Hindawi
Complexity
Volume 2018, Article ID 2170585, 15 pages
https://doi.org/10.1155/2018/2170585

Research Article

Distributed Testing System for Web Service Based on Crowdsourcing

Xiaolong Liu,[1,2] Yun-Ju Hsieh,[3] Riqing Chen,[1,2] and Shyan-Ming Yuan[3]

[1]*College of Computer and Information Sciences, Fujian Agriculture and Forestry University, Fuzhou 350002, China*
[2]*Digital Fujian Institute of the Big Data for Agriculture and Forestry, Fujian Agriculture and Forestry University, Fuzhou, Fujian 350002, China*
[3]*Department of Computer Science, National Chiao Tung University, Hsinchu 300, Taiwan*

Correspondence should be addressed to Shyan-Ming Yuan; smyuan@gmail.com

Received 29 June 2018; Revised 6 September 2018; Accepted 16 September 2018; Published 8 November 2018

Guest Editor: Zhihan Lv

To appropriately realize the performance of a web service, it is essential to give it a comprehensive testing. Although an elastic test could be guaranteed in traditional cloud testing systems, the geographical test that supports real user behavior simulation remains a problem. In this paper, we propose a testing system based on a crowdsourcing model to carry out the distributed test on a target web server automatically. The proposed crowdsourcing-based testing system (CTS) provides a reliable testing model to simulate real user web browsing behaviors with the help of web browsers scattered all around the world. In order to make the entire test process the same as the real situation, two test modes are proposed to simulate real user activity. By evaluating every single resource of web service automatically, a tester can not only find out internal problems but also understand the performance of the web service. In addition, the complete geographical test is available with the performance measurements coming from different regions in the world. Several experiments are performed to validate the functionality and usability of CTS. It demonstrates that CTS is a complete and reliable web service testing system, which provides unique functions and satisfies different requirements.

1. Introduction

Technological developments for broadband networks, distributed systems, and applications have led to tremendous changes in service-oriented society. With the advancement of web technology, service-oriented applications have become an indispensable part of our daily life. RESTful service, which uses HTTP as its underlying protocol, is considered to be the most common and important form of web application nowadays [1]. By using a readable URL, people are capable of accessing all kinds of resources on the Internet, such as web pages, pictures, video files, and diversified information assets. As for the system that provided web service, it is unavoidable to handle thousands of user requests from all over the world. If the design of the web server is not perfect, it will cause low performance and then affect user experience. Therefore, for the staging of the service lifecycle, every system should be tested properly before being implemented as a service. The testing procedure should verify whether the performance of the system is sufficiently good to satisfy users in terms of capability and availability.

Web service testing is a kind of software testing which carries out a series of tests on a web server such as a regression test, performance test, and load test [2–4] under specific conditions. With the help of web service testing, the tester can be aware of weak points of the web service and improve the defect. Currently, web service testing can be divided into two forms, i.e., single-node testing and cloud testing [5]. For single-node testing, the test is generated and executed on the local terminal or within the local area network environment. Generally, a bunch of threads will be created and used to conduct test tasks like real users. However, most web services need to deal with a great number of requests, and the restricted number of threads would make it difficult to simulate such a large-scale testing. As for cloud testing, also known as testing-as-a-service (TaaS), this testing takes advantage of cloud computing technology. The testing service will be installed in a virtual machine and deployed on each data center located around the world. However, cloud testing has some problems on the geographical test since the limited number of worldwide data centers is not enough to build the environment that is the same as the real situation of global users.

Crowdsourcing [6, 7] is recently a novel distributed problem-solving and production model, which has emerged in recent years as the information technology of the new generation. It is a process of obtaining ideas by seeking a large number of people, volunteers, or part-time workers, to contribute all kinds of abilities, aimed at achieving a better result than the traditional manner. In general, we can take crowdsourcing as a virtual labor market [8], taking advantage of advanced Internet technologies to outsource work to individuals. Since workers are a large group of individuals from all over the world, it makes possible to keep a flexible scale and produce diverse results. Therefore, based on the concept of crowdsourcing, this paper tries to design and implement a new kind of testing scenario that can not only go deep into the core structure of a web server to discover potential bottleneck but also perform reliable testing with real global user behaviors.

In this paper, a novel automatic crowdsourcing-based testing system named CTS is proposed. With the power of a crowd-joining model, we utilize the computing nodes provided by the worldwide end users to conduct a series of tests, such as performance testing, availability testing, and geographical testing. In CTS, a web browser is the interface that is taken as the test node responsible for interacting between the testing server side and the target web server. By the use of this model, testers can save their time on building a testing environment and cutting down the workload on a host machine. Our design has therefore focused on efficiency, reliability, cost minimization, and ease of use. The main contributions of our proposed automatic testing system are the following:

(1) *Real User Behavior Simulation Ability*. Generally, the traditional testing tools and services focus on generating workloads to test the web server with their own testing infrastructure. Although it can simulate user requests by using concurrent threads, this kind of simulation is not good enough to act like the way that a real user behaves in web browsing. In the proposed system, we design a reliable testing model to simulate real user web browsing behaviors with the help of web browsers scattered all around the world

(2) *Geographical Test Guarantee*. With the geographical test, the tester is able to perform the web service test on location-based service (LBS) [9], which is able to make use of the geographical location information of the end user so as to provide corresponding services. The proposed system testing service deploys numerous web browsers distributed all over the world with crowdsourcing. In this way, it can not only really support the geographical test but also carry out a large-scale load testing, which is hard to be accomplished in single-node testing and cloud testing

(3) *Comprehensive Resource Evaluation*. The simulated testing workload generated by the traditional testing service is not able to assess the functionality and availability of resources that end users actually obtain. Even if extra resources can be added, the tester still needs to configure one by one or write some scripts manually. Thus, in order to find out the real weakness and problems of the target server, the proposed system is aimed at reaching the goal that every resource in the domain could be traversed and evaluated

(4) *Detailed Test Result Provision*. The proposed system provides two testing modes for different testing purposes, i.e., blanket mode and emulation mode. We provide detailed and reliable test results through the adoption of the proposed modes, which makes the tester clearly understand the overall situation of the target system

The remainder of this paper is as follows: related works are described in Section 2. Section 3 presents the design of CTS, including the conceptual architecture and test modes. In Section 4, system implementation is illustrated. Some experiments are performed to validate the usability of the proposed testing service in Section 5. Also, discussions about the comparison between the proposed modes are given in this section. Eventually, the conclusion of this work is described in Section 6.

2. Related Works

In order to evaluate the performance and the ability of a web service, a reliable web service testing approach is essential. Early studies on web service testing have mainly focused on automated software testing [10], in which testing is created and installed automatically using the target software in a single node. Afterward, the test results, information on bugs, and logs of the service are collected using different test cases. In general, single-node testing tools create a large number of threads to simulate concurrent requests to the tested server. For example, Apache JMeter [11] is a well-known web service testing tool which creates numerous threads to simulate multiple access requests in a single node simultaneously. It enables the tester to validate the correctness of the web service under heavy loads and measure their performance. Analogously, Httperf [12] is another web service performance testing tool which provides a flexible facility for creating multiple kinds of HTTP workloads.

Cloud testing [13, 14] is an emerging model of testing which leverages the vast resources of cloud environments to offer testing services to consumers. It is viewed as a kind of service that uses a cloud computing platform to deploy the testing service on top of cloud infrastructure. Due to the advantages of cloud computing technology, elastic resource provisioning, cost efficiency, and high availability, cloud testing resolves some problems existing on the usage of single-node testing such as limited budget, high cost, and geographical distribution of users. Using the cloud testing technique, the tester can flexibly simulate real-world users with the distributed testing environment, execute a large number of test tasks without maintaining expensive hardware, and increase or decrease the computing resources according to the test requirement.

SOASTA [15] and LoadStorm [16] are industrial cloud testing tools for performance testing. They provide nice web GUI to make testers configure test plans and interact with the testing platform. All test requests are sent by virtual machines scattered in worldwide data centers. After the testing finishes, testers can view results collected from these virtual machines directly. With these tools, the tester is able to perform on-demand automated tests anytime and anywhere and conduct a large scale real-time evaluation on web services with a scalable test environment. Recently, several cloud testing services have also been proposed by scholars in academic circles. Zhu and Zhang [17] proposed a framework of collaborative testing in which test tasks were completed through the collaboration of various test services that were registered, discovered, and invoked at runtime. In order to improve the quality and performance of web application load testing, Shojaee et al. [18] proposed a method by using the existing facilities in the cloud. The pool of computing resources without initial cost, the unlimited data storage, and the cloud computing managerial procedures were integrated to improve load testing flexibility, time, and operational costs. Yan et al. [19] also selected web services as test targets and developed a load testing platform that enabled the load testing process to be as close as possible to real running scenarios. In their design, the number of concurrent requests and the geographical distribution of a test agent were customizable. However, their TaaS platform was developed based on the cloud platform as a service (PaaS), and consequently, the flexibility of the test scenarios and test client customization could not be guaranteed. To increase the flexibility of test client customization and test scenarios, Lo et al. [20] proposed an IaaS-based cloud service testing framework that could adequately adapt to different general test targets and guarantee flexibility for the test environment. The user was able to customize test scenarios involving clients, network topologies, and test scripts.

Although the elastic test could be guaranteed in cloud testing services mentioned above, the geographical test support remains a problem. The available location of VM is restricted owing to the distribution of data centers owned by the IaaS provider (e.g., Amazon). On the other hand, the test flow may possibly be influenced by the actual behavior and policy of a cloud service provider. VMs may be migrated while the test is ongoing according to the resource management policy of the cloud service provider. To solve the problems, we propose a novel web service testing system in this paper by deploying the testing service to numerous web browsers distributed all over the world with crowdsourcing. In this way, the proposed system can not only really support the geographical test but also carry out a large-scale load testing, which is hard to be accomplished in single-node testing and cloud testing. Moreover, a reliable testing model can be guaranteed to simulate real user web browsing behaviors with the help of web browsers scattered all around the world.

3. System Design

This section presents an overview and the detailed concept of the proposed automatic crowdsourcing-based testing system CTS. The main purpose of this study is to propose a referential platform for web service providers to test and specify critical resource problem in their cloud service. The conceptual architecture of CTS and the proposed test modes are illustrated in Sections 3.1 and 3.2, respectively.

3.1. Conceptual Architecture

3.1. Conceptual Architecture. Figure 1 shows the conceptual architecture of the proposed automatic web service testing system CTS. It contains the following four roles:

(1) *Tested Service*. The tested service is basically the target web service that needs to be tested

(2) *Tester*. The tester is either a web developer who wants to find out the problems on the developing website or a supervisor who desires to understand the performance of the web service. In the conceptual architecture, the tester is one of the client-side components based on a browser in CTS which is responsible for specifying all configurations related to the test and communicating with the testing platform, such as submitting test requests and retrieving test results. In addition, the tester is capable of monitoring the process while the test is running. CTS provides a concise web interface which makes the tester set up all parameters in an easy way and launch the test rapidly. Besides, the tester can easily acquire the test result from the detailed result page for the purpose of website evaluation

(3) *Test Node*. The test node is composed of the end users who volunteered to join the test by crowdsourcing. Basically, the end users just need to login to CTS as test nodes, the test will then proceed by CTS automatically. In the conceptual architecture, the test node is also one of the client-side components based on a browser in CTS which is the actual executor of the test plan. The test node is responsible for communicating with the testing platform such as accepting the test task and sending back the test results. As the test node receives a test task, it will generate workload to the tested service according to the assigned test mode. Then, it will collect the responded metrics from the target service and forward the test results to the testing platform

(4) *Testing Platform*. The testing platform is the server-side component which is in charge of managing all information and connections from client-side components: i.e., the tester and test node. Once the testing platform receives the test request from the tester, it first parses that request and wraps all test requirements into a test plan. Then, it collects the available number of test nodes and deploys the corresponding test nodes for testing according to the test plan. In

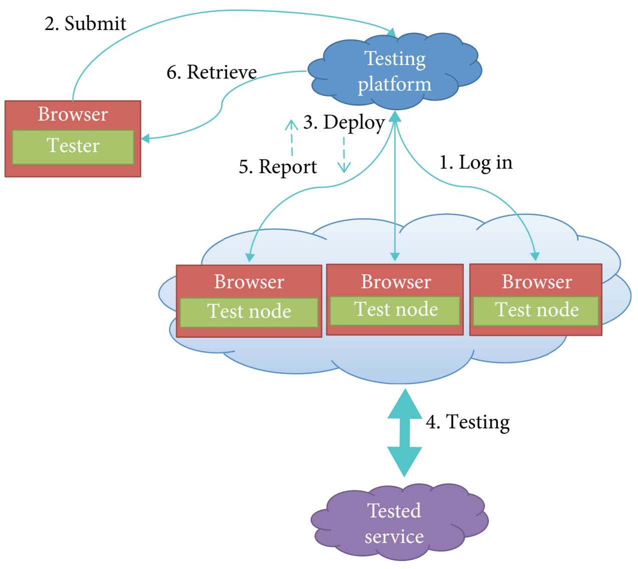


Figure 1: Conceptual architecture of CTS.

addition, the testing platform is also in charge of storing all test results from test nodes into the database

3.2. Test Modes. In the proposed CTS system, the tester and test nodes of the client side need to login to the testing platform, and the server side is responsible for accepting test requests, deploying test tasks, gathering test results, and storing test results back to the database. In order to make CTS reliable and completely enough, two test modes are proposed on the client side for testing, i.e., the blanket mode and emulation mode. The proposed test modes not only enable testers to design their own test with their testing requirement efficiently but also strengthen the ability of the test node so as to perform a thorough test on the target service. The following sections describe the two proposed test modes.

3.2.1. Blanket Mode. The purpose of this mode is to automatically perform at least one single test on all resources located in the same domain of the target tested server. By using this mode, the tester is able to find out errors existing behind the web server and understand the detailed status of the web server. In the blanket mode, the tester should first set up the configuration of the test request, including the provision of the entry URL of the tested service, the test depth of this testing, and the number of leaf resources required to be tested at each level. After that, the testing platform will create a test plan for this specific test request and selects eligible test nodes. Then, the testing platform will deploy test tasks to the corresponding test nodes and wait for each test result from different test nodes.

The workflow of the test node in the blanket mode is shown in Figure 2. Once the test node receives the test task, it will confirm the content of test configuration. Then, the test node begins to generate the test flow starting from the entry URL set by the tester. At each level, the test node will parse the URL and crawl the response body of each tested resource at the same level to find more available resources (R) that belong to the domain of the tested web server. If there are available resources, the test node will randomly select a given number of leaf resources that are set by the tester in configuration as target resources (T). The test node will put the selected resources into another test task and continue to do the same testing scenario at the next level. The testing workflow will be iteratively conducted until no more resources can be found at some levels or the current level number (N) reaches the depth (D) assigned by the tester. For every tested resource, the test node will send test results back to the server side of CTS, such as performance metrics, timestamp, status code, and country code indicating where the test node is located.

3.2.2. Emulation Mode. The design of the emulation mode lies on the idea of making the testing scenario act like the way how real user browses the website. By using this mode, the test node generates the test flow by simulating user behavior so that the tester is able to examine the status of the target server within a real-world situation. In addition, we propose two extra types in this mode based on two properties of user behavior on web browsing: hop and time. Hop means the action of transferring from one web page to another web page. Time indicates the duration time a user spends on browsing a specific website. Therefore, the hop emulation mode will simulate the real user behavior according to how many hops the user will do while browsing a specific website. The time emulation mode will simulate the real user behavior according to how many times the user will spend while browsing a specific website.

At the very beginning, the tester should also set up the configuration of the test request, including the provision of the entry URL of the tested service, the number of hops while selecting the emulation type as hop, or the duration time when the emulation type is time. After that, the testing platform will create a test plan for this specific test request and select eligible test nodes. Then, the testing platform will deploy test tasks to the corresponding test nodes and wait for each test result from different test nodes.

The workflow of the test node in the emulation mode is similar to that in the blanket mode. However, for the resource-selecting stage, the test node will only select one resource at each level to test in this mode. In addition, it needs to wait for 11 seconds to start the next test flow. The operation is simulated in accordance with a research of user browsing behavior in web page based on eye tracking [21]. The research indicated that a user in average would take 11 seconds in one web page before switching to another web page. Finally, the condition of finishing the entire test depends on what kind of emulation type is used. For the hop emulation mode, the testing process will be terminated when the number of finished test flows is equal to the number of hops set by the tester. For the time emulation mode, the entire test will not be stopped until the process lasts for the duration time set by the tester.

4. System Implementation

The overall software architecture of the proposed system is shown in Figure 3, designed with two major modules: server

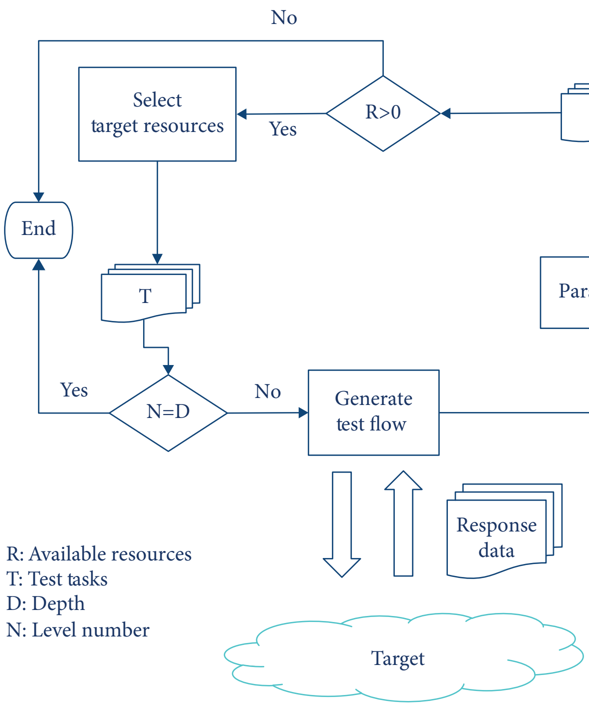


Figure 2: The workflow of the test node in the blanket mode.

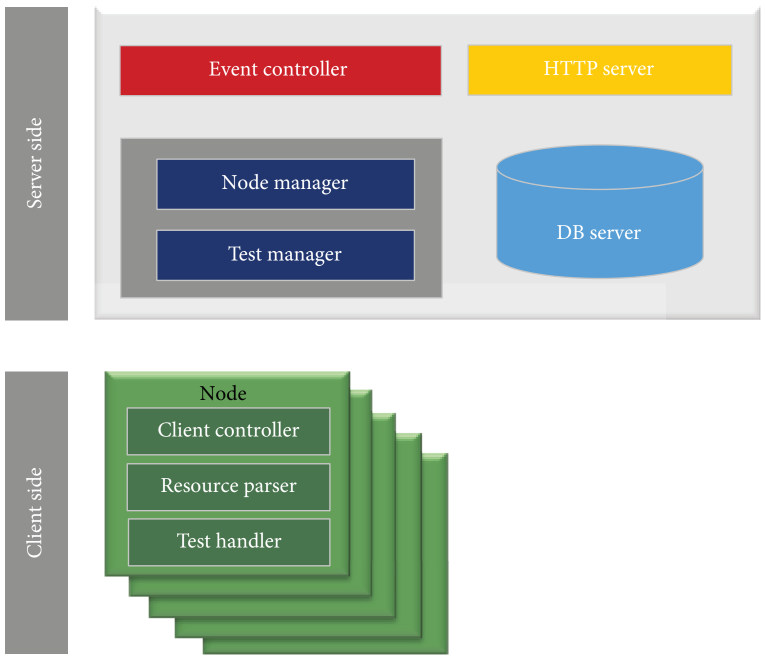


Figure 3: The software architecture of the proposed system.

side and client side. The service-side module plays the roles of the data collector and service provider, which is also in charge of session management and information exchange with the client side; the client-side module is responsible for the representation of the user interface, the communication with the service side, and the test management in different test modes. The implementation of the server-side module uses Node.js [22] to build the core components, including an event controller, test manager, node manager, HTTP server, and database server. The client-side module is composed of a client controller, resource parser, and test handler, which is implemented by JavaScript and HTML5.

4.1. Server-Side Module. Node.js is used to accomplish the functionality of the server-side module, since it is lightweight and efficient in setting up the back-end environment using a great number of embedded modules and third-party modules. In addition, Node.js takes advantage of an event-driven and nonblocking I/O model, which makes it efficient dealing with all kinds of connection issues. Some core components of the service-side module are described in detail as follows:

(1) *Event Controller*. For a service-oriented system, it is essential to deal with a variety of requests from end users. In order to resolve the communication between the server side and client side, we make use of a third-party WebSocket module [23] to manage all connections from end users. Aside from common events related to connection, we also define our own events to handle many kinds of exclusive events triggered by the client side or signaled by the server side. Some of these events are related to job assignment, and some of them are used to help the node manager and test manager execute service commands or process different kinds of notification. The event controller will take the corresponding action in accordance with the specific event name immediately as long as it confirms any event from the tester or test node. For example, an event named "test_request" will trigger the test manager to prepare for deploying test task, an event called "node_info" will transfer the information about the test node to the node manager to update the node list, and a "test_result_query" event sent from the tester will look for a particular test result in the database according to the given test ID

(2) *Node Manager*. The node manager is responsible for managing test nodes and testers. Each node that is placed on standby for job assignment will be saved into a list. Some information about each kind of nodes will be kept by this module, including ID, operating system, browser type, geographical location, and estimated bandwidth. In addition to node information management, the node manager is in charge of selecting available test nodes in accordance with the requirement of test request and providing the information of the selected test nodes to the test manager. As long as the test task is ready and the request of provision of proper test nodes is published from the test manager, the node manager will first check the number of current nodes to guarantee the correctness. Then, the node manager will choose the corresponding test nodes from node list under the policy of balance between node utilization and load balance of each node

(3) *Test Manager*. The test manager has the functionality of dealing with any request related to the test. A test task object will be created and be pushed into a test list when an event for test request is triggered. All parameters in the test configuration will be stored in the test task. The test manager is an indispensable component as it not only manages all test tasks requested from different testers but also deploys all test tasks to the corresponding test nodes. Once the event controller receives one test request from a tester, the test configuration form within the test request will be transferred to the test manager at once. Then, the test manager will command the test nodes to execute their test flows according to the designated test mode after acquiring available instances of the test node from the node manager. When test task is in progress, the test manager is in charge of monitoring the progress of the test task and collecting all test results sent back from different test nodes

(4) *Http Server*. We make use of the express module provided by package manager of Node.js to implement the host of our platform. Express provides a thin layer of fundamental web application features, which makes it easy to handle common routines of web application such as reaction to POST and GET, basic authentication, and cookie manipulation. In addition, the simple syntax of Express eases the complexity of implementation. The Http module is responsible for building the web environment and plays the role of the service provider. It provides an interface to make end users be able to get access to the testing platform and authenticates valid users from all over the world

(5) *Database Server*. For the storage of all kinds of data, we choose MongoDB [24] as the database of our platform. MongoDB is a document-oriented database and also classified as a NoSQL database. By the use of JSON-like documents with the so-called BSON format, it makes data operation more efficient and easier. The database server is implemented under the support of the Node.js module that is able to connect to the MongoDB server and take actions like the traditional database such as insert, update, delete, and query on the database server. We create three documents for the purpose of storing all information about the online test node, content of the test, and test result. The tester is capable of retrieving test results with the help of the database module at any time

4.2. Client-Side Module. The client-side module is a web application which provides an interface for both the tester and test node to join and become one member of our platform. Developers can login as testers to make tests on target tested servers. For test nodes, end users from all over the world are welcome to attend our testing platform and contribute their spare computing resources to help carry out valuable and meaningful test tasks. The following components are mostly implemented with the modern web technology such as CSS for static layout, JavaScript for handling the dynamic functionality of the test, and HTML5 for charging of both static and dynamic features.

(1) *Client Controller*. This component is divided into two parts: control of web interface and message exchange with the server side. For interface control part, we use

HTML5 and CSS to construct the interface. The tester can clearly understand how to operate the functional tabs, either leading to the tester doing the configuration and submission of the test request or providing the tester the dashboard of the test result. The dynamic operation and interactive representation rely on the technique of JavaScript and jQuery. For message exchange part, this module is used to communicate with the server side by the WebSocket protocol [25]. Both the tester side and test node side need to initialize the client instance and connect to the event controller first. After connecting to the event controller, the client controller will keep waiting for messages from the event controller and forwarding test data to it

(2) *Resource Parser*. The resource parser is responsible for parsing the response body of the tested resource to obtain more resources located in the target server and then generate another test task for the next level. We implement a web crawler on this component. The following description splits the implementation into four steps:

(a) First, the component will continue to analyze each response text sent from separate tested resources in order to parse more hyperlinks until the test handler finishes the certain test task

(b) The resource parser will do URL normalization on those hyperlinks found in the first step, which is the process of transforming a relative URL into an absolute URL. Then, it stars to select resources which are located in the domain of the target server from the normalized URLs

(c) In this stage, the duplicated resources as well as those having been tested will be eliminated from the list. Additionally, the target resources in the current task will be recorded for the purpose of future deduplicated process

(d) In the final step, the target resources among candidate resources will be decided and then put into another test task for the next test process

(3) *Test Handler*. The test handler is responsible for generating the test flow by sending HTTP request to the target server. The module is implemented by the use of Web Worker [26] specified within HTML5, and it is executed in the background without disturbing the normal operation of the browser. For the test flow part, we make use of a Web API called Xmlhttprequest [27] to send each http request in the mechanism of cross-origin resource sharing (CORS) [28]. When each test flow is started, the test handler will compute the performance measurements for each tested resource before the response arrived. Besides, the status code of each request is available in the response header. The code with 200 means that the target resource can be accessed successfully. Status for timeout can also be measured with timeout time set to 30 seconds. The test flow is done as the response text finally arrived, and then the test handler will forward the response text to the resource parser, wrap all the retrieved data, including timestamp, status code, and performance result, into the test result for the target resource, and send the test result back to the server side

5. Experimental Results and Discussions

In this section, we perform three experiments for the purpose of validating the functionality and evaluating the usability of our automatic crowdsourcing-based testing system CTS. The experiments are carried out under the proposed test modes: blanket mode and emulation mode. Through these experiments, we aim to identify the difference between test modes and make test results independently. The test results contain status of coverage, performance measurement, geographical measurement, and error report. The performance measurements include three measurements, e.g., latency, response time, and processing time. Latency means the time period from the starting time of request to the arriving time of response. Response time means the round trip time (RTT) between the test node and target server. Processing time is obtained by subtraction from latency, and response rime stands for the accessing time of the responded results. In the experiments, 30 volunteers from 10 countries are invited to participate in CTS as test nodes. The target service for testing is the Museum Service provided by the National Chiao Tung University Library [29]. In the experiment, the tests are conducted with different configurations for different modes. The detailed results of the experiments for the blanket mode, emulation mode with hop, and emulation mode with time are presented in Section 5.1, 5.2, and 5.3, respectively. Section 5.4 will discuss the usability of CTS in two test modes according to the results of three experiments.

5.1. Blanket Mode. The first experiment is carried out under the blanket mode. In test configuration, the test node number is set to 20, and the values of depth and leaf resource number are both assigned as 30. The main web page of the target service is used to be the entry URL of the entire experiment. Table 1 shows the test result of the blanket mode in terms of hit ratio. During the testing, the total number of resources being found on the target server is 1343, and 1337 resources of which are tested. The hit ratio is 99.55%, which almost covers the whole domain of the target server.

Table 2 gives the statistics of three average performance measurements. In overall measurement, the average latency is 188.81 ms with minimum and maximum values of 2 ms and 9761 ms. As for the response time, maximum is up to 31,063 ms and minimum is 3 ms. In average, it requires 3189.12 ms to finish a test task. The situation of processing time is quite the same as the response time. It takes 3 seconds approximately to load the content of resource.

Figures 4–6 show the average geographical measurements of the blanket mode results from 10 countries in terms of latency, response time, and processing time, respectively.

Table 1: Test result of the blanket mode (coverage).

Number of tested resources	1337
Total number of found resources	1343
Hit ratio (%)	99.55%

Table 2: Test result of the blanket mode (performance measurements).

	Min (ms)	Max (ms)	Average (ms)
Latency	2	9761	188.81
Response time	3	31,063	3189.12
Processing time	1	30,907	3000.31

From Figure 4, results from Taiwan perform the best in latency with 18.31 ms among all countries, while those from Sweden result in the highest latency with 630.19 ms. Besides, countries from Asia extensively excel in latency compared to countries in other continents. Figure 5 shows the regional evaluation of response time. We can see that most countries tend to take at least 1 second on accessing one resource. In particular, the performance of United States and Korea did not coincide with the performance on latency. The response time is 5871.55 ms and 4287.67 ms for United States and Korea, respectively, which are longer than the required time for other countries. However, performances of Singapore (270.19 ms) and Taiwan (48.25 ms) overwhelm those of other regions. Figure 6 gives the overview of the average processing time according to different countries. As for average processing time, it is nearly the same as the condition of response time. The highest one is 5644.21 ms in United States, and Taiwan still performs the best with 29.94 ms.

Figure 7 shows the distribution of test nodes and timeout requests. The reason leading to the difference between the regions depends on the number of test nodes in one region and the level that every test node actually reaches. In this experiment, the country where most tests are launched is United States (2306), and the test nodes generating the least tests are from Sweden (139). As for the test timeout, there are 531 timeout cases that come from 7 countries. Korea (130) and United States (339) are countries where much timeout occurs. The cause of timeout might result from several circumstances such as distance, size of the resource, and bandwidth.

Table 3 lists the examples of errors and exception results that exist in the target server, including not found, access forbidden, and redirection issue. The redirection issue means that it is disallowed to do redirection on a browser while sending request based on the security issue. In this experiment, we find out that there are 5 resources which are unavailable, 1 error caused by access forbidden and 11 exceptions due to redirection issue.

5.2. Emulation Mode with Hop. The second experiment is carried out under the emulation mode with the emulation type as hop. We select 20 test nodes to simulate real user

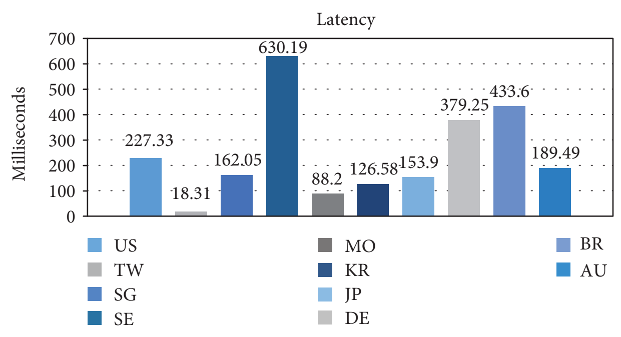


Figure 4: Geographical measurements of the blanket mode (latency).

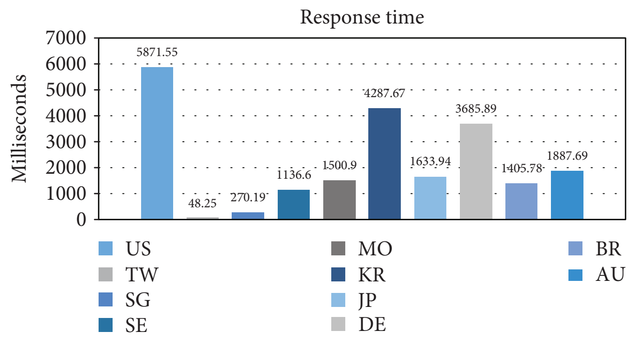


Figure 5: Geographical measurements of the blanket mode (response time).

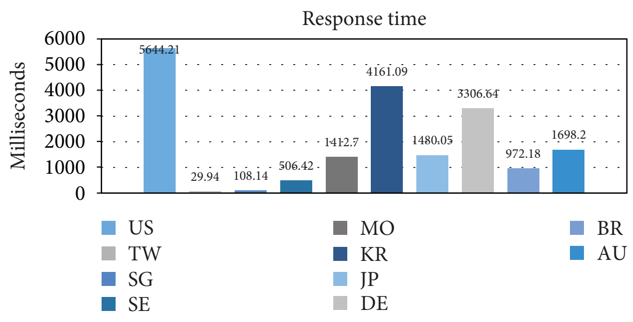
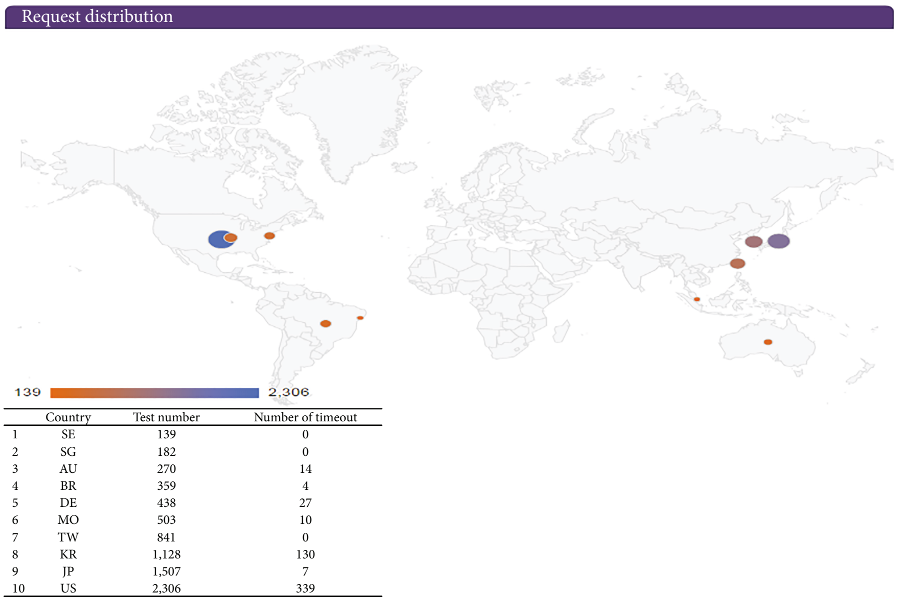


Figure 6: Geographical measurements of blanket mode (processing time).

behaviors on exploring web content based on switching times between web pages. The number of hops is set as 18, which is configured based on a research of user behavior on web browsing by Leslie et al. [30]. Their research indicates that the average number of web pages viewed on a specific website by users is 18. Therefore, the result of our experiment would be more close to the real situation.

Table 4 shows the test result of the emulation mode with hop in terms of hit ratio. During the testing, the total number of resources being found on the target server is 238, and 101 resources of which are tested. The hit ratio is 42%. Table 5 gives the statistics of the three average performance



	Country	Test number	Number of timeout
1	SE	139	0
2	SG	182	0
3	AU	270	14
4	BR	359	4
5	DE	438	27
6	MO	503	10
7	TW	841	0
8	KR	1,128	130
9	JP	1,507	7
10	US	2,306	339

Figure 7: Test result of the blanket mode (request distribution).

Table 3: Test result of the blanket mode (errors and exceptions).

	Path	Description
1	/cht/history.htm	Not found
2	/cht/link/file:///D|/AppServ/www/web/index.htm	Access forbidden
3	/cht/campus/mov-1.htm	Redirection issue

Table 4: Test result of the emulation mode with hop (coverage).

Number of tested resources	101
Total number of found resources	238
Hit ratio (%)	42%

Table 5: Test result of the emulation mode with hop (performance measurements).

	Min (ms)	Max (ms)	Average (ms)
Latency	6	9833	620.49
Response time	7	10,430	936.34
Processing time	0	5375	315.84

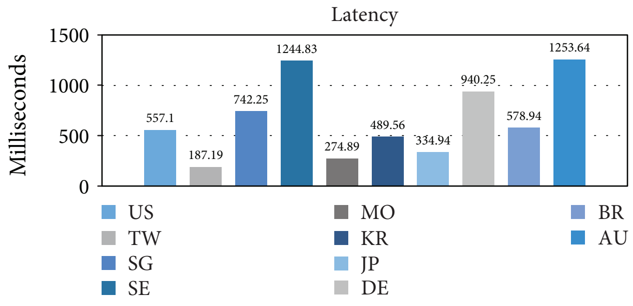


Figure 8: Geographical measurements of the emulation mode with hop (latency).

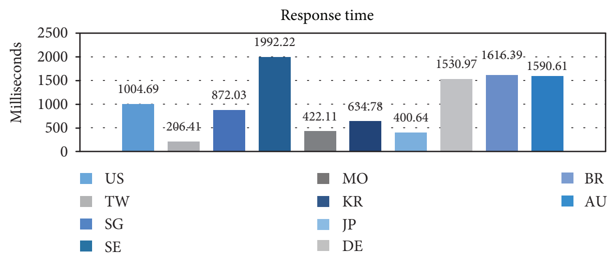


Figure 9: Geographical measurements of the emulation mode with hop (response time).

measurements in the emulation mode with hop. For overall measurement, the average latency is 620.49 ms with minimum and maximum values of 6 ms and 9833 ms, respectively. As for the response time, the average is 936.34 ms. The maximum value is 10,430 ms and the minimum value is 7 ms, which are not as high as that in the experiment of the blanket mode. For processing time, all test nodes are able to receive the response text within 6 seconds and take 315.84 ms in average.

Figures 8–10 show the average geographical measurements of the results from the emulation mode with hop from 10 countries. Figure 8 shows that assessments from Taiwan (187.19 ms) and Macao (274.89 ms) are better than those from other countries. The highest latency is in both Sweden (1244.83 ms) and Australia (1253.64 ms). There is an upward trend in latency among all regions compared to the results in first experiment. Figure 9 displays the geographical

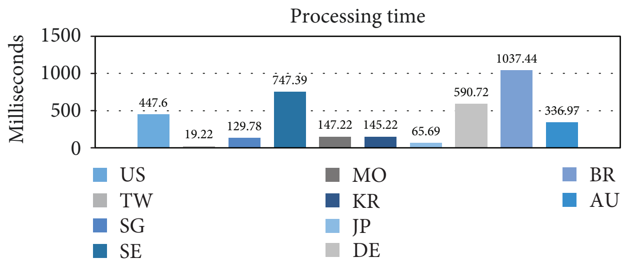


Figure 10: Geographical measurements of the emulation mode with hop (processing time).

measurements of the response time. The round trip time obtained from Germany (1530.97 ms), Brazil (1616.39 ms), and Australia (1590.61 ms) presents the same outcomes. The highest one appears in Sweden (1992.22 ms). The measurements in others are mostly under 1 second. As for the average processing time of different countries, Figure 10 shows that the measurements from Asian countries such as Japan (65.69 ms), Singapore (129.78 ms), and Korea (145.22 ms) are rather low and the performance of Brazil (1037.44 ms) is the worst.

Figure 11 illustrates the distribution of test nodes and timeout requests. In this experiment, the regions where the least tests are launched are Sweden (18), Brazil (18), and Macao (18), and the test nodes that perform the most tests come from United States (72). As for the test timeout, there is 0 timeout case which occurs in any country. This indicates that the target server can retain a certain good quality to serve end users under the circumstance in this experiment. In addition, there is only 1 access forbidden error found in this experiment which is listed in Table 6.

5.3. Emulation Mode with Time. The last experiment is carried out under the emulation mode with the emulation type as time. Through this experiment, we generate the context on the basis of the residence time which users spend on browsing contents of some web servers. There are 20 test nodes used to perform the script simulating behaviors like real users. In the light of the result from a research done by Leslie et al. [30], it showed that the average time spent browsing one website was 9 minutes. Accordingly, we let run interval be set as 9 minutes. In this way, the environment of the experiment will get closer to the real situation.

Table 7 shows the test result of the emulation mode with time in terms of hit ratio. During the testing, the total number of resources being found on the target server is 259, and 121 resources of which are tested. The hit ratio is 46.72%. Table 8 gives the statistics of the three average performance measurements in the emulation mode with time. Regarding the whole measurement, the average latency is 611.46 ms with minimum and maximum values of 7 ms and 13,457 ms. As for the response time, the max value is 15,340 ms and the min value is 8 ms. The average values of response time and processing time are 942.18 ms and 330.72 ms, respectively. Basically, the results are similar to those in the second experiment.

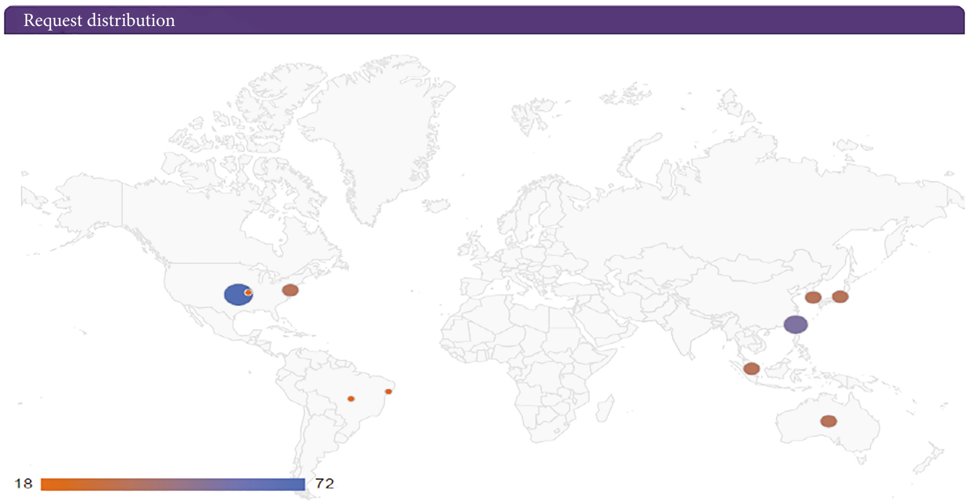
Figures 12–14 show the average geographical measurements of the results from the emulation mode with time from 10 countries. From Figure 12, results from Taiwan and Macao are the best in latency with 158.06 ms and 162.7 ms, respectively, among all countries, while those from Sweden lead to the highest latency with 1633.63 ms. In addition, latency values of Germany (813.99 ms), Brazil (885.96 ms), and Australia (799.42 ms) are about the same and are around 800 ms. Figure 13 shows the geographical measurements of response time. We can see that the condition is quite the same as latency. The results from Japan (571.07 ms), Korea (808.61 ms), and Singapore (559.45 ms) fall in the middle place. However, Sweden (2550.68 ms) is still the region where users get the worst experience. As for the average processing time shown in Figure 14, evaluations from countries located in Asia perform better than those from other countries. It is worth nothing that Brazil (804.18 ms) turns to be at low performance evidently for processing time. The best result comes from Taiwan (43.15 ms).

In this experiment, as shown in Figure 15, the region where the most tests are launched is in United States (111), and the test nodes which perform the least tests come from Sweden (19). Actually, test distribution is even among most regions. No timeout case takes place among all countries during the experiment. The situation is the same as that in the second experiment, which represents the stable service provision by the target server, and users are able to explore the content smoothly. There are 4 errors that have been found in this experiment which are listed in Table 9.

5.4. Discussions. The experimental results in different test modes indicate that the proposed testing system CTS is able to make a complete test on the target server. In this section, we will discuss four abilities of the system and the usability of proposed test modes according to the results of three experiments.

5.4.1. Resource Evaluation. In accordance with the test results in terms of hit ratio, the goal to evaluate each single resource of the target server is truly achieved under both the blanket mode and emulation mode. The tester can look over all the tested resources with the assistance of visualized diagrams provided by CTS one by one. The hit ratio is the key point to verify the completeness of each result. The higher the hit ratio, the larger the coverage of testing to the target service would be. Figure 16 shows the comparison of the number of tested resources and the hit ratio among three experiments. Among the results, the experiment of the blanket mode shows higher proportion of the tested resources to the total found resources than that obtained by both experiments of the emulation mode. The reason is that in the blanket mode, more than one resource will be put into the test task at each level, and only one resource will be evaluated in the emulation mode. Therefore, if the tester aims to perform a comprehensive test, the blanket mode is the suitable method to satisfy the requirement.

5.4.2. Performance Measurement. Figure 17 shows the comparison of test results of performance measurement under the experiments. It can be found that the results of latency, response time, and processing time appear to be the same



	Country	Test number	Number of timeout
1	SE	18	0
2	BR	18	0
3	MO	18	0
4	DE	36	0
5	AU	36	0
6	SG	36	0
7	JP	36	0
8	KR	36	0
9	TW	54	0
10	US	72	0

Figure 11: Test result of the emulation mode with hop (request distribution).

Table 6: Test result of the emulation mode with hop (errors and exceptions).

	Path	Description
1	/cht/link/file:///D|/AppServ/www/web/index.htm	Access forbidden

Table 7: Test result of the emulation mode with time (coverage).

Number of tested resources	121
Total number of found resources	259
Hit ratio (%)	46.72%

Table 8: Test result of the emulation mode with time (performance measurements).

	Min (ms)	Max (ms)	Average (ms)
Latency	7	13,457	611.46
Response time	8	15,304	942.18
Processing time	0	2318	330.72

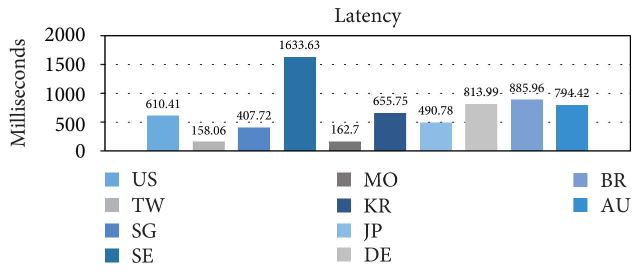


Figure 12: Geographical measurements of the emulation mode with time (latency).

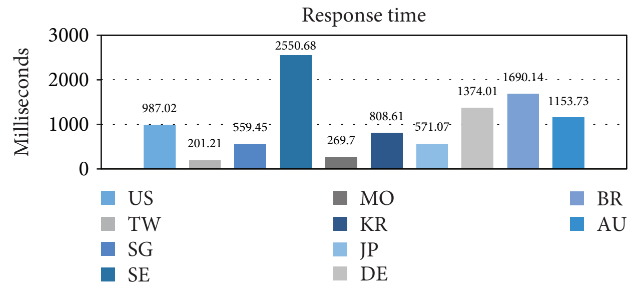


Figure 13: Geographical measurements of the emulation mode with time (response time).

when tests are run in the emulation mode with types as hop and time, which means that the proposed system can fulfill the demand for a real state of affairs by simulating the real user's web browsing activities. For the result conducted under the blanket mode, it leads to extremely long time spending on processing the response data between test nodes and the target server although the lower latency is given. The reason is that there is a large quantity of test flows generated by test nodes at the same time. Moreover, most of them are image and video files whose sizes are so big that the target server needs to consume a lot of computing resources and

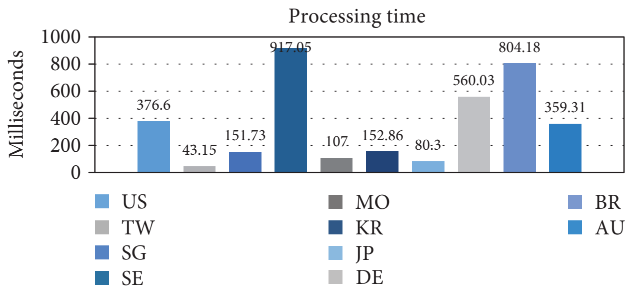


Figure 14: Geographical measurements of the emulation mode with time (processing time).

take more time on fetching those files in different disk sectors and forwarding them back. In addition, the bandwidth speed and distance between test nodes and the target server are also the factors that might cause excessively high measurement. The results of timeout also demonstrate that test nodes from several countries are not able to successfully receive entire response data in a certain period of time. As a result, it is suitable for the tester to give load testing or stress testing using the blanket mode, while the emulation mode can be used to understand the status in real-time situation.

5.4.3. Geographical Testing. With the high distribution and powerful contribution of worldwide crowd, the tester could collect a great amount of test data provided by test nodes from several countries, which enables the testing platform to analyze different performances based on the location of the test node. Through the demonstration of geographical measurement and request distribution, the experiments show different results. The outcomes for some countries retain the same among the experiments. However, the ranking of performance might change dramatically for some countries. This discrepancy resulted from two conditions: the first one is the number of test nodes distributed over each country. If there is unbalanced distribution of test nodes, the result might be unreliable. As we can see, the request numbers for United States and Japan are much more than those for other countries like Sweden and Singapore in the first experiment. Such case might lead to the opposite result due to unbalanced request distribution. The other condition is due to different numbers of resources each test node would obtain. The available resources tested would vary from test node to test node at each level. Some test nodes might finish the entire test early if no more resource can be found, and some would run the complete process according to the test configuration. These cases are more possible to happen in the blanket mode since more than one resource would be put into the test task randomly. Figure 18 shows the distribution of test request and timeout in the blanket mode. It can be found that there are 531 timeout cases that come from 7 countries during the blanket mode, whereas no timeout case takes place among all countries during the experiments of the emulation mode. As a result, if the tester would like to perform the geographical test, the use of the emulation mode might satisfy such requirement and is more suitable than the use of the blanket mode.

5.4.4. Error Detection. In addition to the ability of service evaluation, the testing platform is also equipped with the capability of identifying the errors hidden in the target service. According to the experimental results, there are some problems that indeed exist inside the tested service. Several resources are not available due to broken link or incorrect path name. Furthermore, the problematic expression of a resource identifier, which is vulnerable to be exploited by intended users, is also detected through the experiments.

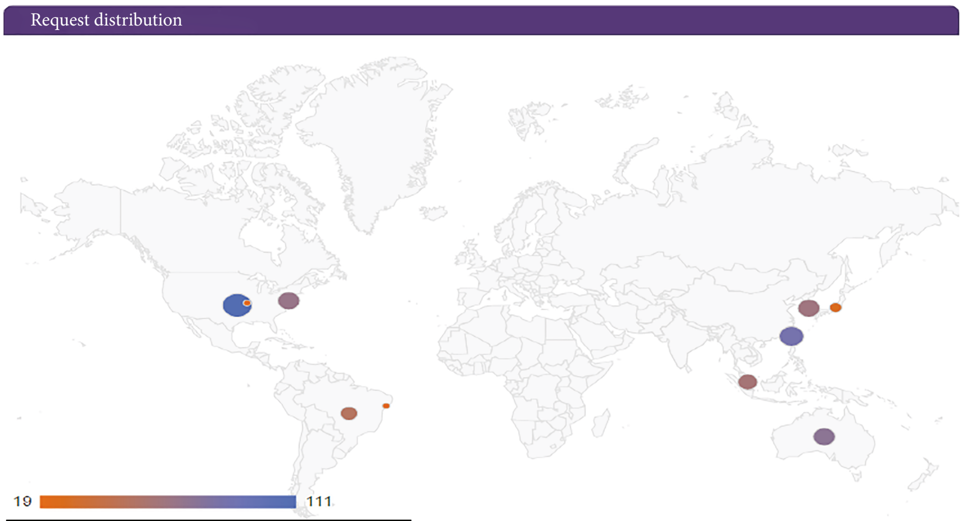
Besides, the results illustrate some exceptions that occur during the procedure of accessing resources. In the experiments, most exceptions result from the redirection of a resource identifier. Since we send HTTP request to the target server with the support of cross-origin resource sharing (CORS), there are some limitations and issues associated with CORS. Most web browsers are not allowed to deal with the redirection condition while using CORS request, and the reason is all about security concern. Without any safe guarantee of unknown location of the resource indicated within the response header, web browsers are required to cancel the redirection from being exposed to any risk. In addition, response headers without indication of access control would also be taken as invalid request by the specification of CORS. These network errors are classified into exception.

Figure 19 shows the comparison of results of found errors and exceptions in different test modes. It can be found that there are more errors detected using the blanket mode than using the emulation mode. The reason is that more resources are evaluated and a comprehensive test can be carried out with the blanket mode. Therefore, testers can choose to use the blanket mode if they look forward to complete error detection results.

6. Conclusions

This paper proposed a novel distributed crowdsourcing-based testing system CTS to ensure automatic testing for the target web service. CTS constructs a reliable testing platform by crowdsourcing, where testers are capable of deploying the web service test on worldwide web browsers. Two test modes are proposed in CTS: the blanket mode is used to give an extensive test on all resources kept in the target server and the emulation mode is capable of providing human-like circumstance in order to make anthropomorphic testing in real situation. In order to validate the functionality and usability of CTS, several experiments are performed. The experimental results reveal that the proposed system is able to make a complete test on the target server and both test modes could deeply evaluate the target service. It also demonstrates that CTS is a complete and reliable web service testing system, which provides unique functions and satisfies different requirements so as to perform availability testing, performance testing, scalability testing, and geographical testing.

Inevitably, at present, the number of volunteers joining our experiments is not enough to form a considerable scale with the concept of a crowdsourcing model. Therefore, in order to take advantage of global users' capability and make more effective tests, in the future, we have to raise more



	Country	Test number	Number of timeout
1	SE	19	0
2	MO	20	0
3	JP	27	0
4	BR	51	0
5	SG	60	0
6	KR	64	0
7	DE	68	0
8	AU	74	0
9	TW	86	0
10	US	111	0

FIGURE 15: Test result of the emulation mode with time (request distribution).

TABLE 9: Test result of the emulation mode with time (errors and exceptions).

	Path	Description
1	/cht/history.htm	Not found
2	/cht/sitemap/history.htm	Not found
3	/eng/share/people14.jpg	Not found
4	/cht/link/file:///D|/AppServ/www/web/index.htm	Access forbidden

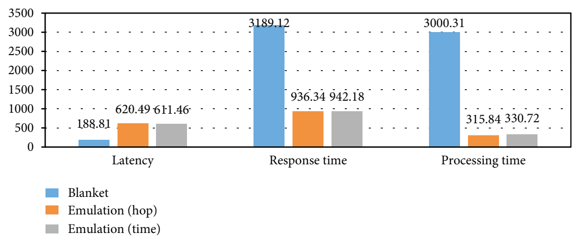


FIGURE 17: Comparison of performance between different test modes.

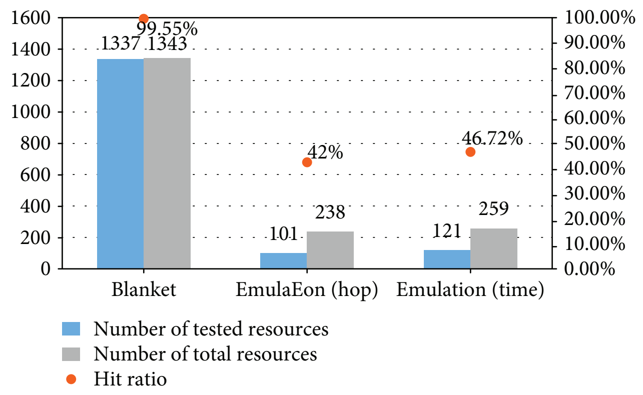


FIGURE 16: Comparison of coverage between different test modes.

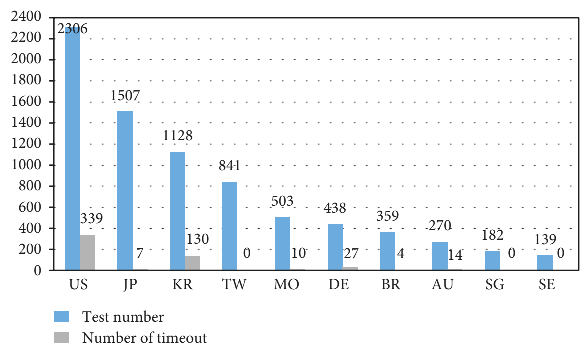
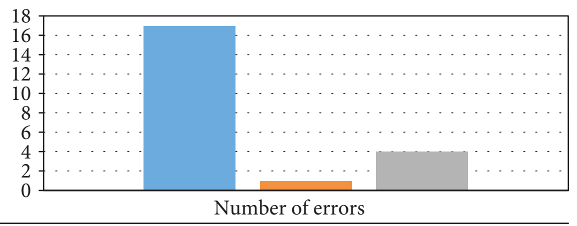


FIGURE 18: Comparison of the test number and timeout among different regions in the blanket mode.



🟦 Blanket	17
🟧 Emulation (hop)	1
⬜ Emulation (time)	4

Figure 19: Comparison of the number of found errors and exceptions.

incentives to increase the popularity of our service and promote its powerful functionality to the public.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the fund from the National Natural Science Foundation of China (Grant nos. 61702102 and 61702100) and Fujian Provincial Natural Science Foundation of China (Grant no. 2018J05100) and in part by the Ministry of Science and Technology (MOST) of Taiwan (Grant 107-2623-E-009-006-D).

References

[1] F. Belqasmi, R. Glitho, and C. Fu, “Restful web services for service provisioning in next-generation networks: a survey,” *IEEE Communications Magazine*, vol. 49, no. 12, pp. 66–73, 2011.

[2] G. Kang, M. Tang, J. Liu, X. F. Liu, and B. Cao, “Diversifying web service recommendation results via exploring service usage history,” *IEEE Transactions on Services Computing*, vol. 9, no. 4, pp. 566–579, 2016.

[3] S. Hussain, Z. Wang, I. K. Toure, and A. Diop, “Web service testing tools: a comparative study,” *International Journal of Computer Science Issues*, vol. 10, no. 1, pp. 641–647, 2013.

[4] Z. Pan, S. Liu, A. K. Sangaiah, and K. Muhammad, “Visual attention feature (VAF): a novel strategy for visual tracking based on cloud platform in intelligent surveillance systems,” *Journal of Parallel and Distributed Computing*, vol. 120, pp. 182–194, 2018.

[5] C. Y. Chiang, C. P. Chang, H. Y. Chen, Y. L. Chen, S. M. Yuan, and C. Wang, “ATP: a browser-based distributed testing service platform,” in *2016 International Computer Symposium (ICS)*, pp. 192–197, Chiayi, Taiwan, 2016.

[6] A. Doan, R. Ramakrishnan, and A. Y. Halevy, “Crowdsourcing systems on the world-wide web,” *Communications of the ACM*, vol. 54, no. 4, pp. 86–96, 2011.

[7] K. Wang, X. Qi, L. Shu, D. J. Deng, and J. J. P. C. Rodrigues, “Toward trustworthy crowdsourcing in the social internet of things,” *IEEE Wireless Communications*, vol. 23, no. 5, pp. 30–36, 2016.

[8] T. de Vreede, C. Nguyen, G.-J. de Vreede, I. Boughzala, O. Oh, and R. Reiter-Palmon, “A theoretical model of user engagement in crowdsourcing,” in *International Conference on Collaboration and Technology*, pp. 94–109, Springer Berlin Heidelberg, 2013.

[9] I. A. Junglas and R. T. Watson, “Location-based services,” *Communications of the ACM*, vol. 51, no. 3, pp. 65–69, 2008.

[10] P. K. Kapur, H. Pham, U. Chanda, and V. Kumar, “Optimal allocation of testing effort during testing and debugging phases: a control theoretic approach,” *International Journal of Systems Science*, vol. 44, no. 9, pp. 1639–1650, 2013.

[11] E. H. Halili, *Apache Jmeter: A Practical Beginner’s Guide to Automated Testing and Performance Measurement for Your Websites*, Packt Publishing Ltd, 2008.

[12] D. Mosberger and T. Jin, “Httperf—a tool for measuring web server performance,” *Acm Sigmetrics Performance Evaluation Review*, vol. 26, no. 3, pp. 31–37, 1998.

[13] L. Riungu-Kalliosaari, O. Taipale, K. Smolander, and I. Richardson, “Adoption and use of cloud-based testing in practice,” *Software Quality Journal*, vol. 24, no. 2, pp. 337–364, 2016.

[14] H. Jin, X. Yao, and Y. Chen, “Correlation-aware QoS modeling and manufacturing cloud service composition,” *Journal of Intelligent Manufacturing*, vol. 28, no. 8, pp. 1947–1960, 2017.

[15] SOASTA2018, http://www.soasta.com.

[16] LoadStorm2018, http://loadstorm.com/.

[17] H. Zhu and Y. Zhang, “Collaborative testing of web services,” *IEEE Transactions on Services Computing*, vol. 5, no. 1, pp. 116–130, 2012.

[18] A. Shojaee, N. Agheli, and B. Hosseini, “Cloud-based load testing method for web services with VMs management,” in *2015 2nd International Conference on Knowledge-Based Engineering and Innovation (KBEI)*, pp. 170–176, Tehran, Iran, 2016.

[19] M. Yan, H. Sun, X. Wang, and X. Liu, “Building a TaaS platform for web service load testing,” in *2012 IEEE International Conference on Cluster Computing*, pp. 576–579, Beijing, China, 2012.

[20] W. T. Lo, X. Liu, R. K. Sheu, S. M. Yuan, and C. Y. Chang, “An architecture for cloud service testing and real time management,” in *2015 IEEE 39th Annual Computer Software and Applications Conference*, pp. 598–603, Taichung, Taiwan, 2015.

[21] X. Zhang, S. M. Yuan, M. D. Chen, and X. Liu, “A complete system for analysis of video lecture based on eye tracking,” *IEEE Access*, vol. 6, pp. 49056–49066, 2018.

[22] C. Lin, Y. Bi, G. Han, J. Yang, H. Zhao, and Z. Liu, “Scheduling for time-constrained big-file transfer over multiple paths in cloud computing,” *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 2, no. 1, pp. 25–40, 2018.

[23] L. Pinca, “The fastest RFC-6455 WebSocket implementation for Node.js,” 2018, https://github.com/websockets/ws.

[24] M. Elhoseny, A. Abdelaziz, A. S. Salama, A. M. Riad, K. Muhammad, and A. K. Sangaiah, “A hybrid model of internet of things and cloud computing to manage big data in health services applications,” *Future Generation Computer Systems*, vol. 86, pp. 1383–1394, 2018.

[25] M. Heinrich and M. Gaedke, "WebSoDa: a tailored data binding framework for web programmers leveraging the WebSocket protocol and HTML5 Microdata," in *International Conference on Web Engineering*, pp. 387–390, Springer, Berlin, Heidelberg, 2011.

[26] J. Harjono, G. Ng, D. Kong, and J. Lo, "Building smarter web applications with HTML5," in *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*, pp. 402-403, Toronto, Ontario, Canada, USA, 2010.

[27] K. A. Van and D. Jackson, "The xmlhttprequest object," in *World Wide Web Consortium, Working Draft WD-XMLHttpRequest-20070618*, W3C, 2007.

[28] K. A. Van, "Cross-origin resource sharing," in *W3C Working Draft WD-cors-20100727*, Betascript Publishing, 2010.

[29] NCTU Museum, 2018, http://museum.lib.nctu.edu.tw/.

[30] E. Leslie, A. L. Marshall, N. Owen, and A. Bauman, "Engagement and retention of participants in a physical activity website," *Preventive Medicine*, vol. 40, no. 1, pp. 54–59, 2005.