Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

2016

A Testing Platform for Teaching Secure Distributed Systems Programming

Endadul Hoque
Northeastern University and Purdue University, mhoque@purdue.edu

Hyojeong Lee
Google

Charles Killian
Google and Purdue University, ckillian@cs.purdue.edu

Cristina Nita-Rotaru
Northeastern University and Purdue University, c.nitarotaru@neu.edu

Report Number:
16-002

Hoque, Endadul; Lee, Hyojeong; Killian, Charles; and Nita-Rotaru, Cristina, "A Testing Platform for Teaching Secure Distributed Systems Programming" (2016). *Department of Computer Science Technical Reports.* Paper 1779.
https://docs.lib.purdue.edu/cstech/1779

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

A Testing Platform for Teaching Secure Distributed Systems Programming

Endadul Hoque[†‡], Hyojeong Lee[§], Charles Killian[†§], and Cristina Nita-Rotaru[†‡]
[‡]*Northeastern University*
[†]*Purdue University*
[§]*Google, Inc*
{mhoque, hyojlee, ckillian, crisn}@cs.purdue.edu

Abstract

In this paper, we report on our experience with transitioning a research platform for performing adversarial testing on distributed system implementations into a tool for teaching students how to implement robust distributed systems. We present how we integrated the tool in a graduate-level distributed systems course by describing the modifications we made to the tool, the projects used in conjunction with the tool, as well as the activities performed by the instructor and the students. We evaluated the effectiveness of the tool through multiple surveys conducted throughout the class and reported the results.

Keywords— Distributed Systems; Secure Programming; Computer Science Education; Robustness; Virtualization; Testing

1 Introduction

One of the primary goals of distributed systems is to provide increased availability and performance for their users. However, the increasing complexity of distributed systems, limited availability of testing tools, and inadequate skills of developers often result in implementations that prevent distributed systems from achieving their design goals in practice.

As instructors of distributed systems courses, we recognize the need to teach students the necessary skill-sets to develop robust systems. While mastering the foundation of distributed algorithms is essential in creating such systems, there are numerous aspects of distributed systems that are better learned by experiencing their design and implementation. Numerous distributed algorithms use models that do not specify the necessary details that are critical for the system. In addition, many systems consist of different algorithms running in parallel which can not be trivially integrated. As a result, many decisions are made during the implementation phase.

Several efforts were made over the years to improve distributed systems education. Many focused on creating tools that help visualize the execution of distributed algorithms [3, 4, 7, 21]. While such tools are instrumental in helping students understand the underlying dynamic behaviors of the entities involved in the algorithms, they do not provide effective means for students to test and debug their own implementations. Recently, some tools [11, 18, 20] are created to help the debugging and testing process in a course setting. MDAT [18] offers automated testing of multi-threaded programs where test results are reproducible. However, it is not tailored for message-passing distributed systems, nor does it support any testing to evaluate the robustness of the code. VDE [11] targets computer networking and provides students a framework leveraging a network of virtual machines to test their developed network protocols. However, VDE lacks any automated support to drive such testing and provides support for a limited set of testing scenarios as it is not focused on distributed systems.

In this paper, we report on our experience with transitioning a research platform developed for adversarial testing of distributed systems into a tool for teaching students how to implement robust message-passing distributed systems. We describe how we integrated the tool in a graduate-level distributed systems course.

To measure the effectiveness of the tool, we conducted multiple student surveys. We reported our evaluation results.

Unlike other testing environments, our tool allows students to run *unmodified binaries* of the system under development in their native operating systems while emulating the network conditions. This not only offers the experience of real world asynchrony but also enables the reproducibility of the test outcomes. Moreover, the network emulation limits the impact of external noise and interference on the test results as compared to testing on a network of computers connected through real network infrastructures (e.g., routers). Both students and instructors can test the same unmodified binary under identical conditions, without implementing the code for the test case scenarios. The testing scenarios are mainly focused on the messages pertaining to the system implementation under test. These scenarios combine manipulations on the delivery and on the content of the messages. As a result, our platform[1] even allows testing of Byzantine resilient protocols such as the Byzantine Generals Problem [17], which are designed to tolerate Byzantine behaviors of the participating node(s). Finally, the design of our platform combining virtualization with an emulated network provides a cost effective approach as compared to the infrastructure costs associated with any testbed of similar scale.

Our tool also has several benefits for the instructor. Firstly, it requires very little effort from the instructor to setup the environment for a new project. The instructor needs to provide only a description of the format of the messages pertaining to the protocol of interest. Secondly, it not only alleviates the burden of the instructor by automating the generation of various testing scenarios but also provides enough flexibility for creating new testing scenarios with specific requirement. Finally, the instructor can leverage our tool for performing automatic evaluation of students' submission in a scalable fashion for a large-size class.

The rest of the article is organized as follows. In §2, we first describe the overview of the platform along with the modifications we made to use it in a class setting. We next present how we integrated the tool in the class along with a concrete example in §3. We present results from the surveys conducted in the class as well as the lessons we learned in §4 and §5, respectively. We review related work in §6 and finally, we conclude in §7.

2 Platform Overview

In this section, we first overview Turret [13, 19], the research platform we transition as an educational tool, and then describe how we adapted it to use it in a distributed systems class.

2.1 Turret Overview

Turret is a platform (shown in Fig. 1) that allows adversarial testing of unmodified distributed system binaries running in a realistic environment. Turret leverages virtualization to run arbitrary operating systems (e.g., Linux, Windows) and applications, and utilizes network emulation to connect these virtualized hosts through a realistic network setting. The platform is automated and capable of reproducing network conditions while isolating from outside world interference.

Virtualization and network emulation. For virtualization, Turret specifically uses KVM [12] virtualization techniques. This allows Turret to have several virtual machines (VMs) running on the same physical host whereas each VM acts as an individual node of the distributed system of interest. Each of the VMs can run an application and communicate through the emulated network created using the NS-3 network simulator [1]. Specifically, each VM is mapped to a node inside NS-3, called a *shadow node*, through a *Tap Bridge connection* (available in NS-3), which connects the inputs and outputs of the network device of the NS-3 node to the inputs and outputs of the VM's network interface as if the NS-3 network device is a local device to the VM.

Controller. The controller module of Turret bootstraps the system by starting NS-3 and running application binaries (i.e., the target distributed system implementation) inside the virtual machines. The controller lets each shadow node know if it will act as a benign (i.e., correct) node or as a malicious node. A shadow

[1]We use *platform* and *tool* interchangeably

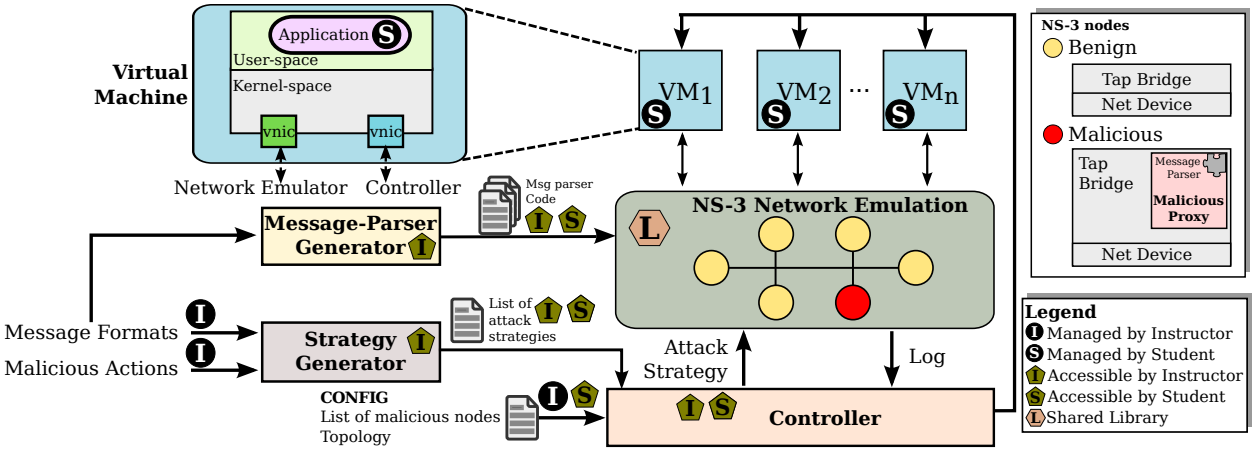

Figure 1: Turret Platform (VNIC: Virtual Network Interface Card)

node instructed to act as malicious will activate the *testing proxy*, a component implemented by Turret on top of the Tap Bridge layer of the NS-3 shadow node. This proxy intercepts messages generated by the application running inside the virtual machines and modifies them according to an *attack strategy*[2]. An attack strategy may consist of two types of malicious actions: *Message Delivery Actions* that affect when and where a message is delivered (see Table 1) and *Message Lying Actions* that affect the contents of a message (see Table 2). To function automatically, the testing proxy requires the user to provide it with a description of the format of the messages that the protocol[3] (or the system) relies on. We will give an example of a message format later in § 3.3.

Message parser generator. The testing proxy requires to parse messages in order to inject a malicious action based on the type of the message. The message parser generator reads a message format description and generates necessary source code containing a set of API calls (e.g., `getMsgType()`, `getMsgSize()`) that exposes various content of the message to the testing proxy.

Table 1: Message delivery actions in Turret

Action	Action Description	Parameter
Drop	Drops a message	Drop probability
Delaying	Injects a delay before it sends a message	Delay amount
Duplicating	Sends the same message several times instead of sending only one copy	Number of duplicated copies
Diverting	Sends the message to a random node instead of its intended destination	None

Table 2: Message lying actions in Turret

Action	Action Description	Parameter
LieValue	Changes the value of the field with a specified value	The new value
LieAdd	Adds some amount to the value of the field	The amount to add
LieSub	Subtracts some amount from the value of the field	The amount to subtract
LieMult	Multiplies some amount to the value of the field	The amount to multiply
LieRandom	Modifies the value with a random value in the valid range of the type of the field	None

[2]An attack strategy is considered as a test case
[3]We use *system* and *protocol* interchangeably to refer to the distributed system of interest

Strategy generator. The strategy generator is responsible for generating a list of attack strategies that a protocol of interest should be tested against. This list of strategies are generated based on the malicious actions given in Tables 1 and 2 along with a value of the parameter(s) that decides the severity of the action. We will give some examples later in § 3.3.

2.2 Adaptating Turret for the Class

The first task we focused on was to enable running multiple instances of Turret simultaneously. We tailor Turret so that resources such as devices, addresses, and ports are configurable and sharable. The second task we focused on was to make Turret easier to use. Specifically, we added two scripts to automate the configuration. One is related to the management and configuration of the VMs, and the other one drives the controller module of Turret. In addition to these scripts, we also prepared a tutorial on how to use Turret as a testing platform. Finally, we altered the design of Turret to allow students to use their own message API class generated by the message parser generator with the other pre-compiled Turret objects shared among multiple instances of Turret.

3 Using Turret in the Class

We describe how we integrated Turret with a graduate-level distributed systems course offered in Spring 2013 and Fall 2014.

3.1 Class Structure

Students had to use this tool while working on the programming projects assigned in this course. Students were assigned 3 projects which helped them learn the core concepts of distributed systems. The projects were carefully designed where the complexity of the assignments would increase gradually. Considering the complexity of the projects, students were given 3∼4 weeks to work on each project.

Project 1 - Byzantine agreement. The first assigned project was to implement a Byzantine consensus protocol, one of the basic concepts in reliable distributed systems. Specifically, students had to implement the authenticated byzantine agreement protocol to solve the classic Byzantine Generals' problem [17]. Since it was the first assignment, the objectives of this project were mostly centered around helping students achieve a first-hand experience on distributed systems programming. While developing the protocol, students were required to consider a synchronous communication channel that the protocol was proposed for. In theory, a synchronous channel is typically assumed for ease and simple characteristics. However, implementing a synchronous communication channel on top of an asynchronous communication channel such as the Internet is quite challenging. Dealing with such a challenging issue not only offered students a better understanding of the practical system implementation but also prepared them for the complexity of the subsequent projects.

Project 2 - Reliable multicast. Moving on to the next level, students were assigned the second project where they had to implement the *reliable total order multicast* protocol [5], which is also included in the ISIS toolkit [2]. This protocol offers a reliable multicast service between a set of servers where each server delivers the multicast messages (to the application) in the same order. The objectives of this project include achieving a deeper understanding of reliable multicasts by developing a practical multicast service on top of an asynchronous communication channel. In fact, this was the project that offered students to work with a protocol that was targeted to deal with the asynchrony of the real communication channel.

Project 3 - Paxos. Of all, the third assigned project was the most difficult in terms of not only the number of lines of code to be written but also the complexity involved. The goal of the project was to implement the *Paxos replication protocol*, a state machine replication protocol built on the Paxos consensus protocol [15, 16]. We asked students to follow the description of the Paxos protocol from [14] targeted to the development of the Paxos replication protocol by comprehending the subtle complexities of the original protocol. Implementing such a complex system where multiple sub-protocols run in parallel offered the

students to experience a tiny flavor of real-world large distributed systems used by tech companies (e.g., Google, Facebook) where a lot of services are backed by replication services like Paxos.

Final project. While the tool was not required for the testing of the final project, two teams chose to use Turret.

Access to Turret. We granted students access to Turret since the release of the first project assignment. Students were free to use the tool while implementing their projects. In addition to providing a tutorial for Turret, we arranged a demo session for the students to demonstrate how to get started with this tool. For each project, students were provided a sample list of test cases with the expected outcome for testing their projects prior to submission except for the first project where we asked students to submit their code in two phases. In phase 1, their submitted code was tested using a set of test cases. We provided them the feedback containing the test results so that they could fix their code, if necessary, prior to submitting in phase 2. We used an additional set of test cases in phase 2. We graded the first project such that a student would lose half the score assigned to a test case if her code fails the test case in phase 1 but passes in phase 2. However, in case of other projects, we required students to submit once and graded using a set of test cases including the sample provided to the students.

3.2 Setup of the Testing Environment

Turret provides each student enrolled in the class with her own testing environment. To do so, we granted each student access to a multi-core Intel(R) Xeon(R) CPU E52690 @2.90GHz server machine equipped with 200 GB RAM and running Gentoo Linux 2.2. Each student was assigned 5 VMs where each VM is equipped with 128 MB RAM. We used Ubuntu 11.10 server with Linux Kernel 3.0 as the guest OS for the VMs.

At the beginning of the semester, we provided each student with a package `Turret-User` containing the necessary code and scripts to create and configure her own testing environment. To setup the testing environment, each student needs to create her own VM instances utilizing the provided VM-related script. It involves generation of fresh VM copy-on-write disk images from a shared base VM image (to reduce the space overhead), instantiation and preparation of the fresh VMs, and creation of tap-devices to be equipped as virtual network interfaces to these VMs. The preparation of a VM specifically includes configuring the network interfaces of the VMs, enabling public key-based authentication for easy and better user authentication to the VMs, and managing static ARP entries to avoid ARP requests during the testing session. Once prepared, students only need to start/stop the VMs as per requirement. Students were free to use the same set of VM images for different projects in this class or they can create and prepare new VM images using the provided scripts whenever necessary.

3.3 Example: Testing Project-1 with Turret

We now demonstrate the usage of Turret with an example, *the Byzantine generals problem* [17]. Specifically, the problem is defined as given a set of generals (one commander and the others are lieutenants) capable of communicating only through messages, the loyal (i.e., correct) generals need to come to an agreement regarding the battle plan (i.e., either *attack* or *retreat*) in the presence of traitor generals. In reliable computer systems, an analogy of this problem can be drawn as the problem of defending against Byzantine failures of components of the system by guaranteeing that the correct components continue their services even if some components (less than majority) exhibit Byzantine failures. One of the solutions to the Byzantine generals problem presented in [17] involves authenticated messages, which means that each message is signed by the sender of the message.

Message format and parsing. Turret requires a message format description to generate and equip the testing proxy with necessary code enabling malicious injections during the experiment. The following example shows the structure of a signed message used by the Byzantine generals to exchange the order (i.e., the battle plan) between them:

```
...
SignedMessage {
  uint32_t type = 1;  // Message type
  uint32_t total_sigs; // Total # of signatures on the message
  uint32_t order; // The order (retreat = 0 and attack = 1)
  struct sig *sigs; // Point to array of total_sigs signatures
}

struct sig {
  uint32_t id; // The identifier of the signer
  uint8_t signature[256]; // Using a RSA private key of 2048 bits
};
...
```

The message parser takes this message format description and generates the necessary code containing a set of API calls. The student then creates an executable binary by combining the message API code, a NS-3 script to configure the emulated network, and the shared libraries. Note that each student creates this executable under her own testing environment.

Traitor mode of generals. A *traitor* general can exhibit Byzantine failures. Students were given the following test case scenarios where a traitor general can:

- Remain silent.
- Delay the sending of a message.
- Send a message to a random set of lieutenants.
- Flip the order of a message.
- Forge a randomly selected signature from the message.

Since each message is digitally signed, any modification to the message content made by a traitor lieutenant will be detected and thus any loyal lieutenant will discard such messages. However, one interesting case about the conflicting orders is worth noting here. When the commander is a traitor, it can send `attack` to a random set of lieutenants and the conflicting order, *i.e.*, `retreat`, to the remaining lieutenants. Both the orders are valid in this case because of the valid digital signature on each message.

Note that students do not have to implement these malicious behaviors of the traitor generals. The testing proxy of Turret includes the necessary code for such malicious actions. This demonstrates one of the benefits of Turret requiring less effort both from the instructors and from the students.

Generating attack strategies. A list of attack strategies is generated by the strategy generator module of Turret that takes in the message format description. For example, consider the following two strategies where the testing proxy is being instructed to drop 90% of "SignedMessage" packets and lie about the order (i.e., 3rd field in the structure) of the "SignedMessage" packets by setting it to `retreat` (i.e., 0).

```
Drop SignedMessage 90
LieValue SignedMessage 2 0
```

Though Turret comes with the script to automatically generate this list of strategies, we provided the students with a sample list of strategies they used to test their implementations prior to submission.

Running test cases. To start testing the code, a student first uploads her project source code to the necessary number of VMs (assuming that the VMs of the student are already running) allocated to her. The student then compiles her project on each VM. Students were encouraged to use `scp` program to upload and `make` to compile their projects. `Turret-User` package contains the controller module, which is a *perl* script. The student executes this script that takes in a list of traitor generals and a list of attack strategies, starts the network emulator on the host machine and the student's program on each of her VMs for each attack strategy, and collects and stores the log messages produced by both the network emulator and the student's program. Moreover, this script contains numerous command line options to support various requirements of the user.

Collecting and combining results. Two types of log messages are generated while testing with Turret: one is generated by the NS3 emulator and the other is produced by the user's program running on each VM. The first log captures messages at the emulation layer and network layer inside NS3, which helps troubleshooting the emulated network. The other log captures the output of the user program, which may contain some error messages as well. When the student starts the testing phase, the controller collects these two types of messages for each test case and stores in separate locations. Later, the student uses another script called *aggregator* that combines the result of each test case and reports to the student for her analysis. For example, consider the following output as the combined result where the traitor commander is running on VM1 performing the `LieValue SignedMessage 2 0` test scenario (attack strategy), and the three loyal lieutenants are running on VM2, VM3, and VM4.

```
#====================================#
 Strategy: LieValue SignedMessage 2 0
#====================================#
VM2: Lieutenant 1 : Agreed on retreat
VM3: Lieutenant 2 : Agreed on retreat
VM4: Lieutenant 3 : Agreed on retreat
```

Since in this attack the traitor commander (i.e., the testing proxy on the shadow node associated with the commander) lies about the battle plan by setting it to `retreat` even though the commander has sent `attack` as the order, all the loyal lieutenants must agree on retreat according to the protocol. Therefore, if the student's implementation produces the identical result, we can conclude that her implementation pass this test case.

4 Results

We performed an evaluation to quantify the degree to which this course enhanced students' learning experience by mostly focusing on their understanding of the fundamental concepts and their hands-on experience with the projects. To do so, after each project, we collected students' responses through *anonymous, IRB-approved surveys* containing a variety of questions regarding their experience with the project as well as with the tool (Turret). The participation in surveys was voluntary, but we observed that on average 66% of the total students completed the surveys. We measured the effectiveness of Turret integrated with this course by analyzing the responses from the students to the following survey questions:

- Q1: Was the time you spent on the lab worthwhile?
- Q2: Was the automated framework (Turret) effectively helpful in finding bugs in your code?
- Q3: Was the time you spent on learning and using Turret worthwhile?
- Q4: Did you attain the learning objectives of the lab?
- Q5: As a result of the lab, are you more interested in Distributed Systems?

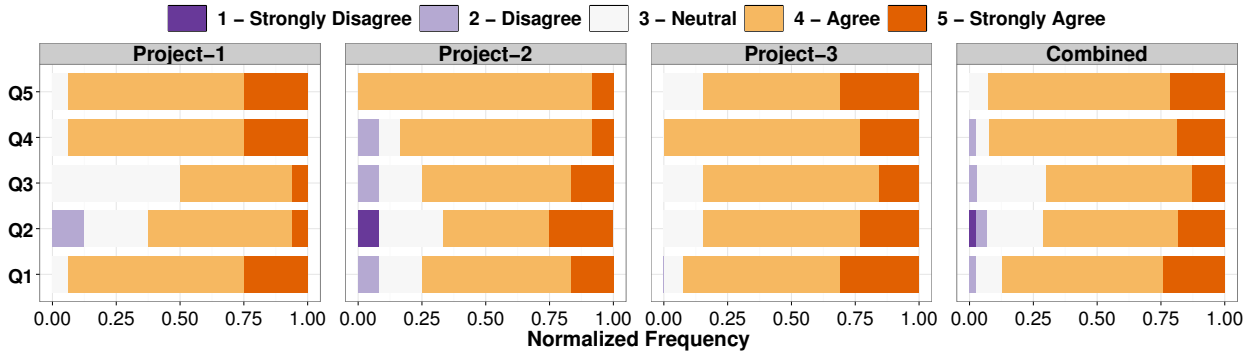

Figure 2: The summary of the students' responses to the survey questions

Fig. 2 shows the received feedback for each project along with the combined feedback. The graphs show the increase of positive reviews from project 1 to project 3 as students became more familiar with the tool. Using a Likert scale ranging from 1 (strongly disagree) to 5 (strongly agree), we quantitatively measured the average response score of the feedbacks for each question. The score of the responses to Q1 is 4.18 after project 1 and increases to 4.23 after project 3 (with a combined score of 4.0) indicating the time students spent on the lab was worthwhile. Similarly, students' found the tool very useful in testing their code as the response-score of Q2 starts with 3.5 and ends up on 4.1 after project 3 with a combined score of 3.79. The question Q3 that is whether learning the tool was worthy at all receives the similar response-score like Q2 as it starts with 3.5 and ends up on 4.0 after project 3 with a combined score of 3.79. The question Q4 receives a score of 4.14 when we combine the scores from all the projects indicating that the students' confidence on the concept. Furthermore, the project assignments help increase the students' interest in distributed systems as the combined response-score of the question Q5 is 4.0.

In addition to the positive feedback, we also noticed some negative experience, especially after project 2 (see Fig. 2). We attribute this to the problems that some students faced with the Turret due to its limitations. In the next section, we discuss these aspects based on our observations in the class and on additional comments provided by students in the surveys.

5 Lessons Learned

There are several lessons we learned from this experience with respect to limitations of the tool and its use in the class.

Provide better network diagnostic reports. One of our pedagogical goals was to have students experience developing robust distributed systems by utilizing an automated testing tool such as Turret. After the first project, several students learned a very pointed lesson of local testing and defensive programming such as handling corner cases, input validation. However, the inherent nature of distributed systems causes some bugs to occur after a particular sequence of events. Such bugs are only reproducible during concrete executions. While testing with Turret, students also encountered such bugs that helped them realize the importance of the full-fledge testing of their code by utilizing a tool like Turret. To help diagnose such bugs, the network emulation layer of Turret provides a detailed trace of all packets exchanged between the VMs. However, some students found this detailed trace not easy to follow and therefore failed to leverage the benefit of the trace. We plan to address this issue by integrating a much simpler diagnostic report with adequate tracing information to help students—even with little background on computer networks—debug their code.

Detect congestion at the network emulation. While the network emulation provides a realistic, but controlled, network for the virtual machines to connect and communicate, the underlying buffer of the NS-3 Tap-Bridge connection can become congested if a student's code sends an uncontrolled number of packets

resulting in a denial-of-service attack to the user application under test. We observed such incidents in case of one of the projects where the code was flooding the network with packets. However, this was not the case when those submissions were tested on an actual network of computers, which has a higher tolerance to denial of service. We plan to address this issue by adding some protection mechanisms that will detect such cases and provide corresponding feedback. As a preliminary measure, we already updated the project descriptions by including a clear specification of what is allowed in the protocol design.

Detect incorrect VM setups. Despite the tutorial and a demo session on Turret, a small number of students had problems with correctly setting up their VMs. This consequently prevented them from testing their code before the issue was fixed. As a precautionary measure, we plan to include scripts that will allow diagnosis and detection of incorrect VM setups.

Use of print functions. Many delays in debugging were caused by misunderstandings on how `printf` function in C works in case of program crashing. We fixed this problem by adding a simple instructions in the tutorial and in the project handouts.

Using the same configurations for developing and testing. Despite emphasizing that the configuration used by the VMs we provide should be used during development, some students had their preferred development setups. This resulted in a few situations where latent bugs such as unsafe memory access (e.g., buffer overflow) did not occur in the development environment, but occurred in the testing environment. We plan to include examples from what we learned from these semesters to emphasize the importance of using the same environment for both the development and the testing.

Allowing students to fix their mistakes. Finally, regarding the way students were allowed to use the feedback from the tool, we learned that students overwhelmingly preferred fixing their mistakes. Recall that we took a different approach in case of the first project assignment where we performed the testing on students' code in phase 1 and provided them with what is needed to be fixed to pass the test case in phase 2. For the last two projects, we had students test their code by themselves prior to submissions. Almost everyone preferred the first approach as they believed they learned more from fixing their code while losing only half the score for each failed test case. We plan to continue using the first approach for the other projects in future.

6 Related Work

In this section, we only focus on the relevant works from an educational perspective. However, for a detailed comparison between Turret and other research tools, we refer interesting readers to [13, 19].

From an educational perspective, several efforts were made over the years towards the enrichment of distributed systems education. Tools like [3, 4, 7, 21] focus on the visualization of the execution of distributed algorithms to help students better understand the underlying dynamic behavior, the internal mechanism, and to provide them with some vital information which may be helpful for them in debugging their implementation. Another avenue of work directly focuses on improving the debugging and the testing of students' code in a course setting. MDAT [18] targets to multi-threaded programming assists debugging by instrumenting the students' program to fully control the scheduling of threads and thus reproduces a failed execution of the program. By employing a specially designed unit testing framework, a test-first approach of writing concurrent programs in Java is proposed in [20]. In contrast, our platform facilitates the robustness testing of the unmodified implementations of distributed systems developed by students, even at the presence of malicious attacks injected to the messages in transit. Moreover, our testing platform can support testing of any message-passing distributed protocol.

The emergence of cheap and effective virtualization makes it to be adopted in computer science education for pedagogical purposes [8–11, 22]. Among them, most closely related is VDE [11] that provides a framework for students to test their developed network protocols, network security tools by creating a network of virtual machines connected through the emulated data-link layer over a vde switch. However, unlike our tool, VDE does not support any automated testing to evaluate the robustness of the implementation, let alone any

adversarial testing. Seattle [6], a community-based effort to set up a platform environment, offers a cloud in the educational environment that promoted resource usage. Our tool, on the other hand, offers an inexpensive and private platform for testing robustness of distributed systems implementations. Moreover, unlike Seattle, our tool is not dependent on any specific language.

7 Conclusion

In this paper, we reported our experience with transitioning a research tool as an educational tool in a class setting in order to teach students how to develop robust distributed systems. In addition to describing how we integrated such tool in a graduate-level distributed systems course, we presented the evaluation of the tool measured through multiple surveys focusing on how such integration was received among students. The survey results and our anecdotal experiences indicate that the students felt the overall benefit of the tool. We learned a few lessons about changes that we need to make to the tool, to the tutorial and documentation provided, and to the project assigned in class. We plan to continue using the tool in distributed systems classes and integrate with other courses such as security and computer networks. We also plan to develop a distributed systems teaching module using this tool, which can be used by students and instructors at other institutions.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation under Grant Number CNS-1223834. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Network Simulator 3. `http://www.nsnam.org/`.

[2] The ISIS project. `http://www.cs.cornell.edu/info/projects/isis/`.

[3] M. Bedy, S. Carr, X. Huang, and C.-K. Shene. A visualization system for multithreaded programming. In *SIGCSE '00*, pages 1–5. ACM, 2000.

[4] M. Ben-Ari. Interactive execution of distributed algorithms. *Journal on Educational Resources in Computing (JERIC)*, 1(2es):2, 2001.

[5] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

[6] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Seattle: a platform for educational cloud computing. In *SIGCSE'09*, pages 111–115. ACM, 2009.

[7] S. Carr, C. Fang, T. Jozwowski, J. Mayo, and C.-K. Shene. Concurrent mentor: A visualization system for distributed programming education. In *PDPTA*, pages 1676–1682, 2003.

[8] M. Casado and N. McKeown. The virtual network system. In *SIGCSE '05*, pages 76–80. ACM, 2005.

[9] D. Dobrilovic and Z. Stojanov. Using virtualization software in operating systems course. In *ITRE'06*, pages 222–226. IEEE, 2006.

[10] A. Gaspar, S. Langevin, W. Armitage, R. Sekar, and T. Daniels. The role of virtualization in computing education. In *SIGCSE'08*, pages 131–132. ACM, 2008.

[11] M. Goldweber and R. Davoli. VDE: an emulation environment for supporting computer networking courses. In *ITiCSE '08*, pages 138–142. ACM, 2008.

[12] I. Habib. Virtualization with kvm. *Linux Journal*, 2008.

[13] M. E. Hoque, H. Lee, R. Potharaju, C. E. Killian, and C. Nita-Rotaru. Adversarial testing of wireless routing implementations. In *WiSec*, pages 143–148. ACM, 2013.

[14] J. Kirsch and Y. Amir. Paxos for system builders. *Dept. of CS, Johns Hopkins University, Tech. Rep*, 2008.

[15] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[16] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[17] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[18] E. Larson and R. Palting. Mdat: a multithreading debugging and testing tool. In *SigCSE '13*, pages 403–408. ACM, 2013.

[19] H. Lee, J. Seibert, E. Hoque, C. Killian, and C. Nita-Rotaru. Turret: A platform for automated attack finding in unmodified distributed system implementations. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2014.

[20] M. Ricken and R. Cartwright. Test-first java concurrency for the classroom. In *SigCSE '10*, pages 219–223. ACM, 2010.

[21] W. Schreiner. A java toolkit for teaching distributed algorithms. In *ITiCSE '02*, pages 111–115. ACM, 2002.

[22] E. Shoop, R. Brown, E. Biggers, M. Kane, D. Lin, and M. Warner. Virtual clusters for parallel and distributed education. In *SIGCSE'12*, pages 517–522. ACM, 2012.