SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2003; **33**:567–581 (DOI: 10.1002/spe.522)

Mooshak: a Web-based multi-site programming contest system

José Paulo Leal*,† and Fernando Silva

DCC-FC & LIACC, Universidade do Porto, Rua do Campo Alegre, 823, 4150-180 Porto, Portugal

SUMMARY

This paper presents a new Web-based system, Mooshak, to handle programming contests. The system acts as a full contest manager as well as an automatic judge for programming contests. Mooshak innovates in a number of aspects: it has a scalable architecture that can be used from small single server contests to complex multi-site contests with simultaneous public online contests and redundancy; it has a robust data management system favoring simple procedures for storing, replicating, backing up data and failure recovery using persistent objects; it has automatic judging capabilities to assist human judges in the evaluation of programs; it has built-in safety measures to prevent users from interfering with the normal progress of contests. Mooshak is an open system implemented on the Linux operating system using the Apache HTTP server and the Tcl scripting language.

This paper starts by describing the main features of the system and its architecture with reference to the automated judging, data management based on the replication of persistent objects over a network. Finally, we describe our experience using this system for managing two official programming contests.

KEY WORDS: Web application; contest management; program evaluation; automatic judging

INTRODUCTION

For many years now, the International Collegiate Programming Contest (ICPC) has organized and conducted yearly world programming championships, under the patronage of Association for

*Correspondence to: José Paulo Leal, DCC-FC & LIACC, Universidade do Porto, Rua do Campo Alegre, 823, 4150-180 Porto, Portugal.
†E-mail: zp@ncc.up.pt

Contract/grant sponsor: Project Ganesh; contract/grant number: PRAXIS/P/EEI/14232/1998
Contract/grant sponsor: LIACC

Received 29 January 2002
Revised 6 August 2002
Accepted 20 December 2002

Published online 5 March 2003

Computing Machinery (ACM), for college students [1]. This contest is a two-tiered competition among teams of students. Up to 60 of the winning teams of the regional contests advance to the world finals. The participation numbers are impressive: in 2001 there were more then 3000 teams, from 1150 universities, 70 countries, participating in 29 regional contests distributed among 94 locations. The main motivation behind such an organization is to provide students with an opportunity to demonstrate and sharpen their problem solving and computing skills.

In a typical contest, teams composed of three undergraduate students get a set of nine problems which they have to solve in five hours on a single computer, programming either in C, C++, Java, or Pascal. During the contest, teams can submit solutions, in source code, to the given problems. The submissions are typically evaluated by a human judge and it involves compiling the program, running it with a set of predefined test inputs, comparing the results obtained with those expected in the test outputs, and then marking the submission against a marking scheme. A submission is marked as accepted only when it successfully passes all the test cases. Usually, there are maximum execution times associated with the tests, and therefore for a program to pass the tests successfully it must not only produce the correct results but also has to do so within the specified time limit. This ensures that the solutions produced by teams are reasonably efficient and not just a brute force approach.

Preparing and running programming contests with many teams (60 teams at the finals) competing is an enormous task. Designing an adequate environment for mediating the communication between teams and judges is highly challenging; it must allow the former to ask questions, to submit their solutions, and to receive information; and the latter to answer questions from the teams, to judge their submissions, and to report back the evaluation results. Some systems have been developed to try and fulfill this purpose. PC^2 is the system that has been used in recent world finals [2]. It has capabilities for managing single and multi-site contests and since it has been developed in Java it can be run on either Windows or Unix operating systems. PC^2, however, lacks an important feature, automated judging capabilities, and therefore many human judges are required to run a contest. Other systems have been developed elsewhere, namely at Ural State University [3] and at Valladolid University [4] were they have a 24 hour online automated judge that interacts with users through e-mail. These universities also provide a Web-based problem archive with more than 1000 problems in total, most of which have been used in previous programming contests.

In this paper, we describe the design and implementation of a new Web-based multi-site programming system, Mooshak. The system acts as a full contest manager and as an automatic judge for programming contests, with the capability of running single and multi-site contests, as well as behaving as a 24 hour online judge. Mooshak is an open system implemented on the Linux operating system using the Apache HTTP server with Tcl scripts communicating via the CGI protocol. The system is available from its home page at http://www.ncc.up.pt/mooshak; on this page the reader can use a public version of Mooshak, both as a contestant and as judge, read its on-line help, and download an archive with the system. Mooshak is distributed under the so called 'Artistic License' and thus qualifies as certified Open Source Software by the Open Source Initiative [5].

Mooshak innovates in a number of aspects: it has a scalable architecture that can be used from small single server contests to complex multi-site contests with simultaneous public online contests and redundancy; it has a robust data management system favoring simple procedures for storing, replicating, backing up data, and failure recovery using persistent objects; it has automatic judging capabilities to assist human judges in the evaluation of programs; it has built-in safety measures to prevent users from interfering with the normal progress of contests.

Mooshak grew from previous experience within the group with the Ganesh system [6], a Web-based learning environment of Computer Science topics that we have been using for several years now. Ganesh includes a module to automatically evaluate student exercises with a stronger requirement in that marks must be given to partial solutions. Ganesh already includes domains to deal with programming languages, graphical interfaces, and operating systems, namely C, Prolog, Tcl/Tk, HTML, assembly, and SQL. For those unfamiliar with the Hindu mythology, lord Ganesh is the elephant-headed God with a broken tusk that is always accompanied by a small mouse named Mooshak.

The reminder of the paper is organized as follows: first we give an overview of Mooshak, mainly focusing on its external view and functionalities; then we describe its architecture and explain in more detail the decisions made in the implementation of the system; next, we describe our approach concerning system security, safe execution of contestants' programs, and data backup to enable system recovery; then, we proceed by describing our approach towards automatic judging; finally, we describe our experience in using the system in two official contests, draw some conclusions, and advance ideas towards future work.

SYSTEM OVERVIEW

Mooshak is a client–server application to fully manage and run programming contests. It is also Web-based and therefore all of its functionalities are accessible through interfaces deployed on a Web-browser, irrespective of the operating system where the browser is running. These interfaces use the HTML 4.0 frameset and no processing is made on the browser, except for some data input validations that are implemented with ECMAScript[‡]. Java and plugins were deliberately avoided to simplify the use of the interface by any machine on the Internet.

Mooshak accommodates a number of user-oriented views with different system requirements and access permissions to the data. The users currently supported are contestants, judges, administrators, and the general public. User access is controlled by authentication, except for the general public. Next, we describe the features available within each user view.

Contestants' view

During a programming contest most of the team work is done locally on their single workstation using standard programming tools such as text editors, compilers, and debuggers. Communication outside the team is mediated by Mooshak through the contestants' interface view. This view allows contestants to submit source programs for evaluation as intended solutions for problems; ask questions to the judges and access all questions posed by contestants and corresponding answers given by the judges; access the list of all submissions and corresponding marks; access the current contest classification; print the code programs in development; and visualize on the browser the problem descriptions[§].

[‡]JavaScript 1.2.

[§]This is especially important on public contests open to everyone on the Internet.

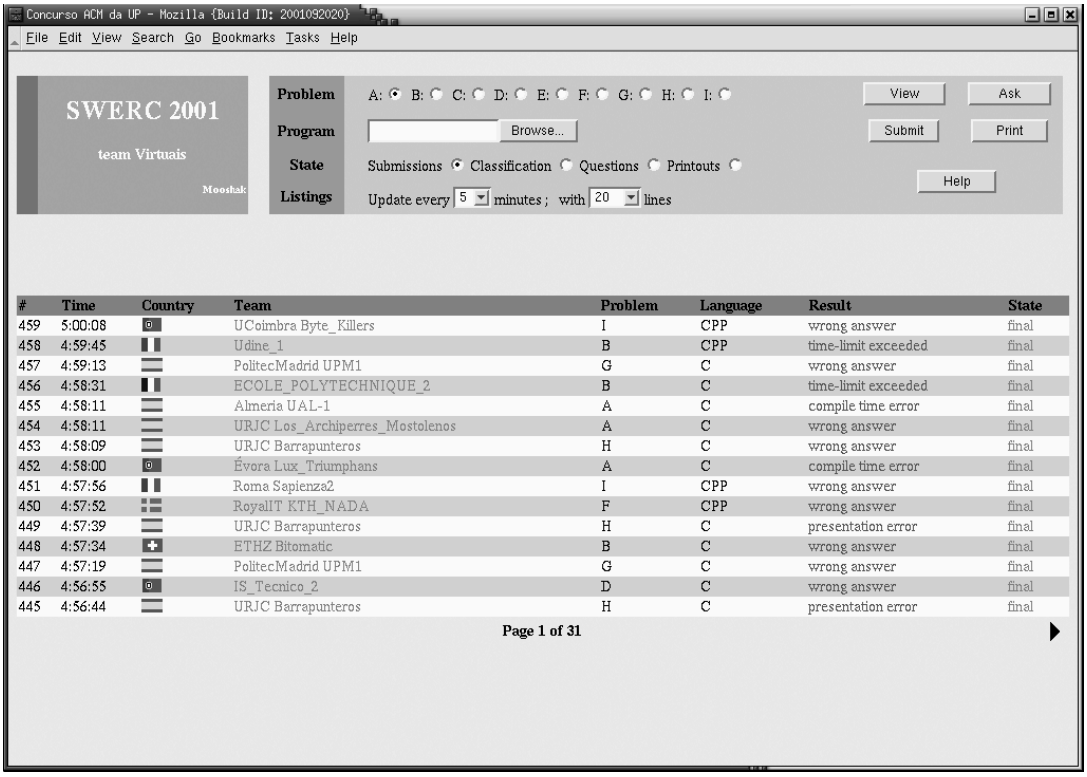

Figure 1. The contestants' view.

The contestants' interface, as shown in Figure 1, and the area at the top (the header) is used to identify the contest and the team (left part), to aggregate the selections (center part), and to display the available command buttons (right part).

The central area is used to list information related to the last command processed. For instance, after submitting a program the listing of submissions is updated, showing the available information for the latest submissions. The listings are paginated and are automatically updated. The interface allows the contestants to control the update rate and page size of the listings.

Judges' view

Even though Mooshak evaluates submissions automatically, it provides a number of functions to help human judges in exercising a finer control on the judging process. From our experience so far, the automatic judging is very much trusted, nevertheless problems may arise unexpectedly such as a

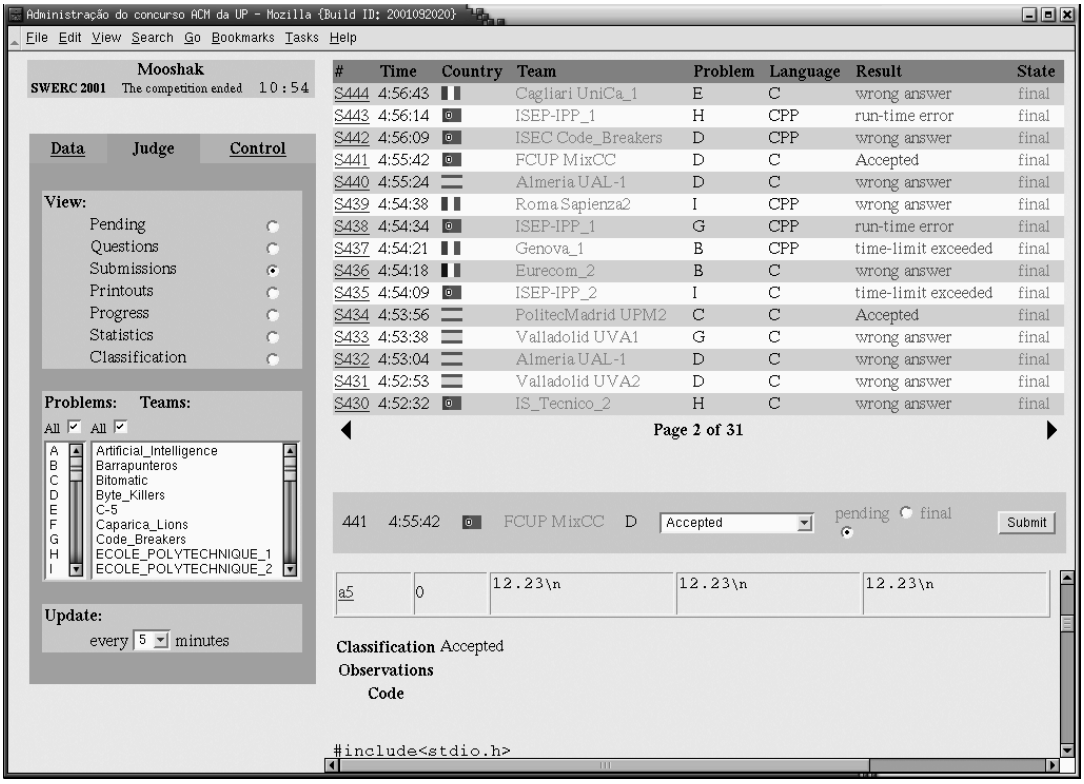

Figure 2. The judges' view.

system resource failure or a mistake in a test case. Mooshak is highly flexible in allowing the judges to re-evaluate submissions without a team being penalized for it, and thus undoing whatever had gone wrong. Through the judges' interface, judges can also answer questions posed by the teams, access all submissions made, view the current classification, and control the handling of printouts produced by teams. The judges' interface is illustrated in Figure 2.

The judges navigate through the information using listings similar to those presented to the contestants. The main differences are the links in each row that open forms for managing submissions, questions, and printout requests. The judges view is divided in two main areas: a control area on the left and a workspace on the right. On the control area the judges select the type of listing to be displayed on the workspace and they may filter the list by specifying a criteria on the problems and teams. By filtering the listings, a human judge can monitor the activity of the problems which he or she knows best.

The basic listings available to the judges show the submissions, questions, and printouts. This view includes a listing presenting the pending transactions of any of the three basic types, i.e. non-validated submissions, unanswered questions, and undelivered printouts. The judge also has access to the classification, statistics, and contest progress listings.

Administration view

The administration interface allows contest directors to set up all necessary data to run the contest. By contest data we mean the problem set (problem descriptions and test cases), teams composition and authentication (passwords), and programming languages with corresponding execution commands and compilation flags. Our first experiences in managing programming contests were focused on the features for contestants and jury. Contest administration was carried out by editing configuration files and moving data files using shell commands. This approach had two major problems: it was a complicated and error prone task, and was difficult to use in an 'emergency' situation during the contest. Even though there is a great effort in making the contest data consistent, it may be necessary to edit some test input or output during the contest itself. This happened once and it was enough to convince anyone about the importance of having flexible administration features. The editing capabilities introduced in the current administration interface overcome this difficulty as it is possible to virtually edit or replace any piece of data. The main requirements identified for the administration interface were the following.

History. A contest is actually a series of events. Before the main event there are usually one or more training sessions. In the case of preliminaries the contest may be broken into several events during a year. Thus, the administration interface must allow the management of several successive events, reuse data from one event to another (e.g. teams), and record all the transactions from previous events.

Navigation. For each event Mooshak records the contest data (problem set, teams, languages) and transactions (submissions, questions and printouts). The contest data have their own structure: the problem set includes several problems, each one with several tests; the teams are composed of contestants (three effective, one reserve and one coach) and are aggregated into institutions. Thus, navigation through the data must be simple and intuitive.

Editing. Mooshak has basically two types of data: text strings and text files. The first are used for configuring atomic values like the starting and finishing date/time or the command line to compile a program for a given language. Examples of data files are program solutions, problem descriptions, or input test data. The interface must include means of inserting and editing both types of data.

Commands. The preparation of contest data requires the execution of several commands at different times, for example importing/exporting problem sets, generating passwords for teams, printing certificates of achievement, checking problems timeouts, etc. These commands must be simple to find and use with the appropriate data.

To meet these requirements the administration view of Mooshak is as illustrated in Figure 3. The navigation tree follows a familiar interaction pattern that most users will recognize immediately

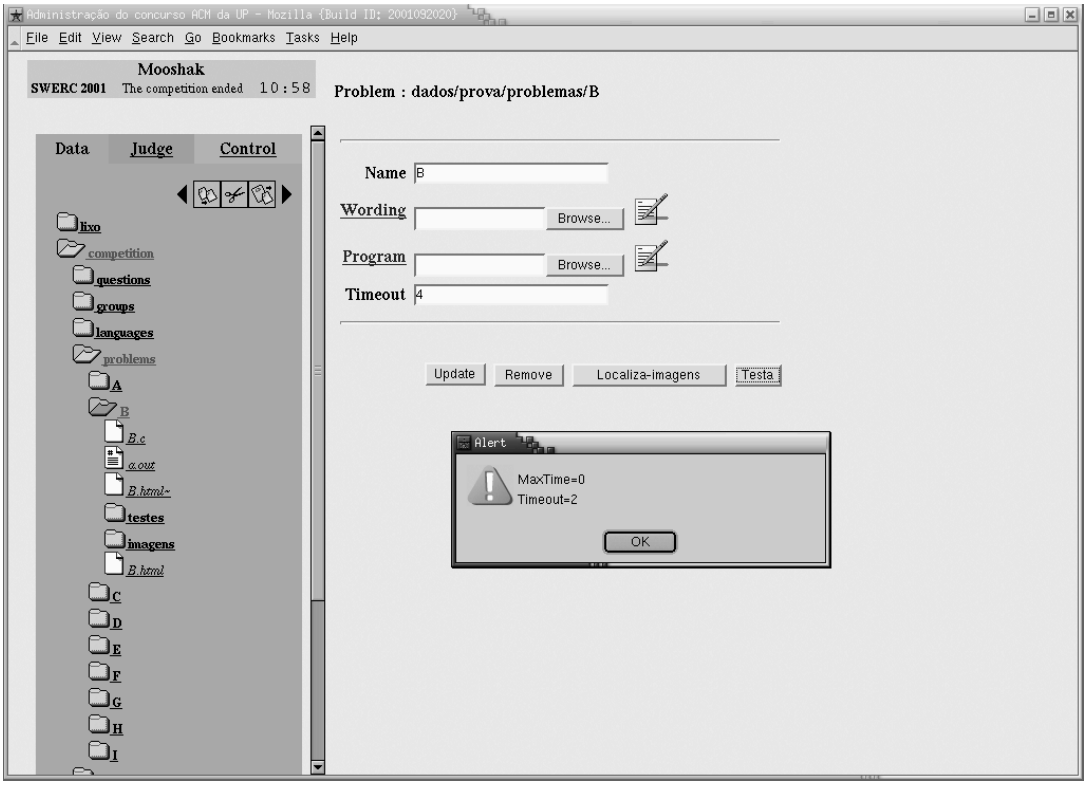

Figure 3. The administration view.

since it is used in several file managers in various operating systems. To capitalize on the metaphor, the navigation tree actually has a folder for each branch and a sheet for each leaf, and each icon anchors a link to that position. By navigating within the tree, the user can quickly select any branch.

When a branch is selected the corresponding form is displayed in the workspace on the right. The forms include a header showing the type of data and the path in the tree to get to this form. The form also includes all the fields related to the selected branch. As mentioned before, these can either be text values, editable in the field, or text files. In the last case, the user may either upload a file or edit its current value. In the footer of the workspace several buttons indicate to the user which commands may be executed with this branch. The Update and Remove buttons are always available. The other buttons may vary according to the type of data being edited.

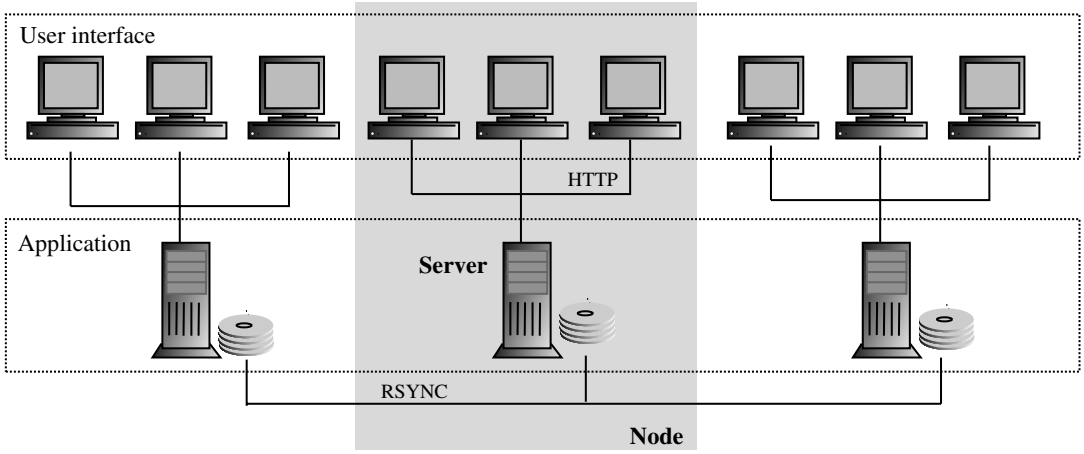

Figure 4. The architecture of Mooshak.

ARCHITECTURE

The architecture of Mooshak is that of a typical Web application: a client–server framework connecting the users with the machine where problem submissions are recorded, analyzed, and validated. Figure 4 represents the architecture of Mooshak, structured in vertical and horizontal layers. The *user interface layer* on the top includes the machines used by the teams, human judge, administrators, and general audience to access the system. The graphical user interface is rendered in HTML and interaction data are communicated back to a *server* on the *application layer* using the HTTP protocol. The application layer is composed of a set of servers, each using its own data management system.

Mooshak also has a vertical structure, where each layer groups a set of client machines to their server. We call these vertical layers *nodes* since they are the basic component of a Mooshak *network*. A simple contest may be managed using a single Mooshak node.

We will now concentrate on detailing the implementation of a Mooshak server, emphasizing its automated judging and its data management approach using persistent objects. Then, we describe how a network of Mooshak nodes is used to deal with issues such as backup, load balancing, and multi-site contests.

Mooshak server

The Mooshak server is an Apache HTTP server extended with external programs using the CGI protocol, running on a Linux operating system. Apache is responsible for the communication, authentication, access control, and encryption. The external programs (CGIs) are responsible for generating HTML interfaces and processing form data. They are implemented in Tcl [7] and manage data using persistent objects over the file system. Tcl was chosen for being a scripting language with

powerful tools for process management and for interfacing the file system. These features were used to implement the automated judging and data management with persistent objects as described in more detail next.

Automated judging

The automated judge is the cornerstone of Mooshak. Its role is to classify a submission according to a set of rules and produce a *report* with the evaluation to be validated by a human judge. A submission is composed of data relevant for the evaluation process, that is the program source code, the team-id, the problem-id, and the programming language (this is automatically inferred from the source code file extension). Submissions are automatically judged and almost instantaneously displayed to the teams, although initially in a pending state. The human judges have the responsibility of validating pending classifications, making them final, and occasionally modifying initial classifications. A classification may have to be modified as a result of changes in the compilation and execution conditions (e.g. changes in test cases). Re-evaluation produces another report that has to be compared with previous ones.

The automated judging can be divided in two parts according to the type of analysis.

(i) **Static analysis** checks integrity of data related to the submission and, if successful, produces an executable program.

(ii) **Dynamic analysis** is performed after a successful static analysis and is composed of one or more executions of the program.

Static analysis starts by verifying if the submitted problem has already been solved (by the same team), in which case the submission is rejected and no classification is given. Then it goes on to confirm the verifications made by the interface, i.e. by double checking the submitted data for team ownership and problem-id. If these verifications fail it probably means that the submissions did not come from the contestants interface (where the values would have been checked) and is thus marked as an 'invalid submission'. At this stage the size of program source is also verified to prevent a denial of service attack by submitting a 'program too long'. Finally, if it succeeds in this verification, it compiles the submitted program using the compilation command line defined in the administration interface. Mooshak may be more or less tolerant according to the flags chosen for each compiler. An error or compiler warning detected in this stage aborts the automated judging and dynamic analysis is skipped. Table I lists the verifications performed during static analysis and the associated classifications upon failure.

Dynamic analysis involves the execution of the submitted program with each test case assigned to the problem. A test is defined by an input and an output file. The input file is passed by the standard input to the program execution and its standard output is compared with the output file. The errors detected during dynamic analysis determine the classifications listed in Table II. Each classification has an associated severity rank and the final classification is that with the highest severity rank found in all test cases. The highest severity is given to the rare situation where the system has an indication that the test failed due to lack of operating system resources (inability to launch more processes, for instance). The lowest severity is the case where no other error was found, using the test cases, and therefore the submission is accepted as a solution to the problem.

Table I. Static analysis verifications.

Verifications	Classification
Team	Invalid submission
Language	Invalid submission
Problem	Invalid submission
Program size	Program too long
Compilation	Compile time error

Table II. Classification and severity of program tests.

Severity	Classification
6	Requires re-evaluation
5	Time-limit exceeded
4	Output too long
3	Run-time error
2	Wrong answer
1	Presentation error
0	Accepted

The automatic judge marks an execution as 'Accepted' only if the standard output is exactly equal to the test output file. Otherwise the output file and standard output are *normalized* and compared again. In the normalization both outputs being compared are stripped of all formatting characters. If after this process the outputs become equal them the submission is marked as having a 'presentation error'; otherwise it is marked as a 'wrong answer'.

In the current implementation the normalization trims white characters (spaces, newlines, and tabulation characters) and replaces sequences of white characters by a single space. This is a general normalization rule since white characters are only used for formatting. In a specific problem other classes of characters could have the same meaning. For instance, in a problem where the only meaningful characters are digits, other characters, such as letters or punctuation, could be treated as formatting characters. This cannot be done in general since many problems have a meaningful output that includes letters. This feature will require having a meaningful class of characters defined for each problem output.

The compilation and the execution of programs are the two most insecure points of a contest management system. Provided it fits in a single file, a team can submit virtually any program in one of the contest languages, including a bogus or malicious program capable of jeopardizing the system and ruining the contest. For that reason Mooshak compiles and executes programs in a secure environment, with the privileges of an insecure user and with several limits. Most of these limits are independent of the problems, with the exception of execution timeout that is adjusted to each problem. The execution timeout for each problem is determined before the contest and it is the maximum time taken by the judges solutions, with all test cases, rounded up to the next integer (in seconds). The execution timeout

is used by the system to ensure more efficient solutions were a brute-force solution would be an obvious candidate. Other time limits, such as the compilation timeout and the real time execution timeout, are independent of the problem and have been introduced to ensure the integrity of the system against bogus submissions. Real time execution timeout deals with programs blocking on input (trying to read past the end-of-file), therefore not using CPU time, and would not be caught by execution timeout. The compilation timeout deals with problems resulting from the misuse of template expansion in languages such as C++. Mooshak also enforces other limits such as source code size, output size, process data, and stack size.

Data management

A typical Web application uses a relational database management system (RDBMS) for managing persistent data. Mooshak instead bases its data management directly over the file system. The main motivation was that the data used by Mooshak is conveniently represented in plain files—source and object programs, data files, HTML files, and images. The number of files required by Mooshak and the number of procedures reading and writing them forced us to define a methodology to mediate the interaction with the file system. This methodology, that we call *persistent objects*, gives an object-oriented flavor to Mooshak since it encapsulates data files with the methods that operate on them.

In fact, persistent objects blend data, recorded on the file system, with code written in the Tcl [7] scripting language. All data used by Mooshak are maintained in plain files grouped in file system directories. Each of these directories is viewed as an object, belonging to a class managed by a Tcl module. This module encapsulates the methods supported by a particular class of objects.

Figure 5 represents a small part of the tree structure where Mooshak's data are hanged. Consider the object *submissions/sub2* containing the program submitted by a team, solving a given problem. After loading this object, it can receive the message *analyze* that once executed will produce a report file within the same object/directory. All submissions have a common parent, the object *submissions*. This object provides the methods that operate over the set of submissions, such as the method *list* that produces a listing of all submissions.

Mooshak network

A single Mooshak node, a server accessible through a set of Web clients on users machines, is sufficient for running a small programming contest (i.e. a contest with up to 20 teams) where reliability is not at a premium. Running an official contest, with a concern for reliability and a larger number of teams, distributed in several sites and a simultaneous online contest requires a more complex setup, with a network of interconnected nodes.

A link between the Mooshak nodes X and Y requires the replication of the contest data from the server X to the server Y. The main reasons for replicating contest data between Mooshak servers are to support the following.

System backup. Replication is used to maintain a backup system, with an updated version of the contest data, so that it can replace one of the servers in case of hardware failure.

Online contest. Replication propagates the contest data to a server with Internet access used to maintain an online contest simultaneously with an official local contest.

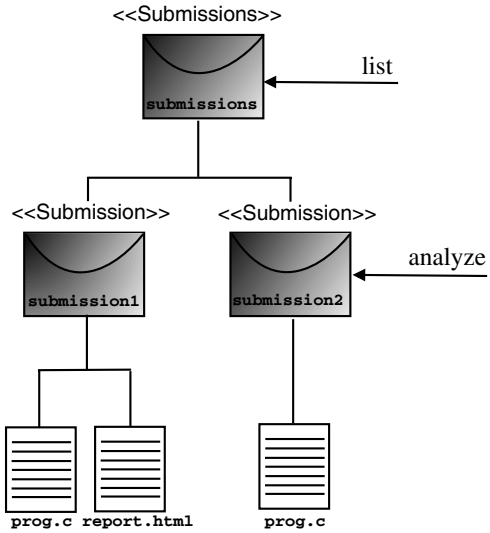

Figure 5. Persistent objects.

Load balancing. Several servers distribute load among them and replicate their data to the others. In this case each server is assigned to a set of users, for instance contestants to a server and judges to another, or contestants in different rooms to different servers.

Multi-site contest. This case is similar to the previous one but servers are in distant locations.

The Mooshak network configuration for a particular contest may contain several of these links. Figure 6 represents the network for a contest taking place simultaneously in two sites, A and B, the first using two servers (Server A1 and Server A2) for load balancing and the second using just one server (Server B). Each site has a backup with an updated version of the contest data, capable of replacing any of the main servers in case of failure. Site A also maintains an online version of the contest where anyone on the Internet can compete against the official contestants physically located at either site A or at site B without interfering with them. Some nodes are connected in unidirectional links, such as those connecting servers with the backup nodes or online-contest servers, and other are bidirectional, such as those connecting contest servers.

The Mooshak replication uses the *rsync* remote-update protocol. This protocol updates differences between two sets of files over a network link, using an efficient checksum-search algorithm. The replication procedure is invoked frequently to propagate changes to other servers, typically every 60 seconds, and copies only the data that have been changed since the last replication. The object files produced by the compilation of programs are not replicated, just the evaluation reports. If necessary the programs may be re-evaluated on a different machine.

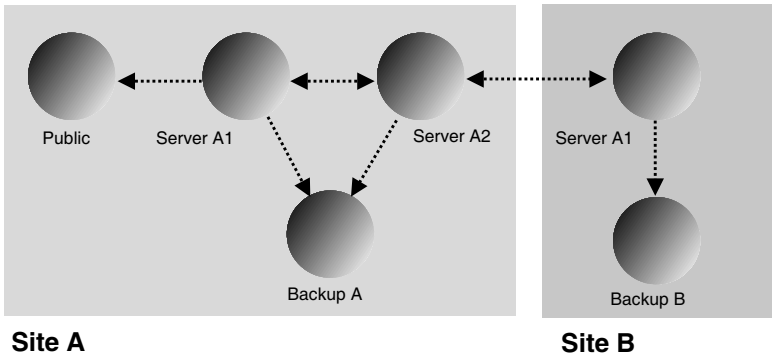

Figure 6. The network of Mooshak nodes.

The main issue with replication is the consistency of contest data, namely that no data fail to be replicated or are overwritten by replicated data. To guarantee that no data fail to be replicated we must ensure that there is a replication path connecting all servers interfacing with official contestants—the main servers.

To address the problem of data being overwritten, we must differentiate between contest definition data (such as teams, problems and programming languages) and contest transactions (such as submissions, questions, and printouts). Of these two, contest transactions, especially submissions, are particularly important. To guarantee uniqueness all transaction data are keyed with a timestamp, the team-id, and the problem-id. Thus, if team-id is unique in the system, and transactions from the same team are consistently sent to the same server, then there is no danger of losing transactions due to overwritten data since each transaction key is also unique.

Contest data are not, in principle, changed after the beginning of the contest. It should be updated in a single node for the sake of consistency, and that node must have a path to every other node in the network. The only exception to this case is the creation of teams for online-contest servers, as we allow contestants to register during the contest. If load balancing is used for online-contest servers then it is important to assign team creation to a single server. Otherwise, two teams with the same name, and same group, registering at the same time on different servers could (although not very likely) share the same record.

For the above setup to work properly, all server clocks must be synchronized. This can be achieved using the Network Time Protocol [8].

EXPERIENCE

Mooshak has been used to manage several programming contests, culminating in SWERC 2001, the Southwestern Regional ACM Programming Contest [9]. The system was also used in the

preliminary Portuguese Programming Contest, MIUP 2001 [10], and several local competitions within the University of Porto. The two major events, SWERC 2001 and MIUP 2001, had simultaneous online contests and were preceded by several practice sessions. The public contests were open to anyone on the Internet and the submissions from the official contests were propagated to the online-contest server. The practice sessions gave the contestants an opportunity to become acquainted with the system as well as train their problem solving skills.

More recently Mooshak has been used by others without the authors direct supervision to run their programming contests, namely at the University of Évora in Portugal. Further use of the system is scheduled for the events SWERC 2002 and MIUP 2002. The second event will be held at the Classical University of Lisbon and run by a different group of people.

The experience of using Mooshak in last year events showed us that the major performance issue of the system was the server high load towards the end of the contests. As an example, in the final minutes of SWERC 2001 the load of the server reached an average number of 85 processes running simultaneously. The reason for this high load was twofold: teams (48 at this contest) tend to submit all the programs they are still working on, hoping to have another correct solution; and teams and the audience produce a lot more requests for classification listings. This led us to conclude that the number of teams managed by a single server should be around 20 which is less than we originally assumed. This influenced our load balancing strategy for similar future events as explained in the previous section.

In summary, this experience helped us to validate the main designs goals of Mooshak and to identify points where we had to focus our efforts to improve the system.

Flexibility. During the contests changes had to be made to the contest definition data, such as input/output tests, which required re-evaluating submissions. This situation arose specially in the practice contests, where the preparation time is small and standards of problem verification are not as high as in the official contests. Using Mooshak the human judges were able to quickly correct all situations of this nature during the contest itself.

Robustness. The large number of submissions during the several contests confirmed the robustness of the automated judging system. During the training sessions some of the teams explored the limits of the system and submitted programs that they thought could damage it; we were pleased to find that they did not succeed. It should be noted that, although overloaded, Mooshak managed every contest from start to finish. Splitting the teams by servers can be done using the load balancing capabilities of the Mooshak network.

Accuracy. The automated judging system provided classification reports that were easily validated by a small team of human judges. In some cases the load of the machines made it impossible to execute the programs within the timeout limits defined for some problems. In the situations where the server load was too high and the submissions reported 'time limit exceeded' the human judges re-evaluated these submissions on the backup system. To overcome this situation, the current version of Mooshak enforces two time limits: CPU and real time. The CPU time limit is not affected by machine load and is set per problem to deal with inefficient solutions. The real time limit is obviously affected by machine load but can be set to a much higher value, independent of the problem. Real time limits are necessary to ensure that programs trying to read more than the available data do not run forever.

CONCLUSION

In this paper we have detailed the design and implementation of Mooshak, a system aiming to become a full programming contest manager. The system has been heavily tested with practice and official contests, and has proven so far to be very flexible in managing contests with different requirements, quite robust as it supported successfully high transaction loads, and quite accurate in its automated judging capabilities. Furthermore, the system does not require a large number of people to assist in the management during the contest. Mooshak distinguishes itself from other systems by allowing all interaction with the system to be Web-based and by having a simple and scalable architecture that enables one to easily support multiple-site contests and simultaneous online contests.

For the near future we envisage further system development, especially concerning the following issues.

Evaluation and classification. Mooshak evaluates each submission and computes the final classification using International Collegiate Programming Contest (ICPC) rules [1] (summarized in the introduction). These rules are inappropriate for other types of programming contests that have shown interest in using Mooshak. For instance, submissions could be evaluated quantitatively instead of just being marked as accept or, say, wrong answer. Similarly, the final classification of teams could be computed differently to also accommodate the partial marks for problems. To deal with these different types of contests the next version of Mooshak will have evaluation and classification policies as part of the contest definition.

Data sharing. Mooshak already has some import/export features, namely for teams (using ICPC data) and for problem sets. For problem sets, Mooshak uses an archive (zip or gziped tar) with all files related to each problem (problem description, solutions, test data) and an XML file stating the archive content. We expect to improve this specification and extend this feature to other contest data.

ACKNOWLEDGEMENTS

The authors wish to thank the anonymous referees for their valuable comments. This work was partially supported by Project Ganesh (contract PRAXIS/P/EEI/14232/1998) and by funds granted to LIACC through the 'Programa de Financiamento Plurianual, FCT' and 'Programa POSI'.

REFERENCES

1. The ACM-ICPC International Collegiate Programming Contest. http://icpc.baylor.edu/icpc.
2. Programming Contest Control System (PC^2), California State University, Sacramento, U.S.A. http://www.ecs.csus.edu/pc2/.
3. Ural University Problem Set Archive. http://acm.timus.ru/en.
4. Online Judge from the Universidad de Valladolid, Spain. http://acm.uva.es/problemset.
5. Open Source Initiative. http://www.opensource.org/.
6. Ganesh Learning Environment. http://www.ncc.up.pt/ zp/ganesh.
7. Tcl Developer Xchange. http://tcl.activestate.com.
8. Network Time Synchronization Project. http://www.eecis.udel.edu/ mills/ntp.htm.
9. 2001 Southwestern Regional ACM Programming Contest, Universidade do Porto, Portugal. http://swerc.up.pt.
10. Maratona Inter-Universitária de programação. http://acm.up.pt/miup.

