2.3 Installing CMS and its Python dependencies

There are a number of ways to install CMS and its Python dependencies:

2.3.1 Method 1: Global installation with pip

There are good reasons to install CMS and its Python dependencies via pip (Python Package Index) instead of your package manager (e.g. apt-get). For example: two different Linux distro (or two different versions of the same distro) may offer two different versions of `python-sqlalchemy`. When using pip, you can choose to install a *specific version* of `sqlalchemy` that is known to work correctly with CMS.

Assuming you have `pip` installed, you can do this:

```
sudo pip3 install -r requirements.txt
sudo python3 setup.py install
```

This command installs python dependencies globally. Note that on some distros, like Arch Linux, this might interfere with the system package manager. If you want to perform the installation in your home folder instead, then you can do this instead:

```
pip3 install --user -r requirements.txt
python3 setup.py install --user
```

2.3.2 Method 2: Virtual environment

An alternative method to perform the installation is with a virtual environment, which is an isolated Python environment that you can put wherever you like and that can be activated/deactivated at will.

You will need to create a virtual environment somewhere in your filesystem. For example, let's assume that you decided to create it under your home directory (as `~/cms_venv`):

```
python3 -m venv ~/cms_venv
```

To activate it:

```
source ~/cms_venv/bin/activate
```

After the activation, the `pip` command will *always* be available (even if it was not available globally, e.g. because you did not install it). In general, every python command (python, pip) will refer to their corresponding virtual version. So, you can install python dependencies by issuing:

```
pip3 install -r requirements.txt
python3 setup.py install
```

Note: Once you finished using CMS, you can deactivate the virtual environment by issuing:

```
deactivate
```

2.3.3 Method 3: Using `apt-get` on Ubuntu

Warning: It is usually possible to install python dependencies using your Linux distribution's package manager. However, keep in mind that the version of each package is controlled by the package mantainers and could be too new or too old for CMS. **On Ubuntu, this is generally not the case** since we try to build on the python packages that are available for the current LTS version.

To install CMS and its Python dependencies on Ubuntu, you can issue:

```
sudo python3 setup.py install

sudo apt-get install python3-setuptools python3-tornado python3-psycopg2 \
     python3-sqlalchemy python3-psutil python3-netifaces python3-crypto \
     python3-six python3-bs4 python3-coverage python3-mock python3-requests \
     python3-werkzeug python3-gevent python3-bcrypt python3-chardet patool \
     python3-babel python3-xdg python3-future python3-jinja2

# Optional.
# sudo apt-get install python3-yaml python3-sphinx python3-cups python3-pypdf2
```

2.3.4 Method 4: Using `pacman` on Arch Linux

Warning: It is usually possible to install python dependencies using your Linux distribution's package manager. However, keep in mind that the version of each package is controlled by the package mantainers and could be too new or too old for CMS. **This is especially true for Arch Linux**, which is a bleeding edge distribution.

To install CMS python dependencies on Arch Linux (again: assuming you did not use the aforementioned AUR packages), you can issue:

```
sudo python3 setup.py install

sudo pacman -S --needed python-setuptools python-tornado python-psycopg2 \
     python-sqlalchemy python-psutil python-netifaces python-crypto \
     python-six python-beautifulsoup4 python-coverage python-mock \
     python-requests python-werkzeug python-gevent python-bcrypt \
     python-chardet python-babel python-xdg python-future python-jinja
```

(continues on next page)

(continued from previous page)

```
# Install the following from AUR.
# https://aur.archlinux.org/packages/patool/

# Optional.
# sudo pacman -S --needed python-yaml python-sphinx python-pycups
# Optionally install the following from AUR.
# https://aur.archlinux.org/packages/python-pypdf2/
```

2.4 Configuring the worker machines

Worker machines need to be carefully set up in order to ensure that evaluation results are valid and consistent. Just running the evaluations under isolate does not achieve this: for example, if the machine has an active swap partition, memory limit will not be honored.

Apart from validity, there are many possible tweaks to reduce the variability in resource usage of an evaluation.

We suggest following isolate's guidelines for reproducible results.

2.5 Running CMS non-installed

To run CMS without installing it in the system, you need first to build the prerequisites:

```
python3 prerequisites.py build
```

There are still a few steps to complete manually in this case. First, add CMS and isolate to the path and create the configuration files:

```
export PATH=$PATH:./isolate/
export PYTHONPATH=./
cp config/cms.conf.sample config/cms.conf
cp config/cms.ranking.conf.sample config/cms.ranking.conf
```

Second, perform these tasks (that require root permissions):

- create the `cmsuser` user and a group with the same name;
- add your user to the `cmsuser` group;
- set isolate to be owned by root:cmsuser, and set its suid bit.

For example:

```
sudo useradd cmsuser
sudo usermod -a -G cmsuser <your user>
sudo chown root:cmsuser ./isolate/isolate
sudo chmod u+s ./isolate/isolate
```

2.6 Updating CMS

As CMS develops, the database schema it uses to represent its data may be updated and new versions may introduce changes that are incompatible with older versions.

To preserve the data stored on the database you need to dump it on the filesystem using `cmsDumpExporter` **before you update CMS** (i.e. with the old version).

You can then update CMS and reset the database schema by running:

```
cmsDropDB
cmsInitDB
```

To load the previous data back into the database you can use `cmsDumpImporter`: it will adapt the data model automatically on-the-fly (you can use `cmsDumpUpdater` to store the updated version back on disk and speed up future imports).

CHAPTER 3

Running CMS

3.1 Configuring the DB

The first thing to do is to create the user and the database. If you're on Ubuntu, you need to login as the postgres user first:

```
sudo su - postgres
```

Then, to create the user (which does not need to be a superuser, nor be able to create databases nor roles) and the database, you need the following commands:

```
createuser --username=postgres --pwprompt cmsuser
createdb --username=postgres --owner=cmsuser cmsdb
psql --username=postgres --dbname=cmsdb --command='ALTER SCHEMA public OWNER TO␣
↪cmsuser'
psql --username=postgres --dbname=cmsdb --command='GRANT SELECT ON pg_largeobject␣
↪TO cmsuser'
```

The last two lines are required to give the PostgreSQL user some privileges which it does not have by default, despite being the database owner.

Then you may need to adjust the CMS configuration to contain the correct database parameters. See *Configuring CMS*.

Finally you have to create the database schema for CMS, by running:

```
cmsInitDB
```

Note: If you are going to use CMS services on different hosts from the one where PostgreSQL is running, you also need to instruct it to accept the connections from the services. To do so, you need to change the listening address of PostgreSQL in `postgresql.conf`, for example like this:

```
listen_addresses = '127.0.0.1,192.168.0.x'
```

Moreover, you need to change the HBA (a sort of access control list for PostgreSQL) to accept login requests from outside localhost. Open the file `pg_hba.conf` and add a line like this one:

```
host  cmsdb  cmsuser  192.168.0.0/24  md5
```

3.2 Configuring CMS

There are two configuration files, one for CMS itself and one for the rankings. Samples for both files are in the directory config/. You want to copy them to the same file names but without the `.sample` suffix (that is, to `config/cms.conf` and `config/cms.ranking.conf`) before modifying them.

- `cms.conf` is intended to be the same on all machines; all configurations options are explained in the file; of particular importance is the definition of `core_services`, that specifies where and how many services are going to be run, and the connecting line for the database, in which you need to specify the name of the user created above and its password.
- `cms.ranking.conf` is not necessarily meant to be the same on each server that will host a ranking, since it just controls settings relevant for one single server. The addresses and log-in information of each ranking must be the same as the ones found in `cms.conf`.

These files are a pretty good starting point if you want to try CMS. There are some mandatory changes to do though:

- you must change the connection string given in `database`; this usually means to change username, password and database with the ones you chose before;
- if you are running low on disk space, you may want to make sure `keep_sandbox` is set to `false`;

If you are organizing a real contest, you must also change `secret_key` to a random key (the admin interface will suggest one if you visit it when `secret_key` is the default). You will also need to think about how to distribute your services and change `core_services` accordingly. Finally, you should change the ranking section of `cms.conf`, and `cms.ranking.conf`, using non-trivial username and password.

Warning: As the name implies, the value of `secret_key` must be kept confidential. If a contestant knows it (for example because you are using the default value), they may be easily able to log in as another contestant.

The configuration files get copied automatically by the `prerequisites.py` script, so you can either run `sudo ./prerequisites.py install` again (answering "Y" when questioned about overwriting old configuration files) or you could simply edit the previously installed configuration files (which are usually found in `/usr/local/etc/` or `/etc/`), if you do not plan on running that command ever again.

3.3 Running CMS

Here we will assume you installed CMS. If not, you should replace all commands path with the appropriate local versions (for example, `cmsLogService` becomes ./scripts/cmsLogService).

At this point, you should have CMS installed on all the machines you want run services on, with the same configuration file, and a running PostgreSQL instance. To run CMS, you need a contest in the database. To create a contest, follow *these instructions*.

CMS is composed of a number of services, potentially replicated several times, and running on several machines. You can start all the services by hand, but this is a tedious task. Luckily, there is a service (ResourceService) that takes care of starting all the services on the machine it is running, limiting thus the number of binaries you have to run. Services started by ResourceService do not show their logs to the standard output; so it is expected that you run LogService to inspect the logs as they arrive (logs are also saved to disk). To start LogService, you need to issue, in the machine specified in cms.conf for LogService, this command:

```
cmsLogService 0
```

where `0` is the "shard" of LogService you want to run. Since there must be only one instance of LogService, it is safe to let CMS infer that the shard you want is the 0-th, and so an equivalent command is

```
cmsLogService
```

After LogService is running, you can start ResourceService on each machine involved, instructing it to load all the other services:

```
cmsResourceService -a
```

The flag `-a` informs ResourceService that it has to start all other services, and we have omitted again the shard number since, even if ResourceService is replicated, there must be only one of it in each machine. If you have a funny network configuration that confuses CMS, just give explicitly the shard number. In any case, ResourceService will ask you the contest to load, and will start all the other services. You should start see logs flowing in the LogService terminal.

Note that it is your duty to keep CMS's configuration synchronized among the machines.

You should now be able to start exploring the admin interface, by default at http://localhost:8889/. The interface is accessible with an admin account, which you need to create first using the AddAdmin command, for example:

```
cmsAddAdmin name
```

CMS will create an admin account with username "name" and a random password that will be printed by the command. You can log in with this credentials, and then use the admin interface to modify the account or add other accounts.

3.4 Recommended setup

Of course, the number of servers one needs to run a contest depends on many factors (number of participants, length of the contest, economical issues, more technical matters...). We recommend that, for fairness, each Worker runs an a dedicated machine (i.e., without other CMS services beyond ResourceService).

As for the distribution of services, usually there is one ResourceService for each machine, one instance for each of LogService, ScoringService, Checker, EvaluationService, AdminWebServer, and one or more instances of ContestWebServer and Worker. Again, if there are more than one Worker, we recommend to run them on different machines.

The developers of isolate (the sandbox CMS uses) provide a script, `isolate-check-environment` that verifies your system is able to produce evaluations as fair and reproducible as possible. We recommend to run it and follow its suggestions on all machines where a Worker is running. You can download it here.

We suggest using CMS over Ubuntu. Yet, CMS can be successfully run on different Linux distributions. Non-Linux operating systems are not supported.

We recommend using nginx in front of the (one or more) `cmsContestWebServer` instances serving the contestant interface. Using a load balancer is required when having multiple instances of `cmsContestWebServer`, but even in case of a single instance, we suggest using nginx to secure the connection, providing an HTTPS endpoint and redirecting it to `cmsContestWebServer`'s HTTP interface.

See config/nginx.conf.sample for a sample nginx configuration. This file probably needs to be adapted to your distribution if it is not Ubuntu: try to merge it with the file you find installed by default. For additional information see the official nginx documentation and examples. Note that without the `ip_hash` option some CMS features might not always work as expected.

3.5 Logs

When the services are running, log messages are streamed to the log service. This is the meaning of the log levels:

- debug: they are just for development; in the default configuration, they are not printed;

- info: they inform you on what is going on in the system and that everything is fine;
- warning: something went wrong or was slightly unexpected, but CMS knew how to handle it, or someone fed inappropriate data to CMS (by error or on purpose); you may want to check these as they may evolve into errors or unexpected behaviors, or hint that a contestant is trying to cheat;
- error: an unexpected condition that should not have happened; you are encouraged to take actions to fix them, but the service will continue to work (most of the time, ignoring the error and the data connected to it);
- critical: a condition so unexpected that the service is really startled and refuses to continue working; you are forced to take action because with high probability the service will continue having the same problem upon restarting.

Warning, error, and critical log messages are also displayed in the main page of AdminWebServer.

CHAPTER 4

Data model

CMS is used in many different settings: single onsite contests, with remote participations, short training camps, always-on training websites, university assignments... Its data model needs to support all of these, and therefore might be a bit surprising if you only think of the first use case.

In the following, we explain the main objects in CMS's data model.

4.1 Users

Users are the accounts for contestants; a user represents a person, which can participate to zero, one or many contests.

4.2 Participations

Participations contain the interactions of users and a contest. In particular all of the following are associated to a participation: the submissions sent, their results, questions asked, communications from contest admins.

4.3 Contests

A contest is a collection of tasks to be solved, and participations of users trying to solve them.

It is *very configurable* with respect of timing (start and end times, how much time each contestant has), logistic (how contestants login), permissions (what contestants can or cannot do, and how often), scoring, and more.

They are mainly thought as limited in duration to a few hours, but this is not a requirement: contests can be very long running.

4.4 Tasks and datasets

A task is one of the problems to solve within a contest. A task cannot be associated to more than one contest, but you can have tasks temporarily not associated to any.

Tasks store additional configurations that might override or alter the configurations at the contest level.

A task can have one or more datasets. The complete task data is shared between these two objects: task contains more contest-visible data, such as name, statement, constraints on the submissions; datasets instead contain instruction on how to compile, evaluate and score the submissions. The information in the task object should not change, as they are visible to the contestants and influence their interaction during the contest, whereas those in the dataset object could change without the contestants noticing.

The dataset used to display information to contestants is said to be active. For example, a second, inactive dataset can be created to fix an incorrect testcase; evaluation could progress in parallel until the inactive dataset is deemed correct, and eventually the switch to make it active could happen at once.

See the section on *task versioning* for more details on how to use datasets.

4.5 Submissions, results and tokens

Submissions are associated to a participation and a task. In addition to these, the judging of a submission depends on the dataset used in the compilation, evaluation and scoring phases. Therefore, a submission result is associated also to a specific dataset.

Similarly to submissions and submission results, we have user tests and user tests results. User tests are a way to allow contestants to run their source on the sandboxed environment, for example to diagnose errors or test timings.

Tokens are requests from a contestant to see additional details about the result of a submission they are associated to the submission (not to the submission result, so if the admins change dataset, the contestant can still see the detailed results).

4.6 Announcements, questions and messages

Announcements are contest-level communications sent by the admins and visible to all contestants.

Questions and messages instead are communication between a specific contestant and the admins. The difference is that questions are initiated by the contestants and expect an answer from the admins, whereas messages are initiated by the admins and contestants cannot reply.

4.7 Admins

CMS stores administrative accounts to use AdminWebServer. Accounts can be useful to offer different permissions to different sets of people. There are three levels of permissions: read-only, full permissions, and communication only (that is, these admins can only answer questions, send announcements and messages).

CHAPTER 5

Creating a contest

5.1 Creating a contest from scratch

The most immediate (but often less practical) way to create a contest in CMS is using the admin interface. You can start the AdminWebServer using the command `cmsAdminWebServer` (or using the ResourceService).

After that, you can connect to the server using the address and port specified in `cms.conf`; by default, http://localhost:8889/. Here, you can create a contest clicking on the link in the left column. After this, you must similarly add tasks and users.

Since the details of contests, tasks and users usually live somewhere in the filesystem, it is more practical to use an importer to create them in CMS automatically.

5.2 Creating a contest from the filesystem

CMS philosophy is that, unless you want, it should not change how you develop tasks, or how you store contests, tasks and user information.

To achieve this goal, CMS has tools to import a contest from a custom filesystem description. There are commands which read a filesystem description and use it to create contests, tasks, or users. Specifically: the `cmsImportContest`, `cmsImportTasl`, `cmsImportUser` commands (by default) will analyze the directory given as first argument and detect if it can be loaded (respectively) as a new contest, task, user. Run the commands with a `-h` or `--help` flag in order to better understand how they can be used.

In order to make these tools compatible with your filesystem format, you have to write a Python module that converts your filesystem description to the internal CMS representation of the contest. You have to extend the classes `ContestLoader`, `TaskLoader`, `UserLoader` defined in cmscontrib/loaders/base_loader.py, implementing missing methods as required by the docstrings (or use one of the existing loaders in `cmscontrib/loaders/` as a template). If you do not use complex task types, or many different configurations, loaders can be very simple.

Out of the box, CMS offers loaders for two formats:

- The Italian filesystem format supports all the features of CMS. No compatibility in time is guaranteed with this format. If you want to use it, an example of a contest written in this format is in this GitHub repository, while its explanation is *here*.

- The Polygon format, which is the format used in several contests and by Codeforces. Polygon does not support all of CMS features, but having this importer is especially useful if you have a big repository of tasks in this format.

CMS also has several convenience scripts to add data to the database specifying it on the command line, or to remove data from the database. Look in `cmscontrib` or at commands starting with `cmsAdd` or `cmsRemove`.

5.3 Creating a contest from an exported contest

This option is not really suited for creating new contests but to store and move contest already used in CMS. If you have the dump of a contest exported from CMS, you can import it with `cmsDumpImporter <source>`, where `<source>` is the archive filename or directory of the contest.

CHAPTER 6

Configuring a contest

In the following text "user" and "contestant" are used interchangeably. A "participation" is an instance of a user participating in a specific contest. See *here* for more details.

Configuration parameters will be referred to using their internal name, but it should always be easy to infer what fields control them in the AWS interface by using their label.

6.1 Limitations

Contest administrators can limit the ability of users to submit submissions and user_tests, by setting the following parameters:

- `max_submission_number` / `max_user_test_number`

  These set, respectively, the maximum number of submissions or user tests that will be accepted for a certain user. Any attempt to send in additional submissions or user tests after that limit has been reached will fail.

- `min_submission_interval` / `min_user_test_interval`

  These set, respectively, the minimum amount of time, in seconds, the user is required to wait after a submission or user test has been submitted before they are allowed to send in new ones. Any attempt to submit a submission or user test before this timeout has expired will fail.

The limits can be set both for individual tasks and for the whole contest. A submission or user test is accepted if it verifies the conditions on both the task *and* the contest. This means that a submission or user test will be accepted if the number of submissions or user tests received so far for that task is strictly less that the task's maximum number *and* the number of submissions or user tests received so far for the whole contest (i.e. in all tasks) is strictly less than the contest's maximum number. The same holds for the minimum interval too: a submission or user test will be accepted if the time passed since the last submission or user test for that task is greater than the task's minimum interval *and* the time passed since the last submission or user test received for the whole contest (i.e. in any of the tasks) is greater than the contest's minimum interval.

Each of these fields can be left unset to prevent the corresponding limitation from being enforced.

6.2 Feedback to contestants

Each testcase can be marked as public or private. After sending a submission, a contestant can always see its results on the public testcases: a brief passed / partial / not passed status for each testcase, and the partial score

that is computable from the public testcases only. Note that input and output data are always hidden.

Tokens were introduced to provide contestants with limited access to the detailed results of their submissions on the private testcases as well. If a contestant uses a token on a submission, then they will be able to see its result on all testcases, and the global score.

6.2.1 Tokens rules

Each contestant have a set of available tokens at their disposal; when they use a token it is taken from this set, and cannot be use again. These sets are managed by CMS according to rules defined by the contest administrators, as explained later in this section.

There are two types of tokens: contest-tokens and task-tokens. When a contestant uses a token to unlock a submission, they are really using one token of each type, and therefore needs to have both available. As the names suggest, contest-tokens are bound to the contest while task-tokens are bound to a specific task. That means that there is just one set of contest-tokens but there can be many sets of task-tokens (precisely one for every task). These sets are controlled independently by rules defined either on the contest or on the task.

A token set can be disabled (i.e. there will never be tokens available for use), infinite (i.e. there will always be tokens available for use) or finite. This setting is controlled by the `token_mode` parameter.

If the token set is finite it can be effectively represented by a non-negative integer counter: its cardinality. When the contest starts (or when the user starts its per-user time-frame, see *USACO-like contests*) the set will be filled with `token_gen_initial` tokens (i.e. the counter is set to `token_gen_initial`). If the set is not empty (i.e. the counter is not zero) the user can use a token. After that, the token is discarded (i.e. the counter is decremented by one). New tokens can be generated during the contest: `token_gen_number` new tokens will be given to the user after every `token_gen_interval` minutes from the start (note that `token_gen_number` can be zero, thus disabling token generation). If `token_gen_max` is set, the set cannot contain more than `token_gen_max` tokens (i.e. the counter is capped at that value). Generation will continue but will be ineffective until the contestant uses a token. Unset `token_gen_max` to disable this limit.

The use of tokens can be limited with `token_max_number` and `token_min_interval`: users cannot use more that `token_max_number` tokens in total (this parameter can be unset), and they have to wait at least `token_min_interval` seconds after they used a token before they can use another one (this parameter can be zero). These have no effect in case of infinite tokens.

Having a finite set of both contest- and task-tokens can be very confusing, for the contestants as well as for the contest administrators. Therefore it is common to limit just one type of tokens, setting the other type to be infinite, in order to make the general token availability depend only on the availability of that type (e.g. if you just want to enforce a contest-wide limit on tokens set the contest-token set to be finite and set all task-token sets to be infinite). CWS is aware of this "implementation details" and when one type is infinite it just shows information about the other type, calling it simply "token" (i.e. removing the "contest-" or "task-" prefix).

Note that "token sets" are "intangible": they're just a counter shown to the user, computed dynamically every time. Yet, once a token is used, a Token object will be created, stored in the database and associated with the submission it was used on.

Changing token rules during a contest may lead to inconsistencies. Do so at your own risk!

6.3 Computation of the score

6.3.1 Released submissions

The score of a contestant for the contest is always the sum of the score for each task. The score for a task is the best score among the set of "released" submissions.

Admins can use the configuration "Score mode" in AdminWebServer to change the way CMS defines the set of released submission. There are two ways, corresponding to the rules of IOI 2010-2012 and IOI 2013-.

In the first mode, used in IOI from 2010 to 2012, the released submissions are those on which the contestant used a token, plus the latest one submitted.

In the second mode, used since 2013, the released submissions are all submissions.

Usually, a task using the first mode will have a certain number of private testcases, and a limited sets of tokens. In this situation, you can think that contestants are required to "choose" the submission they want to use for grading, by submitting it last, or by using a token on it.

On the other hand, a task using the second mode usually has all testcases public, and therefore it would be silly to ask contestants to choose the submission (as they would always choose the one with the best score).

6.3.2 Score rounding

Based on the ScoreTypes in use and on how they are configured, some submissions may be given a floating-point score. Contest administrators will probably want to show only a small number of these decimal places in the scoreboard. This can be achieved with the `score_precision` fields on the contest and tasks.

The score of a user on a certain task is the maximum among the scores of the "tokened" submissions for that task, and the last one. This score is rounded to a number of decimal places equal to the `score_precision` field of the task. The score of a user on the whole contest is the sum of the *rounded* scores on each task. This score itself is then rounded to a number of decimal places equal to the `score_precision` field of the contest.

Note that some "internal" scores used by ScoreTypes (for example the subtask score) are not rounded using this procedure. At the moment the subtask scores are always rounded at two decimal places and there's no way to configure that (note that the score of the submission is the sum of the *unrounded* scores of the subtasks).

The unrounded score is stored in the database (and it's rounded only at presentation level) so you can change the `score_precision` at any time without having to rescore any submissions. Yet, you have to make sure that these values are also updated on the RankingWebServers. To do that you can either restart ScoringService or update the data manually (see *RankingWebServer* for further information).

6.4 Languages

6.4.1 Statements

When there are many statements for a certain task (which are often different translations of the same statement) contest administrators may want to highlight some of them to the users. These may include, for example, the "official" version of the statement (the one that is considered the reference version in case of questions or appeals) or the translations for the languages understood by that particular user. To do that the `primary_statements` field of the tasks and the `preferred_languages` field of the users has to be used.

The `primary_statements` field for the tasks is a list of strings: it specifies the language codes of the statements that will be highlighted to all users. A valid example is `en_US, it`. The `preferred_languages` field for the users is a list of strings: it specifies the language codes of the statements to highlight. For example `de, de_CH`.

Note that users will always be able to access all statements, regardless of the ones that are highlighted. Note also that language codes in the form `xx` or `xx_YY` (where `xx` is an ISO 639-1 code and `YY` is an ISO 3166-1 code) will be recognized and presented accordingly. For example `en_AU` will be shown as "English (Australia)".

6.4.2 Interface

The interface for contestants can be localized (see *Localization* for how to add new languages), and by default all languages will be available to all contestants. To limit the languages available to the contestants, the field "Allowed localizations" in the contest configuration can be set to the list of allowed language codes. The first of this language codes determines the fallback language in case the preferred language is not available.

6.5 Timezone

CMS stores all times as UTC timestamps and converts them to an appropriate timezone when displaying them. This timezone can be specified on a per-user and per-contest basis with the `timezone` field. It needs to contain a string recognized by pytz, for example `Europe/Rome`.

When CWS needs to show a timestamp to the user it first tries to show it according to the user's timezone. If the string defining the timezone is unrecognized (for example it is the empty string), CWS will fallback to the contest's timezone. If it is again unable to interpret that string it will use the local time of the server.

6.6 User login

Users can log into CWS manually, using their credentials (username and a password), or they can get logged in automatically by CMS based on the IP address their requests are coming from.

6.6.1 Logging in with IP-based autologin

If the "IP-based autologin" option in the contest configuration is set, CWS tries to find a user that matches the IP address the request is coming from. If it finds exactly one user, the requester is automatically logged in as that user. If zero or more than one user match, CWS does not let the user in (and the incident is logged to allow troubleshooting).

In general, each user can have multiple ranges of IP addresses associated to it. These are defined as a list of subnets in CIDR format (e.g., *192.168.1.0/24*). Only the subnets whose mask is maximal (i.e., */32* for IPv4 or */128* for IPv6) are considered for autologin purposes (subnets with non-maximal mask are still useful for IP-based restrictions, see below). The autologin will kick in if *any* of the subnets matches the IP of the request.

Warning: If a reverse-proxy (like nginx) is in use then it is necessary to set `num_proxies_used` (in `cms.conf`) to `1` and configure the proxy in order to properly pass the `X-Forwarded-For`-style headers (see *Recommended setup*). That configuration option can be set to a higher number if there are more proxies between the origin and the server.

6.6.2 Logging in with credentials

If the autologin is not enabled, users can log in with username and password, which have to be specified in the user configuration (in cleartext, for the moment). The password can also be overridden for a specific contest in the participation configuration. These credentials need to be inserted by the admins (i.e. there's no way to sign up, of log in as a "guest", etc.).

A successfully logged in user needs to reauthenticate after `cookie_duration` seconds (specified in the `cms.conf` file) from when they last visited a page.

Even without autologin, it is possible to restrict the IP address or subnet that the user is using for accessing CWS, using the "IP-based login restriction" option in the contest configuration (in which case, admins need to set `num_proxies_used` as before). If this is set, then the login will fail if the IP address that attempted it does not match at least one of the addresses or subnets specified in the participation settings. If the participation IP address is not set, then no restriction applies.

6.6.3 Failure to login

The following are some common reasons for login failures, all of them coming with some useful log message from CWS.

- IP address mismatch (with IP-based autologin): if the IP address doesn't match any subnet of any participation or if it matches some subnets of more than one participation, then the login fails. Note that if the user is using the IP address of a different user, CWS will happily log them in without noticing anything.
- IP address mismatch (using IP-based login restrictions): the login fails if the request comes from an IP address that doesn't match any of the participation's IP subnets (non-maximal masks are taken into consideration here).
- Blocked hidden participations: users whose participation is hidden cannot log in if "Block hidden participations" is set in the contest configuration.

6.7 USACO-like contests

One trait of the USACO contests is that the contests themselves are many days long but each user is only able to compete for a few hours after their first login (after that they are not able to send any more submissions). This can be done in CMS too, using the `per_user_time` field of contests. If it is unset the contest will behave "normally", that is all users will be able to submit solutions from the contest's beginning until the contest's end. If, instead, `per_user_time` is set to a positive integer value, then a user will only have a limited amount of time. In particular, after they log in, they will be presented with an interface similar to the pre-contest one, with one additional "start" button. Clicking on this button starts the time frame in which the user can compete (i.e. read statements, download attachments, submit solutions, use tokens, send user tests, etc.). This time frame ends after `per_user_time` seconds or when the contest `stop` time is reached, whichever comes first. After that the interface will be identical to the post-contest one: the user won't be able to do anything. See issue #61.

The time at which the user clicks the "start" button is recorded in the `starting_time` field of the user. You can change that to shift the user's time frame (but we suggest to use `extra_time` for that, explained in *Extra time and delay time*) or unset it to make the user able to start its time frame again. Do so at your own risk!

6.8 Extra time and delay time

Contest administrators may want to give some users a short additional amount of time in which they can compete to compensate for an incident (e.g. a hardware failure) that made them unable to compete for a while during the "intended" time frame. That's what the `extra_time` field of the users is for. The time frame in which the user is allowed to compete is expanded by its `extra_time`, even if this would lead the user to be able to submit after the end of the contest.

During extra time the user will continue to receive newly generated tokens. If you don't want them to have more tokens that other contestants, set the `token_max_number` parameter described above to the number of tokens you expect a user to have at their disposal during the whole contest (if it doesn't already have a value less than or equal to this).

Contest administrators can also alter the competition time of a contestant setting `delay_time`, which has the effect of translating the competition time window for that contestant of the specified numer of seconds in the future. Thus, while setting `extra_time` *adds* some times at the end of the contest, setting `delay_time` *moves* the whole time window. As for `extra_time`, setting `delay_time` may extend the contestant time window beyond the end of the contest itself.

Both options have to be set to a non negative number. They can be used together, producing both their effects. Please read *Detailed timing configuration* for a more in-depth discussion of their exact effect.

Note also that submissions sent during the extra time will continue to be considered when computing the score, even if the `extra_time` field of the user is later reset to zero (for example in case the user loses the appeal): you need to completely delete them from the database or make them unofficial, and make sure the score in all rankings reflects the new state.

6.9 Analysis mode

After the contest it is often customary to allow contestants to see the results of all their submissions and use the grading system to try different solutions. CMS offers an analysis mode to do this. Solutions submitted during the analysis are evaluated as usual, but are marked as not official, and thus do not contribute to the rankings. Users will also be prevented from using tokens.

The admins can enable the analysis mode in the contest configuration page in AWS; they also must set start end stop time (which must be after the contest end).

By awarding extra time or adding delay to a contestant, it is possible to extend the contest time for a user over the start of the analysis. In this case, the start of the analysis will be postponed for this user. If the contest rules contemplate extra time or delay, we suggest to avoid starting the analysis right after the end of the contest.

6.10 Programming languages

CMS allows to restrict the set of programming languages available to contestants in a certain contest; the configuration is in the contest page in AWS.

CMS offers out of the box the following combination of languages: C, C++, Pascal, Java (in two flavours, either compiled with gcj or using a JDK), Python 2, PHP.

C, C++ and Pascal are the default languages, and, together with Java with gcj, have been tested thoroughly in many contests.

PHP and Python have only been tested with Batch task types, and have not thoroughly analyzed for potential security and usability issues. Being run under the sandbox, they should be reasonably safe, but, for example, the libraries available to contestants might be hard to control.

Java with JDK works with Batch and Communication task types. Under usual conditions (default submission format) contestants must name their class as the short name of the task.

> **Warning:** Java with JDK uses multithreading even for simple programs. Therefore, if this language is allowed in the contest, multithreading and multiprocessing will be allowed in the sandbox for *all* evaluations (even with other languages).
>
> If a solution uses multithreading or multiprocessing, the time limit is checked against the sum of the user times of all threads and processes.

6.10.1 Language details

- C and C++ are supported through the GNU Compiler Collection. Submissions are optimized with `-O2`, and use the 2011 standards for C and C++.
- Java with gcj is also supported through the GNU Compiled Collection. Programs are compiled with `gcj`, optimized with `-O3`, and then run as normal executables. Notice that gcj only fully supports Java 1.4.
- Java with JDK uses the system version of the Java compiler and JVM.
- Pascal support is provided by `fpc`, and submissions are optimized with `-O2`.
- Python submissions are interpreted using Python 2 (you need to have `/usr/bin/python2`).
- PHP submissions are interpreted by `/usr/bin/php`.
- Haskell support is provided by `ghc`, and submissions are optimized with `-O2`.
- Rust support is provided by `rustc`, and submissions are optimized with `-O`.
- C# uses the system version of the Mono compiler `mcs` and the runtime `mono`. Submissions are optimized with `-optimize+`.

6.10.2 Custom languages

Additional languages can be defined if necessary. This works in the same way *as with task types*: the classes need to extend `cms.grading.language.Language` and the entry point group is called *cms.grading.languages*.

CHAPTER 7

Detailed timing configuration

This section describes the exact meaning of CMS parameters for controlling the time window allocated to each contestant. Please see *Configuring a contest* for a more gentle introduction and the intended usage of the various parameters.

When setting up a contest, you will need to decide the time window in which contestants will be able to interact with the contest (by reading statements, submit solutions, ...). In CMS there are several parameters that allow to control this time window, and it is also possible to personalize it for each single user in case it is needed.

The first decision to chose among these two possibilities:

1. all contestants will start and end the contest at the same time (unless otherwise decided by the admins during the contest for fairness reasons);
2. each contestant will start the contest at the time they decide.

The first situation is that we will refer to as a fixed-window contest, whereas we will refer to the second situation as customized-window contest.

7.1 Fixed-window contests

These are quite simple to configure: you just need to set `start_time` and `end_time`, and by default all users will be able to interact with the contest between these two instants.

For fairness reasons, during the contest you may want to extend the time window for all or for particular users. In the first case, you just need to change the end_time parameter. In the latter case, you can use one of two slightly different per-contestant parameters: `extra_time` and `delay_time`.

You can use `extra_time` to award more time at the end of the contest for a specific contestant, whereas you can use `delay_time` to shift in the future the time window of the contest just for that user. There are two main practical differences between these two options.

1. If you set `extra_time` to S seconds, the contestant will be able to interact with the contest in the first S seconds of it, whereas if you use `delay_time`, they will not, as in the first case the time window is extended, in the second is shifted (if S seconds have already passed from the start of the contest, then there is no difference).
2. If tokens are generated every M minutes, and you set `extra_time` to S seconds, then tokens for that contestants are generated at `start_time` + k*M (in particular, it might be possible that more tokens are generated for contestants with `extra_time`); if instead you set `delay_time` to S seconds, tokens for

that contestants are generated at start_time + S + k*M (i.e., they are shifted from the original, and the same amount of tokens as other contestants will be generated).

If needed, it is possible to use both at the same time.

7.2 Customized-window contests

In these contests, contestants can use a time window of fixed length (`per_user_time`), starting from the first time they log in between `start_time` and `end_time`. Moreover, the time window is capped at `end_time` (so if `per_user_time` is 5 hours and a contestant logs in for the first time one minute before `end_time`, they will have just one minute).

Again, admins can change the time windows of specific contestants for fairness reasons. In addition to `extra_time` and `delay_time`, they can also use `starting_time`, which is automatically set by CMS when the contestant logs in for the first time.

The meaning of `extra_time` is to extend both the contestant time window (as defined by `starting_time` + `per_user_time`) and the contest time window (as defined by `end_time`) by the value of `extra_time`, but only for that contestant. Therefore, setting `extra_time` to S seconds effectively allows a contestant to use S seconds more than before (regardless of the time they started the contest).

Again, delay time is similar, but it shifts both contestant and contest time window by that value. The effect on available time similar to that achieved by setting `extra_time`, with the difference explained before in point 1. Also, there is a difference in token generation as explained in point 2 above.

Finally, changing `starting_time` is very similar to changing `delay_time`, but it shifts just the contestant time window, hence if that window was already going over `end_time`, at all effects advancing `starting_time` would not award more time to the contestant, because the end would still be capped at `end_time`. The effect on token generation is the same.

There is probably no need to fiddle with more than one of these three parameters, and our suggestion is to just use `extra_time` or `delay_time` to award more time to a contestant.

CHAPTER 8

Task types

8.1 Introduction

In the CMS terminology, the task type of a task describes how to compile and evaluate the submissions for that task. In particular, they may require additional files called managers, provided by the admins.

A submission goes through two steps involving the task type: the compilation, that usually creates an executable from the submitted files, and the evaluation, that runs this executable against the set of testcases and produces an outcome for each of them.

Note that the outcome doesn't need to be obviously tied to the score for the submission: typically, the outcome is computed by a grader (which is an executable or a program stub passed to CMS) or a comparator (a program that decides if the output of the contestant's program is correct) and not directly by the task type. Hence, the task type doesn't need to know the meaning of the outcome, which is instead known by the grader and by the *score type*.

An exception to this is when the contestant's source fails (for example, exceeding the time limit); in this case the task type will assign directly an outcome, usually 0.0; admins must consider that when planning the outcomes for a task.

8.2 Standard task types

CMS ships with four task types: Batch, OutputOnly, Communication, TwoSteps. The first three are well tested and reasonably strong against cheating attempts and stable with respect to the evaluation times. TwoSteps is a somewhat simpler way to implement a special case of a Communication task, but it is substantially less secure with respect to cheating. We suggest avoiding TwoSteps for new tasks, and migrating old tasks to Communication.

OutputOnly does not involve programming languages. Batch is tested with all languages CMS supports out of the box, (C, C++, Pascal, Java, C#, Python, PHP, Haskell, Rust), but only with the first five when using a grader. Communication is tested with C, C++, Pascal and Java. TwoSteps only with C. Regardless, with some work all task types should work with all languages.

Task types may have parameters to configure their behaviour (for example, a parameter for Batch defines whether to use a simple diff or a checker to evaluate the output). You can set these parameters, for each task, on the task's page in AdminWebServer.

8.2.1 Batch

In a Batch task, each testcase has an input (usually kept secret from the contestants), and the contestant's solution must produce a correct output for that input.

Warning: If the input, or part thereof, is supposed to be a secret from the contestant's code at least for part of the evaluation, then Batch is insecure and Communication should be used.

The contestant must submit a single source file; thus the submission format should contain one element with a `.%l` placeholder for the language extension.

Batch has three parameters:

- the first specifies whether the source submitted by the contestant is compiled on its own, or together with a grader provided by the admins;
- the second specifies the filenames of input and output (for reading and writing by the contestant source or by the grader), or whether to redirect them to standard input and standard output (if left blank);
- the third whether to compare correct output and contestant-produced output with a simple diff, or with an admin-provided comparator.

A grader is a source file that is compiled with the contestant's source, and usually performs I/O for the contestants, so that they only have to implement one or more functions. If the task uses a grader, the admins must provide a manager called `grader.`*`ext`* for each allowed language, where *`ext`* is the standard extension of a source file in that language. If header files are needed, they can be provided as additional managers with an appropriate extension (for example, `.h` for C/C++ and `lib.pas` for Pascal).

The output produced by the contestant (possibly through the grader) is then evaluated against the correct output. This can be done with *white-diff*, or using a *comparator*. In the latter case, the admins must provide an executable manager called `checker`. If the contestant's code fails, this step is omitted, and the outcome will be 0.0 and the message will explain the reason.

Batch supports user tests; if a grader is used, the contestants must provide their own grader (a common practice is to provide a simple grader to contestants, that can be used for local testing and for server-side user tests). The output produced by the contestant's solution, possibly through the grader, is sent back to the contestant; it is not evaluated against a correct output.

Note: Batch tasks are supported also for Java, with some requirements. The top-level class in the contestant's source must be named like the short name of the task. The one in the grader (containing the main method) must be named `grader`.

8.2.2 OutputOnly

In an OutputOnly task, contestants can see the input of each testcase, and have to compute offline a correct output.

In any submission, contestants may submit outputs for any subset of testcases. The submission format therefore must contain one element for each testcase, and the elements must be of the form `output_`*`codename`*`.txt` where *`codename`* is the codename for the testcase.

Moreover, CMS will automatically fill the missing files in the current submission with those in the previous one, as if the contestant had submitted them. For example, if there were 4 testcases, and the following submissions:

- submission s1 with files f1 and f2,
- submission s2 with files f2' and f3,
- submission s3 with file f4,

then s1 will be judged using f1 and f2; s2 will be judged using f1, f2' and f3; and finally s3 will be judged using f1, f2', f3 and f4.

OutputOnly has one parameter, that specifies whether to compare correct output and contestant-produced output with *white-diff*, or using a *comparator* (exactly the same as the third parameter for Batch). In the latter case, the admins must provide an executable manager called `checker`.

8.2.3 Communication

Communication tasks are similar to Batch tasks, but should be used when the input, or part of it, must remain secret, at least for some time, to the contestant's code. This is the case, for example, in tasks where the contestant's code must ask questions about the input; or when it must compute the solution incrementally after seeing partial views of the input.

In practice, Communication tasks have two processes, running in two different sandboxes:

- the first (manager) is entirely controlled by the admins; it reads input, communicates with the other one, and writes a *standard manager output*;
- the second is where the contestant's code runs, after being compiled together with an admin-provided stub that helps with the communication with the first process; it doesn't have access to the input, just to what the manager communicates.

This setup ensures that the contestant's code cannot access forbidden data, even in the case they have full knowledge of the admin code.

The admins must provide an executable manager called `manager`. It can read the testcase input from stdin, and will also receive as argument the filename of two FIFOs, from and to the contestant process (in this order). It must write to stdout the outcome and to stderr the message for the contestant (see *details about the format*). If the contestant's process fails, the output of the manager is ignored, and the outcome will be 0.0 and the message will explain the reason.

Admins must also provide a manager called `stub.`*`ext`* for each allowed language, where *`ext`* is the standard extension of a source file in that language. This will be compiled with the contestant's source. Usually, it takes care of communication with manager, so that the contestants have to implement only a function. The stub will receive as argument the filename of the FIFOs, from and to manager (in this order). As for Batch, admins can also add header file that will be used when compiling the stub and the contestant's source.

Communication has one (optional) parameter, the number of user processes. If not specified, it is equal to 1 (and the behavior will be as explained above). If it is an integer N, there are a few differences:

- there will be N processes with the contestant's code and the stub running;
- there will be N pairs of FIFOs, one for each stub running; manager will receive as argument all pairs in order, and each stub will receive its own;
- the stub will receive as a third paramenter the 0-based index within the running stubs;
- the time limit is checked against the total user time of all the contestant's processes.

The submission format must contain one or more filenames ending with `.%l`. Multiple source files are simply linked together. Usually the number of files to submit is equal to `num_processes`.

Communication supports user tests. In addition to the input file, contestant must provide the stub and their source file. The admin-provided manager will be used; the output returned to the contestant will be what the manager writes to the file `output.txt`.

Note: Particular care must be taken for tasks where the communication through the FIFOs is particularly large or frequent. In these cases, the time to send the data may dominate the actual algorithm runtime, thus making it hard to distinguish between different complexities.

8.2.4 TwoSteps

Warning: This task type is not secure; the user source could intercept the main function and take control of input reading and communication between the processes, which is not monitored. Admins should use Communication instead.

In a TwoSteps task, contestants submit two source files implementing a function each (the idea is that the first function gets the input and compute some data from it with some restriction, and the second tries to retrieve the original data).

The admins must provide a manager, which is compiled together with both of the contestant-submitted files. The manager needs to be named `manager.`*`ext`*, where `{ext}` is the standard extension of a source file in that language. Furthermore, the admins must provide appropriate header files for the two source files and for the manager, even if they are empty.

The resulting executable is run twice (one acting as the computer, one acting as the retriever). The manager in the computer executable must take care of reading the input from standard input; the one in the retriever executable of writing the retrieved data to standard output. Both must take responsibility of the communication between them through a pipe.

More precisely, the executable is called with two arguments: the first is an integer which is 0 if the executable is the computer, and 1 if it is the retriever; the second is the name of the pipe to be used for communication between the processes.

TwoSteps has one parameter, similar to Batch's third, that specifies whether to compare the second process output with the correct output using white-diff or a checker. In the latter case, an executable manager named `checker` must be provided.

TwoSteps supports user tests; contestants must provide the manager in addition to the input and their sources.

How to migrate from TwoSteps to Communication. Any TwoSteps task can be implemented as a Communication task with two processes. The functionalities in the stub should be migrated to Communication's manager, which also must enforce any restriction in the computed data.

8.3 White-diff comparator

White-diff is the only built-in comparator. It can be used when each testcase has a unique correct output file, up to whitespaces. White-diff will report an outcome of 1.0 if the correct output and the contestant's output match up to whitespaces, or 0.0 if they don't.

More precisely, white-diff will return that a pair of files match if all of these conditions are satisfied:

- they have the same number of lines (apart from trailing lines composed only of whitespaces, which are ignored);
- for each corresponding line in the two files, the list of non-empty, whitespace-separated tokens is the same (in particular, tokens appear in the same order).

It treats as whitespace any repetition of these characters: space, newline, carriage return, tab, vertical tab, form feed.

Note that spurious empty lines in the middle of an output will make white-diff report a no-match, even if all tokens are correct.

8.4 Checker

When there are multiple correct outputs, or when there is partial scoring, white-diff is not powerful enough. In this cases, a checker can be used to perform a complex validation. It is an executable manager, usually named `checker`.

It will receive as argument three filenames, in order: input, correct output, and contestant's output. It will then write a *standard manager output* to stdout and stderr.

It is preferred to compile the checker statically (e.g., with `-static` using `gcc` or `g++`) to avoid potential problems with the sandbox.

8.5 Standard manager output

A standard manager output is a format that managers can follow to write an outcome and a message for the contestant.

To follow the standard manager output, a manager must write on stdout a single line, containing a floating point number, the outcome; it must write to stderr a single line containing the message for the contestant. Following lines to stdout or stderr will be ignored.

Note: If the manager writes to standard error the special strings "translate:success", "translate:wrong" or "translate:partial", these will be respectively shown to the contestants as the localized messages for "Output is correct", "Output isn't correct", and "Output is partially correct".

8.6 Custom task types

If the set of default task types doesn't suit a particular need, a custom task type can be provided. For that, in a separate "workspace" (i.e., a directory disjoint from CMS's tree), write a new Python class that extends `cms.grading.tasktypes.TaskType` and implements its abstract methods. The docstrings of those methods explain what they need to do, and the default task types can provide examples.

An accompanying `setup.py` file must also be prepared, which must reference the task type's class as an "entry point": the `entry_points` keyword argument of the `setup` function, which is a dictionary, needs to contain a key named `cms.grading.tasktypes` whose value is a list of strings; each string represents an entry point in the format `{name}={package.module}:{Class}`, where `{name}` is the name of the entry point (at the moment it plays no role for CMS, but please name it in the same way as the class) and `{package.module}` and `{Class}` are the full module name and the name of the class for the task type.

A full example of `setup.py` is as follows:

```
from setuptools import setup, find_packages

setup(
    name="my_task_type",
    version="1.0",
    packages=find_packages(),
    entry_points={
        "cms.grading.tasktypes": [
            "MyTaskType=my_package.my_module:MyTaskType"
        ]
    }
)
```

Once that is done, install the distribution by executing

```
python3 setup.py install
```

CMS needs to be restarted for it to pick up the new task type.

For additional information see the general distutils documentation and the section of the setuptools documentation about entry points.

CHAPTER 9

Score types

9.1 Introduction

For every submission, the score type of a task comes into play after the *task type* produced an outcome for each testcase. Indeed, the most important duty of the score type is to describe how to translate the list of outcomes into a single number: the score of the submission. The score type also produces a more informative output for the contestants, and the same information (score and detail) for contestants that did not use a token on the submission. In CMS, these latter set of information is called public, since the contestant can see them without using any tokens.

9.2 Standard score types

CMS ships with the following score types: Sum, GroupMin, GroupMul, GroupThreshold.

The first of the four well-tested score types, Sum, is the simplest you can imagine, just assigning a fixed amount of points for each correct testcase. The other three are useful for grouping together testcases and assigning points for that group only if some conditions held. Groups are also known as subtasks in some contests. The group score types also allow test cases to be weighted, even for groups of size 1.

Also like task types, the behavior of score types is configurable from the task's page in AdminWebServer.

9.2.1 Sum

This score type interprets the outcome for each testcase as a floating-point number measuring how good the submission was in solving that testcase, where 0.0 means that the submission failed, and 1.0 that it solved the testcase correctly. The score of that submission will be the sum of all the outcomes for each testcase, multiplied by an integer parameter given in the Score type parameter field in AdminWebServer. The parameter field must contain only this integer. The public score is given by the same computation over the public testcases instead of over all testcases.

For example, if there are 20 testcases, 2 of which are public, and the parameter string is `5`, a correct solution will score 100 points (20 times 5) out of 100, and its public score will be 10 points (2 times 5) out of 10.

9.2.2 GroupMin

With the GroupMin score type, outcomes are again treated as a measure of correctness, from 0.0 (incorrect) to 1.0 (correct); testcases are split into groups, and each group has an integral multiplier. The score is the sum of the score of each group, which in turn is the minimum outcome within the group times the multiplier. The public score is computed over all groups whose testcases are all public.

The parameters string for GroupMin is of the form `[[m1, t1], [m2, t2], ...]`, where for each pair the first element is the multiplier, and the second is either always an integer, or always a string.

In the first case (second element is always an integer), the integer is interpreted as the number of testcases that the group consumes (ordered by codename). That is, the first group comprises the first `t1` testcases and has multiplier `m1`; the second group comprises the testcases from the `t1` + 1 to the `t1` + `t2` and has multiplier `m2`; and so on.

In the second case (second element is always a string), the string is interpreted as a regular expression, and the group comprises all testcases whose codename satisfies it.

9.2.3 GroupMul

GroupMul is almost the same as GroupMin; the only difference is that instead of taking the minimum outcome among the testcases in the group, it takes the product of all outcomes. It has the same behavior as GroupMin when all outcomes are either 0.0 or 1.0.

9.2.4 GroupThreshold

GroupThreshold thinks of the outcomes not as a measure of success, but as an amount of resources used by the submission to solve the testcase. The testcase is then successfully solved if the outcome is between 0.0 (excluded, as 0.0 is a special value used by many task types, for example when the contestant solution times out) and a certain number, the threshold, specified separately for each group.

The parameter string is of the form `[[m1, t1, T1], [m2, t2, T2], ...]` where the additional parameter `T` for each group is the threshold.

The task needs to be crafted in such a way that the meaning of the outcome is appropriate for this score type.

For Batch tasks, this means that the tasks creates the outcome through a comparator program. Using diff does not make sense given that its outcomes can only be 0.0 or 1.0.

9.3 Custom score types

Additional score types can be defined if necessary. This works in the same way *as with task types*: the classes need to extend `cms.grading.scoretypes.ScoreType` and the entry point group is called *cms.grading.scoretypes*.

CHAPTER 10

Task versioning

10.1 Introduction

Task versioning allows admins to store several sets of parameters for each task at the same time, to decide which are graded and among these the one that is shown to the contestants. This is useful before the contest, to test different possibilities, but especially during the contest to investigate the impact of an error in the task preparation.

For example, it is quite common to realize that one input file is wrong. With task versioning, admins can clone the original dataset (the set of parameters describing the behavior of the task), change the wrong input file with another one, or delete it, launch the evaluation on the new dataset, see which contestants have been affected by the problem, and finally swap the two datasets to make the new one live and visible by the contestants.

The advantages over the situation without task versioning are several:

- there is no need to take down scores during the re-evaluation with the new input;
- it is possible to make sure that the new input works well without showing anything to the contestants;
- if the problem affects just a few contestants, it is possible to notify just them, and the others will be completely unaffected.

10.2 Datasets

A dataset is a version of the sets of parameters of a task that can be changed and tested in background. These parameters are:

- time and memory limits;
- input and output files;
- libraries and graders;
- task type and score type.

Datasets can be viewed and edited in the task page. They can be created from scratch or cloned from existing ones. Of course, during a contest cloning the live dataset is the most used way of creating a new one.

Submissions are evaluated as they arrive against the live dataset and all other datasets with background judging enabled, or on demand when the admins require it.

Each task has exactly one live dataset, whose evaluations and scores are shown to the contestants. To change the live dataset, just click on "Make live" on the desired dataset. Admins will then be prompted with a summary of what changed between the new dataset and the previously active, and can decide to cancel or go ahead, possibly notifying the contestants with a message.

Note: Remember that the summary looks at the scores currently stored for each submission. This means that if you cloned a dataset and changed an input, the scores will still be the old ones: you need to launch a recompilation, reevaluation, or rescoring, depending on what you changed, before seeing the new scores.

After switching live dataset, scores will be resent to RankingWebServer automatically.

CHAPTER 11

External contest formats

There are two different sets of needs that external contest formats strive to satisfy.

- The first is that of contest admins, that for several reasons (storage of old contests, backup, distribution of data) want to export the contest original data (tasks, contestants, ...) together with all data generated during the contest (from the contestants, submissions, user tests, ... and from the system, evaluations, scores, ...). Once a contest has been exported in this format, CMS must be able to reimport it in such a way that the new instance is indistinguishable from the original.
- The second is that of contest creators, that want an environment that helps them design tasks, testcases, and insert the contest data (contestant names and so on). The format needs to be easy to write, understand and modify, and should provide tools to help developing and testing the tasks (automatic generation of testcases, testing of solutions, ...). CMS must be able to import it as a new contest, but also to import it over an already created contest (after updating some data).

CMS provides an exporter `cmsDumpExporter` and an importer `cmsDumpImporter` working with a format suitable for the first set of needs. This format comprises a dump of all serializable data regarding the contest in a JSON file, together with the files needed by the contest (testcases, statements, submissions, user tests, ...). The exporter and importer understand also compressed versions of this format (i.e., in a zip or tar file). For more information run

```
cmsDumpExporter -h
cmsDumpImporter -h
```

As for the second set of needs, the philosophy is that CMS should not force upon contest creators a particular environment to write contests and tasks. Therefore, CMS provides general-purpose commands, `cmsAddUser`, `cmsAddTask` and `cmsAddContest`. These programs have no knowledge of any specific on-disk format, so they must be complemented with a set of "loaders", which actually interpret your files and directories. You can tell the importer or the reimported wich loader to use with the `-L` flag, or just rely and their autodetection capabilities. Running with `-h` flag will list the available loaders.

At the moment, CMS comes with two loaders pre-installed:

- `italy_yaml`, for tasks/users stored in the "Italian Olympiad" format.
- `polygon_xml`, for tasks made with Polygon.

The first one is not particularly suited for general use (see below for more details), so, if you don't want to migrate to one of the aforementioned formats then we encourage you to **write a loader** for your favorite format and then get in touch with CMS authors to have it accepted in CMS. See the file cmscontrib/loaders/base_loader.py for some hints.

11.1 Italian import format

You can follow this description looking at this example. A contest is represented in one directory, containing:

- a YAML file named `contest.yaml`, that describes the general contest properties;
- for each task *task_name*, a directory *task_name* that contains the description of the task and all the files needed to build the statement of the problem, the input and output cases, the reference solution and (when used) the solution checker.

The exact structure of these files and directories is detailed below. Note that this loader is not particularly reliable and providing confusing input to it may lead to create inconsistent or strange data on the database. For confusing input we mean parameters and/or files from which it can infer no or multiple task types or score types.

As the name suggest, this format was born among the Italian trainers group, thus many of the keywords detailed below used to be in Italian. Now they have been translated to English, but Italian keys are still recognized for backward compatibility and are detailed below. Please note that, although so far this is the only format natively supported by CMS, it is far from ideal: in particular, it has grown in a rather untidy manner in the last few years (CMS authors are planning to develop a new, more general and more organic, format, but unfortunately it doesn't exist yet).

For the reasons above, instead of converting your tasks to the Italian format for importing into CMS, it is suggested to write a loader for the format you already have. Please get in touch with CMS authors to have support.

Warning: The authors offer no guarantee for future compatibility for this format. Again, if you use it, you do so at your own risk!

11.1.1 General contest description

The `contest.yaml` file is a plain YAML file, with at least the following keys.

- `name` (string; also accepted: `nome_breve`): the contest's short name, used for internal reference (and exposed in the URLs); it has to match the name of the directory that serves as contest root.
- `description` (string; also accepted: `nome`): the contest's name (description), shown to contestants in the web interface.
- `tasks` (list of strings; also accepted: `problemi`): a list of the tasks belonging to this contest; for each of these strings, say *task_name*, there must be a directory called *task_name* in the contest directory, with content as described *below*; the order in this list will be the order of the tasks in the web interface.
- `users` (list of associative arrays; also accepted: `utenti`): each of the elements of the list describes one user of the contest; the exact structure of the record is described *below*.
- `token_mode`: the token mode for the contest, as in *Tokens rules*; it can be `disabled`, `infinite` or `finite`; if this is not specified, the loader will try to infer it from the remaining token parameters (in order to retain compatibility with the past), but you are not advised to rely on this behavior.

The following are optional keys.

- `start` (integer; also accepted: `inizio`): the UNIX timestamp of the beginning of the contest (copied in the `start` field); defaults to zero, meaning that contest times haven't yet been decided.
- `stop` (integer; also accepted: `fine`): the UNIX timestamp of the end of the contest (copied in the `stop` field); defaults to zero, meaning that contest times haven't yet been decided.
- `per_user_time` (integer): if set, the contest will be USACO-like (as explained in *USACO-like contests*); if unset, the contest will be traditional (not USACO-like).
- `token_*`: additional token parameters for the contest, see *Tokens rules* (the names of the parameters are the same as the internal names described there).

- `max_*_number` and `min_*_interval` (integers): limitations for the whole contest, see *Limitations* (the names of the parameters are the same as the internal names described there); by default they're all unset.

11.1.2 User description

Each contest user (contestant) is described in one element of the `utenti` key in the `contest.yaml` file. Each record has to contains the following keys.

- `username` (string): obviously, the username.
- `password` (string): obviously as before, the user's password.

The following are optional keys.

- `first_name` (string; also accepted: `nome`): the user real first name; defaults to the empty string.
- `last_name` (string; also accepted: `cognome`): the user real last name; defaults to the value of `username`.
- `ip` (string): the IP address or subnet from which incoming connections for this user are accepted, see *User login*.
- `hidden` (boolean; also accepted: `fake`): when set to true set the `hidden` flag in the user, see *User login*; defaults to false (the case-sensitive *string* `True` is also accepted).

11.1.3 Task directory

The content of the task directory is used both to retrieve the task data and to infer the type of the task.

These are the required files.

- `task.yaml`: this file contains the name of the task and describes some of its properties; its content is detailed *below*; in order to retain backward compatibility, this file can also be provided in the file *`task_name`*`.yaml` in the root directory of the *contest*.
- `statement/statement.pdf` (also accepted: `testo/testo.pdf`): the main statement of the problem. It is not yet possible to import several statement associated to different languages: this (only) statement will be imported according to the language specified under the key `primary_language`.
- `input/input`*`%d`*`.txt` and `output/output`*`%d`*`.txt` for all integers *`%d`* between 0 (included) and `n_input` (excluded): these are of course the input and reference output files.

The following are optional files, that must be present for certain task types or score types.

- `gen/GEN`: in the Italian environment, this file describes the parameters for the input generator: each line not composed entirely by white spaces or comments (comments start with `#` and end with the end of the line) represents an input file. Here, it is used, in case it contains specially formatted comments, to signal that the score type is *GroupMin*. If a line contains only a comment of the form `# ST:` *`score`* then it marks the beginning of a new group assigning at most *`score`* points, containing all subsequent testcases until the next special comment. If the file does not exists, or does not contain any special comments, the task is given the *Sum* score type.
- `sol/grader.`*`%l`* (where *`%l`* here and after means a supported language extension): for tasks of type *Batch*, it is the piece of code that gets compiled together with the submitted solution, and usually takes care of reading the input and writing the output. If one grader is present, the graders for all supported languages must be provided.
- `sol/*.h` and `sol/*lib.pas`: if a grader is present, all other files in the `sol` directory that end with `.h` or `lib.pas` are treated as auxiliary files needed by the compilation of the grader with the submitted solution.
- `check/checker` (also accepted: `cor/correttore`): for tasks of types *Batch* or *OutputOnly*, if this file is present, it must be the executable that examines the input and both the correct and the contestant's

output files and assigns the outcome. It must be a statically linked executable (for example, if compiled from a C or C++ source, the `-static` option must be used) because otherwise the sandbox will prevent it from accessing its dependencies. It is going to be executed on the workers, so it must be compiled for their architecture. If instead the file is not present, a simple diff is used to compare the correct and the contestant's output files.

- `check/manager`: (also accepted: `cor/manager`) for tasks of type *Communication*, this executable is the program that reads the input and communicates with the user solution.
- `sol/stub.%l`: for tasks of type *Communication*, this is the piece of code that is compiled together with the user submitted code, and is usually used to manage the communication with `manager`. Again, all supported languages must be present.
- `att/*`: each file in this folder is added as an attachment to the task, named as the file's filename.

11.1.4 Task description

The task YAML files require the following keys.

- `name` (string; also accepted: `nome_breve`): the name used to reference internally to this task; it is exposed in the URLs.
- `title` (string; also accepted: `nome`): the long name (title) used in the web interface.
- `n_input` (integer): number of test cases to be evaluated for this task; the actual test cases are retrieved from the *task directory*.
- `score_mode`: the score mode for the task, as in *Computation of the score*; it can be `max_tokened_last` (for the legacy behavior), or `max` (for the modern behavior).
- `token_mode`: the token mode for the task, as in *Tokens rules*; it can be `disabled`, `infinite` or `finite`; if this is not specified, the loader will try to infer it from the remaining token parameters (in order to retain compatibility with the past), but you are not advised to relay on this behavior.

The following are optional keys.

- `time_limit` (float; also accepted: `timeout`): the timeout limit for this task in seconds; defaults to no limitations.
- `memory_limit` (integer; also accepted: `memlimit`): the memory limit for this task in megabytes; defaults to no limitations.
- `public_testcases` (string; also accepted: `risultati`): a comma-separated list of test cases (identified by their numbers, starting from 0) that are marked as public, hence their results are available to contestants even without using tokens. If the given string is equal to `all`, then the importer will mark all testcases as public.
- `token_*`: additional token parameters for the task, see *Tokens rules* (the names of the parameters are the same as the internal names described there).
- `max_*_number` and `min_*_interval` (integers): limitations for the task, see *Limitations* (the names of the parameters are the same as the internal names described there); by default they're all unset.
- `output_only` (boolean): if set to True, the task is created with the *OutputOnly* type; defaults to False.

The following are optional keys that must be present for some task type or score type.

- `total_value` (float): for tasks using the *Sum* score type, this is the maximum score for the task and defaults to 100.0; for other score types, the maximum score is computed from the *task directory*.
- `infile` and `outfile` (strings): for *Batch* tasks, these are the file names for the input and output files; default to `input.txt` and `output.txt`; if left empty, `stdin` and `stdout` are used.
- `primary_language` (string): the statement will be imported with this language code; defaults to `it` (Italian), in order to ensure backward compatibility.

11.2 Polygon format

Polygon is a popular platform for the creation of tasks, and a task format, used among others by Codeforces.

Since Polygon doesn't support CMS directly, some task parameters cannot be set using the standard Polygon configuration. The importer reads from an optional file `cms_conf.py` additional configuration specifics to CMS. Additionally, user can add file named contestants.txt to allow importing some set of users.

By default, all tasks are batch files, with custom checker and score type is Sum. Loaders assumes that checker is check.cpp and written with usage of testlib.h. It provides customized version of testlib.h which allows using Polygon checkers with CMS. Checkers will be compiled during importing the contest. This is important in case the architecture where the loading happens is different from the architecture of the workers.

Polygon (by now) doesn't allow custom contest-wide files, so general contest options should be hard-coded in the loader.

CHAPTER 12

RankingWebServer

12.1 Description

The **RankingWebServer** (RWS for short) is the web server used to show a live scoreboard to the public.

RWS is designed to be completely separated from the rest of CMS: it has its own configuration file, it doesn't use the PostgreSQL database to store its data and it doesn't communicate with other services using the internal RPC protocol (its code is also in a different package: `cmsranking` instead of `cms`). This has been done to allow contest administrators to run RWS in a different location (on a different network) than the core of CMS, if they don't want to expose a public access to their core network on the internet (for security reasons) or if the on-site internet connection isn't good enough to serve a public website.

To start RWS you have to execute `cmsRankingWebServer`.

12.1.1 Configuring it

The configuration file is named `cms.ranking.conf` and RWS will search for it in `/usr/local/etc` and in `/etc` (in this order!). In case it's not found in any of these, RWS will use a hard-coded default configuration that can be found in cmsranking/Config.py. If RWS is not installed then the config directory will also be checked for configuration files (note that for this to work your working directory needs to be root of the repository). In any case, as soon as you start it, RWS will tell you which configuration file it's using.

The configuration file is a JSON object. The most important parameters are:

- `bind_address`

  It specifies the address this server will listen on. It can be either an IP address or a hostname (in the latter case the server will listen on all IP addresses associated with that name). Leave it blank or set it to `null` to listen on all available interfaces.

- `http_port`

  It specifies which port to bind the HTTP server to. If set to `null` it will be disabled. We suggest to use a high port number (like 8080, or the default 8890) to avoid the need to start RWS as root, and then use a reverse proxy to map port 80 to it (see *Using a proxy* for additional information).

- `https_port`

It specifies which port to bind the HTTPS server to. If set to `null` it will be disabled, otherwise you need to set `https_certfile` and `https_keyfile` too. See *Securing the connection between PS and RWS* for additional information.

- `username` and `password`

  They specify the credentials needed to alter the data of RWS. We suggest to set them to long random strings, for maximum security, since you won't need to remember them. `username` cannot contain a colon.

> **Warning:** Remember to change the `username` and `password` every time you set up a RWS. Keeping the default ones will leave your scoreboard open to illegitimate access.

To connect the rest of CMS to your new RWS you need to add its connection parameters to the configuration file of CMS (i.e. `cms.conf`). Note that you can connect CMS to multiple RWSs, each on a different server and/or port. The parameter you need to change is `rankings`, a list of URLs in the form:

```
<scheme>://<username>:<password>@<hostname>:<port>/<prefix>
```

where `scheme` can be either `http` or `https`, `username`, `password` and `port` are the values specified in the configuration file of the RWS and `prefix` is explained in *Using a proxy* (it will generally be blank, otherwise it needs to end with a slash). If any of your RWSs uses the HTTPS protocol you also need to specify the `https_certfile` configuration parameter. More details on this in *Securing the connection between PS and RWS*.

You also need to make sure that RWS is able to keep enough simultaneously active connections by checking that the maximum number of open file descriptors is larger than the expected number of clients. You can see the current value with `ulimit -Sn` (or `-Sa` to see all limitations) and change it with `ulimit -Sn <value>`. This value will be reset when you open a new shell, so remember to run the command again. Note that there may be a hard limit that you cannot overcome (use `-H` instead of `-S` to see it). If that's still too low you can start multiple RWSs and use a proxy to distribute clients among them (see *Using a proxy*).

12.2 Managing data

RWS doesn't use the PostgreSQL database. Instead, it stores its data in `/var/local/lib/cms/ranking` (or whatever directory is given as `lib_dir` in the configuration file) as a collection of JSON files. Thus, if you want to backup the RWS data, just make a copy of that directory. RWS modifies this data in response to specific (authenticated) HTTP requests it receives.

The intended way to get data to RWS is to have the rest of CMS send it. The service responsible for that is ProxyService (PS for short). When PS is started for a certain contest, it will send the data for that contest to all RWSs it knows about (i.e. those in its configuration). This data includes the contest itself (its name, its begin and end times, etc.), its tasks, its users and teams, and the submissions received so far. Then it will continue to send new submissions as soon as they are scored and it will update them as needed (for example when a user uses a token). Note that hidden users (and their submissions) will not be sent to RWS.

There are also other ways to insert data into RWS: send custom HTTP requests or directly write JSON files. For the former, the script *cmsRWSHelper* can be used to handle the low level communication.

12.2.1 Logo, flags and faces

RWS can also display a custom global logo, a flag for each team and a photo ("face") for each user. The only way to add these is to put them directly in the data directory of RWS:

- the logo has to be saved right in the data directory, named "logo" with an appropriate extension (e.g. `logo.png`), with a recommended resolution of 200x160;
- the flag for a team has to be saved in the "flags" subdirectory, named as the team's name with an appropriate extension (e.g. `ITA.png`);

- the face for a user has to be saved in the "faces" subdirectory, named as the user's username with an appropriate extension (e.g. `ITA1.png`).

We support the following extensions: .png, .jpg, .gif and .bmp.

12.2.2 Removing data

PS is only able to create or update data on RWS, but not to delete it. This means that, for example, when a user or a task is removed from CMS it will continue to be shown on RWS. To fix this you will have to intervene manually. The `cmsRWSHelper` script is designed to make this operation straightforward. For example, calling `cmsRWSHelper delete user` *`username`* will cause the user *username* to be removed from all the RWSs that are specified in `cms.conf`. See `cmsRWSHelper --help` and `cmsRWSHelper` *`action`* `--help` for more usage details.

In case using `cmsRWSHelper` is impossible (for example because no `cms.conf` is available) there are alternative ways to achieve the same result, presented in decreasing order of difficulty and increasing order of downtime needed.

- You can send a hand-crafted HTTP request to RWS (a `DELETE` method on the */entity_type/entity_id* resource, giving credentials by Basic Auth) and it will, all by itself, delete that object and all the ones that depend on it, recursively (that is, when deleting a task or a user it will delete its submissions and, for each of them, its subchanges).
- You can stop RWS, delete only the JSON files of the data you want to remove and start RWS again. In this case you have to *manually* determine the depending objects and delete them as well.
- You can stop RWS, remove *all* its data (either by deleting its data directory or by starting RWS with the `--drop` option), start RWS again and restart PS for the contest you're interested in, to have it send the data again.

Note: When you change the username of an user, the name of a task or the name of a contest in CMS and then restart PS, that user, task or contest will be duplicated in RWS and you will need to delete the old copy using this procedure.

12.2.3 Multiple contests

Since the data in RWS will persist even after the PS that sent it has been stopped it's possible to have many PS serve the same RWS, one after the other (or even simultaneously). This allows to have many contests inside the same RWS. The users of the contests will be merged by their username: that is, two users of two different contests will be shown as the same user if they have the same username. To show one contest at a time it's necessary to delete the previous one before adding the next one (the procedure to delete an object is the one described in *Removing data*).

Keeping the previous contests may seem annoying to contest administrators who want to run many different and independent contests one after the other, but it's indispensable for many-day contests like the IOI.

12.3 Securing the connection between PS and RWS

RWS accepts data only from clients that successfully authenticate themselves using the HTTP Basic Access Authentication. Thus an attacker that wants to alter the data on RWS needs the username and the password to authenticate its request. If they are random (and long) enough the attacker cannot guess them but may eavesdrop the plaintext HTTP request between PS and RWS. Therefore we suggest to use HTTPS, that encrypts the transmission with TLS/SSL, when the communication channel between PS and RWS is not secure.

HTTPS does not only protect against eavesdropping attacks but also against active attacks, like a man-in-the-middle. To do all of this it uses public-key cryptography based on so-called certificates. In our setting RWS has a public certificate (and its private key). PS has access to a copy to the same certificate and can use it to verify

the identity of the receiver before sending any data (in particular before sending the username and the password!). The same certificate is then used to establish a secure communication channel.

The general public does not need to use HTTPS, since it is not sending nor receiving any sensitive information. We think the best solution is, for RWS, to listen on both HTTP and HTTPS ports, but to use HTTPS only for private internal use. Not having final users use HTTPS also allows you to use home-made (i.e. self-signed) certificates without causing apocalyptic warnings in the users' browsers.

Note that users will still be able to connect to the HTTPS port if they discover its number, but that is of no harm. Note also that RWS will continue to accept incoming data even on the HTTP port; simply, PS will not send it.

To use HTTPS we suggest you to create a self-signed certificate, use that as both RWS's and PS's `https_certfile` and use its private key as RWS's `https_keyfile`. If your PS manages multiple RWSs we suggest you to use a different certificate for each of those and to create a new file, obtained by joining all certificates, as the `https_certfile` of PS. Alternatively you may want to use a Certificate Authority to sign the certificates of RWSs and just give its certificate to PS. Details on how to do this follow.

Note: Please note that, while the indications here are enough to make RWS work, computer security is a delicate subject; we urge you to be sure of what you are doing when setting up a contest in which "failure is not an option".

12.3.1 Creating certificates

A quick-and-dirty way to create a self-signed certificate, ready to be used with PS and RWS, is:

```
openssl req -newkey rsa:1024 -nodes -keyform PEM -keyout key.pem \
            -new -x509 -days 365 -outform PEM -out cert.pem -utf8
```

You will be prompted to enter some information to be included in the certificate. After you do this you'll have two files, `key.pem` and `cert.pem`, to be used respectively as the `https_keyfile` and `https_certfile` for PS and RWS.

Once you have a self-signed certificate you can use it as a CA (Certificate Authority) to sign other certificates. If you have a `ca_key.pem`/`ca_cert.pem` pair that you want to use to create a `key.pem`/`cert.pem` pair signed by it, do:

```
openssl req -newkey rsa:1024 -nodes -keyform PEM -keyout key.pem \
            -new -outform PEM -out cert_req.pem -utf8
openssl x509 -req -in cert_req.pem -out cert.pem -days 365 \
             -CA ca_cert.pem -CAkey ca_key.pem -set_serial <serial>
rm cert_req.pem
```

Where `<serial>` is a number that has to be unique among all certificates signed by a certain CA.

For additional information on certificates see the official Python documentation on SSL.

12.4 Using a proxy

As a security measure, we recommend not to run RWS as root but to run it as an unprivileged user instead. This means that RWS cannot listen on port 80 and 443 (the default HTTP and HTTPS ports) but it needs to listen on ports whose number is higher than or equal to 1024. This is not a big issue, since we can use a reverse proxy to map the default HTTP and HTTPS ports to the ones used by RWS. We suggest you to use nginx, since it has been already proved successfully for this purpose (some users have reported that other software, like Apache, has some issues, probably due to the use of long-polling HTTP requests by RWS).

A reverse proxy is most commonly used to map RWS from a high port number (say 8080) to the default HTTP port (i.e. 80), hence we will assume this scenario throughout this section.

With nginx it's also extremely easy to do some URL mapping. That is, you can make RWS "share" the URL space of port 80 with other servers by making it "live" inside a prefix. This means that you will access RWS using an URL like "http://myserver/prefix/".

We'll provide here an example configuration file for nginx. This is just the "core" of the file, but other options need to be added in order for it to be complete and usable by nginx. These bits are different on each distribution, so the best is for you to take the default configuration file provided by your distribution and adapt it to contain the following code:

```
http {
  server {
    listen 80;
    location ^~ /prefix/ {
      proxy_pass http://127.0.0.1:8080/;
      proxy_buffering off;
    }
  }
}
```

The trailing slash is needed in the argument of both the `location` and the `proxy_pass` option. The `proxy_buffering` option is needed for the live-update feature to work correctly (this option can be moved into `server` or `http` to give it a larger scope). To better configure how the proxy connects to RWS you can add an `upstream` section inside the `http` module, named for example `rws`, and then use `proxy_pass http://rws/`. This also allows you to use nginx as a load balancer in case you have many RWSs.

If you decide to have HTTPS for private internal use only, as suggested above (that is, you want your users to use only HTTP), then it's perfectly fine to keep using a high port number for HTTPS and not map it to port 443, the standard HTTPS port. Note also that you could use nginx as an HTTPS endpoint, i.e. make nginx decrypt the HTTPS trasmission and redirect it, as cleartext, into RWS's HTTP port. This allows to use two different certificates (one by nginx, one by RWS directly), although we don't see any real need for this.

The example configuration file provided in *Recommended setup* already contains sections for RWS.

12.4.1 Tuning nginx

If you're setting up a private RWS, for internal use only, and you expect just a handful of clients then you don't need to follow the advices given in this section. Otherwise please read on to see how to optimize nginx to handle many simultaneous connections, as required by RWS.

First, set the `worker_processes` option[1] of the core module to the number of CPU or cores on your machine. Next you need to tweak the `events` module: set the `worker_connections` option[2] to a large value, at least the double of the expected number of clients divided by `worker_processes`. You could also set the `use` option[3] to an efficient event-model for your platform (like epoll on linux), but having nginx automatically decide it for you is probably better. Then you also have to raise the maximum number of open file descriptors. Do this by setting the `worker_rlimit_nofile` option[4] of the core module to the same value of `worker_connections` (or greater). You could also consider setting the `keepalive_timeout` option[5] to a value like `30s`. This option can be placed inside the `http` module or inside the `server` or `location` sections, based on the scope you want to give it.

For more information see the official nginx documentation:

12.5 Some final suggestions

The suggested setup (the one that we also used at the IOI 2012) is to make RWS listen on both HTTP and HTTPS ports (we used 8080 and 8443), to use nginx to map port 80 to port 8080, to make all three ports (80, 8080 and

[1] http://wiki.nginx.org/CoreModule#worker_processes
[2] http://wiki.nginx.org/EventsModule#worker_connections
[3] http://wiki.nginx.org/EventsModule#use
[4] http://wiki.nginx.org/CoreModule#worker_rlimit_nofile
[5] http://wiki.nginx.org/HttpCoreModule#keepalive_timeout

8443) accessible from the internet, to make PS connect to RWS via HTTPS on port 8443 and to use a Certificate Authority to generate certificates (the last one is probably an overkill).

At the IOI 2012, we had only one server, running on a 2 GHz machine, and we were able to serve about 1500 clients simultaneously (and, probably, we were limited to this value by a misconfiguration of nginx). This is to say that you'll likely need only one public RWS server.

If you're starting RWS on your server remotely, for example via SSH, make sure the `screen` command is your friend :-).

CHAPTER 13

Localization

13.1 For developers

When you change a string in a template or in a web server, you have to generate again the file cms/locale/cms.pot. To do so, run this command from the root of the repository.

```
./setup.py extract_messages
```

When you have a new translation, or an update of an old translation, you need to update the `cms.mo` files (the compiled versions of the `cms.po` files). You can run `python setup.py compile_catalog` to update the `cms.mo` files for all translations (or add the `-l <code>` argument to update only the one for a given locale). The usual `python setup.py install` will do this automatically. Note that, to have the new strings, you need to restart the web server.

13.2 For translators

To begin translating to a new language, run this command from the root of the repository.

```
./setup.py init_catalog -l <code>
```

Right after that, open the newly created `cms/locale/<code>/LC_MESSAGES/cms.po` and fill the information in the header. To translate a string, simply fill the corresponding msgstr with the translations. You can also use specialized translation softwares such as poEdit and others.

When the developers update the `cms.pot` file, you do not need to start from scratch. Instead, you can create a new `cms.po` file that merges the old translated string with the new, to-be-translated ones. The command is the following, run inside the root of the repository.

```
./setup.py update_catalog -l <code>
```

After you are done translating the messages, please run the following command and check that no error messages are reported (the developers will be glad to assist you if any of them isn't clear):

```
./setup.py compile_catalog -l <code>
```

CHAPTER 14

Troubleshooting

Subtle issues with CMS can arise from old versions of libraries or supporting software. Please ensure you are running the minimum versions of each dependency (described in *Dependencies and available compilers*).

In the next sections we list some known symptoms and their possible causes.

14.1 Database

- *Symptom.* Error message "Cannot determine OID of function lo_create"

  Possible cause. Your database must be at least PostgreSQL 8.x to support large objects used by CMS.

- *Symptom.* Exceptions regarding missing database fields or with the wrong type.

  Possible cause. The version of CMS that created the schema in your database is different from the one you are using now. If the schema is older than the current version, you can update the schema as in *Installing CMS and its Python dependencies*.

- *Symptom.* Some components of CMS fail randomly and PostgreSQL complains about having too many connections.

  Possible cause. The default configuration of PostgreSQL may allow insufficiently many incoming connections on the database engine. You can raise this limit by tweaking the `max_connections` parameter in `postgresql.conf` (see docs). This, in turn, requires more shared memory for the PostgreSQL process (see `shared_buffers` parameter in docs), which may overflow the maximum limit allowed by the operating system. In such case see the suggestions in http://www.postgresql.org/docs/9.1/static/kernel-resources.html#SYSVIPC. Users reported that another way to go is to use a connection pooler like PgBouncer.

14.2 Servers

- *Symptom.* Some HTTP requests to ContestWebServer take a long time and fail with 500 Internal Server Error. ContestWebServer logs contain entries such as `TimeoutError('QueuePool limit of size 5 overflow 10 reached, connection timed out, timeout 60',)`.

  Possible cause. The server may be overloaded with user requests. You can try to increase the `pool_timeout` argument in cms/db/__init__.py or, preferably, spread your users over more instances of ContestWebServer.

- *Symptom.* Message from ContestWebServer such as: `WARNING:root:Invalid cookie signature KFZzdW5kdWRlCnAwCkkxMzI5MzQzNzIwCnRw...`

  Possible cause. The contest secret key (defined in cms.conf) may have been changed and users' browsers are still attempting to use cookies signed with the old key. If this is the case, the problem should correct itself and won't be seen by users.

- *Symptom.* Ranking Web Server displays wrong data, or too much data.

  Possible cause. RWS is designed to handle groups of contests, so it retains data about past contests. If you want to delete previous data, run RWS with the `` `-d` `` option. See *RankingWebServer* for more details

14.3 Sandbox

- *Symptom.* The Worker fails to evaluate a submission logging about an invalid (empty) output from the manager.

  Possible cause. You might have been used a non-statically linked checker. The sandbox prevent dynamically linked executables to work. Try compiling the checker with `` `-static` ``. Also, make sure that the checker was compiled for the architecture of the workers (e.g., 32 or 64 bits).

- *Symptom.* The Worker fails to evaluate a submission with a generic failure.

  Possible cause. Make sure that the isolate binary that CMS is using has the correct permissions (in particular, its owner is root and it has the suid bit set). Be careful of having multiple isolate binaries in your path. Another reason could be that you are using an old version of isolate.

- *Symptom.* Contestants' solutions fail when trying to write large outputs.

  Possible cause. CMS limits the maximum output size from programs being evaluated for security reasons. Currently the limit is 1 GB and can be configured by changing the parameter `max_file_size` in `cms.conf`.

14.4 Evaluations

- *Symptom.* Submissions that should exceed memory limit actually pass or exceed the time limits.

  Possible cause. You have an active swap partition on the workers; isolate only limit physical memory, not swap usage. Disable the swap with `sudo swapoff -a`.

- *Symptom.* Re-running evaluations gives very different time or memory usage.

  Possible cause. Make sure the workers are configured in a way to minimize resource usage variability, by following isolate's guidelines for reproducible results.

CHAPTER 15

Internals

This section contains some details about some CMS internals. They are mostly meant for developers, not for users. However, if you are curious about what's under the hood, you will find something interesting here (though without any pretension of completeness). Moreover, these are not meant to be full specifications, but only useful notes for the future.

Oh, I was nearly forgetting: if you are curious about what happens inside CMS, you may actually be interested in helping us writing it. We can assure you it is a very rewarding task. After all, if you are hanging around here, you must have some interest in coding! In case, feel free to get in touch with us.

15.1 RPC protocol

Different CMS processes communicate between them by mean of TCP sockets. Once a service has established a socket with another, it can write messages on the stream; each message is a JSON-encoded object, terminated by a `\r\n` string (this, of course, means that `\r\n` cannot be used in the JSON encoding: this is not a problem, since new lines inside string represented in the JSON have to be escaped anyway).

An RPC request must be of the form (it is pretty printed here, but it is sent in compact form inside CMS):

```
{
  "__method": <name of the requested method>,
  "__data": {
              <name of first arg>: <value of first arg>,
              ...
            },
  "__id": <random ID string>
}
```

The arguments in `__data` are (of course) not ordered: they have to be matched according to their names. In particular, this means that our protocol enables us to use a `kwargs`-like interface, but not a `args`-like one. That's not so terrible, anyway.

The `__id` is a random string that will be returned in the response, and it is useful (actually, it's the only way) to match requests with responses.

The response is of the form:

```
{
  "__data": <return value or null>,
  "__error": <null or error string>,
  "__id": <random ID string>
}
```

The value of `__id` must of course be the same as in the request. If `__error` is not null, then `__data` is expected to be null.

15.2 Backdoor

Setting the `backdoor` configuration key to true causes services to serve a Python console (accessible with netcat), running in the same interpreter instance as the service, allowing to inspect and modify its data, live. It will be bound to a local UNIX domain socket, usually at `/var/local/run/cms/`*`service_shard`*. Access is granted only to users belonging to the cmsuser group. Although there's no authentication mechanism to prevent unauthorized access, the restrictions on the file should make it safe to run the backdoor everywhere, even on workers that are used as contestants' machines. You can use `rlwrap` to add basic readline support. For example, the following is a complete working connection command:

```
rlwrap netcat -U /var/local/run/cms/EvaluationService_0
```

Substitute `netcat` with your implementation (`nc`, `ncat`, etc.) if needed.