Autograding Distributed Algorithms in Networked Containers

Evan Maicus
Rensselaer Polytechnic Institute
maicue@rpi.edu

Matthew Peveler
Rensselaer Polytechnic Institute
pevelm@rpi.edu

Stacy Patterson
Rensselaer Polytechnic Institute
sep@cs.rpi.edu

Barbara Cutler
Rensselaer Polytechnic Institute
cutler@cs.rpi.edu

ABSTRACT

We present a container-based system to automatically run and evaluate networked applications that implement distributed algorithms. Our implementation of this design leverages lightweight, networked Docker containers to provide students with fast, accurate, and helpful feedback about the correctness of their submitted code. We provide a simple, easy-to-use interface for instructors to specify networks, deploy and run instances of student and instructor code, and to log and collect statistics concerning node connection types and message content. Instructors further have the ability to control network features such as message delay, drop, and reorder. Running student programs can be interfaced with via stream-controlled standard input or through additional containers running custom instructor software. Student program behavior can be automatically evaluated by analyzing console or file output and instructor-specified rules regarding network communications. Program behavior, including logs of all messages passed within the system, can optionally be displayed to the student to aid in development and debugging. We evaluate the utility of this design and implementation for managing the submission and robust and secure testing of programming projects in a large enrollment theory of distributed systems course. This research has been implemented as an extension to Submitty, an open source, language-agnostic course management platform with automated testing and automated grading of student programming assignments. Submitty supports all levels of courses, from introductory to advanced special topics, and includes features for manual grading by TAs, version control, team submission, discussion forums, and plagiarism detection.

CCS CONCEPTS

• **Computing Education** → Computing Education Programs; Computer Science Education;

KEYWORDS

Distributed Systems, Testing, Autograding, Container

SIGCSE '19, February 27-March 2, 2019, Minneapolis, MN, USA

ACM ISBN 978-1-4503-5890-3/19/02…$15.00
https://doi.org/10.1145/3287324.3287505

ACM Reference Format:
Evan Maicus, Matthew Peveler, Stacy Patterson, and Barbara Cutler. 2019. Autograding Distributed Algorithms in Networked Containers. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19), February 27-March 2, 2019, Minneapolis, MN, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3287324.3287505

1 INTRODUCTION

Testing and grading networked programming assignments is a complex endeavor. As a first step, the tester must distribute student code to many nodes — running either locally or remotely — each of which must be configured to properly execute the student application. These nodes must be configured to communicate with one another, often utilizing manually-specified ip addresses and ports. It is possible to run all instances of student code from a single host; however, the algorithm will be less likely to encounter the realities of networked communication over the Internet, including delays, packet loss, and packet corruption or duplication. Once the student assignment is installed, running, and networked on all nodes, the grader must interface with it. Manual testing is often conducted via multiple ssh terminals to provide a sequence of input to the different nodes. It can be difficult to satisfactorily test resilience to simultaneous inputs across many hosts.

1.1 Manual Testing & Grading

Manual grading of distributed applications is typically done by issuing a command, waiting for the system to react and stabilize, and then checking the resulting state of the system. Student code is usually viewed by the instructor as a black box, as even a correct implementation could result in different outputs on subsequent runs due to issues of concurrency and network delay. Take, as an example, manually grading a distributed calendar application, in which no two events may be scheduled at the same time for the same user, but in which any user many enroll another in an activity. First, a command from the instructor might prompt a node in the system to enter an event into the calendar. The instructor, then, must wait while the nodes of the network determine whether the newly entered event is non-conflicting. When the system has stabilized, the instructor can check each node to see whether they agree on the state of the calendar. If all nodes agree on a valid calendar, the student receives credit for this rubric item. If the student's results are invalid, or, worse, one or more of the student nodes crash, the instructor must adapt, restart the grading system, or award the student zero credit on subsequent tests.

1.2 Grading by Live Demonstration

As a result of these difficulties, instructors may opt to schedule in-person code demonstration sessions during which students bring their completed assignments and run them through a series of live testcases. Because the student is presumably well acquainted with their own implementation and interface, they can more rapidly configure and manipulate the system to demonstrate adherence to assignment specifications. The student arrives with the code already configured and running on multiple hosts, allowing the demonstration time to focus on providing test inputs to the system and discussing the output and implementation details. However, even live demos have hiccups with unexpected bugs and the instructor often must think on their feet, as it is not often possible to restart testing altogether. As a result, if live testcase 1 is a failure, the expected output from live testcase 2 must be adjusted. Sometimes, however, failures are so catastrophic that testcase 2 becomes impossible.

1.3 Contributions

Our contributions are as follows:

- We present the design and implementation of a system for automatically running and grading networked, distributed applications within Submitty, an open source course management and autograding system [12].
- The design allows instructors to manipulate network features such as message delay and loss on a per-connection basis, to enforce restrictions regarding which nodes are allowed to communicate, to require that student code uses a particular protocol (e.g., TCP or UDP) to pass messages, and to connect and disconnect nodes from a network.
- Our system allows instructors to interface with and provide input to running student distributed applications in a scripted way, facilitating automated testing of student code.
- The system logs information about all messages and message content passed between the networked nodes. This log can be leveraged to automatically evaluate the efficiency of an implementation and may also be provided to students as a learning aid and debugging tool.
- The module for the generation of container networks and deployment of student code can also be as a standalone tool. This tool may be utilized by students during development and by instructors to facilitate live student demonstrations.

2 RELATED WORK

As enrollment sizes have increased in Computer Science programs, professors have turned to autograding solutions, both commercial [4], and homegrown [15]. The use of such automated grading platforms has been claimed to be an economic [16] and logistical [15] necessity for the professors and institutions that are able to employ them. However, not all programming assignments are a good fit for the traditional autograding scheme of running a single instance of student code, piping it inputs, and then difference checking its output against a known solution. In particular, autograding assignments that require networked instances of student code are not feasible in the majority of autograding solutions.

This apparent lack of support is due at least in part to the difficulties of automatically testing the correctness of distributed applications. Issues of concurrency and sub-optimal network conditions yield outputs that are difficult to debug, as "happens before" relationships can be impossible to establish without a reliable global clock [3]. This issue is exacerbated by the problem of distributed state, which makes log and output analysis a challenging task. To combat issues of concurrency and nondeterministic message orderings, some systems decompose tests into a set of abstract pre- and post- conditions, rather than defining a single, fixed valid output [8]. In such a testcase, a precondition might be that the state of a distributed calendar is inconsistent because a conflicting event has been entered. The postcondition, then, would be that a valid calendar is decided upon by the nodes in the network. Other systems, meanwhile, attempt to sidestep the problem of concurrency entirely by testing distributed applications component by component [14]. In such systems, rather than passively assessing a distributed system as it runs, unit tests are performed by "separating system components from the environment." In contrast, other testing frameworks aim to allow developers to subject their applications to the full wrath of a stressful network. This can be done by integrating a transparent "degrader", which sits between the nodes of a system and intercepts all messages. The degrader acts as a relay, and can modify message content, delay, and drop rates. [7]. Alternatively, a less intrusive approach can be taken to inject delays via a network emulator directly on connections between network nodes without the need for interception [1, 13]. This process, known as "fuzzing," is necessary to tease out unknown errors from a system. Packet loss rates of about 0.1% are expected in realistic conditions — a packet loss rate of 1% is considered problematic and 5% is a "severe failure" [10].

3 SYSTEM REQUIREMENTS

Our system was designed to support RPI's Distributed Systems and Algorithms course, an elective of the Computer Science major taken by juniors, seniors, and graduate students. To fairly, robustly, and efficiently test and evaluate student work, the autograder must:

- **Automatically create a reasonably sized student network.** Manual testing during live demos of typical projects for this course requires as many as eight student nodes. The hardware for the server should therefore be able to run a minimum of eight simultaneous containers.
- **Keep network specification simple but powerful.** Network configuration should have reasonable default values so that instructors can quickly build autograders for simple examples, while remaining customizable to facilitate and control complex test cases.
- **Facilitate scripted interaction via standard input and capture of all program output.** Automated validation and scoring can be performed on captured program output.
- **Support common message passing protocols and a variety of programming languages.** We have implemented and tested support for both TCP and UDP protocols, and students have chosen from Java or Python for their implementations. The Submitty platform supports a comprehensive set of programming languages.

- **Be highly reusable.** The automated creation of networks and deployment of student code is a useful tool independent of the autograding pipeline. This module should be self contained, so that it can be leveraged by students during development and by instructors for additional, unscripted testing and grading.
- **Provide an interface for professors to subject student programs to stressful network conditions.** Subjecting distributed applications to stressful but realistic conditions can be useful in revealing hidden implementation issues [7, 10]. To this end, our system should include tools with which instructors can subject student assignments to network conditions including packet loss, corruption, duplication, and delay.
- **Provide students with meaningful feedback for debugging purposes.** Automated grading is not only about assigning students a numeric grade, it is also an opportunity to help them understand why they did not receive full credit and (if allowed by the course syllabus) to revise and resubmit their projects. Like other autograding platforms, Submitty supports multiple submissions [15] and access to files, standard output, and message logs produced during testing to facilitate debugging of distributed applications [3].

4 IMPLEMENTATION

4.1 Isolating Docker Communications

For the implementation of our system, we focus on leveraging Docker containers [5] — lightweight, secure environments that are virtualized at the operating system level and used to execute applications. Docker containers can effectively and securely execute and autograde student code within many different types of runtime environments [9, 11]. We considered multiple network emulation schemes before settling on this decision, including the use of physical host PCs, virtual machines, emulation schemes involving multiprocessing, and the use of cloud services such as Amazon EC2 instances — virtual servers provided through Amazon's cloud services [2]. The decision to use Docker containers was made based on the following factors:

- **Cost** Many Docker containers can run on a single physical host server, making it far more cost effective than using multiple networked physical PCs. Docker is open-source, and free to use on your own hardware. In contrast, cloud-based container/virtual machine services have a fee per unit of time.
- **Ease of Configuration** Docker containers and networks are simple to configure [6]. In Submitty, we allow instructors to provide DockerFiles so they can easily specify course, assignment, or testcase specific packages.
- **Network Emulation** Docker provides native support for networks with arbitrary topology among containers. We leverage these networks to log all messages passed by the student application.
- **Security and Isolation** Docker containers isolate student code from the host operating system. Docker networking solutions prevent one student's code from communicating with that of another student's being graded in parallel. This represents a real advantage over multiprocessing solutions or physical machines with known hostnames or IP addresses.
- **Ease of Interface** Because all Docker instances are run on a single host server, a wrapper script may maintain handles on all running instances. This makes it simple to attach and pipe standard input to running Docker instances in an immediate fashion without maintaining multiple ssh connections.
- **Performant** Docker containers are quick to spin up – it is possible to create and start a new batch of containers for a given testcase in only a few seconds. Furthermore, Docker containers are more efficient than virtual machines and have a small impact on their host machine's resources. In stress-tests with single threaded, non-networked programs with and without Docker containers, we found that overall CPU and memory usage increased by less than 1% each [9].
- **Scalable** While it falls outside of the scope of our current work, Docker networks are scalable and may be distributed across multiple physical host servers, making it possible to deploy and test swarm applications [17].

4.2 Router: Intercept and Log All Messages

Our goal is to allow instructors to subject student programs to stressful network conditions and to change these conditions over time. Similar to the "degrader" presented in [7], we introduce an additional, invisible node to the network that we call the *router*. The router intercepts and logs all messages, applying rules regarding message delay, loss, and reorder that may be unique for each host-port pair. While the injection of such a node into the student network is more intrusive than other solutions [1, 13], it facilitates other desired features such as allowing students to study, learn, and debug with access to a detailed message log; limiting network communication to specified protocols; and evaluating student code based on volume and patterns of communication. Furthermore, specification of custom network rules is simple within such a model; within our system, the router is implemented in Python and can be customized or replaced by an instructor to delay, drop, reorder, or otherwise directly manipulate student messages. To accommodate assignments that require a less intrusive model, testcases can be run in "routerless" mode, without message logging or manipulation.

Our router node is invisible to the student application — no modifications are necessary to the student code — yet all network communications pass through this router, allowing it to manipulate traffic in any way necessary. If we have two nodes, A and B, and A sends a message A → B, the message, in actuality, follows the path A → router → B. However, neither A nor B are aware that the message passed through the router rather than directly from A to B. To accomplish this we leverage the power of Docker network aliasing:

- Each container in a Docker network may have a unique alias and may be contacted by this alias by any other node in the network.
- While Docker aliases must be unique among the aliases on a network, they need not be unique across all networks.
- A Docker container may simultaneously be connected to multiple networks.

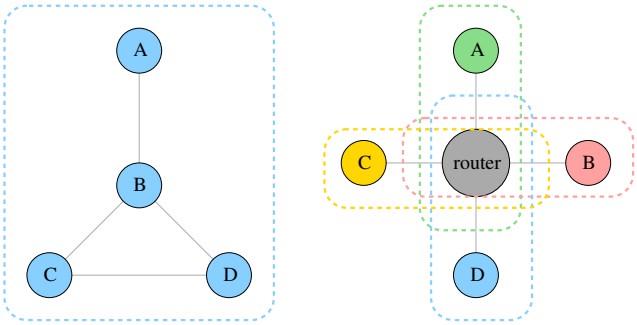

Figure 1: On the left, we illustrate a conceptual network for a student program with four nodes. On the right, we illustrate the actual connectivity using multiple Docker networks and our router.

```
{
  "use_router" : true,
  "containers" : [
    {
      "container_name" : "container0",
      "container_image" : "python:3.6",
      "outgoing_connections" : ["container1"],
      "commands" : ["python3 server.py"]
    },
    {
      "container_name" : "container1",
      "container_image" : "python:3.6",
      "outgoing_connections" : ["container0"],
      "commands" : ["python3 client.py"]
    }
  ]
}
```

Figure 2: Specification of a Docker network for a testcase.

Thus, we create two Docker networks (A, router), and (B, router). However, on the network (A, router), we give the router the alias "B" and A the alias "A_Actual" On the (B, router) network, we give the router the alias "A" and B the alias "B_Actual." Therefore, in the eyes A, the router is B, and in the eyes of B, the router is A. The router, meanwhile, can retrieve the true identities of A and B using the "_Actual" aliases. This use of one network for each node in the system carries with it the added benefit of making it impossible for a student process to interface with containers other than the router. This network architecture is illustrated in Figure 1.

The router can intercept and forward all messages as they move through the system. To facilitate this, the necessary network information is provided to a student process via an automatically generated `knownhosts.txt` file which contains the addresses of hosts on the system and the ports on which they will be listening. The student program must parse this file and use this information to establish the necessary connections. Our design requires that we add the constraint that port assignments are unique across the entirety of the system. This constraint does not represent a significant limitation on the program.

By default, the router will listen for both UDP and TCP connections, and provide students with both UDP and TCP versions of the `knownhosts.txt` file. Upon initialization, our router creates a "switchboard" dictionary, that maps a port to a sender, recipient, and connection type (UDP or TCP). Leveraging this switchboard, it is simple for instructors to modify the router to add rules regarding network stress by recipient, sender, or port.

4.3 Network Configuration

Ease-of-configuration is of prime importance when developing autograding tools for instructors. Simple test cases should leverage default system configurations and require only short, intuitive, and human-readable autograding configuration files. To this end, our system pares network specification down to five simple but powerful parameters (Figure 2). The first of these, "use_router," determines whether or not the router node will be injected into the student network. The instructor may either specify a custom router or the default logging router will be inserted. The "container_name" field allows the instructor to give a container an application-meaningful designation. If unspecified the container names default to "container0", "container1", etc. The container name is used when defining network connections and as the name of the directory where student output is archived. The "container_image" field specifies the Docker container image. If unspecified at the container level within the test case, the system will check for a "container_image" at the global level for this autograding configuration. If unspecified, the system will default to a custom Ubuntu image appropriate for typical introductory Python/C++/Java programming assignments. The "outgoing_connections" field allows the instructor to specify one way channels of communication between containers in the network. If unspecified, this field defaults to all other containers created for this testcase. The combination of the outgoing_connections across all containers for a testcase is used to generate the `knownhosts.txt` file. Finally, the "commands" array is an array of one or more of unix commands to be executed sequentially within the container.

Network configuration JSON objects (Figure 2) are presented to our system embedded in a JSON array. For each testcase, new Docker containers and networks are created. As a result, each new testcase can test different types of containers, network sizes, and network conditions. For each testcase, the output of the run is stored in an individual folder, which is then fed directly into our existing Submitty autograder validation pipeline.

4.4 Methods of Evaluation and Autograding

We provide many methods for instructors to interface with student distributed applications. In some cases, it is enough to merely initialize a student node with some known state, either via command line arguments or through the use of an initialization file. In such cases, the nodes of the student application must be capable of achieving the goal of a testcase without additional instructor intervention. For example, in the distributed calendar assignment, an initial testcase may determine whether the nodes of the student application can reach consensus on startup from initially inconsistent or incomplete state — a task that may be performed without additional instructor prompts or instructions.

In cases where interaction is needed, instructors may provide and deploy their own code within an additional container on a testcase's network. For assignments in which students are asked

```
{
  "dispatcher_actions" :
  [
    {
      "action" : "stdin",
      "containers" : ["container0"],
      "string" : "This will be piped to container0\n"
    },
    {
      "action" : "delay",
      "seconds" : 2
    },
    {
      "action" : "stdin",
      "containers" : ["container1"],
      "string" : "This is piped to container1 2 sec later\n"
    }
  ]
}
```

Figure 3: The specification of a list of dispatcher actions.

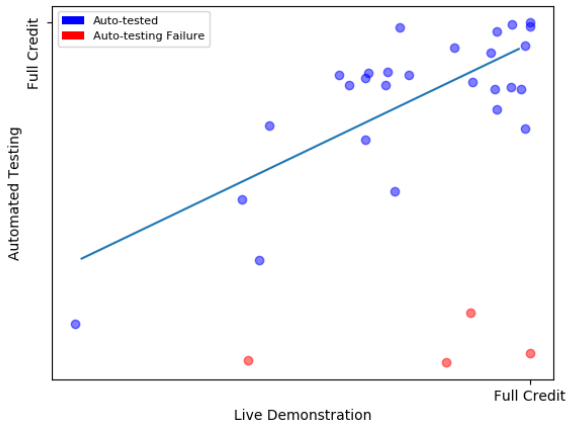

Figure 4: A scatterplot correlating live demonstration scores to automated testing of a Distributed Systems and Algorithms midterm project. 5% noise has been added to all values for anonymity.

to implement only certain portions of a distributed algorithm, this instructor node may represent one or more modules of the whole system. For example, the instructor may write a server node while the student writes a client node (or vice versa). The student's code must then interface with the instructor-written module(s), and then (typically) the standard output of the instructor module can be directly used for validation/autograding of the student code. This type of "object instantiation" [16] can provide students with invaluable experience with complicated systems without asking them to implement every module themselves.

The student may also be asked to implement an entire distributed system — for example a distributed calendar — while the instructor deploys one or more "client" nodes within the network. These instructor written clients perform an automated batch of actions (with appropriate time delays), connecting to different student containers and sending them messages. In the case of the distributed calendar assignment, the client might attempt to schedule an event while connected to one node, then access the event while connected to another. In a more complicated testcase, the instructor may create a client in which two threads of execution connect to student nodes and issue multiple sets of commands concurrently.

Some assignments lend themselves to an even more direct and simple method of communication. To this end, we allow instructors to "dispatch" strings to the standard input of running Docker containers. A sample of the syntax for specifying these actions is shown in Figure 3. Dispatched actions take four forms: delay, standard input, stop, and start actions. Actions are performed sequentially, with no default delay between them. Standard input actions may target one or more containers, allowing professors to quickly and easily broadcast the same message to many student processes. Start and stop actions may be used to connect or disconnect nodes in the student network — a feature which is helpful for testing program resilience to system faults. This method of delivering input to student code is especially useful when testing peer-to-peer applications. For example, in the calendar assignment, each node in the system may be used to represent a user. To specify a testcase on such a network, an instructor may dispatch standard input to multiple nodes, creating or canceling calendar events. If, at the end of these actions, all nodes agree on a valid calendar, the student would receive full credit for the testcase. To test an assignment for fault tolerance, an instructor might choose to stop a student node mid-testcase execution. After more standard input actions are processed, the node can be restarted into a system that has changed since it was last active. After the system has stabilized, the instructor may query the nodes of the network to see if they have reached an acceptable consensus.

5 RESULTS: CASE STUDY

A core component of RPI's Distributed Systems and Algorithms course has been the twice yearly live demonstration of student projects. These demonstrations consist of a one on one, 20 minute code presentation, during which the instructor guides the student through a suite of testcases. This term, limited automated testing with Submitty was used as a prerequisite for participation. Students received the benefit of multiple submissions with testing feedback, and the demonstrations went more smoothly, as the student implementations had undergone preliminary testing.

After the live demonstrations for this term's distributed calendar midterm project, we adapted the set of demonstration testcases to a more complete set of automated tests. The live testcase script included seven tests. These ranged in difficulty from entering an event into the calendar to testing resilience to node failure. The tests were designed by the instructor to provide good coverage of the problem specification.

Testcases for the live demonstration were developed to build upon one another, with the ending state of a testcase presumed to carry into the next. In order to capture this feature, each of our testcases began with the state-changing actions from those prior. This term, projects were submitted by 31 teams of students. Our system successfully tested 27 of these projects automatically. The four remaining submissions had automated testing failures that will require further investigation to debug and prevent (either by making the system more robust, or by providing appropriate

feedback to the instructor and/or student about misunderstandings in the system or assignment specification). Had a complete test suite been available earlier in the project period, allowing early submission and resubmission, we are confident that the students would have successfully adapted their implementations to meet the requirements of automated testing.

The results of all submissions are depicted in Figure 4. All datapoints have been perturbed with 5% noise on each axis to help preserve student anonymity. The (unperturbed) results of these testcases display a Pearson Correlation Coefficient of 0.78, and show that our system is capable of successfully evaluating the challenging testcases from RPI's Distributed Systems and Algorithms course. The four assignments with auto-testing failures shown in red were omitted from the Pearson calculation. Autotesting was performed in parallel, and the complete test suite took fewer than 2 minutes for each student team submission. Much of this time was used for initialization, as student nodes were given five seconds to fully initialize per testcase. This startup time helps to ensure that student code is not given inputs before it is fully operational.

6 DISCUSSION & LIMITATIONS

We have detailed a new framework for automatically running and grading networked, distributed applications within Submitty, an open-source course management and autograding platform. We have presented a simple yet powerful means by which instructors may specify automatically-generated networks of nodes and deploy student code to those nodes. We have introduced the design and implementation of a "router," for logging and manipulating all messages sent through a student network. We have evaluated and implemented a number of methods for interacting with student distributed applications including the use of "object instantiation", an instructor-written client, and directly via standard input. This implementation has been successful in grading RPI's Distributed Systems and Algorithms course.

The open source Submitty course management and autograding system has served CS courses of all levels at RPI since 2014. It has become indispensable as the size of all of our courses has grown in recent years. The success of this extension for autograding projects in the Distributed Systems and Algorithms course has piqued interest among instructors, and our solution will be deployed in our Operating Systems and Network Programming courses in future semesters.

7 FUTURE WORK

Our current router implementation is limited in that it expects every network connection to use a unique port. This means that if a node is expecting connections from two others, it must open two sockets. Investigation is ongoing to create a new version of the default router which will allow many to one communication.

We would like to streamline the process of developing autograding configuration files. Our current syntax is flexible, powerful, and customizable; however, a suite of testcases for an assignment can have duplication between tests that use similar container networks, dispatched commands, and autograding validation. We plan to explore methods of reducing redundancy and to develop a web GUI for autograding configuration.

REFERENCES

[1] K. Alnawasreh, P. Pelliccione, Z. Hao, M. RÃënge, and A. Bertolino. 2017. Online Robustness Testing of Distributed Embedded Systems: An Industrial Approach. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 133–142. https://doi.org/10.1109/ICSE-SEIP.2017.17

[2] Amazon. 2006-2018. Amazon EC2. https://aws.amazon.com/ec2/

[3] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D Ernst. 2016. Debugging Distributed Systems. *Queue* 14, 2, Article 50 (March 2016), 20 pages. https://doi.org/10.1145/2927299.2940294

[4] John DeNero, Sumukh Sridhara, Manuel Pérez-Quiñones, Aatish Nayak, and Ben Leong. 2017. Beyond Autograding: Advances in Student Feedback Platforms. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, New York, NY, USA, 651–652. https://doi.org/10.1145/3017680.3017686

[5] Docker. 2013-2018. https://www.docker.com/

[6] Federico Kereki. 2015. Concerning Containers' Connections: On Docker Networking. *Linux J.* 2015, 254, Article 2 (June 2015). http://dl.acm.org/citation.cfm?id=2807678.2807680

[7] R. LÃijbke, D. Schuster, and A. Schill. 2013. Reproducing Network Conditions for Tests of Large-Scale Distributed Systems. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. 74–77. https://doi.org/10.1109/CCGrid.2013.70

[8] A. Marroquin, D. Gonzalez, and S. Maag. 2015. Testing distributed systems with test cases dependencies architecture. In *2015 7th IEEE Latin-American Conference on Communications (LATINCOM)*. 1–6. https://doi.org/10.1109/LATINCOM.2015.7430116

[9] Matthew Peveler, Evan Maicus, and Barbara Cutler. 2019. Comparing Jailed Sandboxes vs Containers within an Autograding System. In *Proceedings of the 2019 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, USA.

[10] A. Rusan and R. Vasiu. 2015. Emulation of backhaul packet loss on the LTE S1-U interface and impact on end user throughput. In *2015 IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*. 529–536. https://doi.org/10.1109/ICCP.2015.7312715

[11] František Špaček, Radomír Sohlich, and Tomáš Dulík. 2015. Docker as Platform for Assignments Evaluation. *Procedia Engineering* 100 (2015), 1665–1671.

[12] Submitty. 2014-2018. http://www.submitty.org/

[13] TERRA NULLIS. 2018. Pumba - Chaos Testing for Docker. https://alexei-led.github.io/post/pumba_docker_chaos_testing/

[14] C. Torens and L. Ebrecht. 2010. RemoteTest: A Framework for Testing Distributed Systems. In *2010 Fifth International Conference on Software Engineering Advances*. 441–446. https://doi.org/10.1109/ICSEA.2010.75

[15] Chris Wilcox. 2015. The Role of Automation in Undergraduate Computer Science Education. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, New York, NY, USA, 90–95. https://doi.org/10.1145/2676723.2677226

[16] Chris Wilcox. 2016. Testing Strategies for the Automated Grading of Student Programs. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. ACM, New York, NY, USA, 437–442. https://doi.org/10.1145/2839509.2844616

[17] H. Zeng, B. Wang, W. Deng, and W. Zhang. 2017. Measurement and Evaluation for Docker Container Networking. In *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*. 105–108. https://doi.org/10.1109/CyberC.2017.78