MetaOJ: A Massive Distributed Online Judge System

Miao Wang, Wentao Han, and Wenguang Chen*

Abstract: Online Judge (OJ) systems are a basic and important component of computer education. Here, we present MetaOJ, an OJ system that can be used for holding massive programming tests online. MetaOJ is designed to create a distributed, fault-tolerant, and easy-to-scale OJ system from an existing ordinary OJ system by adding several interfaces into it and creating multiple instances of it. Our case on modifying the TUOJ system shows that the modification adds no more than 3% lines of code and the performance loss on a single OJ instance is no more than 12%. We also introduce mechanisms to integrate the system with cloud infrastructure to automate the deployment process. MetaOJ provides a solution for those OJ systems that are designed for a specific programming contest and are now facing performance bottlenecks.

Key words: online judge; programming test; distributed systems

1 Introduction

Online Judge (OJ) systems are programs used for automating the judging process of submitted solutions for programming problems, which normally involve compiling source codes into executables, running them against sets of input data, and comparing the given results with standard results. These systems can enhance education processes[1, 2], allowing students to write running codes rather than simply writing them on paper. To fulfill the requirement, a simple OJ system can be only composed of several scripts. Most systems, however, require additional mechanisms, such as sandboxing, to ensure their own integrity, because running untrusted codes from users is always considered risky. In addition, many OJ systems include more functions that allow them to be adapted to specific application scenarios. These functions can include organizing contestants into teams and optionally disclosing scores to contestants.

OJ systems, whether simple or complex, can be roughly divided into two parts: one is for front-end services, displaying a user interface and receiving submissions from contestants, and the other is for judging the submissions and giving them correction. Generally, an OJ system can easily scale by simply adding more computing resources into the second part and the submissions received from the contestants can be judged more quickly. However, as the number of contestants grows, an OJ system may be busy in calculating rankings and other things and become less responsive to requests from contestants. For example, China Computer Federation is now holding a programming test series called Certified Software Professional. It has more than 11,000 contestants in its most recent session, the number of which is expected to grow in future sessions. With so many contestants located around China competing at the same time, simply adding more workers into the judging part cannot overcome the bottleneck that resides in the front-end service and network. Some OJ systems can allow many contestants to simultaneously compete. However, one OJ system cannot fit all scenarios and the source codes of most famous OJ systems are not open[3]. For a certain scenario, an OJ system may have already been developed based on its unique requirement. For these customized systems, performance bottlenecks can reside everywhere. Solving them can be a problem, not to mention how to make the system available at the locations near the

* Miao Wang, Wentao Han, and Wenguang Chen are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: {miaowang, hanwentao, cwg}@tsinghua.edu.cn.
* To whom correspondence should be addressed.
Manuscript received: 2020-05-13; accepted: 2020-05-25
contestants and maintain consistency. Our challenge is to quickly transform an existing OJ system into one that can handle large quantities of requests from a huge number of contestants and can be deployed in a distributed manner to ensure better network conditions when contestants visit the OJ system.

Our basic idea is to deploy multiple instances of an OJ system at locations near contestants and develop a new system managing all the instances to simultaneously deploy things, such as statements of problems, configuration of contests, and login credentials for contestants onto each of the instances. By doing so, we can reduce the amount of work required in the modification and reuse the existing codes which are mature and repeatedly tested on business logics. The scaling problem is also solved by bringing more instances online. If contestants do not need to know their real-time rankings, then all the deployed instances may not need to communicate with one another. These instances can be independent of one another and the inter-instance communication interfaces are not needed.

The remainder of the paper is presented as follows: In Section 2, we introduce the interfaces needed for building a distributed OJ system. In Section 3, we can show the manner by which the whole distributed OJ system can be managed. In Section 4, we demonstrate the transformation of an existing OJ system into a distributed one and integrate that system with cloud infrastructure to automate the provision of online systems. In Section 5, we sum up all the results and describe possible future work.

2 Base OJ System

Most OJ systems were developed to meet the requirements of various programming tests and practices. An OJ system can be generally divided into two parts: one is the front-end service which is responsible for displaying an interface for users, receiving all submitted solutions, dispatching judge tasks, and calculating scores, and the other is the judger service, which is used for receiving source codes from the front-end server, running against the provided datasets, and giving correction to that submission. Because the running of individual codes does not affect one another, most OJ systems allow a flexible provision of the two parts. Multiple judger services may be simultaneously deployed and registered to the front-end service to enhance the performance of the OJ system. Figure 1 demonstrates the relationship between the two parts.
2.2 Interfaces of the front-end service

Apparently, some facilities for human administrators manage objects residing in the front-end services, including creating, removing, and updating them. Programming interfaces for automating those actions may be abstracted from the user interface or be instantly available. Because we aim to build a management system for all individual OJ instances, after a reasonable period of time, the configurations of User, Problem, and Contest and their relationships shall be made the same as the administrator configures on the management system. To achieve this, upon synchronization, the missing objects shall be created and the outdated objects shall be updated. To prevent the possible inconvenience caused by comparing those objects, we may introduce the “update or insert” interface, which can update the record if the given unique ID exists and can otherwise create a new one with that ID if not. With this interface, the manager can directly load data into OJ instances, regardless of the previous state. Another interface is for the collection of the scores. On an invocation, the interface will give out the scores of all the contestants. Another interface may be introduced to incrementally dump the detailed scoring information of all the submissions.

Thus, the interfaces needed for the front-end service shall include “updating or inserting” the configuration of Users, Problems, and Contests and fetching all the scores. For the interfaces handling Users and the scores, the payload can be simply encoded in accordance with what is stored in the database. The encoding of Contests should also include the primary key of the Users and Problems belonging to them. The transportation of Problems may involve extra care about carrying the input and output data for testing, which may be possibly stored outside the database. These interfaces can be designed to accept bulks of data at one time to enhance the performance.

3 Design of the Distributed Layer

As discussed in Section 1, normal OJ systems are not capable of holding large contests, even if all the network connections were considered reliable. Before we design the distributed layer, we can study the scaling method in Contest Manage System (CMS)\textsuperscript{[9]}, which is an OJ system used in the recent International Olympiad in Informatics series. CMS, adopting a modular design, is composed of modules including the web front-end service, scoring service, evaluation service, and ranking service, most of which can be deployed into multiple instances and communicate with one another via a remote procedure call interface. CMS can naturally scale by adding more instances of the service which results in a bottleneck. CMS lacks a mechanism discovering all services, but it hard-codes the list of services in the configuration on each working node, which makes hot-adding more resources impossible unless we reserve a list of services we may add in the future in the configuration before we set up the whole system. Furthermore, the services are required to connect to a shared Postgres database, which becomes a single point of failure. Although there are mechanisms built into Postgres that allow to set up a high available database cluster across the wide-area network, the lack of profession and experience managing such a complex cluster can result in disasters. Even with professionals, the disaster of a complex database cluster cannot be avoided\textsuperscript{[10]}.

Instead of creating a distributed OJ system from scratch, which can consume a lot of effort on the basic logic of an OJ system, we can also reuse the existing work in an OJ system as much as possible and only add a few codes to make the whole system working in a distributed manner. Our basic idea, which is simple and straightforward, is to deploy multiple replicas of the OJ instances near the locations where the contestants compete. In this way, we can solve three problems. The first problem relates to the performance. When the contestants compete on their own instances in the OJ system, there would not be too many of them in one instance, so that they can get feedback on their submissions in a short time. The second problem relates to networking. The connection between the contestants and the OJ server can be of bad quality, especially when requests from all over China flood into one location. If multiple replicas are deployed, then the experience will be better and we can have the flexibility to improve by locating those replicas in accordance with the network environment of the contestants. The last problem is to avoid complex maintenance of the database system. For the distribution of workloads, because we are modifying an existing OJ system rather than creating a new one, roaming contestants between OJ instances will be too complex for the modification, so contestants can be assigned to their own OJ instances in advance in accordance with the physical location they reside in. We keep the original interaction between OJ instances
and their judges untouched. Now the question arises on how we can create a system to manage all the replicas. These replicas shall share the same set of problems and users. The settings for contests share some common attributes but with a different list of users. Because there might be updates to that data after the instances are put into use and there are additional data stored outside the database, simply dumping the database and importing to other instances do not make sense. Thus, we need to utilize interfaces of the front-end services for importing those data structures. The structure is described in Fig. 3. As we can see, adopting this structure means that any issues with the OJ manager or with the connection between it and the OJ instances will not affect the normal operation of the OJ instances, because the interactions between them are limited to deploying configurations and collecting scores, which are not critical during the running of a contest.

As shown in Fig. 2, there are three objects for which we have importing interfaces on the OJ instance side. Their relationships are maintained via their primary keys, i.e., the ids. Thus, after importing, we need to keep those relationships and ensure the link of the objects on the manager side and those on the OJ instance side. Because the only source of these objects on the OJ instance side is from the manager, it is reasonable to utilize the “update or insert” interface. We also deploy an instance of the OJ system as the template along with the manager, serving as the source of the data used for deploying other instances and providing a user interface for the administrator to prepare for the resources. Among the three objects, User is the simplest one to process. When importing user information into the OJ instances, the OJ manager can just connect to each of them, using that interface to load user information in. As described before, contestants are assigned to a specific OJ instance, but we still load information for all the users to all the OJ servers here for simplicity when there is Contest facility. Problem can be imported similarly, with input and answers files for test cases packed on the OJ template and unpacked on the OJ instances. Contest should be treated differently, if any, because for the same contest on the OJ manager it will become slightly different on the OJ instances. As shown in Fig. 4, we add Contest Instance to describe this change. When importing contests, each Contest Instance is combined with its inheriting Contest and loaded into the corresponding OJ server. The primary key on the OJ server is recorded in the attribute of the relation deploys on. The enrollment information of the contestants is also loaded when we have Contest facility on the OJ instances. Because there will be no enrollment information for contestants on OJ instances not assigned for them, contestants logging into a wrong OJ instance will have nothing to do but switch to the correct server. Normally, the contestants shall not communicate with the OJ manager, but with the OJ instances assigned to them. The procedure of the interaction between a contestant and an OJ instance will remain the same as the way he or she interacts with the OJ system before our modification. Scores can be simply collected repeatedly from the OJ instances over a certain period of time for

---

**Fig. 3** Structure of OJ instances and OJ manager.

**Fig. 4** ER model of OJ manager.
the administrators to have a better grasp of the whole contest if the network condition allows. They can also be collected only at the end of a contest.

4 Case Study: TUOJ as an Example

The TUOJ system is an OJ system that has been developed to meet the requirement of various programming tests. Its front-end service is developed using Node.js and with the support of Contest, allowing to hold several contests at the same time or for test purposes. A non-relational database\(^\text{[1]}\), namely MongoDB, is chosen to cooperate with TUOJ to store various data structures with more flexibility. Compared with relational databases, non-relational databases can store hash dictionaries, and dictionary entries may contain more complex types, which provides a more concise implementation for the TUOJ system of the objects and data structures presented in Fig. 2.

In this section, the system is modified in accordance with the scheme introduced earlier so that it can be deployed distributively in multiple locations. Here, the modification of the TUOJ system is first introduced, and the performance of the TUOJ system is then tested to determine that the modification process has only a slight impact on the performance of the system.

4.1 Data structure in the TUOJ system and modifications

To deploy TUOJ distributively, we first analyze the management interface provided by the system. In the front-end service, the web User Interface (UI) is connected through RESTful interfaces to its backend, loosing the coupling of the UI and its backend. In the TUOJ system, we have three data objects to consider: Contest, Problem, and User, among which, Contest and User contain only normal fields, so these objects can be directly encoded into a JavaScript Object Notation (JSON) object. Problem, however, contains fields referring to files stored in the file system, including the files containing test data and description for that problem. Thus, these files should be included in the encoded Problem object. To achieve this, the files are packed into a tar archive, which is latter serialized using base64\(^\text{[2]}\) encoding and appended to the JSON object created from the original Problem object. The relationships between the Contest object and the other two are maintained in the additional fields, the type of which is array of the unique IDs of the other two objects, residing in the Contest object. Thus, by importing the encoded Contest object, the relationships are restored.

There should be four new interfaces in the TUOJ system for the OJ manager, three of which are for importing Contest, Problem, and User objects into the controlled TUOJ systems and the other one is for collecting score information. Utilizing the unique id field in the Contest, Problem, and User objects, the three importing interfaces can be implemented using the findOneAndUpdate operation with the upsert flag enabled, which will update the existing record of data in the database or create a new one if the record with the specified id does not exist. The implementation of the interface importing Problem is shown as Algorithm 1, noting that the interface can accept either a single Problem object or an array of them to provide a batch operation interface, reducing the number of requests needed. Interfaces for importing the other two objects can be implemented similarly.

When exporting scoring information from the TUOJ system to the OJ manager, the OJ manager shall invoke the interface at a certain interval to fetch the results of the evaluation of the submissions to analyze the results and calculate the rankings and other statistics. As we

Algorithm 1 Interface for importing Problems

| Input   | Array of Problem objects or a Problem object P |
|---------|-----------------------------------------------|
| Output  | Array of operation results of each Problem object R |

1   begin
2   \[ R \leftarrow []; \]
3   \[ \text{if } P \text{ is not an array then} \]
4   \[ P \leftarrow \{P\}; \]
5   \[ \text{end} \]
6   \[ \text{for each element } p \text{ in } P \text{ do} \]
7   \[ r \leftarrow \text{success}; \]
8   \[ \text{Allocate a new directory in the filesystem, let } s \text{ be the result;} \]
9   \[ \text{Decode and expand the tar archive containing additional files in } p \text{ into } s; \]
10  \[ \text{if Successful then} \]
11  \[ \text{Create a new Problem object pd;} \]
12  \[ \text{Copy all the fields in } p \text{ into pd;} \]
13  \[ \text{Point the storage path field in pd to } s; \]
14  \[ \text{Execute findOneAndUpdate operation in the database with the object pd;} \]
15  \[ r \leftarrow \text{the result of database operation;} \]
16  \[ \text{else} \]
17  \[ r \leftarrow \text{failure;} \]
18  \[ \text{end} \]
19  \[ \text{Push } r \text{ into } R; \]
20 \[ \text{end} \]
21 \[ \text{return } R; \]
22 end
can expect, if all the scoring information is transmitted in one invocation of the interface, then there will be lots of repeated records in two successive invocations, which will be a waste of the networking bandwidth. To incrementally transfer the scoring information, noticing that there are timestamps associated with the scoring information, as long as the clock on the TUOJ instance is not adjusted backward, the timestamp here can be used for fetching newly generated scoring information after the previous interface invocation, which is summarized as Algorithm 2.

4.2 Interface authentication and credential management

When importing or exporting data, the OJ manager sends a request via a HyperText Transfer Protocol (HTTP) interface to the TUOJ instances. As a result, a facility is needed for the OJ instances to authenticate the request. JSON Web Token (JWT)\[13\] is a good mechanism to encode a signed structured data using a shared secret. To utilize it, we may assign a shared secret for each TUOJ instance and record it on the OJ manager. For a better management of these credentials, the PBKDF2 algorithm\[14\] can be used to manage the credentials hierarchically. Let $K_m$ be the only primary credential configured in the OJ manager. The shared secret for usage $U$ and the TUOJ instance named $S$ will be

\[
DRVK(K_m, U, S) = P(K_m, U + \text{"/"} + S, 10, 32)
\]

(1)

where $P$ means the PBKDF2 algorithm with SHA256-HMAC, + means string concatenation, 10 is the number of iterations taken, and 32 is the length of the derived key. In this way, each of the TUOJ instances will be given a different shared secret for the usage $\text{mgmt\_auth}$, which can be quickly generated by the OJ manager and does not need to be stored in the database.

Moreover, the password for contestants can be managed in a similar way. User names are generated from two parts, connected with _. The first part indicates the TUOJ instance that this user is assigned to. The second part is for password derivation. If a user is assigned to the TUOJ instance named $S_j$, then the primary password generating key for this TUOJ instance will be

\[
K_{i}^{pwd} = DRVK(K_m, \text{"pwd\_gen"}, S_j)
\]

(2)

If the second part of the name of the user is $N_j$, then the password for that user will be the first 10 letters of

\[
\text{base32}(DRVK(K_{i}^{pwd}, \text{"usr\_pwd"}, N_j))
\]

(3)

where base32\[12\] is an algorithm converting binary data into letters and numbers ($\{A-2Z, 7\}$). For an OJ instance, only its $K_{i}^{pwd}$ is needed to authenticate the contestants assigned to it.

4.3 Summary of the modification and performance test

Summarizing the above modifications, we can see only that the following tasks are carried out:

- Analyzing the data structure of the main objects and designing the encoding of the objects, among which only the Problem object is treated specially and others are simply encoded into JSON objects;
- Implementing the interfaces for importing the objects and exporting scoring information;
- Inserting a JWT authentication middleware into the control flow; and
- Adjusting the way that passwords for contestants are generated.

During the modification, a full grasp of every detail of the TUOJ system is not needed. The implementations of the data import and export interfaces are also quite mechanical and easy to implement. In fact, by analyzing the code, we can see that after adding the above interfaces, the code of the TUOJ front-end service increased to 22353 lines (excluding the code of the dependent modules), of which 551 lines of code were added due to the addition of the above interface, which accounts for 2.46\% of all codes. Among these additions, 198 lines are related to the Problem object, accounting for 35.9\% of all the added codes. Thus, compared with the code of the entire OJ system, the modifications made to the TUOJ system are relatively minor. Meanwhile, the implementation process is more mechanized, reducing
the possibility of introducing program defects and maintaining the robustness of the original system. To evaluate the performance of the TUOJ system after modification, it is necessary to first analyze the overall system architecture and find performance weaknesses. In the modified TUOJ system, after the distributed architecture is adopted, the requests from the contestants are actually directly processed by each TUOJ instance instead of going through a unified entrance. If the administrator can reasonably deploy instances of the OJ system so that the number of contestants assigned to each instance is similar to the number of contestants that the TUOJ system can handle before the modification, then for each instance of the TUOJ system, there is no substantive difference on the working process before and after the modification. To evaluate the performance changes, it is necessary to first analyze the impact of the newly introduced code on the performance of the TUOJ system. As shown in the above modification process, in the newly added interfaces, the three data importing interfaces are generally not invoked during the process of a contest, but before the start of a contest, to be used to import various contest data. Even if one is invoked during the contest, the invocation is initiated by the administrator through the OJ manager, and the frequency is extremely low. Therefore, the influence of the newly added data importing interfaces on the TUOJ system during operation can be ignored. However, the interface of collecting scoring information will be constantly called by the OJ manager during the contest at intervals. The constant access of this interface, for a instance of the TUOJ system, is an additional load, which needs to be analyzed to identify its impact on the performance.

We selected a lightweight performance test framework, i.e., Artillery.io to carry out the test. Artillery.io can send a series of successive requests to a given HTTP server, to simulate the interaction between the user and web server. Parameters can be extracted from the response of a previous request and filled into the next request. The series of requests is also known as a scenario. Because users arriving at a website can perform different actions, generating different series of requests, multiple scenarios with different weights are allowed in a performance test configuration. Each simulated visit will randomly select a scenario and perform the operations defined in that scenario. The probability of selecting each scenario is proportional to the configured weight. In that way, the behavior patterns of different kinds of users can be modeled and simulated. Multiple phases can be defined in a performance test configuration, controlling the arrival rate of simulated visits. By defining multiple phases, we can simulate gradual changes in request pressure, which can reflect the trend of service quality changes under the condition of increasing the request pressure.

To simulate the user activity on the TUOJ system, in accordance with the historical access logs, we sorted all the Application Programming Interfaces (APIs) that may be called by the users from the front-end pages into two groups. The first group contains nine APIs for the contestants to fetch data about the contest, including the basic information of the current contest, the information of a certain problem, submissions made by the contestant, and detailed scoring results of a submission. Historical statistics show that the frequencies of the invocation of these APIs are roughly close. Thus, it is reasonable to create one scenario to sequentially invoke these APIs after a simulated login. The other group contains only the API used for submitting solutions. As a result, another scenario is created to submit a file of 1 KB in size with this API after a simulated login. The ratio of weights of the two scenarios is determined to be 4:1, i.e., one invocation of the submission API in every 37 invocations on average. To cooperate with the performance test framework, a stub judge service is created to fetch judge jobs from the web service and give random scores. To significantly reflect the performance bottleneck of the TUOJ system, we selected a virtual cloud server with a relatively low configuration to run the TUOJ web service being tested. The server is equipped with one core of Intel Xeon Gold 6133 processor, operating at 2.5 GHz, and 1 GB of memory. The virtual server flooding simulated requests are with the same model of Central Processing Unit (CPU) but four cores. The connection between the two servers is through the internal network on the cloud platform.

Before starting the performance test, a new contest and a problem with 10 test cases are configured on the TUOJ system being tested. The database is backed up so that the state can be recovered to start a new round of tests. The first round is to evaluate and compare the performance of the system under extreme pressure. The phases are configured as follows. In the first 10 s, the arrival rate of simulated visits is set to 5 times/s, linearly increases to 50 times/s in the next 120 s, and maintains 50 times/s for 30 s. This test is performed two times when the OJ manager is disabled.
and the submission collecting interface is not invoked and when the manager is enabled and the submission information is incrementally collected every 5 s. The response time of requests is collected every 10 s. The statistics are shown in Fig. 5. As shown in Fig. 5, the modification has a minor effect on the performance. Taking the 0.95 quantile for example, the delay of responses before and after the modification is not much different, and even at some moments, the delay after the modification has been reduced. This finding shows that the performance loss caused by the modification is actually not significant.

Another round of tests is performed to simulate a normal high workload. There is only one phase in this round. The arrival rate is set to 10 times/s for 60 s. The distribution of the delay of responses is shown in Fig. 6, and the key statistics are shown in Table 1. The above statistical data show that after the modification, under normal high workload, the performance loss caused by frequently invoking the exporting interface of the scoring information is between 5% and 12%, which is relatively controllable and acceptable.

### 4.4 Unified login service

To reduce the possible confusion brought to users, we can create a unified login service, working statelessly and independently of our OJ system based on the above password generating method. It stores a mapping from the instance indicator, i.e., the first part of the username, to the name of the TUOJ instance and the primary credential $K_m$, and it can authenticate all the users on every OJ instance and redirect the users to the instance assigned to them. To avoid a single point of failure, this login service can be simply deployed behind a load balancer or a smart DNS system without sharing the database or other states.

### 4.5 Integration of cloud resources

In recent years, the cloud computing platform industry has been booming, which leads to a cheaper solution to utilize computing and networking resources. Thus, it is natural to build our OJ system on a cloud infrastructure. Integrated with a cloud provider, we can automate the deployment of OJ instances and judgers. Prior to the integration, a network infrastructure should be built, allowing the OJ manager and judgers to connect to the

---

**Fig. 5 Delay of responses under extreme pressure.**

**Fig. 6 Distribution of delay under normal high workload.**

**Table 1 Statistics of delay under normal high workload.**

| Statistics | Minimum | Median | 0.95-quantile | 0.99-quantile | Maximum |
|------------|---------|--------|---------------|---------------|---------|
| Before     | 1.4     | 4.2    | 12.1          | 32.2          | 68.3    |
| After      | 1.5     | 4.3    | 13.5          | 33.9          | 78.0    |
OJ instances. For simplicity, default settings may be adopted and the servers may be connected to each other using their public Internet Protocol (IP) address through the Internet. Because Internet traffic requires extra cost, it will be cheaper and safer to let the connection between them inside a virtual private network. With servers deployed at different locations, connecting the private networks among these locations requires the configuration of tunnel gateways, which is beyond our discussion.

Popular cloud platforms support creating virtual servers from a prebuilt template and passing user-defined data which can be easily visited from inside the virtual server using a restful interface or directly reading files in a virtual Compact Disc (CD) drive mounted to it. Using JWT, the OJ manager can issue an intermediate token expiring in a short time through the user-defined data interface we discussed above. The initializer program will start along with the newly created virtual server and read the token out. Parameters needed for starting a new OJ instance will be requested from the OJ manager. To protect the secret used for authentication in the initialization parameters, the JWT token is designed to be invalidated after the parameters are sent to the initializer program and the transmission of them is under the protection of a Secure Sockets Layer (SSL) layer. Upon successfully starting the OJ instance, the initializer program will exit before reporting to the OJ manager, which will later configure the name server by adding a new domain name pointing the new OJ instance and load data into it using the interface we introduced in Section 3. The whole process is shown in Fig. 7.

5 Conclusion and Future Work

In this paper, we covered the whole design of the MetaOJ system, including the brief structure and interfaces of ordinary OJ systems, serving as the fundamentals of our work and the OJ manager, generating and deploying all the configurations to the OJ instances. We also studied our actual work on the TUOJ system to make it distributed and the integration with the cloud, simplifying and automating the provision of TUOJ instances.

Our future work will focus on more possibilities for these systems. A version management mechanism for the configurations will be adopted in the OJ manager to ensure that the configurations on OJ instances are up to date and to provide a facility to roll back when necessary. For the cloud integration of the OJ system, we will seek compatibility with more popular public cloud providers and try to construct a mixed infrastructure to avoid vendor lock-in. We will also try to introduce the support for more OJ systems as the OJ instance in our system.

References

[1] A. Kurnia, A. Lim, and B. Cheang, Online judge, *Comput. Educ.*, vol. 36, no. 4, pp. 299–315, 2001.
[2] G. P. Wang, S. Y. Chen, X. Yang, and R. Feng, OJPOT: Online judge & practice oriented teaching idea in programming courses, *Eur. J. Eng. Educ.*, vol. 41, no. 3, pp. 304–319, 2016.
[3] S. Wasik, M. Antczak, J. Badura, A. Laskowski, and T. Sternal, A survey on online judge systems and their applications, *ACM Comput. Surv.*, vol. 51, no. 1, p. 3, 2018.
[4] J. H. Wu, S. P. Chen, and R. R. Yang, Development and application of online judge system, presented at 2012 International Symposium on Information Technologies in Medicine and Education, Hokodate, Japan, 2012, pp. 83–86.
[5] M. A. Revilla, S. Manzoor, and R. J. Liu, Competitive learning in informatics: The UVa online judge experience, *Olympiads in Informatics*, vol. 2, pp. 131–148, 2008.
[6] A. Kosowski, M. Malafiejski, and T. Noinski, Application of an online judge & contest system in academic tuition, in *Proc. 6th International Conference on Web Based Learning*, Edinburgh, UK, 2007, pp. 343–354.
[7] Y. W. Luo, X. L. Wang, and Z. Y. Zhang, Programming grid: A computer-aided education system for programming courses based on online judge, in Proc. 1st ACM Summit on Computing Education in China on First ACM Summit on Computing Education in China, Beijing, China, 2008, pp. 1–4.

[8] P. P. Chen, The entity-relationship model—Toward a unified view of data, ACM Transactions on Database, doi.org/10.1145/320434.320440.

[9] S. Maggiolo and G. Mascellani, Introducing CMS: A contest management system, Olympiads in Informatics, vol. 6, pp. 86–99, 2012.

[10] GitLab, GitLab.com database incident, https://about.gitlab.com/blog/2017/02/01/gitlab-dot-com-database-incident/, 2017.

[11] J. Han, H. H. E, G. Le, and J. Du, Survey on NoSQL database, presented at 2011 6th International Conference on Pervasive Computing and Applications, Port Elizabeth, South Africa, 2011, pp. 363–366.

[12] S. Josefsson, RFC 4648: The base16, base32, and base64 data encodings, https://tools.ietf.org/html/rfc4648, 2006.

[13] M. Jones, J. Bradley, and N. Sakimura, RFC 7519: JSON Web Token (JWT), https://tools.ietf.org/html/rfc7519, 2015.

[14] K. Moriarty, B. Kaliski, and A. Rusch, RFC 8018: PKCS #5: Password-based cryptography specification version 2.1, https://tools.ietf.org/html/rfc8018, 2017.

Miao Wang received the bachelor degree from Tsinghua University in 2017. He is pursuing the master degree at Tsinghua University. His research interests include distributed systems and networking.

Wentao Han received the bachelor and PhD degrees in computer science from Tsinghua University in 2008 and 2015, respectively. He is currently an assistant research fellow at the Department of Computer Science and Technology, Tsinghua University. His research interest includes parallel and distributed computing, especially in big data processing systems and neuromorphic systems.

Wenguang Chen received the bachelor and PhD degrees in computer science from Tsinghua University in 1995 and 2000, respectively. He was the Chief Technology Officer (CTO) of Opportunity International Inc. in 2000–2002. Since January 2003, he has joined Tsinghua University. He is now a professor at the Department of Computer Science and Technology, Tsinghua University. His research interest includes parallel and distributed computing, programming model, and mobile cloud computing.