
CMS Documentation

Release 1.0.1

The CMS development team

July 12, 2013

CONTENTS

1	Introduction	1
1.1	General structure	1
1.2	Services	1
1.3	Security considerations	2
2	Installation	3
2.1	Dependencies	3
2.2	Installing CMS	4
2.3	Updating CMS	4
3	Running CMS	5
3.1	Configuring the DB	5
3.2	Configuring CMS	5
3.3	Running CMS	6
3.4	Recommended setup	6
3.5	Logs	7
4	Creating a contest	9
4.1	Creating a contest from scratch	9
4.2	Creating a contest from the filesystem	9
4.3	Creating a contest from an exported contest	9
5	Configuring a contest	11
5.1	Limitations	11
5.2	Tokens	11
5.3	Score rounding	12
5.4	Primary statements	13
5.5	Timezone	13
5.6	User login	13
5.7	USACO-like contests	13
5.8	Extra time	14
6	Task types	15
6.1	Introduction	15
6.2	Standard task types	15
7	Score types	19
7.1	Introduction	19
7.2	Standard score types	19
8	External contest formats	21
8.1	Italian import format	21
9	RankingWebServer	25

9.1	Description	25
9.2	Running it	25
9.3	Managing it	26
9.4	Securing the connection between SS and RWS	27
9.5	Using a proxy	28
9.6	Some final suggestions	29
10	Localization	31
10.1	For developers	31
10.2	For translators	31
11	Troubleshooting	33
11.1	Database	33
11.2	Servers	33
11.3	Sandbox	33
11.4	Importers	34
11.5	Configuration	34

INTRODUCTION

When organizing a programming contest, there are three main stages:

- the first is to develop all the data that the assigned tasks need (i.e., statements, solutions, testcases, information on how to grade submissions, etc.);
- the second, that happens when the contest is onsite, is to properly configure the machines that the contestants are going to use during the contest, in particular with respect to network security;
- the third is to manage the actual contest (accepting and grading submissions, give feedback on them, display a live ranking, etc.).

The aim of the CMS project is to give a good answer to the third problem. Our goal is to develop a contest management system that is secure, extendable, adaptable to different situations, and easy to use.

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. At the moment, there is no way to use other SQL databases, because the Large Object (LO) feature of PostgreSQL is used. 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 without problems. This means that if a machine goes down, then the administrator can quickly replace it with an identical one, which will take its roles (without having to move information from the broken machine). Of course, this also means that if the database goes down, the system is unable to work. 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 redundancy 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; also sends these scores to the rankings;
- 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.

There are also other services for testing, importing and exporting contests.

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.

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.

INSTALLATION

2.1 Dependencies

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 >= 8.4;
- gettext >= 0.18;
- Python >= 2.7, < 3.0;
- setuptools >= 0.6;
- Tornado >= 2.0;
- Psycpg >= 2.4;
- simplejson >= 2.1;
- SQLAlchemy >= 0.7;
- psutil >= 0.2;
- netifaces >= 0.5;
- PyCrypto >= 2.3;
- pytz;
- iso-codes;
- shared-mime-info;
- PyYAML >= 3.10 (only for YamlImporter);
- BeautifulSoup >= 3.2 (only for running tests);
- mechanize >= 0.2 (only for running tests);
- coverage >= 3.4 (only for running tests);
- Sphinx (only for building documentation).

On Ubuntu 12.04, one will need to run the following script to satisfy all dependencies:

```
sudo apt-get install build-essential fpc postgresql postgresql-client \
    gettext python2.7 python-setuptools python-tornado python-psycpg2 \
    python-simplejson python-sqlalchemy python-psutil python-netifaces \
    python-crypto python-tz iso-codes shared-mime-info stl-manual \
    python-beautifulsoup python-mechanize python-coverage
```

Optional.

```
# sudo apt-get install phpPgAdmin python-yaml python-sphinx
```

On Arch Linux, the following command will install almost all dependencies (two of them can be found in the AUR):

```
sudo pacman -S base-devel fpc postgresql postgresql-client python2 \
    setuptools python2-tornado python2-psycopg2 python2-simplejson \
    python2-sqlalchemy python2-psutil python2-netifaces python2-crypto \
    python2-pytz iso-codes shared-mime-info python2-beautifulsoup3 \
    python2-mechanize
```

```
# Install the following from AUR.
# https://aur.archlinux.org/packages/sgi-stl-doc/
# https://aur.archlinux.org/packages/python2-coverage/
```

```
# Optional.
# sudo pacman -S phppgadmin python2-yaml python-sphinx
```

If you prefer using Python Package Index, you can retrieve all Python dependencies with this line:

```
sudo pip install -r REQUIREMENTS.txt
```

2.2 Installing CMS

You can download CMS 1.0.1 from [GitHub](#) and extract it on your filesystem. After that, you can install it (recommended, not necessary though):

```
./setup.py build
sudo ./setup.py install
```

If you install CMS, you also need to add your user to the `cmsuser` group and logout to make the change effective:

```
sudo usermod -a -G cmsuser
```

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

```
groups
```

2.3 Updating CMS

If you were using CMS before the release of version 1.0.1, you can update the content of your database with:

```
cd cms/db
python UpdatedDB.py -l # To see which updating scripts are available.
python UpdatedDB.py -s YYYYMMDD # To update the DB, where YYYYMMDD is
                                # the last date in which you created or
                                # updated the DB.
```


RUNNING CMS

3.1 Configuring the DB

The first thing to do is to create the user and the database. For PostgreSQL, this is obtained with the following commands (note that the user doesn't need to be a superuser, nor be able to create databases nor roles):

```
sudo su postgres
createuser cmsuser -P
createdb -O cmsuser cmsdb
```

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 `examples/`. You want to copy them to the same file names but without the `.sample` suffix (that is, to `examples/cms.conf` and `examples/cms.ranking.conf`) before modifying them.

- `cms.conf` is intended to be the same on all servers; all configurations are explained in the file; of particular importance is the definition of `core_services`, that specifies where the services are going to be run, and how many of them, 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 intended to be different on each server that will host a ranking. 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 and password with the ones you chose before;
- if you are running low on disk space, you may want to change `keep_sandbox` to `false`;
- if you want to run CMS without installing it, you need to change `process_cmdline` to reflect that.

If you are organizing a real contest, you must change `secret_key` from the default, and also you will need to think about how to distribute your services and change accordingly `core_services`. Finally, you should

change the ranking section of `cms.conf`, and `cms.ranking.conf`, to use a non-trivial username and password.

After having modified `cms.conf` and `cms.ranking.conf` in `examples/`, you can reinstall CMS in order to make these changes effective, with

```
sudo ./setup.py install
```

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 `./cms/service/LogService.py`).

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 run 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.

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, there is at least one server associated only to a `Worker`.

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

Our preferred distribution is Ubuntu \geq 12.04 LTS. We will hopefully support Ubuntu 12.04.x out of the box for the length of Ubuntu’s support duration, that is five years.

Very important note: up to now, we support only 32 bit distributions.

Saying that, one is not forced to follow the previous rules, and it should not be very hard to successfully run CMS on different distributions or even on 64 bit installations (see [docs/32bits_sandbox_howto.txt](#) to read how to set up a 32 bits chroot).

You can replicate the service handling the contestant-facing web server, `cmsContestWebServer`; in this case, you need to configure a load balancer in front of them. We suggest to use `nginx` for that, and provide a sample configuration for it at [examples/nginx.conf.sample](#) (this file also configures `nginx` to act as a HTTPS endpoint and to force secure connections, by redirecting HTTP to HTTPS). This file probably needs to be adapted to your distribution if it's 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 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`: you can ignore them (in the default configuration, the log service does not show them);
- `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 really 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 logs are also displayed in the main page of `AdminWebServer`.

CREATING A CONTEST

4.1 Creating a contest from scratch

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

After that, you can connect to the server using the address and port specified in `cms.conf`; typically, <http://localhost:8889/>.

Here, you can create a contest clicking the “+” next to the drop down on the left. After that, you must add the tasks and the users. Up to now, each of these operations is manual; plus, it is usually more practical to work, for example, on a file specifying the contestants’ details instead of using the web interface.

Luckily, there is another way to create a contest.

4.2 Creating a contest from the filesystem

The idea we wanted to implement is that CMS does not get in the way of how you create your contest and your tasks (unless you want to). We think that every national IOI selection team and every contest administrator has a preferred way of developing the tasks, and of storing their data in the filesystem, and we do not want to change the way you work.

Instead, you are encouraged to write an “importer”, that is a piece of software that reads data from your filesystem structure and creates the contest in the database. You can also write a “reimporter” if you want to change the contest details without losing the data (submissions, user tests, ...) already uploaded.

We provide examples of both importer and reimporter that target the format used by the Italian IOI selection team. You can either adapt them to the format you use, or decide to use the Italian format. An example of a contest written in this format is in [this GitHub repository](#), while its explanation is [here](#).

4.3 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 `cmsContestImporter <source>`, where `<source>` is the archive filename or directory of the contest.

CONFIGURING A CONTEST

In the following text “user” and “contestant” are used interchangeably.

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.

5.1 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/user_tests that will be accepted for a certain user. Any attempt to send in additional submissions/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 the user is required to wait after a submission/user_test has been submitted before he’s allowed to send in new ones. Any attempt to submit a submission/user_test before this timeout has expired will fail.

The limits can be set both on individual tasks and on the whole contest. For a submission to be accepted it has to verify the conditions on both the task *and* the contest. This means that a submission/user_test will be accepted only if the number of submissions/user_tests received so far for that task is strictly less than the task’s “max_number” *and* the number of submissions/user_tests received so far for the whole contest (i.e. in all tasks) is strictly less than the contest’s “max_number”. The same holds for “min_interval” too: a submission/user_test will be accepted only if the time passed since the last submission/user_test for that task is greater than the task’s “min_interval” *and* the time passed since the last submission/user_test received for the whole contest (i.e. in any of the tasks) is greater than the contest’s “min_interval”.

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

5.2 Tokens

Tokens are a concept introduced to provide contestants with limited access to the detailed results of their submissions on the private testcases.

For every submission sent in for evaluation, a contestant is always able to see if it successfully compiled. He/she is also able to see its scores on the public testcases of the task, if any. All information about the other so-called private testcases is kept hidden. Yet, a contestant can choose to use one of its tokens to “unlock” a certain submission of his/her choice. After he/she does this, detailed results are available for all testcases, as if they were all public. A token, once used, is consumed and lost forever. Contestants have a set of available tokens at their disposal, where the ones they use are picked from. These sets are managed by CMS according to rules defined by the contest administrators, as explained later in this section. Note that the public/private results are arbitrarily provided by the *score types* and are *usually* based on public/private testcases, but they could actually use other criteria.

Tokens also affect the score computation. That is, all “tokened” submissions will be considered, together with the last submitted one, when computing the score for a task. See also [Score rounding](#).

There are two types of tokens: contest-tokens and task-tokens. When a contestant uses a token to unlock a submission he/she is in reality using two tokens, one 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.

Rules consist of six parameters: `token_initial`, `token_gen_number`, `token_gen_time`, `token_max`, `token_total` and `token_min_interval`. For a set to be disabled, the `token_initial` field has to be unset. For a set to be infinite, `token_initial` needs to be a non-negative integer, `token_gen_number` needs to be a positive integer and `token_gen_time` needs to be zero.

Otherwise the token set is finite. This means that the token set 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_initial` tokens (i.e. the counter is set to `token_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_time` minutes from the start (note that `token_gen_number` can be zero, thus disabling token generation). If `token_max` is set to a positive integer, the set cannot contain more than `token_max` tokens, therefore generation of new tokens will stop at that value (unset `token_max` to disable this limit).

The use of tokens can be limited with `token_min_interval` and `token_total`: users 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), and they cannot use more than `token_total` tokens in total (this parameter can be unset). Note that `token_total` has no effect if the token set is infinite.

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's 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!

5.3 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). That will be changed soon. See [issue #33](#).

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).

5.4 Primary 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 users has to be used.

The `primary_statements` field for the tasks is a JSON-encoded 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 `primary_statements` field for the users is a JSON-encoded object of lists of strings. Each item in this object specifies a task by its name and provides a list of language codes of the statements to highlight. For example `{"task1": ["de"], "task2": ["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)”.

5.5 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 in the format `Europe/Rome` (actually, any string recognized by `pytz` will work).

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.

5.6 User login

Users log into CWS using a username and a password. These have to be specified, respectively, in the `username` and `password` fields (in cleartext!). These credentials need to be inserted (i.e. there's no way to have an automatic login, a “guest” session, etc.) and, if they match, the login (usually) succeeds. The user needs to login again if he/she doesn't navigate the site for `cookie_duration` seconds (specified in the `cms.conf` file).

In fact, there are other reasons that can cause the login to fail. If the `ip_lock` option (in `cms.conf`) is set to `true` then the login will fail if the IP address that attempted it is different from the `ip` field of the specified user. If `ip` is `0.0.0.0` then this check will be skipped, even if `ip_lock` is `true`. Note that if a reverse-proxy (like `nginx`) is in use then it is necessary to set `is_proxy_used` (in `cms.conf`) to `true`.

The login can also fail if `block_hidden_users` (in `cms.conf`) is `true` and the user one wants to login as has the `hidden` field set.

5.7 USACO-like contests

The most peculiar 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 he/she is 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 `start` until

the contest's stop. If, instead, `per_user_time` is set to a positive integer value then an use will only have a limited amount of time. In particular, after he/she logs in, he/she will be presented with an interface similar to the pre-contest one, with one addition: a “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](#)) or unset it to make the user able to start its time-frame again. Do so at your own risk!

5.8 Extra 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 submit solutions is expanded by its `extra_time`, even if this would lead the user to be able to submit after the end of the contest.

Note that in its extra time the user will continue to receive newly generated tokens. If you don't want him/her to have more tokens than other contestants set the `token_total` parameter described above to the number of tokens you expect a user to have at his/her disposal during the whole contest (if it doesn't already have a value less than or equal to this). See also [issue #29](#).

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.

TASK TYPES

6.1 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 (that might also be empty): 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 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*.

6.2 Standard task types

CMS ships with four task types: Batch, OutputOnly, Communication, TwoSteps. The first two are well tested and reasonably strong against cheating attempts and stable with respect to the evaluation times. Communication should be usable but it is less tested than the first two. The last one, TwoSteps, is probably not ready for usage in a public competition. The first two task types cover all but three of the IOI tasks up to IOI 2012.

You can configure, for each task, the behavior of these task types on the task's page in AdminWebServer.

6.2.1 Batch

In a Batch task, the contestant submits a single source file, in one of the three supported languages: C, C++, or Pascal.

The source file is either standalone or to be compiled with a grader provided by the contest admins. The resulting executable does I/O either on standard input and output or on two files with a specified name. The output produced by the contestant's program is then compared to the correct output either using a simple diff algorithm (that ignores whitespaces) or using a comparator, provided by the admins.

The three choices (standalone or with a grader, standard input and output or files, diff or comparator) are specified through parameters.

If the admins want to provide a grader that takes care of reading the input and writing the output (so that the contestants only need to write one or more functions), they must provide three managers, called `grader.c`, `grader.cpp` and `grader.pas`. If header files are needed, they can be provided with names `task_name.h` or `task_namelib.pas`.

If the output is compared with a diff, the outcome will be a float, 0.0 if the output is not correct, 1.0 if it is. If the output is validated by a comparator, you need to provide a manager called `checker` that is an executable taking three arguments: input, correct output and contestant's output and that must write on standard output the outcome

(that is going to be used by the score type, usually a float between 0.0 and 1.0), and on standard error a message to forward to the contestant.

The submission format must contain one filename ending with `.%1`. If there are additional files, the contestants are forced to submit them, the admins can inspect them, but they are not used towards the evaluation.

6.2.2 OutputOnly

In an OutputOnly task, the contestant submits a file for each testcase. Usually, the semantics is that the task specifies a task to be performed on an input file, and the admins provide a set of testcases composed of an input and an output file (as it is for a Batch task). The difference is that, instead of requiring a program that solves the task without knowing the input files, the contestant are required, given the input files, to provide the output files.

There is only one parameter for OutputOnly tasks, namely how correctness of the contestants' outputs is checked. Similarly to the Batch task type, these can be checked using a diff or using a comparator, that is an executable manager named checker, with the same properties of the one for Batch tasks.

OutputOnly tasks usually have many uncorrelated files to be submitted. Contestants may submit the first output in a submission, and the second in another submission, but it is easy to forget the first output in the other submission; it is also tedious to add every output every time. Hence, OutputOnly tasks have a feature that, if a submission lacks the output for a certain testcase, the current submission is completed with the most recently submitted output for that testcase (if it exists). This has the effect that contestants can work on a testcase at a time, submitting only what they did from the last submission.

The submission format must contain all the filenames of the form `output_num.txt` where *num* is a three digit decimal number (padded with zeroes, and goes from 0 (included) to the number of testcases (excluded)). Again, you can add other files that are stored but ignored. For example, a valid submission format for an OutputOnly task with three testcases is `["output_000.txt", "output_001.txt", "output_002.txt"]`.

6.2.3 Communication

In a Communication task, a contestant must submit a source file implementing a function, similarly to what happens for a Batch task. The difference is that the admins must provide both a stub, that is a source file that is compiled together with the contestant's source, and a manager, that is an executable.

The two programs communicate through two fifo files. The manager receives the name of the two fifos as its arguments. It is supposed to read from standard input the input of the testcase, and to start communicating some data to the other program through the fifo. The two programs exchange data through the fifo, until the manager is able to assign an outcome to the evaluation. The manager then writes to standard output the outcome and to standard error the message to the user.

If the program linked to the user-provided file fails (for a timeout, or for a non-allowed syscall), the outcome is 0.0 and the message describes the problem to the user.

The submission format must contain one filename ending with `.%1`. If there are additional files, the contestants are forced to submit them, the admins can inspect them, but they are not used towards the evaluation.

6.2.4 TwoSteps

Warning: use this task type only if you know what are you doing.

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 compiled together with both files. 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 outcome and the explanation message to standard output and error respectively. Both must take responsibility of the communication between them through a pipe.

More precisely, the executable are 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.

SCORE TYPES

7.1 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.

7.2 Standard score types

Like task types, CMS has the most common score types built in. They are Sum, GroupMin, GroupMul, GroupThreshold. There is also a score type called Relative, but it is an experiment not meant for usage.

The first of the four well-tested score types, Sum, is the simplest you can imagine, just assigning a certain amount of points for each correct testcases. 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.

7.2.1 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.

7.2.2 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, over all groups, of the minimum outcome for that group times the multiplier. The public score is computed over all groups in which all testcases within are public.

More precisely, the parameters string for GroupMin is of the form $[[m1, t1], [m2, t2], \dots]$, meaning that 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.

7.2.3 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.

7.2.4 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 and a certain number, the threshold, specified separately for each group.

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

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 `cmsContestExporter` and an importer `cmsContestImporter` 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

```
cmsContestExporter -h
cmsContestImporter -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, we encourage you to write importer and reimporter scripts, modeled upon those we wrote for the environment used in the Italian Olympiads, that can be run with the commands `cmsYamlImporter` and `cmsYamlReimporter` and inspected at `cmscontrib/YamlImporter.py` and `cmscontrib/YamlReimporter.py`. If you want to use the Italian environment there is a description in the next section, but please be aware that it has severe limitations: for example, many handles are in Italian and the support for complex task types is a bit cumbersome.

8.1 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 YAML file `task_name.yaml` that describes the task and a directory `task_name` that contains 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 providing confusing input to `cmsYamlImporter` can, unexpectedly, confuse it and create inconsistent tasks and/or strange errors. For confusing input we mean parameters and/or files from which it can infer no or multiple task types or score types.

8.1.1 General contest description

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

- `nome_breve` (“short name”, string): 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.
- `nome` (“name”, string): the contest’s name (description), shown to contestants in the web interface.
- `problemi` (“tasks”, list of strings): a list of the tasks belonging to this contest; for each of these strings, say `task_name`, there must be a file named `task_name.yaml` in the contest directory and a directory called `task_name`, used to extract information about that task; the order in this list will be the order of the tasks in the web interface.
- `utenti` (“users”, list of associative arrays): each of the elements of the list describes one user of the contest; the exact structure of the record is described [below](#).

The following are optional keys.

- `inizio` (“start”, integer): 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.
- `fine` (“stop”, integer): 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.
- `token_*`: token parameters for the contest, see [Tokens](#) (the names of the parameters are the same as the internal names described there); by default tokens are disabled.
- `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.

8.1.2 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.

- `nome` (“name”, string): the user real first name; defaults to the empty string.
- `cognome` (“surname”, string): the user real last name; defaults to the value of `username`.
- `ip` (string): the IP address from which incoming connections for this user are accepted, see [User login](#); defaults to `0.0.0.0`.
- `fake` (string): when set to `True` (case-sensitive `_string_`) set the `hidden` flag in the user, see [User login](#); defaults to `False`.

8.1.3 Task description

The task YAML files requires the following keys.

- `nome_breve` (“short name”, string): the name used to reference internally to this task; it is exposed in the URLs.
- `nome` (“name”, string): 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](#).

The following are optional keys.

- `timeout` (float): the timeout limit for this task in seconds; defaults to no limitations.
- `memlimit` (integer): the memory limit for this task in megabytes; defaults to no limitations.
- `resultati` (“results”, string): 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.
- `token_*`: token parameters for the task, see [Tokens](#) (the names of the parameters are the same as the internal names described there); by default tokens are disabled.
- `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.
- `outputonly` (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`.

8.1.4 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.

- `testo/testo.pdf` (“statement”): the main statement of the problem. It is not yet possible to import several statement associated to different languages.
- `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 (one of) the correct 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.
- `cor/correttore` (checker): 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. If the file is not present, a simple diff is used to compare the correct and the contestant’s output files.
- `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.

RANKINGWEBSERVER

9.1 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.

9.2 Running it

To start RWS you have to execute `cmsRankingWebServer` if you have installed CMS (by running `./setup.py install`), and execute `$REPO/cmsranking/RankingWebServer.py` otherwise (where `$REPO` has the same meaning as in the README).

9.2.1 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 `$REPO/cmsranking/Config.py`. If RWS is not installed then the `./examples` directory will also be checked for configuration files (note that for this to work your working directory needs to be `$REPO`). 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 SS 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.

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. There are three parameters to do it, three lists of the same length: `ranking_address`, `ranking_username` and `ranking_password`. The elements in `ranking_address` are lists of three elements: the protocol (either "http" or "https"), the address (IP address or hostname) and the port. 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 SS 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).

9.3 Managing it

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 ScoringService (SS for short). When SS is started for a certain contest, it'll 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 the submissions received so far. Then it'll continue to send new submissions as soon as they're scored and it'll update them as needed (for example when a users uses a token). Note that hidden users (and their submissions) won't be sent to RWS.

There are also other ways to insert data into RWS: send custom HTTP requests or directly write JSON files. They're both discouraged but, at the moment, they're the only way to add team information to RWS (see [issue #65](#)).

9.3.1 Logo, flags and faces

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

- the logo has to be saved in the top-level directory, named "logo" with an appropriate extension;
- the flag for a team has to be saved in the "flags" directory, named as the team's name with an appropriate extension;
- the face for a user has to be saved in the "faces" directory, named as the user's username with an appropriate extension.

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

9.3.2 Removing data

SS 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'll continue to be shown on RWS. There are several ways to fix that (presented in increasing order of difficulty and decreasing order of downtime needed).

- 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 SS for the contest you're interested in, to have it send the data again.
- You can stop RWS, delete only the JSON files of the data you want to remove and start RWS again. Note that if you remove an object (e.g. a user) you have to remove all objects (e.g. the submissions) that depend on it, that is you have to simulate the “on delete cascade” behavior of SQL by hand. (When you delete a submission remember to delete also the related subchanges).
- You can keep RWS running and send a hand-crafted HTTP request to it and it'll, all by itself, delete the objects you want to remove and all the ones that depend on it.

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

9.3.3 Multiple contests

Since the data in RWS will persist even after the SS that sent it has been stopped it's possible to have many SS 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.

9.4 Securing the connection between SS 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 he/she can't guess them but, since they're sent as plaintext in the HTTP request, he/she could read them if he/she can eavesdrop the communication channel between SS and RWS. Therefore we suggest to use HTTPS, that encrypts the transmission with TLS/SSL, when the communication channel between SS and RWS isn't secure. HTTPS doesn't only protect against eavesdropping attacks but also against more 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 certificate (and its private key) that is given to SS, that verifies its authenticity 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 doesn't need to use HTTPS since it's 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's of no harm. Note also that RWS will continue to accept incoming data even on the HTTP port, it's just that SS won't send it.

To use HTTPS we suggest you to create a self-signed certificate, use that as both RWS's and SS's `https_certfile` and use its private key as RWS's `https_keyfile`. If your SS 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 SS. Alternatively you may want to use a Certificate Authority to sign the certificates of RWSs and just give its certificate to SS. Details on how to do this follow.

9.4.1 Creating certificates

A quick-and-dirty way to create a self-signed certificate, ready to be used with SS 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 SS 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](#).

9.5 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 isn't 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 successfully used 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/>". Note that, however, an "unprefixed" port has to be publicly available and that is the port that has to be written in the `cms.conf` file since it's needed by SS (because it's currently unable to use a prefixed RWS, see [issue #36](#)).

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 transmission 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.

9.5.1 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:

9.6 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 8443) accessible from the internet, to make SS 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 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 :-).

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

LOCALIZATION

10.1 For developers

When you change a string in a template or in a web server, you have to generate again the file `cms/server/po/messages.pot`. To do so, run this command inside `cms/server/`.

```
xgettext -o po/messages.pot --language=Python --keyword=:1,2 \
*.py templates/admin/*.html templates/contest/*.html
```

When you have a new translation, or an update of an old translation, you need to update the `.mo` files (the compiled versions of the `.po` files). You can run `./setup.py build` to update all translations (and also do a couple of other things, like compiling the sandbox). Alternatively, run the following inside `cms/server/`.

```
msgfmt po/<code>.po -o mo/<code>/LC_MESSAGES/cms.mo
```

If needed, create the tree. Note that to have the new strings, you need to restart the web server.

10.2 For translators

To begin translating to a new language, run this command, from `cms/server/po/`.

```
msginit -d <two_letter_code_of_language>
```

Right after that, open `<code>.po` and fill the information in the header. To translate a string, simply fill the corresponding `msgstr` with the translations.

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

```
msgmerge <code>.po messages.pot > <code>.new.po
```

You can now inspect `<code>.new.po` and, if satisfied, move it to `<code>.po` and finish the translation.

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](#)).

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

11.1 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 [Updating CMS](#). If this fails, and you do not care about the content of the database, the following commands will **erase all content**, clear the schema and rebuild it.

```
python cms/db/SQLAlchemyAll.py redrop
python cms/db/SQLAlchemyAll.py
```

11.2 Servers

- *Symptom.* Message from ContestWebServer such as: `WARNING:root:Invalid cookie signature KFZzdW5kdWRlCnAwCkxxMzI5MzQzNzIwCnRw...`

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. If you want to delete the previous data, run it with the `‘-d’` option. See [RankingWebServer](#) for more details

- *Symptom.* Ranking Web Server misbehaving

Possible cause. Ensure you are running Tornado 2.0 or higher. (see [issue #2](#))

11.3 Sandbox

- *Symptom.* The freopen test fails.

Possible cause. Add the `dup3` syscall to the allowed list.

11.4 Importers

- *Symptom.* Importing a contest with ContestImporter fails.

Possible cause. The contest was imported with a previous version of CMS. Wait for us to provide update scripts for exports, or contact us for the fast solution.

11.5 Configuration

- *Symptom.* ResourceService keeps restarting its services.

Possible cause. As stated in the README, a reason for this could be that the “process_cmdline” in the configuration isn’t suited to your system. To find the one that suits you, you can run a service by hand (for example *cmsLogService*), then run `ps aux` and search for a process that looks like `/usr/bin/python2 /usr/local/bin/cmsLogService`. The “process_cmdline” corresponding to this would be `["/usr/bin/python2", "/usr/local/bin/cms%s", "%d"]`. This value is the default one and should work well on most Ubuntu systems, but for example on some Gentoo systems you may need to use `["/usr/bin/python2.7", "/usr/bin/cms%s", "%d"]`.