Managing programming contests with Mooshak

José Paulo Leal Fernando Silva

December 2001

DCC-FC & LIACC, Universidade do Porto

Rua do Campo Alegre, 823,

4150-180 PORTO,

Portugal

Email: {zp,fds}@ncc.up.pt

Phone: +351 22 6078830

Fax: +351 22 6003654

Abstract

This paper presents a new web-based system - Mooshak - aiming to behave 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 roboust 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 builtin 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 recent 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, ACM has organized and conducted yearly world programming championships best known as the ACM International Collegiate Programming Contest (ICPC) for college students[7]. 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 are 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 by three 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 judge person 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 accordingly to a marking scheme. A submission is as accepted only when it successfully passes all the test cases. Usually, there are minimum execution times associated with the tests, and therefore for a program to pass successfully the tests must not only produce the correct results but also has to do it within the specified time limit. This ensures that the solutions produced by teams are efficient enough 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 first to ask questions, to submit their solutions, and to receive information; and the later to to answer questions from the teams, to judge their submissions and to report back results. Some systems have been developed trying to fulfill this purpose. PC^2 is the system that has been used in recent world finals [1]. It has capabilities for managing single and multi-site contests and since it has been developed in Java it can be run in either Windows or Unix operating systems. PC^2, however, lacks an important feature that is automated judging capabilities, and therefore requires many judge persons to run a contest. Other systems have been developed elsewhere, namely at the University of Valladolid were they have a 24 hour online automated judge[2] that operates with users through email.

In this paper, we describe the design and implementation of a new web-based system, Mooshak, aiming to behave as a full contest manager and automatic judge for programming contests, with capabilities to run single and multi-site contests, as well as to behave 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 .

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 builtin 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 [3], a web-based learning environment of Computer Science topics (mainly programming languages) that we have been using for several years now. Ganesh includes a module to automatically evaluate students exercises with a stronger requirement in that marks must be given to partial solutions. 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 its functionalities are accessible through interfaces deployed on a web-browser, irrespective of the operating system were 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[1]. Java and plugins were avoided on purpose to simplify the use of the interface by any machine on the Internet.

Mooshak provides a number of user interfaces to accommodate different views to users with different requirements and access permissions to the data managed by the system. The system includes four main views organized as follows:

Contestants view allows contestants to communicate with the judges, asking questions, submitting programs, and requesting printouts

Judges view allows judge persons give feedback to contestants, by answering questions, marking programs and tracking the handling of printouts.

Administration view allows contest directors to setup a new contest and to add all data necessary to make it operational.

Public view allows any user on the Internet to to follow online the progress of the contest.

Access to these user-oriented views is controlled by authentication, except in the case of the public view that is open to everyone. Furthermore, the system has builtin safety measures to prevent users from interfering with the normal progress of the contests. Next we describe in more detail the features available within each view.

Contestants view

During a programming contest most of the team work is done locally in their workstation using standard programming tools such as text editors, compilers and debuggers. Communication out-

[1] JavaScript 1.2

side 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 to all questions posed by contestants and corresponding answers given by the judges;
- access the list of all submissions and corresponding marks;
- access to the current contest classification;
- print the code programs in development;
- visualize on the browser the problem descriptions[2].

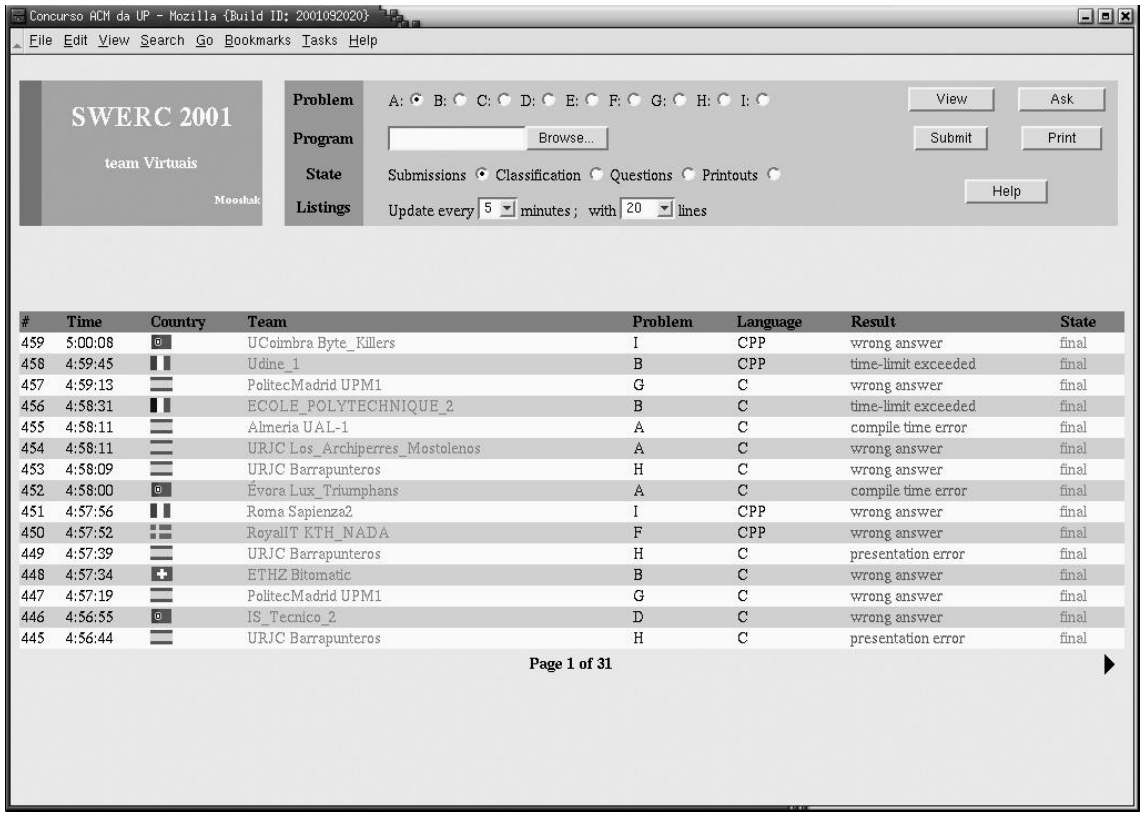

Figure 1: Contestants view

Contestants interface, as shown in Figure 1, is divided in two areas: the central area is used to display information and dialogs, 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).

[2] Specially important on public contests open to everyone on the Internet.

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.

Jury view

Even though Mooshak evaluates submissions automatically, it provides a number of functionalities to help judge persons in doing a finer control of the judging process. From our experience so far, the automatic judging is very much trustful, nevertheless problems may arise unexpectedly such as a 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 could have gone wrong first. Through the jury's interface, judges can also answer questions posed by the teams, access to all submissions made, view the current classification and to control the handling of printouts produced by teams. The jury interface is illustrated in figure 2. The jury

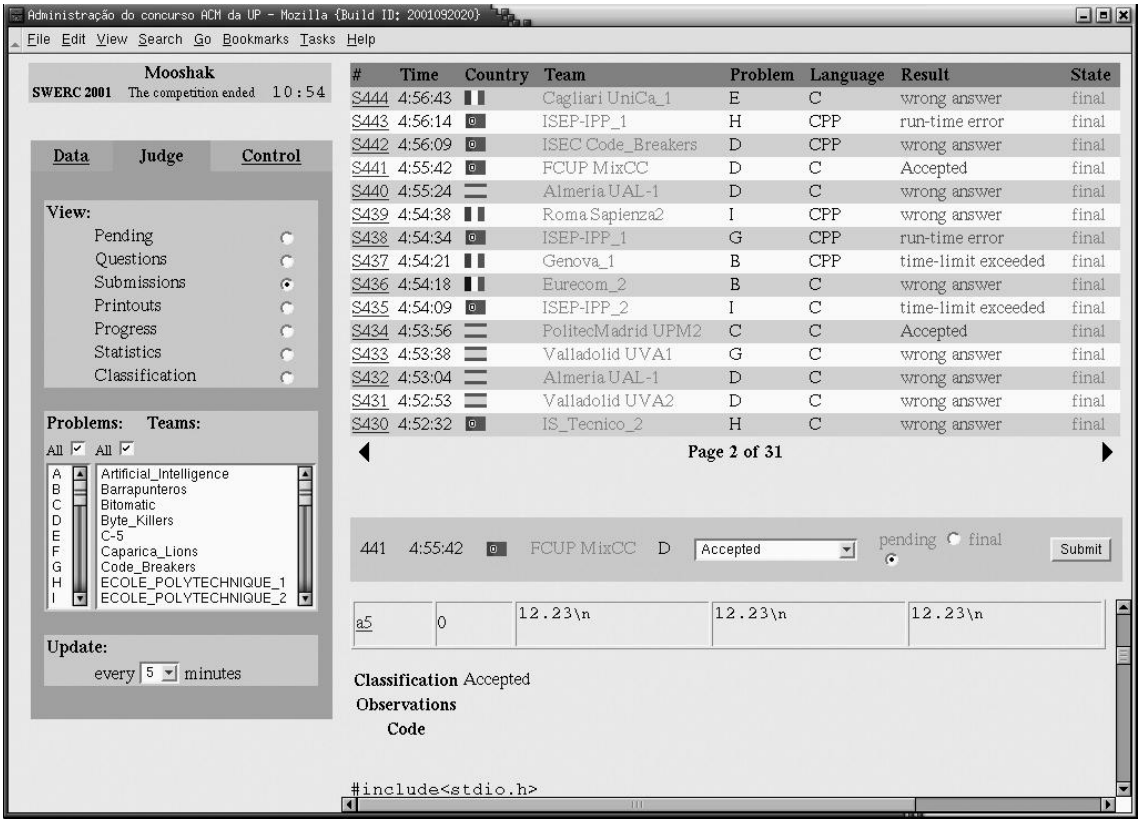

Figure 2: Judges view

navigates through the information using listings similar to those presented to the contestants. The main difference to contestant listings 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 may filter the list by specifying criteria on the problems and teams. Filtering the listings a judge person can monitor the activity of the problems which he or she knows best.

The basic listings available to the jury 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 has also access to the classification, statistics and evolution listings.

Administration

The administration interface allows contest directors to setup 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 and 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 done by editing configuration files and moving data files using shell commands. This approach had two major problems: it was a fastidious and error prone task and was difficult to use in an "emergency" situation during the contest. Even though there was a great effort to make everything consistent and robust, it may be necessary to edit contest data during the contest itself. This seldom occurs but once is enough to convince anyone about the importance for flexible administration features.

The main requirements identified for the administration interface were the following:

History A contest is actually a series of events. Before the main event usually there are one of 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 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 has its own structure: the problem set includes several problems, each one with several tests; the teams are composed by several contestants and aggregated in institutions. The 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 in different moments, 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, as illustrated in Figure 3 provides a navigation tree located on its left side and a workspace on the right. The navigation tree follows a familiar interaction pattern that most users will recognize immediately 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.

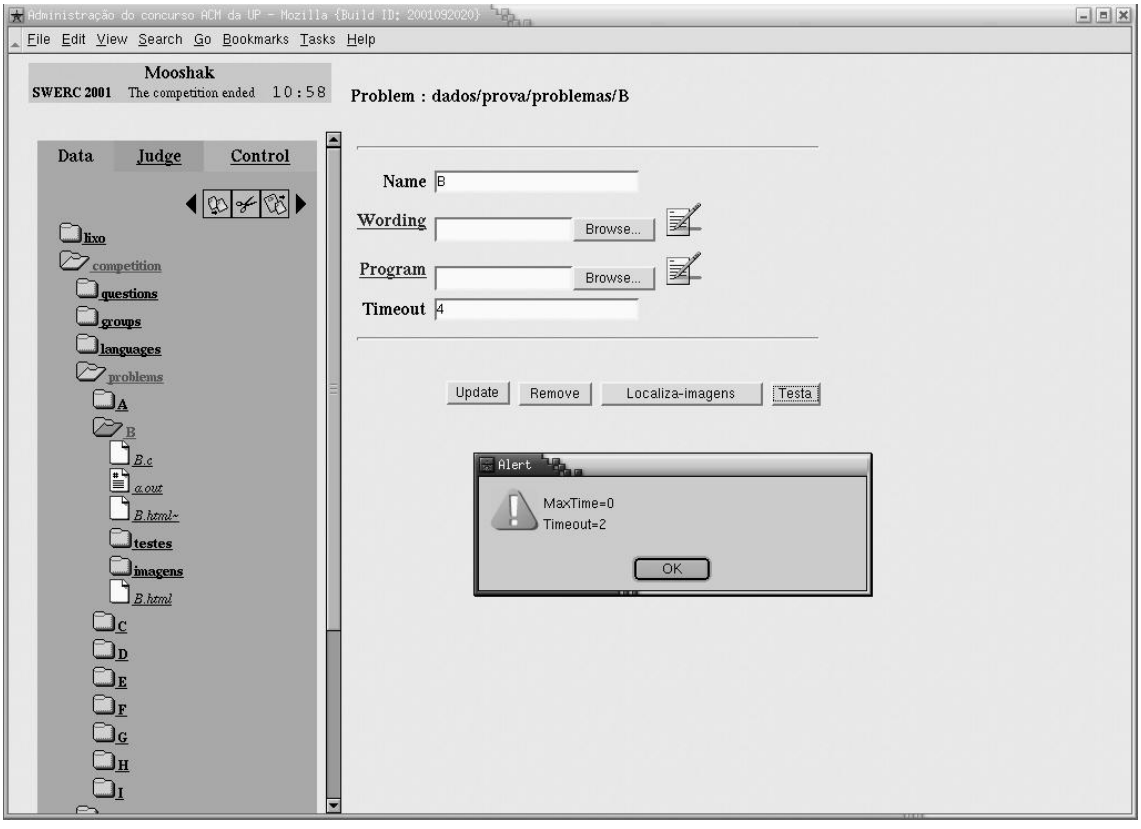

Figure 3: Administration view

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 with 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 accordingly to the type of data being edited.

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. This model was adopted since it solves efficiently the fundamental issues of a distributed system with Mooshak requirements, namely:

GUI The HTML features are sufficient to layout the forms and tables required for submitting data and displaying information. Scripting on client side is enough for form validation purposes and Mooshak does not require any other processing on the client side.

Communication HTTP supports file upload which is required to submit programs during contests and to manage contest data. Mooshak does not require complex interaction patterns. Hence, the lack of state and session on HTTP are drawbacks that can be easily overcome.

Security HTTP provides users authentication and access control. Using HTTP over a secure socket layer (HTTPS) provides data encryption during communication.

Infrastructure Most of the infrastructure required for running a Web application is already installed or is easily available. Moreover, it does not require changing existing security policies.

Figure 4 represents the architecture of Mooshak, structured in vertical and horizontal layers. The **users interface layer** on the top includes the machines used by the teams, human judge, administrators and general audience to access the system. The graphical users interface is rendered in HTML and interaction data is communicated back to a **server** on the **application layer** using the HTTP protocol. The application layer is composed of set of servers, each using its own data management system.

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

We will now concentrate on detailing the implementation of a Mooshak server, emphasizing on 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.

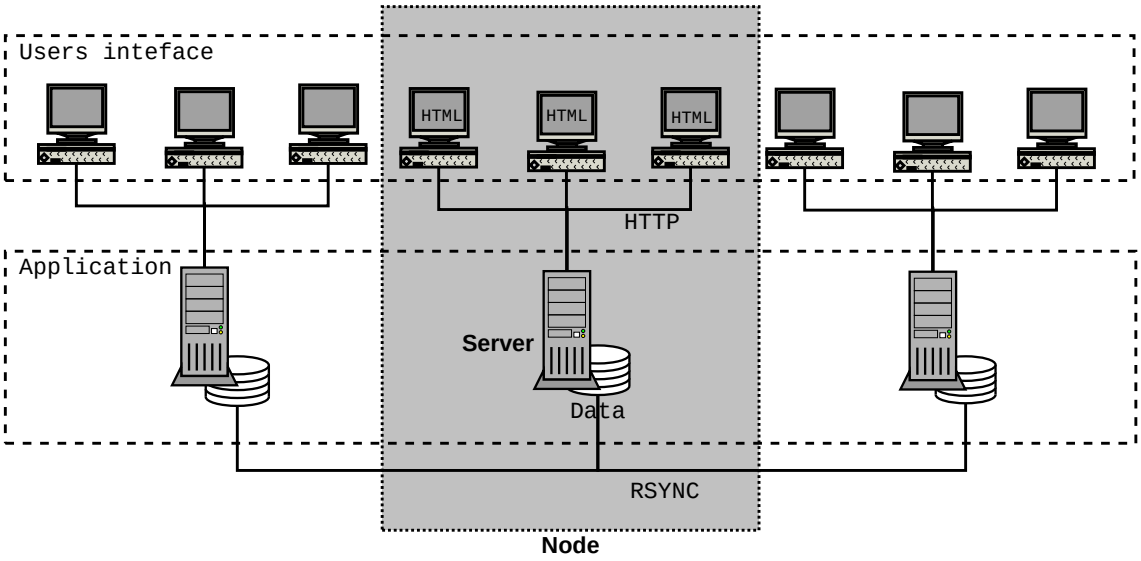

Figure 4: Architecture of Mooshak

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` [4] 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 corner stone 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 judge person. A submission is composed by 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 judge persons have the responsibility of validating pending classifications, making them final, and occasionally modify 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). Reevaluation 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:

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

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, 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 came 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 1 lists the verifications performed during static analysis and the associated classifications upon failure.

Verifications	**Classification**
Team	invalid submission
Language	invalid submission
Problem	invalid submission
Program size	program too long
Compilation	compile time error

Table 1: Static analysis verifications

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 2. 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.

The automatic judge marks an execution as "Accepted" only if the the standard output is exactly equal to the test output file. Otherwise the output file and standard output are **normalized** and

Severity	Classification
6	requires reevaluation
5	time-limit exceeded
4	output too long
3	run-time error
2	wrong answer
1	presentation error
0	Accepted

Table 2: Classification and severity of program tests

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 ruin 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 problems, with the exception of execution timeout that is adjusted to each problem. The 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 for the next integer (in seconds). The timeout for compilation is 60 seconds. The other resource limits enforced are listed in Table 3 with their default values in bytes (except for the number of child-processes) .

Maximum Limits	Value
Process data segment	2097152
Process stack segment	1048576
Process RSS	4194304
Output	102400
Source code	102400
Child processes	0

Table 3: Compilation and execution limits

Persistent objects

The Mooshak data uses an object oriented approach - that we call **persistent objects** - to blend data, recorded on the file system, with `Tcl` code. This approach structures application and is the basis for the replication mechanism. It should be noted that persistent objects are in no way related with `incr Tcl`, a full fledged object oriented language based on `Tcl`. The data management of Mooshak owes more to the `Tk` graphical object library: both have pathnames as object references and a flat (without inheritance) set of classes. Mooshak does not use a separate data management system, typical of the three-tier model so popular among Web applications. The next paragraphs try to justify this choice and describe the persistent objects mechanism.

Most Web application rely on relational database managements system (RDBMS) to store data. These systems provide independence between application and data as well as efficient tools for managing and querying data. On the other hand a RDBMS introduces extra complexity: the RDBMS itself requires a separate installation and management; the mismatch between application data structures and database structure requires extra processing for converting data between the two formats. Arguably, for some applications the RDBMS may not the best approach and this may be the case a programming contest management data given its characteristics:

Small The amount of data is comparatively small and does not require sophisticated indexing. The larger data structures are those that record transactions (e.g. problem submissions) and they do seldom require more than 1000 records per contest.

Variable Records include data of variable sizes such as program code and object files, problem descriptions (HTML files with images).

Structured Data is logically organized in an hierarchical structure which simplifies the implementation of navigation and edition commands.

Accessible Programs and test data must be easily accessible from the command line interpreter to implement features such as automated judging.

Distributed Data must be efficiently copied between different machines to enable replication.

Having in mind that a good part of the data used by Mooshak is conveniently represented in plain files - source and object programs, data files, HTML files and images - it seams reasonable to base the data management directly over the operating system, using files to record data and directories to maintain structure. Thus, we define a persistent object as a special kind of object that is made persistent by recording its definition directly on the file system, and is therefore referenced by a pathname.

A persistent object belongs to a **class** that determines both its **attributes** and the **operations** it supports. A class is implemented as a `Tcl` module with two new declarations: `Attributes` and `Operations`. The first type of declaration states the names and types of attributes of the class. The supported types include text values, enumerations, files, sub-objects and references to other persistent objects. Hence, a persistent object is implemented as a directory containing its files and sub-objects as sub-directories. The other values as well as the object's class are recorded in special hidden files in that directory. An operation is a kind of method. In the definition of an operation the attribute names refer to the values of the current object. A class operation is invoked when a message is passed to the object. The object reference defines a context for the operation and the values for the attributes.

```
Attributes Counter {
  Value text  {}
}
Operation Counter::reset {
  set Value 0
}
Operation Counter::increment {
  incr Value
}
Operation Counter::show {
  return $Value
}
```

Figure 5: Definition of class `Counter`

Figure 5 presents the class `Counter` with a single attribute `Value`. Please note that the value is declared as `text` since persistent objects do not provide numeric types. The reason behind the lack of numeric types is the fact that these types do not exist neither in `Tcl` nor in HTTP. Hence, for Web applications written `Tcl`, numeric attributes are redundant. The operations `reset`, `increment` and `show` allow us to operate on this attribute. Figure 6 illustrates an use of class Counter: first an instance of this class is created from file `/home/mooshak/mycounter`, then this pathname is used for sending and increment message to the instance that is finally saved on disk. Opening and saving are performed using commands provided by the Mooshak data package.

```
data::open Counter /home/mooshak/mycounter
/home/mooshak/mycounter increment
data::save /home/mooshak/mycounter
```

Figure 6: Using class `Counter`

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 premium. Running an official contest, with a concern for reliability and 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 consists on 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:

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.

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 a different servers.

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

The Mooshak network configuration for a particular contest may contain several of these links. Figure 7 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 last 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 maintains also 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. 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 among them.

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 has 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 reevaluated in a different machine.

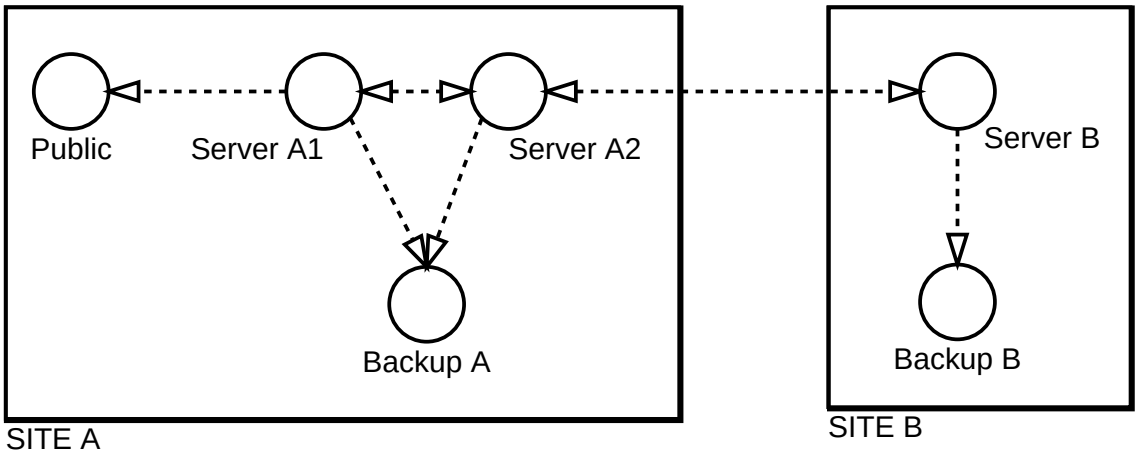

Figure 7: Network of Mooshak nodes

The main issue with replication is the consistency of contest data, namely that no data fails to be replicated or is overwritten by replicated data. To guarantee that no data fails 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 teams, problems, programming languages) and contest transactions (such as submissions, questions and printouts). Of these two, contest transactions, specially submissions, are particularly important. To guarantee uniqueness all transaction data is keyed by 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 is not, in principle, changed after the beginning of the contest. It should be updated in a single node for consistency sake, 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 same time in different servers could (although not very likely) share the same record.

For the above setup to work properly, all servers 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 [5]. The system was also used in the preliminary Portuguese Programming Contest, MIUP 2001 [6], 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.

	SWERC 2001	**MIUP 2001**
Number of Teams	47	22
Total of submissions	459	95
Total of queries	57	44
Total of printouts	329	38
Server CPU	AMD 1.6 GHz	2*PIII 600 MHz
Server RAM	756 Mb	512 Mb
Maximum CPU load	85	10

Table 4: Contest figures

Table 4 lists some figures related to activity of the Linux servers used during the two official contests managed by Mooshak. Its shows that SWERC 2001 was a much more demanding event in what regards system resources and made us rethink our load balancing strategy for similar future events, using the approach explained in the previous section. The experience gained during these events helped us to validate the main designs goals of Mooshak and to identify some points needing improvement:

Flexibility During the contests changes had to be made to the contest definition data, such as input/output tests, which required reevaluating 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. We noticed that the system was under a lot of stress in the final minutes of the contests when the teams a) submit all the programs they are still working on, hoping to have another correct solution b) produce more requests for classification

listings. For this reason, in the case of SWERC 2001, we concluded that the the number of teams managed by a single server should be less than we originally assumed. Splitting the teams by servers can be done using the load balancing capabilities of the Mooshak network. It should be noted that, although overloaded, the Mooshak managed every contest from beginning to finish.

Accuracy The automated judging system provided classification reports that were simply validated by a small team of judge persons. In some cases the load of the machines made it impossible to execute the programs within the timeout limits defined for some problems. Mooshak enforces two time limits: execution and real time. The execution time limit is not affected by machine load but real time limit is. Real time limits are necessary to ensure that programs trying to read more than the available data do not run forever. In the situations where the server load was too high and the submissions reported "time limit exceeded" the judge persons reevaluated these submissions on the backup system.

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 shown so far to be very flexible in managing contests with different requirements, quite robust as it supported successfully high transactions load and its automated judging capabilities showed to be very accurate. Furthermore, the system does not require a large number of judge persons 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 to easily support multiple-site contests and simultaneous online contests.

For the near future we envisage further system development, specially concerning the following issues:

Evaluation and Classification Mooshak evaluates each submission and computes the final classification using ICPC rules. These rules are unappropriated 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 team 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 archives content. We expect to improve this specification and extend this feature to other contest data.

References

[1] Programming Contest Control System (PC^2), California State University, Sacramento, USA `http://www.ecs.csus.edu/pc2/`

[2] Online Judge from the Universidad de Valladolid, Spain `http://acm.uva.es/problemset`

[3] Ganesh Learning Environment `http://www.ncc.up.pt/~zp/ganesh`

[4] Tcl Developer Xchange `http://tcl.activestate.com`

[5] 2001 Southwestern Regional ACM Programming Contest, Universidade do Porto, Portugal `http://swerc.up.pt`

[6] Maratona Inter-Universitária de programação `http://acm.up.pt/miup`

[7] The ACM-ICPC International Collegiate Programming Contest `http://icpc.baylor.edu/iccp`

[8] Network Time Synchronization Project `http://www.eecis.udel.edu/~mills/ntp.htm`