Olympiads in Informatics, 2012, Vol. 6, 86–99
© 2012 *Vilnius University*

Introducing CMS: A Contest Management System

Stefano MAGGIOLO[1], Giovanni MASCELLANI[2]
[1]*Mathematics Area,* SISSA/ISAS
Via Bonomea 265, 34136 Trieste, Italy
[2]*Faculty of Sciences, Scuola Normale Superiore*
Piazza dei Cavalieri 7, 56126 Pisa, Italy
e-mail: maggiolo@sissa.it, giovanni.mascellani@sns.it

Abstract. We present *Contest Management System* (CMS), the free and open source grading system that will be used in IOI 2012. CMS has been designed and developed from scratch, with the aim of providing a grading system that naturally adapts to the needs of an IOI-like competition, including the team selection processes. Particular care has been taken to make CMS secure, robust, developed for the community, extensible, easily adaptable and usable.

Key words: CMS, contest management system, grading system, IOI, IOI-like competitions.

1. Introduction

Programming contests have a long-standing tradition in computer science education, where the basic principle is to get young students interested by letting them compete on their skills alone. The main technical issues of organizing a programming contest can be roughly divided into three parts:

1. creating tasks and developing all the data required (statements, solutions, testcases, information on how to grade submissions, etc.);
2. when the contest is on-site, configuring the machines that the grading system and the contestants are going to use, in particular with respect to network security;
3. managing the actual contest (accepting and grading submissions, giving feedback, displaying a live ranking, etc.).

The aim of this paper is to describe a tool for dealing with the third point when contests follow the spirit and share the main features of the IOI and similar contests, such as CEOI, ICPC, national selections for the IOI, etc. To this end, we introduce *Contest Management System* (CMS), the free and open source grading system that will be used in IOI 2012.[1]

We designed CMS keeping some important keywords in mind.

- Secure: even though the main security measure for a contest is to isolate the contestants and the grading networks, obviously there must be at least one point of contact between the two; this must give the fewest possible ways of attacking the system.

[1]CMS is available at `https://github.com/cms-dev/cms`.

- Robust: an error or a critical condition in one part of the system must not take down the whole system; hot swapping of services and machines is not an exceptional condition but a standard procedure; the coherence of the state of the contest must always be ensured.
- Developed for the community: CMS must be easily accessible, free and open source (it is licensed under the GNU Affero General Public License (Free Software Foundation, Inc., 2007)); feature requests, bug reports and patches must be considered and applied whenever possible and without long delays; it must support localization of the contestants' interface.
- Extensible: new or rare task types and scoring methods must be easy to implement in the form of plug-ins; currently based on the current competition rules for the IOI, CMS should leave the possibility of reflecting the modifications of such rules by future hosts.
- Adaptable: CMS must not interfere with the first two points listed previously for organizing a contest, namely it must not mandate a minimum number of grading machines, or special network configurations (apart from the ones that prevent security issues), and must not require a particular method of preparing the tasks for the contest.
- Usable: CMS must be well-documented for contests administrators, developers, and contestants, and it must not require insights into internals of the system for running a contest.

The design and development of CMS started in the second part of 2010. Even though the authors had several years of experience using and contributing to the grading system developed for the Italian selection process, the design of CMS started afresh. Alpha versions started circulating in the first part of 2011 and were used in the selection process of the Italian team for IOI 2011. After that, more refined and stable versions have been used for official contests: OII 2011 (Italian Olympiad in Informatics) in October 2011, AIIO 2012 (Australian Invitational Informatics Olympiad) and FARIO 2012 (French-Australian Regional Informatics Olympiad), respectively in February and March 2012.

The paper is organized as follows: in Section 2 we motivate our decision to develop a new grading system; in Section 3 we describe the structure of CMS and explain the main design choices; in Section 4 we describe the process of setting up a contest using CMS; in Section 5 we show how data is manipulated inside the system to eventually produce a ranking; in Section 6 we briefly sum up our results and hopes for the future. Of course, this paper is just an introduction to the usage of CMS. Additional documentation, together with the source code, can be found on the CMS website (Boscariol *et al.*, 2010).

2. A Brief History of IOI Grading Systems

The earliest editions of the IOI did not use any grading system at all (Heyderhoff *et al.*, 1992): contestants were required to handle diskettes with their solutions, that were manually judged by the team leaders compiling, running, and scoring the solutions, according to some precise instructions by the Scientific Committee.

The progress of technology, together with higher reliability and lower costs of networks and equipment in general, made possible developing grading systems that are:

- automatic: where solutions are graded without human intervention;
- distributed: where solutions are submitted via a remote interface;
- live: where solutions are graded as they are submitted, providing a public ranking and/or feedback to the contestants.

The first semi-automatic system was used in IOI 1994 (Verhoeff, 1994), even if it still used diskettes to submit solutions. A fully automatic and distributed system was used in IOI 1999 (Piele, 1999), and IOI 2010 offered the first live evaluation (IOI 2010 Host Scientific Committee, 2010).

To the best of our knowledge, several different grading systems have been used during the period 1999–2011. Several times, the host country adapted its own grading system, used in national selection, to handle the tasks given in the IOI. Other times these systems were "borrowed" from other host countries. Finally, in IOI 2010 and 2011 the Marmoset grading system (Spacco *et al.*, 2006) has been used. It comes from outside the IOI community, thus it differentiates from the previous approaches.

The need for a system specifically designed for the IOI, with certain goals that we tried to achieve with CMS, has been obvious for many years (Verhoeff, 2002). Evident advantages are the transfer of expertise, the possibility to achieve a more polished product, the reusability of invested resources, and more. We decided to design CMS basing upon the previous experience of similar systems, which we analyzed in terms of the requirements and expectations from the ISC and the IOI community, as discussed below.

- Adapting the system used for the national selection of the host country is often the easiest solution, because of the experience of the country using its own software. But, often national selections and the IOI have slightly different needs; even more important, the IOI cannot expect that all host countries have a sufficiently developed or tested grading system. In these situations, having a standardized grading system could effectively lower the number of issues that a host country has to consider, possibly increasing the number of potential hosts.
- Using the system developed for the national selection of a different country (possibly a previous host, or not, as happened with the USACO system in IOI 2001; Kolstad and Piele, 2007) solves the issue of having a tested system ready to be used, but introduces some other problems: the current host may have no experience with the system, and often it is difficult to synchronize the adaptations needed for the current IOI with the main codebase, requiring additional work in the following years. It also often happens that these systems are privately developed, therefore they cannot be inspected by the community, nor used freely in other competitions such as national selections.
- Marmoset has been free since December 2009 (Pugh and Sims, 2009; Spacco *et al.*, 2012). It is a "system for handling student programming project submission, testing

and code review", and has been developed at the University of Maryland. Since it was born in the academic setting, and later applied to IOI 2010, it needed to be adapted to the context of the IOI. As far as we know, nobody has maintained the set of patches needed to be applied to Marmoset's codebase to comply with the competition rules of the IOI. On the other hand, Marmoset offers many more features that are related to teaching, thus increasing its size and complexity without any benefit for an IOI contest.

3. General Structure

CMS is organized in a modular way, with different services running, potentially, on different machines. When applicable, services can be replicated to allow scaling for larger contests.

CMS *Services*

The services, listed in Table 1, are Python programs built on top of AsyncLibrary, a custom-made RPC library using the asyncore framework (Sam Rushing and Python Software Foundation, 1996). Some of the services, called servers, provide also a second interface thanks to the web framework Tornado (Facebook, Inc., 2009), so they are able to both talk to the other services and to serve web pages to administrators or contestants. We require the services to be always responsive; hence remote requests must be completed in a short amount of time, or spawn a thread to handle the request. The only place where this behaviour is needed is in *Worker*, where a compilation or an evaluation cannot be split into multiple fast requests. All other services are single-threaded. The interactions among the services are shown in Fig. 1.

Table 1

The list of services composing CMS. Replicable is "Yes" when multiple instances of the service can cooperate to increase the capacity. The last two services (*ContestWebServer* and *AdminWebServer*) are also servers.

Name	Replicable	Duties
LogService	No	Receiving, aggregating and displaying all the logs of the system.
Worker	Yes	Running compilations and evaluations of submissions in a safe environment.
EvaluationService	No	Maintaining the queue of the jobs to be assigned to the *Worker*s.
ScoringService	No	Transforming the evaluation results into scores, and communicating them to the live rankings.
Checker	No	Calling the heartbeat function of all the services.
ResourceService	Yes	Collecting resource usage information about the machine in which it is running, and starting all the services on a machine.
ContestWebServer	Yes	Serving web pages to the contestants, accepting submissions and providing feedback.
AdminWebServer	No	Serving web pages to the administrators, configuring and managing the contests.

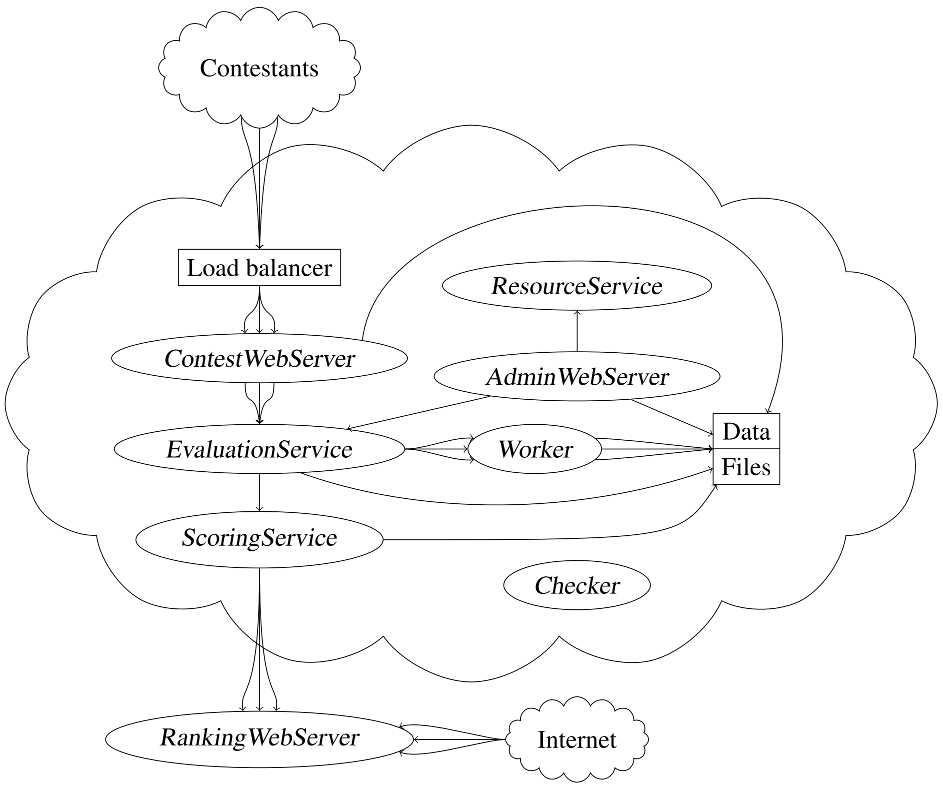

Fig. 1. Services and their interactions in CMS. An arrows represent a request of some kind, and multiple arrow heads mean that the service can be replicated. The services *Checker* and *ResourceService* talks with all the other services, and the relative arrows are omitted.

The nature of a grading system forces it to run untrusted code. To prevent contestants from cheating or even compromising the grading environment, all the evaluations and even the compilations of sources coming from contestants must run inside a sandbox. We use a (slightly modified) version of Martin Mareš's sandbox, which is part of the Moe contest environment (Mareš and Gavenčiak, 2001).

A special discussion is warranted for the program responsible for serving live rankings. Its name is *RankingWebServer* and can run in multiple instances on different machines. It is built on top of Tornado, but it is not part of the core structure of CMS, in the sense that it is allowed to run on machines that are not in the network used by CMS. This is because live rankings and the core system have different needs: the former needs to be accessed via a public network and may generate a large amount of traffic, whereas the latter must be private. CMS pushes data to *RankingWebServer* using HTTP queries, so the assumption that *ContestWebServer* is the only point of contact between CMS's network and the outside is preserved, since *RankingWebServer* does not query CMS.

Data Model and Implementation

From the viewpoint of CMS, the state of a contest is described by two kinds of objects: the *files* and the *data*.

- By files we mean something really similar to the usual concept of file: the only significant difference is that CMS does not consider them as part of a directory structure. They are used to store things that naturally look like a file, such as test-cases, task statements, submissions, compiled programs, etc.
- By data we mean a collection of objects implementing a set of classes defined in CMS; they are used to "glue" together the files and combine them in the actual description of the contest.

Files are addressed via their checksums using a hashing algorithm (SHA1 is used at the moment, but it would be easy to change it); in particular, they are immutable. This choice is justified by the huge simplification it gives: there are no concurrency problems, there is no need to care about local cache synchronization, it is easy to detect data corruption (although to save on CPU time, this is not done by default) and there is an automatic layer of data deduplication (if a file gets saved twice in the storage, only one copy is retained).

The file storage is implemented using PostgreSQL's "Large Objects", that can contain up to 2 GB of data each (PostgreSQL Global Development Group, 2012a). PostgreSQL supports accessing them using streams, so they can be copied to the local disk of a machine without loading them in the RAM. The previous and now discontinued implementation of the same service employed a custom service that kept files on the filesystem.

Data are stored in an SQL database using SQLAlchemy (Bayer, 2005) to do ORM (Object-Relational Mapping, i.e., automatically mapping the Python objects to their description in an SQL database). Most of the common SQL engines can be used with SQLAlchemy, although at the moment using anything other than PostgreSQL would require some minor modifications of CMS code, as explained below.

Files and data are stored in CMS using two different methods, hence, in theory, one could use two independent databases. At the moment this is not really possible, because CMS uses the same configuration line for both databases. This forces SQLAlchemy to be used with PostgreSQL, since it is the only database supported by the file storage. Changing this behaviour is not difficult and will be done as soon as there is evidence of its usefulness.

Handling Failures

It is important to notice that the database is the only unique point of failure of the whole system. It is thus essential that administrators can use standard procedures of replication and clustering to provide reasonable guarantees of uptime and resilience of data. The choice of using a widely-used database like PostgreSQL is supported, among other reasons, by the several mature solutions it offers to handle such situations; see for example (PostgreSQL Global Development Group, 2012b). CMS does not mandate the use of any of these solutions (it does not mandate the use of a replication or high availability solution at all), and, actually, the CMS authors have not yet settled on a preferred solution.

On the other hand, services can be started and stopped independently without affecting the system, of course apart from the downtime of the service itself. This allows administrators to quickly replace faulty hardware or software with identical copies, without caring about losing data or having to transfer information between the old and the new copy of the service, as long as the database is working correctly.

4. Walk Through

In this section, we will examine the procedure to run a contest using CMS, in order to get accustomed to its terminology and key concepts. We will assume that an on-site contest in the style of the IOI is being organized. The model we use is the competition rules for IOI 2011 (IOI 2011 Host Scientific Committee, 2011).

Network Setup

As mentioned, CMS does not interfere with the setup of the network. The only (sane) requirement comes from the fact that all the communication between services are not encrypted and there are no authentication methods for services; hence, the network in which CMS runs must be separate from the network of the contestants.

The only point of contact should be the HTTP port of the *ContestWebServer*, if the contest is small enough that one instance suffices, or the load balancer that splits the traffic amongst the instances of the *ContestWebServer*s (we provide a sample configuration to use nginx (Sysoev, 2004) as a load balancer).

To display a public live ranking, CMS's network must be able to access the (possibly remote) machine in which the *RankingWebServer*s are running; no other forms of communication are needed nor encouraged.

Task Preparation

Again, this is not an issue particularly relevant to CMS: the tasks can be set up using any method and any format. When everything is ready, the administrator should run an *importer*, that is, a program that translates the structure of the contest as it is in the file system to the internal format in the CMS database.

Of course, the importer depends on the particular format of the tasks. We provide an example importer, that works starting from the format used by the Italian Olympiads, and should be easy to adapt to different formats. We hope that in the future there could be a common task format for the IOI, that could be taken as a reference format for all the national Olympiads.

It is also possible to tweak the tasks after the import, or even to create the tasks from the interface of *AdminWebServer*. Nonetheless, using *AdminWebServer* to create contests and tasks is complicated by the large amount of different data needed to specify a task. To avoid mistakes and missing data, we encourage administrators to write their own importers.

There are some requirements that CMS enforces on the tasks: the *task type* (that is, the "instructions" on how to compile and evaluate the submissions) and the *score type* (how to translate the outcomes for each testcase to a score for the task) must be supported by CMS. All task types and score types, even the most standard ones, are in the form of plug-ins, loosely coupled to the core structure. In particular, it is easy to tweak types already defined or to add new ones.

Setting Up CMS

The administrators must install CMS on all the machines that will be running some services, and create a PostgreSQL database. They must decide, depending on the contest size and on the available resources, how many instances of a service to run. There are three replicable services: *ResourceService*, *ContestWebServer* and *Worker*.

We recommend to run an instance of *ResourceService* on every machine that is "interesting", that is, for which one would like to know the state in a given moment; in particular, for all machines where services are running.

For *ContestWebServer*, one can take advantage of multiple cores and provide as many instances as cores. In our experience, we reliably handled a contest with about 80 participants (OII 2011) with a single instance of *ContestWebServer*.

Finally, to provide the maximum fairness, the *Worker*s should be ideally one per machine (instead of one per core). A good rule of thumb is to allocate one *Worker* every 20 contestants, but this can significantly vary among contests.

After these decisions, the administrators need to make the relevant changes to the configuration file, that must be replicated on all the machines. In the future, this could be made easier by means of a configuration service. At the time of writing, a solution to deploy CMS to multiple machines by mean of network booting is nearly ready. This will allow administrators to avoid the process of installing and configuring the machines dedicated to the grading system.

Running CMS

When the configuration is synchronized on all the machines, the services can be started. Some of them need to work with a specific contest (*EvaluationService*, *ScoringService*, *ContestWebServer*), so it is also needed to have the contest stored in the database. Note that *AdminWebServer* does not need a contest, hence can be used to define new contests.

The services can be started together using *ResourceService*, which has an option to start automatically all the services in the given machine, and restart them automatically in case they die. As mentioned before, services can be killed and restarted without causing problems; one can also use the *AdminWebServer*'s "Resource usage" page to restart a specific service or disable the automatic restarting.

To ensure that everything works as expected, CMS offers to the administrators the possibility of running a test suite covering as many code paths as possible. The test suite is also a useful tool during the development, to ensure that new changes do not break existing code.

During the Contest

Once all the services are running, the contest is ready to start. The contest description contains the starting and finishing times (that can be changed on the fly, for example to delay a few minutes the beginning of the contest). Moreover, CMS supports a "relative timer" mode, where each contestant is given a certain amount of time starting from their first login into the system. This can be useful for online contests, to let contestants from different countries choose their preferred time slice to participate in the contest.

During a contest, contestants connect to *ContestWebServer* to submit their solutions, request feedback and check their statuses. The web interface also enables them to send questions and receive answers from the administrators, that can also send per-user messages and public announcements. Contest administrators can also use *AdminWebServer* to check that the contest is going smoothly, looking at statistics about the submissions and the services and their statuses; in case, they can request new compilations or evaluations of submitted solutions.

Once the contest is over, *ContestWebServer* stops accepting solutions and *EvaluationService* finishes processing the compilation and evaluation queue. Using *AdminWebServer*, administrators can decide to perform another evaluation round for either all or some selected submissions. In the end, *RankingWebServer* shows the final ranking of the contest. The complete state of the contest can be exported in a format that allows importing to another instance of CMS; moreover, administrators can also use an exporting tool to save the contest state in the format they prefer, without having to dump the whole database content. An example exporter is provided, that produces an output in the format used by the Italian Olympiad.

5. Internals of CMS

In this section we will describe in more detail the duties of the services, elaborating on how they interact and how the data flows through the system. Figure 2 represents this flow.

Submission of a Solution

Contestants interact with *ContestWebServer*; especially in case of national contests, *ContestWebServer*'s web interface can be translated in the local language using the standard tool gettext. Apart from read-only functions, such as downloading task statements and attachments, or inspecting the results of the submissions, there are two main functions contestants use: submitting solutions for a task, and test-releasing a submission.

Suppose that a contestant submits a solution. He has to provide exactly the files requested by the task, that are specified in the submission format table in the database. Output only (offline) tasks are an exception since the administrators may choose to let CMS "fill" the missing files using the ones from the previous submission.

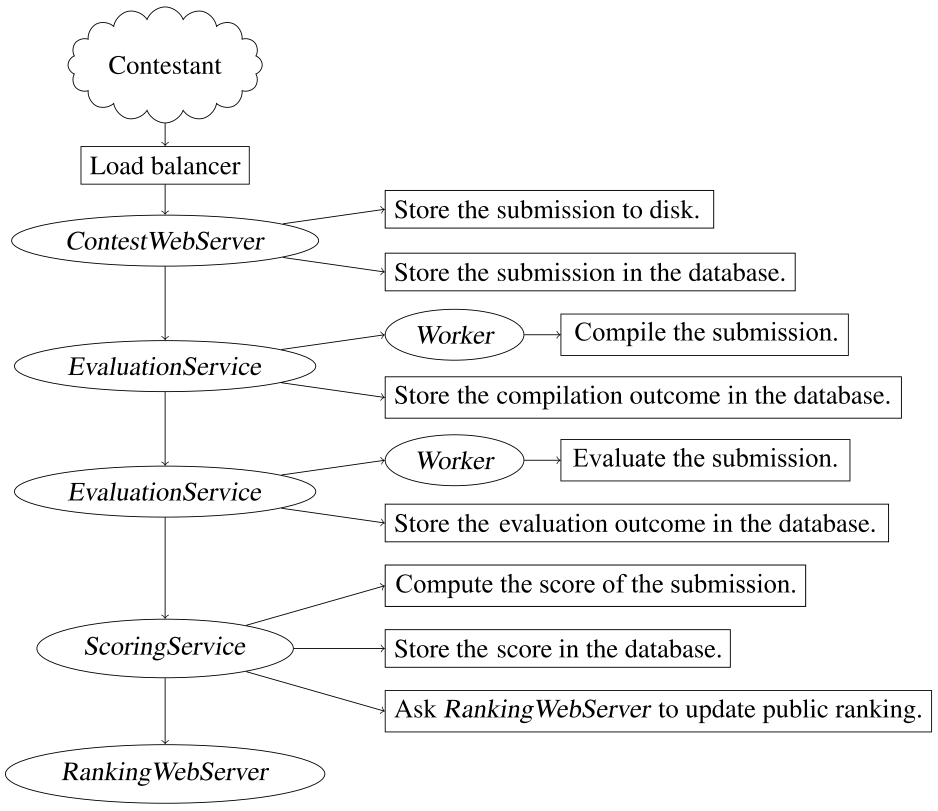
Contestant
Load balancer
ContestWebServer
Store the submission to disk.
Store the submission in the database.
EvaluationService
Worker
Compile the submission.
Store the compilation outcome in the database.
EvaluationService
Worker
Evaluate the submission.
Store the evaluation outcome in the database.
ScoringService
Compute the score of the submission.
Store the score in the database.
Ask RankingWebServer to update public ranking.
RankingWebServer

Fig. 2. Data flow of a submission through CMS.

Note that the task type, that is the plug-in in charge of specifying how to compile and evaluate the submission, uses the handlers provided by the submission format. For example, the submission format may force the contestant to submit a file with basename “encode” and one with basename “decode”; the task type may specify that they should be compiled into a single executable, and that the evaluation should run two instances of the program connected by a pipe. The task type can refer to another category of files, the *managers*, that are associated with the task and can be used, for example, to allow contestants to write just a function instead of taking care of the input/output, or to assess the correctness of the output when a plain diff is not enough.

If the submitted files respect the submission format, and some other predefined constraints are respected (files are not too large, submissions are not too frequent), *ContestWebServer* stores the submitted files in the database, and in a backup directory in the server, to allow recovery in case of a dramatic database failure. After that, it informs *EvaluationService* that a new submission is ready to be compiled.

Compilation and Evaluation

As a rule of thumb, services in CMS are informed that there is an action to perform. However they do not rely on the notification message, and also periodically check for actions that need to be done. For example, *EvaluationService* checks if there are submissions that need to be compiled or evaluated.

Either by receiving the message from *ContestWebServer*, or by checking the database, *EvaluationService* finds the submission to be compiled, and pushes it onto the job queue. A job is an action that a *Worker* performs with a single call: the compilation of a submission, or the evaluation on all testcases. *EvaluationService* ensures that, at any given moment, no more than one job per submission is in the queue.

When a *Worker* is available for a job, *EvaluationService* asks it to perform the job with the highest priority in the queue. We prioritize the compilations over the evaluations since they are often faster, and so they give an almost immediate feedback to the contestants.

As mentioned earlier, a job is usually long enough not to fit in the asynchronous framework of CMS, which requires the services to be responsive all the time: thus, the *Worker* spawns a thread dedicated to running the requested job. The main thread of the *Worker* does not accept other jobs before the current one is completed (nor *EvaluationService* should ask it to perform another job). The second thread instantiates the correct task type object and delegates the job to it, dying just after the job is completed and reported to *EvaluationService*.

A task type class needs to specify two operations: the compilation and the evaluation of a single testcase. Helper functions are provided to ease the writing of a task type: for example, it is easy to create and manage a sandbox, to run a program, or to move files from and to the sandbox. In the current version there are three task types already defined, for batch, output only, and communication tasks; the longest consists of about 60 Python statements. We plan to make it even easier to define new task types.

When the task type object has completed the job, *Worker* collects the result and send them back to *EvaluationService*, that writes them into the database. Indeed, *EvaluationService* is the only service that has “writing rights” on the parts of the database relevant to the compilation and the evaluation of submissions. If the job was a compilation, *EvaluationService* queues the evaluation; otherwise, it informs *ScoringService* that a submission is ready to receive a score.

Scoring

In the evaluation process, an *outcome* is assigned to each submission and testcase. An outcome is just a floating-point number, whose meaning depends on the score type of the task. When *ScoringService* receives the request from *EvaluationService*, or when it detects an evaluated submission without a score, it instantiates a score type object for the submission’s task, and asks it to translate the outcomes of the submission to an actual score. When the result is ready, it writes it into the database, being the only service with writing rights to the scoring tables.

A score type class must implement two methods: the first returns the maximum score obtainable in the task, while the other returns the score actually obtained by a submission, plus some explanatory details string that will be passed directly to the contestant. Each of these methods returns data in two different flavours: the complete data, to be displayed to administrators and also to contestants that have test-released the submission, and a partial summary, for contestants that have not test-released the submission. Indeed, each task specifies a list of *public* testcases, intended to be shown to contestants regardless of the releasing status of the submission. Score types are usually simpler than task types, with an average score type taking around 25 Python statements.

Within the structure of CMS, the score computation is influenced only by the outcomes of the specific submission. It would be easy to use some other data regarding the submission (for example, the time); on the other hand, using data from *other* submissions generates interesting problems (both on the development side and on the competition side) that the authors did not want to address in the first iteration of the system. For instance, a development problem is that the number of changes of scores could be quadratic in the number of submissions, and not linear; a competition problem is that the relative position of two contestants in the ranking could change due to a submission from a third contestant.

Publishing the Scores

The second duty of *ScoringService* is to send all the information needed by the (possibly multiple) *RankingWebServer*s. This information includes the details about the contest, and the lists of users and tasks, that are transmitted when *ScoringService* starts. The other class of information includes submissions, together with their scores, and test-releasing times.

The internal structure of CMS works around the concept of a contest; for example, a row in the contestants table in the database does not refer to a physical person, but to the participation of a person in a contest. Competitions that are distributed in more than one day should use a CMS contest per day, replicating the contestants. Nonetheless, *RankingWebServer* can display a complete ranking for such contests; indeed, being completely detached from the information stored in CMS's database, it can choose its own format: it merges all the entities received by the *ScoringService*, so that using the same *RankingWebServer*s for all the days of a competition effectively results in having a ranking for the whole competition.

RankingWebServer can handle and display other information about the contest and its participants, namely a picture of each contestant, and partition them into teams. This information is not currently provided by CMS itself, so they must be manually loaded into the *RankingWebServer*s beforehand.

6. Conclusion

We presented a grading system specifically designed for the needs of a programming contest in the style of the IOI. Nonetheless, we took care of writing a system that is extensible and configurable, in order to accommodate both possible future changes to the competition rules of the IOI, and different competitions (national selections, online contests, etc.).

The focus of the early development, that still carries on, has been directed towards providing a safe, tested infrastructure and a pleasant experience, in particular from the point of view of contestants. Still, facilities for administrators have been developed, and will be further expanded in the future. An agreement on a common format for developing and storing IOI-related programming tasks would be a great enhancement in this respect.

We hope that such a system can be further developed collaboratively. We are pleased that these collaborations with future hosts have already started, hoping that they will be fruitful for the IOI community.

Acknowledgments. The work in this paper has been partially supported by AICA (Associazione Italiana per il Calcolo Automatico) and MIUR (Ministero dell'Istruzione, dell'Università e della Ricerca). We would like to warmly thank Matteo Boscariol, developer of CMS together with the authors of this paper. We also acknowledge the important contributions of organizations and individuals that helped in all stages of the development of CMS: Luca Wehrstedt and Bernard Blackham for contributing to the design and development of substantial parts of CMS, the teams supervising the Italian and Australian selection process for IOI 2012 for providing testing and useful suggestions. A special thank you goes to Roberto Grossi (Chair of the IOI 2012 Host Scientific Committee) for his constant support and trust in our work. Finally, we would like to thank Bernard Blackham and Roberto Grossi for reading draft versions of this paper and giving valuable suggestions.

References

Bayer, M. (2005). *SQLAlchemy: The Database Toolkit for Python*. `http://www.sqlalchemy.org/`.
Boscariol, M., Maggiolo, S., Mascellani, G. (2010). *CMS, a Contest Management System*. `https://github.com/cms-dev/cms`.
Facebook, Inc. (2009). *Tornado Web Server*. `http://www.tornadoweb.org/`.
Free Software Foundation, Inc. (2007). *GNU Affero General Public License*. `http://www.gnu.org/licenses/agpl-3.0.html`.
Heyderhoff, P., Hein, H.-W., Krückeberg, F., Miklitz, G., Widmayer, P. (1992). *Final Report International Olympiad in Informatics 1992 Bonn/Germany*. `http://www.ioinformatics.org/locations/ioi92/report.html`.
IOI 2010 Host Scientific Committee (2010). *Competition Rules for IOI 2010* (Draft 1.0). `http://www.ioi2010.org/rules.shtml`.
IOI 2011 Host Scientific Committee (2011). *Competition Rules for IOI 2011* (Draft 1.0). `http://www.ioi2011.or.th/rules`.
Kolstad, R., Piele, D. (2007). USA Computing Olympiad (USACO). *Olympiads in Informatics*, 1, 105–111.
Mareš, M., Gavenčiak, T. (2001). *The Moe Contest Environment*. `http://www.ucw.cz/moe/`.
Piele, D. (1999). *Report from Turkey*. `http://www.ioinformatics.org/locations/ioi99/don99.shtml`.
PostgreSQL Global Development Group (2012a). *PostgreSQL 9.1 Manual. Large Objects*. `http://www.postgresql.org/docs/9.1/static/high-availability.html`.

PostgreSQL Global Development Group (2012b). *PostgreSQL 9.1 Manual. High Availability, Load Balancing, and Replication*.
`http://www.postgresql.org/docs/9.1/static/high-availability.html`.
Pugh, W., Ryan, S.W. (2009). *Marmoset*. `https://code.google.com/p/marmoset/`.
Sam Rushing and Python Software Foundation (1996). *Asyncore: Asynchronous Socket Handler*.
`http://docs.python.org/library/asyncore.html`.
Spacco, J., Pugh, W., Ayewah, N., Hovemeyer, D. (2006). The Marmoset project: An automated snapshot, submission, and testing system. In: *OOPSLA Companion*, 669–670.
`http://dl.acm.org/citation.cfm?doid=1176617.1176665`.
Spacco, J., Pugh, W., Ayewah, N., Hovemeyer, D. (2012). *Marmoset*.
`http://sourceforge.net/projects/marmoset/`.
Sysoev, I. (2004). *Nginx*. `http://nginx.org/`.
Verhoeff, T. (1994). *The Sixth International Olympiad in Informatics: A Trip Report*.
`http://www.ioinformatics.org/locations/ioi94/rprt-nl/index.html`.
Verhoeff, T. (2002). *Minutes of the International Scientific Committee Meeting 6, 17th May 2002*.
`http://ioinformatics.org/admin/isc/iscmeetings/meet-06/minutes06.txt`.

S. Maggiolo is a PhD student in geometry at SISSA/ISAS, Trieste. He participated in IOI 2002, winning a bronze medal and in IOI 2003. Since 2006 he collaborates with the training and selection process for the Italian team at IOI, and has been an observer in IOI 2009 and the deputy leader of the Italian team in IOI 2011. He is a member of the HSC for IOI 2012 and one of the main author of CMS, the contest system that will be used in IOI 2012.

G. Mascellani has been a contestant in IOI 2007 and 2008, winning a silver and a bronze medal. He is a student of mathematics in Pisa at the Scuola Normale Superiore and collaborates in training the Italian team for IOI. He is a member of the HSC for IOI 2012 and is one of the main authors of CMS, the contest system that will be used in IOI 2012. He is a Debian Developer.