Experience with Seattle: A Community Platform for Research and Education

Yanyan Zhuang
University of Victoria, NYU-Poly
yyzhuang@cs.uvic.ca

Albert Rafetseder
University of Vienna
albert.rafetseder@univie.ac.at

Justin Cappos
NYU-Poly
jcappos@poly.edu

***Abstract*—Hands-on experience is a critical part of research and education. Today's distributed testbeds fulfill that need for many students studying networking, distributed systems, cloud computing, security, operating systems, and similar topics. In this work, we discuss one such testbed, Seattle. Seattle is an open research and educational testbed that utilizes computational resources provided by end users on their existing devices. Unlike most other platforms, resources are not dedicated to the platform which allows a greater degree of network diversity and realism at the cost of programmability. Seattle is designed to preserve user security and to minimally impact application performance. We describe the architectural design of Seattle, and summarize our experiences with Seattle over the past few years as *both* researchers and educators.**

We have found that Seattle is very easy to adopt due to cross-platform support, and is also surprisingly easy for students to use. While there are programmability limitations, it is possible to construct complex applications integrated with real devices, networks, and users with Seattle as a core component. From an educational standpoint, Seattle has been shown not only to be useful as a teaching tool, it has been successful in variety of different systems classes at a variety of different types of schools. In our experience, when low-level programmability is not the main requirement, Seattle can supersede many existing testbeds for diverse educational and research tasks.

***Keywords*-Distributed Testbed, End-User Machines, Experimental Facilities, Educational Use**

I. Introduction

Distributed platforms are now a de facto standard in modern software and application development, as well as education in computer science. Real world distributed testbeds can provide a programming infrastructure that typically does not exist in a laboratory environment. To date, existing testbeds such as PlanetLab [1], Emulab [2], and various GENI efforts [3] have played critical roles for evaluating networked and distributed systems. These testbeds, however, are composed of dedicated resources. As a result, they are expensive to scale as the management and equipment costs dominate. Using dedicated hardware also makes it difficult for a testbed to be representative of the Internet without constantly purchasing and adding the latest hardware on diverse end user networks. As such, testbeds composed of dedicated hardware do not have representative connectivity, processing power, and churn behavior of existing Internet hosts. Furthermore, due to resource contention, researcher and instructor must undergo an approval process and wait for actions by multiple parties. This has hampered experiment deployment and educational adoption. With the growing end-user diversity, heterogeneous network connectivity, and increasing interest in sensor data from end user devices like smartphones, traditional platforms leave much to be desired.

Over the past four years, we have been using a testbed constructed with a different approach, namely, the Seattle testbed [4]. Seattle utilizes computational resources provided by end users on their existing devices. The growth and success of this testbed is enabled via end user participation. It is designed to incentivize participation while minimizing the risk to end users. To this end, Seattle is able to preserve user security and to minimally impact the performance of the user's other applications. The testbed provides researchers and educators with the ability to create and evaluate enticing prototypes that span a wide range of devices, from servers, desktop PCs and laptops, to smartphones and tablets. Despite the security restrictions and programmability limitations that are necessary to make the platform safe for end users, our experience indicates that it is an easy to use and powerful platform for both education and research. The development and direction of Seattle follows a common model in the open source community, which we call a *community platform*. Hence, Seattle embraces the heterogeneity of today's end user environment, and provides a unique environment that is not available on other testbeds.

In this work, we provide our experiences with Seattle platform from *both* research and educational perspectives. For researchers, Seattle allows the research community to answer fundamental research questions using real end-user machines, and support devices behind NATs, wireless routers, firewalls, and mobile platforms. For educators, Seattle provides a usable and safe programming platform that is accessible to students in many security, operating systems, distributed systems and networking courses. Despite certain challenges and limitations, we believe that Seattle will continue to have a unique, positive impact in the educational and research communities.

The rest of this paper is organized as follows. Section II describes the architecture of Seattle, and its current size and scale. The experiences with Seattle platform by a broad range of researchers and educators are presented in Sections III and IV. A summary of our on-going work to overcome some of Seattle's current limitations is provided in Section V, followed by an overview of related work in Section VI. Section VII concludes this paper with our future vision of the Seattle testbed, as we expand it to an even larger scale.

II. SEATTLE: AN OPEN COMMUNITY PLATFORM

This section describes the sufficiently general testbed architecture of Seattle that can support many research and educational use cases.

A. System Goals

The creators of Seattle testbed [4] have set out five high-level design goals which guide the core development of the testbed. These goals are listed roughly in order of priority from highest to lowest:

1) Safety: The end users that participate in the testbed should not face significant risk. Code should be strictly sandboxed. Code executed on the testbed must not interfere with the performance or correctness of user's applications.

2) Open Development: The testbed must be constructed out of loosely integrated components. To the extent that is feasible, components should not rely on each other and should be replaceable. All code should be open source with a permissive license and documented for easy modification.

3) Open Participation: The testbed must be an open system with participation from real end users and developers. All software should be easy to install, un-install, and stop. Developers should be able to easily gain access to resources.

4) Democratic Deployment: If multiple competing implementations of a component are available, all versions should be available. An end user should be able to select a component based on its value, not prescribed defaults.

5) Diversity: The underlying devices that run the testbed should provide a realistic view of the Internet. This means that the testbed should support heterogeneous network and device types.

B. Testbed Architecture

As testbed users, it is important to know the system architecture and components, and how the components work together to provide services to its users. The details of each component are listed as follows. With these details in mind, a researcher or educator can fully utilize the Seattle platform from different standpoints.

1) Virtual Machine: In order to allow programs to run safely on end user machines, the creators of Seattle have encapsulated them in a virtual machine (VM). The virtual machine has several goals. First and foremost, it must prevent a program from performing malicious actions like installing key loggers or reading the user's sensitive files. In addition, the virtual machine provides performance isolation for applications to prevent them from consuming too much CPU, memory, battery, etc. To cut down on the abuse complaints from our testbed, the virtual machine also prevents the user from spoofing the source address of packets or sending ICMP messages. The virtual machine provides memory, CPU, disk, and network access in a similar way to cloud services like an Amazon EC2 instance. Thus the safety for the end user does not significantly limit the generality of code that can be executed.

2) Node Manager: When users want to deploy their code on a remote system, they need a way to upload code into a virtual machine and start it. This should only be allowed for users that have the appropriate credentials to modify a virtual machine. The node manager component takes care of these tasks by mediating access to those virtual machines to ensure that only authorized parties can execute code in them. Since the node manager maintains and controls the VMs running on a node, it also provides an interface to upload code into a VM, start and stop a VM, and collect log files and data from a VM. The node manager is agnostic whether the interface is accessed interactively, by a script, or even another piece of code running in a VM.

3) Service Manager: A researcher uses a service manager to interface with the node manager on a group of nodes running code on her behalf. This can provide multiple different types of interfaces, like a shell, GUI, or automated scripts. Depending on the task at hand, it might make sense to use an interactive service manager that provides direct control of and feedback from node managers. For classroom assignments, students would most probably use an interactive service manager to deploy code into virtual machines and debug the result. Other tasks, such as automatic deployment, monitoring services, and periodic aggregation of data and log files are better handled with a scriptable service manager. Such a scriptable service manager would monitor the number of service instances and re-deploy a user's code onto new virtual machines as needed.

4) Clearinghouse: The purpose of a clearinghouse is to enable researchers to pool and share resources. Without a clearinghouse, a researcher can distribute a Seattle installer with his credentials inside, so that the node manager grants them access to VMs. This means, however, this researcher can only access resources on the machines of users that ran his installer. This severely limits the scale and diversity of the resources that are accessible, and is relatively uncommon (about 10% of our users).

Most researchers instead provide an installer that they get from a clearinghouse. Using a clearinghouse allows researchers to collectively pool resources and then acquire VMs across all clearinghouse resources according to the clearinghouse's policy. Researchers can trade VMs on their well-equipped lab machines for VMs across the Internet on other kinds of Internet-enabled devices contributed to the testbed. A clearinghouse can also provide mechanisms for acquiring a specific category of nodes, offer service level agreements, trade VMs between users, etc.

5) Lookup Service: One big issue that occurs in a distributed system like Seattle is for parties to discover each other. For example, a service manager or a clearinghouse must be able to locate node managers that have VMs they can control. The lookup service allows any party to either advertise a value under a certain key, or to look up a key to find out the associated values. This key-value store is used for a variety of coordination tasks across all components. Seattle's creators leverage many different designs for such a service and the choice of a lookup service is not limited to

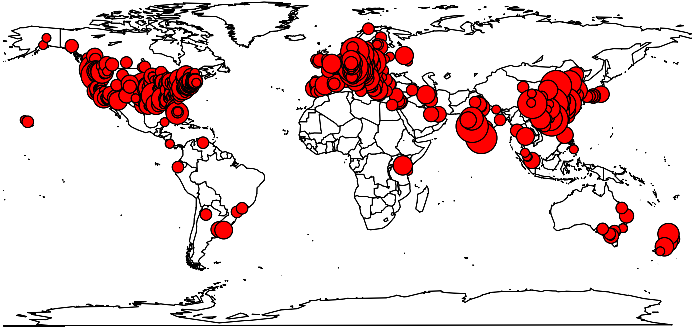

Figure 1: World map showing the distribution of nodes in the Seattle testbed (The area of the circle is logarithmic in the number of nodes).

services available within the testbed.

6) Software Updater: Even though the VM and the node manager provide a safety barrier against malicious or buggy user code, there must be a way to patch potential vulnerabilities in the VM, the node manager, and other parts of the system (including the software updater itself). Ideally, testbed software can be updated with no end user interaction whatsoever. This also means that the end user need not manually search for new releases of the software; instead, updates are pushed to nodes automatically.

7) Infrastructure Services: There are a number of infrastructure services that make it easier to perform a specific action. Some of these run on VMs within the testbed, while others are hosted on stand alone servers. These include monitoring services, communication relays for NAT (Network Address Translation) traversal, or a customized installer creator for bundling applications with Seattle.

C. Current Testbed Size and Scale

The Seattle testbed has been used in a variety of different contexts and its use is spread across the world. This can best be seen by a world map which represents the global user base in Figure 1. As shown in the map, Seattle has a large user base outside of the US. Additionally, Seattle is deployed on a wide variety of networks, as shown in Table I. In the past two years, there have been 20,445 IPs contacted Seattle for updates, with an average of about 20 new IPs a day. This metric over-counts mobile nodes and under-counts nodes behind a NAT. A reverse DNS lookup was used on each IP and then tried to categorize the resulting host names. The result shows about 4% computers in other testbeds (like PlanetLab) and 17% computers at universities, with the remainder roughly split between confirmed home machines and machines that do not respond to reverse DNS requests (they are "unclassified" as in Table I, and we expect them most likely be home systems). There are 505 nodes with a host name explicitly indicating a phone or a tablet. Since nodes often have diurnal patters, the number of online virtual machines at a time is around 3 to 6 thousand.

Node Type	Quantity
University nodes	3,510
Home machines	7,594
Other Testbeds (PlanetLab, Emulab, etc.)	859
Phones	505
Unclassified	7,977
Total	20,445

Table I: Seattle node types determined via reverse DNS lookups on systems that retrieve software updates.

III. Research Experience with Seattle

Over the past four years, Seattle has been promoted to researchers through presentations and demos at various levels of conferences, workshops and seminars. By its design, as described in Section II, Seattle has a number of features that made it immediately attractive for the exploratory type of network research. For example, the number of networks Seattle is deployed on is significant (Table I); all of the code is open and easy to modify; and it is easy to deploy experiments. All of these aspects make Seattle a convenient platform for rapid prototyping and measurement.

In this section we describe our past experiences regarding how to design and deploy experiments on Seattle. We also present some published results using Seattle. The mechanisms used to design and deploy Seattle experiments are in Section III-A, and the application use cases in Sections III-B1 and III-B2 summarize key findings from our prior experiences [5]. Sections III-B3 and III-B4 introduce new experiments on smartphone and tablet devices.

A. Experimentation Design and Deployment

The prerequisite for using nodes federated in the Seattle testbed is to register an account at the Seattle clearinghouse website [6]. In contrast to other testbed systems, Seattle follows a self sign up policy, so the prospective user need not be associated with an academic or industrial institution participating in the system, nor is consent from a Principal Investigator required. This makes Seattle a truly public testbed. Any new registration is processed and access is available immediately. Once registered with a username, the user is gifted credit for ten VMs at a time in the Seattle testbed. Via the clearinghouse website, credits can be used to access up to 10 concurrent VMs on the testbed.

1) Design Of Experiment: The programming language used on Seattle VMs is Repy, a restricted subset of Python [7]. When it comes to designing an experiment, one needs to take into account the restrictions put forth by the Repy sandbox [8]. Much like other programming environments, Seattle comes with an extensive set of standard libraries [9]: Libraries for network time synchronization, data serialization and encoding, cryptography, handling of URLs, HTTP, and XML, GeoIP, a NAT layer, DNS and other announcement / advertisement / lookup libraries, and parallelization and synchronization primitives.

2) Deployment: The interactive frontend for deploying code on acquired VMs is `seash`, the Seattle shell. To locate VMs provided by the user's clearinghouse, `seash`

uses Seattle's VM lookup services. Then, `seash` exchanges cryptographically signed messages with the VMs to control experiments and move data. There are also GeoIP and DNS remapping services available to make it easy to understand devices with diverse or changing locations.

Furthermore, `seash` makes the parallel interaction with multiple VMs a straightforward task. It automatically issues parallel requests, ensures the order of commands, and clearly indicates the success status of each command per VM.

3) Scale Out: As stated before, every user registered with the main Seattle clearinghouse is granted credit for ten VMs. If users want to contribute to the resources available to others reciprocally, they can install Seattle on their machines. Altruistic contributions like these are given back to the community of users in the form of free VM credits.

Users who want to increase their VM credit beyond ten VMs can install (or have others install) Seattle using a Seattle installer linked with their clearinghouse account. Each additional install gives the user access to ten more VMs. Since Seattle runs on most desktops, laptops, tablets, and phones, our experience is that it is easy to find devices to install Seattle on. Compare to existing testbeds such as Emulab and PlanetLab, Seattle's tit-for-tat approach can get users more resources in a much more convenient way.

Note again that not all installers need to be linked to a clearinghouse. A user can create Seattle installers that allow access to specific user keys that are unassociated with a clearinghouse. The user will be able to access these VMs directly, but will not obtain additional credits from a clearinghouse.

B. Research Use Cases

1) Video Streaming and Overlay Routing: Seattle has been used to construct a routing overlay across a number of nodes on different continents, which in turn was configured to forward a live video stream. The route was modified in the progress, and the effects of black-hole routing, delay mismatch, and buffering artifacts due to overload situations on nodes were evaluated.

The software for this experiment has two parts. On the stream source, Repy code runs to encapsulate the output of a web cam into the overlay transport protocol. On the stream sink where the stream is displayed and evaluated, traffic is decapsulated. For convenience, we had the source and sink on a campus LAN, while the VMs were running in different countries and even on different continents.

The routing overlay between source and sink is deployed to Seattle VMs that fit the desired properties, e.g., are located in remote places. The overlay router code consists of only 15 lines of code, as it is only responsible for receiving an encapsulated packet, modifying its header, and sending it off again. Eventually, through this project, improvements to Seattle's networking code were identified and fed back into the main Seattle release [10], [11].

2) Content Distribution Network (CDN) Measurements: CDNs are large-scale networks that provide content to customers everywhere with high performance through replication of content at a large number of distributed sites. Thus, measurements from a single vantage point such as a Point of Presence (PoP) or at a campus network gateway do not successfully capture the global-scale effects of CDN management. Seattle has been used as the platform for a distributed measurement campaign that monitored the video hosting platform YouTube over the course of four weeks from forty vantage points, and has performed various application-layer network measurements to gauge YouTube's load balancing strategies and performance improvements [12].

The metrics gathered during this measurement campaign were the IP addresses www.youtube.com resolves to over time and location of the node, the round-trip time to these addresses and all of our VMs as measured by the TCP three-way handshake procedure, the packet loss experienced on TCP connection initiation, and the (approximate) packet sizes and inter-arrival times when downloading HTTP content to estimate the bottleneck bandwidth. Using a combination of GeoIP and the nodes' Fully-Qualified Domain Names (FQDNs), we then grouped results by VM timezones to identify location-dependent performance variations. The results indicate that the number of frontend IP addresses www.youtube.com resolves to changes over time following a diurnal pattern, a probable cycle of human activity.

3) ET Phone-Home: Many times an end user will have connectivity problems with a network device and have a hard time troubleshooting it. A network administrator can only help so much because they can only see one perspective, from the network to the device. In this use case, by running a Seattle instance on a phone or other device, a user can allow the network administrator access to test connectivity from their machine. Thus the Seattle instance becomes an Extra Technician (ET) which collects statistics and can transmit them to the administrator, possibly over a different transmission medium such as 3G.

In our prior work [13], we described the ET system designed to provide ISPs and others with an environment to troubleshoot home networking in a remote, safe and flexible manner. The system was implemented on Nokia N800 and N810 devices running the Maemo operating system [14]. ET can be sent to smartphones in reply to text messages sent by customers or after the ISP is hired. Alternatively, users can download the script using the smartphone connection to the Internet.

4) Open3GMap: Open3GMap [15] is another Seattle research project running on users' smart devices. This project is motivated by the research challenges in participatory sensing. Smartphones are devices with wireless network connectivity that can constantly provide data communication on the move. They are also embedded with a rich set of sensors that are capable of, for instance, revealing the availability and signal quality of nearby wireless networks, GPS location, etc. However, privacy has been the primary issue in sensing and data sharing for traditional applications in participatory sensing. To address these challenges, we implemented Open3GMap as a Seattle project that opens

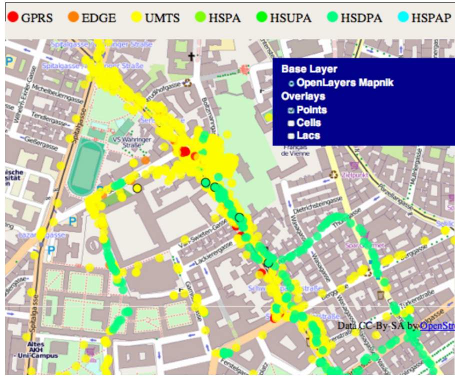

Figure 2: 3G reception quality at different GPS locations.

up secure sensor access to both local and remote processes via a generic sensing software framework[1].

As this is a secure, open-source project, Android is an ideal platform. User-configurable privacy is achieved through Privacy Configuration (a set of privacy filters), where fine-grained security settings allows one of the followings: 1) full exposure of sensor data; 2) reducing the precision of sensor values, e.g., rounding GPS coordinates; 3) salt and hash sensor values for anonymizing gathered data; or 4) completely deny access to individual (or all) sensors. For Open3GMap, we collect data from the 3G radio and GPS sensor on an Android device, e.g., using Repy code over an XML-RPC interface to communicate with the sensors. The XML-RPC interface confirms to the Seattle sensor specification [16]. Figure 2 shows the different types of 3G network devices displayed according to their GPS locations, where data points are shown to represent the available cellular access technology at different geolocations.

IV. Classroom Experiences with Seattle

From our past experience, Seattle is not only very easy to adopt in research projects, but also surprisingly easy for students to use. Seattle was first publicly promoted in 2009 as an educational testbed for networking classes [17]. It helped fill a void in the networking curriculum by allowing students to get practical experience with real-world end user networks. Our existing educational materials have been used in about 40 classes at a dozen universities, ranging from tier-1 research universities to 4-year liberal arts colleges. In this section, we first give an overview of Seattle's use in classrooms, then describe several educational use cases we have experienced in the past.

A. Overview: Educational Use of Seattle

Seattle has been used in approximately 40 classes, about 25 of which are networking, about 8 security classes, one OS class, and the remainder being distributed systems or cloud computing classes. These materials have been used in classes at all levels, with about an equal split between graduate and undergraduate classes. In our experience with Seattle, the feedback from students has been very positive, because they think it is easy to learn and is fast to get their code up and running. The most popular assignment for networking classes [18] covers non-transitive connectivity and NATs, requires no programming, and takes the students only about 90 minutes to complete.

There have been about a half dozen educational workshops / tutorials for Seattle, with another two dozen or so presentations, posters, and talks at educational venues [19]. The first three workshops were run by the creators of Seattle, and the latter ones were run by outside educators. These workshops serve to promote the platform and demonstrate how useful the educational materials for Seattle are in the classroom.

Over the years, there have been extensive educational modules for Seattle to facilitate reuse in the classroom. These modules cover topics such as routing, sliding window protocols, web servers, peer-to-peer networks, and high-level abstractions like MapReduce (details see next section).

Educators have also given positive feedback about Seattle on other forums. As of February 16th, 2013, Seattle is the top ranked ACM SIGCOMM educational resource [20]. Materials using Seattle are being integrated into assignments for the most popular networking textbook [21].

B. Educational Modules

Within the past few years, a sequences of modules were developed to increase the breadth and depth of topics in systems that are accessible to students. These materials will in many cases be adapted to work across multiple classes. In particular, Repy makes Seattle applicable to a broad range of systems courses and provides several additional benefits, including portability, simplicity, diversity, etc.

1) Implementing Access Control (Sec / OS): In this module, we instruct students to implement access control techniques to prevent an attacker from performing a specific action. For example, a student is told that an attacker must not be able to write a file to the system where the first two characters in the file are 'MZ' [22]. Students must provide a correct reference monitor that blocks access only in this specific circumstance and allows all other writes through.

2) Escaping Access Control (Sec / OS): This module leverages the previous module by having students try to bypass each others' reference monitors [23]. The students try to write programs that either write 'MZ' as the first two characters or a program that does not write this but is blocked from writing.

Informal Feedback: Our experience with the above series of use modules has been very positive. The students seem to throughly enjoy being able to attack each others' code. Further, when they see how other students are able to bypass their reference monitor, it reinforces the threats posed by race conditions and incorrectly ordered access checks. It also helps students to get used to reviewing code for security flaws. These assignments have proven popular enough that we have created multiple versions of these assignments.

[1] https://github.com/fmetzger/android-sensorium/

3) Threat Modeling Wireless Routers (Sec / Net): In this module, students are asked to examine how a NAT works and decide how to protect against an attacker that can initiate TCP and UDP connections from behind it (a common feature for browsers, shared WiFi hotspots, etc.). Each student is asked to write code that filters network access in a way that allows users normal access, but prevents an attacker from being malicious on the network [24].

4) Non-Transitive Connectivity and NATs (Net / Dist Sys): This module has students learn about non-transitive connectivity and NATs through practical experimentation on Seattle. This assignment requires no programming. Students perform a series of steps to find non-transitive connectivity. (Where a routing problem on the Internet prevents two nodes from directly communicating when they can talk through intermediaries.) Once they discover this, the student will cause the nodes to communicate through intermediaries for their network traffic. Finally, students get experience with what traffic looks with NAT. This gives students experience with connections similar to what they see at home or when connecting through many wireless networks.

Feedback: This assignment has been used and evaluated using pre and post surveys [25]. The results show two key features. First, students that use these assignments demonstrate improvement when reasoning about NATs, non-transitive connectivity, and similar topics. Second, students also enjoy doing this assignment. While these results demonstrate success, a more thorough evaluation leveraging evaluation experts can be conducted to further understand the positive impact of such an educational module.

5) Link State Routing (Net): Students using this module build a Dijkstra's shortest path overlay between different Seattle nodes. In the basic assignment, software provided to the students will select nodes that should be considered down. The student must generate the routing overlay (and has facilities to check this offline). Following this, the student will deploy their overlay on real Seattle nodes. The final step of the assignment has students measure latencies for Seattle nodes and feed this into the routing algorithm.

A big feature that works well for students and instructors is that the assignment naturally decomposes into six steps. Some instructors choose to give the code for some steps to the students to reduce the workload on the students. For example, they can give the code which does packet forwarding, and simply ask students to perform Dijkstra's algorithm to decide where packets should be forwarded to.

6) Stop-and-Wait (Net): This module has students implement a simple loss-less data transmission protocol over UDP. Students implement a basic acknowledgment system for this assignment. They can then either test their code on the real Internet (using Seattle) or locally with a relay that drops traffic in a manner they specify.

7) Sliding Window (Net): While the stop-and-wait protocol will prevent loss, it is extremely slow and inefficient. The sliding window assignment has students extend their stop-and-wait solution to handle a configurable number of lost messages. As before, students can experiment with loss rates they configure or use real Seattle nodes on the Internet.

8) Web Server (Net): In this module, students create a minimal web server to serve text and HTML files. The student's webserver has a single dynamic component for listing files in a directory, it will need to read and send file contents to clients as well as accept incoming data. From this assignment, students learn about how a webserver can serve both static and dynamic content concurrently.

9) Chat Server (Net): Students using this module build a web-based chat service called Seattlechat. Seattlechat has three main components, a central Seattlechat server whose focus is to relay messages, a collection of Seattlechat translators that change messages into different formats for display, and a Seattlechat client which uses a standard web browser for communicating with a user. Students get experience with multi-tier web applications and learn how to compose components to build a working web service.

10) Distributed Hash Table (Net / Dist Sys): For this assignment the students first implement a DHT-like message routing system based on Chord, and test their implementations on local Seattle resources. The students then run their code on globally distributed Seattle resources. Chord works well over LAN, but has performance and correctness problems in a global scale deployment with non-transitive connectivity. After explaining the reasons behind Chord's poor performance, we can have the class discuss solutions to these problems. Students can then implement these solutions to achieve better performance and reliability.

This assignment reinforces several important ideas. First, the students will reuse their implementation, which emphasizes good software engineering practices. Second, the assignment demonstrates that test and deployment environments may differ significantly. Third, students will hone their debugging skills as they attempt to understand why their "correct code" does not work in a global deployment. Fourth, by creating their implementation from scratch, and motivated by a real problem, students can arrive at unique solutions. Lastly, instructors can easily evaluate student implementations using a small set of metrics.

V. Discussions and Future Work

A. Limitations

For all of its strengths, Seattle is not without weaknesses as well. Its greater degree of network diversity and realism comes at the cost of programmability. There are a few places where we think Seattle could improve on its functionality. Per design, its performance is restricted. There is a way for resource donors to increase the restrictions during install, but the standard values are set quite low (e.g. an average 10 KBps of network traffic). We acknowledge that port restrictions make sense for safety purposes, but this leaves us with no way of transparently serving Domain Name System (DNS), HTTP, and other services on well-known ports (which would require special privileges). Sometimes, ICMP ping and traceroute measurements would be of interest, but are not available due to Repy's restrictions. If a node reboots,

the user has to restart the experiment either manually or by scripting. At the moment, Seattle's scriptable deployment service does not run on Repy, which means the user has to provision a machine outside the testbed to do that. We are in the process of rectifying these situations so that the Seattle testbed will become more flexible.

B. Future Work of Seattle

1) Virtual Machine:

i. *Repy V2.* Since the initial deployment of Repy V1, there has been on-going work to add better support for several things. First, users will be allowed to add security policies to all code that executes on their sandbox [8]. For example, the end user may want to restrict the code in the Seattle node using a policy that allows the node to only communicate with other Seattle nodes [26].

Another feature addition was to support extensibility of the sandbox. Repy V2 can be extended with additional functionality so that it can utilize other devices, if the end user allows it. For example, a researcher added support for accessing tun / tap interfaces from Repy V2 to support a project called ToMaTo [27]. These hooks are available on any Seattle nodes that have enabled it. In the future, providing safe access to GPS can be made possible from users who opt-in to this.

ii. *Lind.* Ongoing work at the University of Victoria aims at extending Repy V2 to support programming languages other than Python. The researcher will be able to compile their programs using the tool chain from Google Native Client (NaCl) [28] to validate the safety of code. The resulting code will execute computationally in the NaCl sandbox, but call into Repy V2 to perform system calls. This extension will give researchers the ability to execute anything that can be compiled for the x86 architecture and possibly other architectures such as ARM. We feel this will be useful both for reusing legacy code and for writing performance critical experiments.

2) Educational Modules: In addition to the educational modules in Section IV-B, there are also several others under development. We list some of them as follows.

i. *Understanding Web Modification (adaptable to Net):* There are many countries and ISPs that filter or alter Internet traffic [29], [30]. Students will use Seattle nodes in countries all around the world to examine how a set of popular sites look from different locations, and look to whether filtering occurs and also find the type of filtering.

ii. *Evading Web Censorship (adaptable to Net / Dist Sys):* As a follow-on to the previous module, students will build an overlay to evade web modification. They will deploy the web proxy, but will relay the requests through an overlay network on Seattle. Students will use unencrypted overlay, end-to-end encryption and Onion routing across the overlay.

iii. *Man-in-the-Middle Attacks (adaptable to Net / Dist Sys):* This module has clients act as a man-in-the-middle to change network content before it reaches the client. The student writes a Seattle program that sits in-between their web browser and the Internet and changes traffic.

iv. *XSS (Cross-site scripting) Attacks:* We will create a module where students learn how to inject attack code into a Seattle web application that is provided to them. Following this, they will explore different techniques for mitigating XSS.

v. *Dynamic Code Injection:* Students will be given a web application that dynamically interprets requests and has an injection vulnerability. Then they will design a defense to this attack. After this, they will try to attack each other's "patched" version of the web server to find additional vulnerabilities.

VI. Related Work

Like any large scale systems work, the Seattle testbed makes heavy use of other work and ideas from many fields. Our goal is to combine aspects of the existing systems' ideas in order to create a scalable general-purpose testbed that is easy to use, diverse, and open for anyone to join.

A. Peer-to-Peer Networks

Peer-to-peer (P2P) networks are similar to the Seattle testbed in that administrators do not need to fully provision their servers. One example application is given in [31] where a P2P application uses a large dataset gathered from BitTorrent users for crowdsourcing event monitoring. However, in P2P systems, the coordinated execution of code is only considered in some special cases like P2P-based network management [32]. In addition, resource isolation, safety and privacy are typically not a major concern. The Seattle testbed focuses comprehensively on these issues and in particular on the execution of code on behalf of another user.

B. Cloud Computing

Cloud computing is similar to Seattle in that resources which are not owned by the user provide the user with improved availability and performance. The Seattle testbed shares many of the scalability benefits as cloud computing, albeit lacking pairwise bandwidth. However, cloud computing infrastructures typically consist of large data centers and require developers to pay for the resources. In contrast, the Seattle testbed leverages donated resource sharing.

C. Volunteer Computing

Volunteer computing efforts like BOINC [33] and SETI@Home [34] also leverage unused resources on end-user computers. However, the primary leveraged resource is the CPU. Access to volunteer computing resources is tightly controlled because developers are given low-level access to the machine so security is not provided. Also, a volunteer computing platform will usually only execute when the system is idle so resource isolation is not a major concern.

D. Measurement Infrastructure

There have been a variety of efforts to gather measurements from end user machines. NETI@home [35] monitors the traffic a user sends throughout the course of their normal actions. This is useful, but does not allow researchers to measure paths between users (unless they intersect) or provide a mechanism for in-situ experimentation [32]. Fathom [36] is a Firefox extension that implements a number of measurement primitives that enable websites or other parties to program network measurements using JavaScript. Other existing techniques for measuring network performance, such as SAMKnows or BISMark [37], cannot adapt their measurements procedure and software in the course of a measurement study. Seattle instead allows for very general programmability and rapid code updates on end nodes.

VII. Conclusions

In this paper we present our research and educational experiences with Seattle testbed over the past four years. Seattle serves a role similar to existing testbeds such as PlanetLab, but provides diverse network connectivity, real world use patterns and high scalability. It is an excellent platform providing in-depth understanding of real world problems for researchers, and hands-on experiences for students. Although it has limitations in its programmability, future work will expand the usability of Seattle by adding support for efficient computation. Seattle has already proven itself an excellent platform for experimental research and educational use across a diverse set of classes.

References

[1] "PlanetLab," http://www.planet-lab.org/.
[2] "Emulab," https://www.emulab.net/.
[3] "GENI," http://www.geni.net/.
[4] "Seattle," https://seattle.poly.edu/.
[5] A. Rafetseder, F. Metzger, and K. Tutschku, "Three thrilling years of using the seattle Internet testbed," under submission, 2013.
[6] "Seattle Clearinghouse," https://seattleclearinghouse.poly.edu/.
[7] "Repy Tutorial," https://seattle.cs.washington.edu/wiki/RepyTutorial.
[8] J. Cappos, A. Dadgar, J. Rasley, J. Samuel, I. Beschastnikh, C. Barsan, A. Krishnamurthy, and T. Anderson, "Retaining sandbox containment despite bugs in privileged memory-safe code," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010.
[9] "Seattle Standard Library (SeattleLib)," https://seattle.cs.washington.edu/wiki/SeattleLib.
[10] J. Eisl, A. Rafetseder, and K. Tutschku, "Service architectures for the future converged internet: Specific challenges and possible solutions for mobile broad-band traffic management," in *Future Internet Services and Service Architectures*, 2011.
[11] K. Tutschku, A. Rafetseder, W. Wiedermann, and J. Eisl, "Towards sustained multi media experience in the future mobile internet," in *14th International Conference on Intelligence in Next Generation Networks (ICIN)*, October 2010.
[12] A. Rafetseder, F. Metzger, D. Stezenbach, and K. Tutschku, "Exploring youtube's content distribution network through distributed application-layer measurements: A first view," in *1st Workshop on Modeling, Analysis, and Control of Complex Networks*, 2011.
[13] L. Collares, C. Matthews, J. Cappos, Y. Coady, and R. McGeer, "Et (smart) phone home!" in *Proceedings of NEAT'11 workshop*. ACM, 2011, pp. 283–288.
[14] "Using Seattle on the Nokia N800 and N810," https://seattle.cs.washington.edu/wiki/SeattleOnNokia.
[15] "Open3GMap," http://homepage.univie.ac.at/albert.rafetseder/o3gm/.
[16] "Using Sensors in Seattle," https://seattle.cs.washington.edu/wiki/UsingSensors.
[17] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Seattle: a platform for educational cloud computing," in *ACM SIGCSE Bulletin*. ACM, 2009, pp. 111–115.
[18] "Take Home Assignment — Seattle," https://seattle.cs.washington.edu/wiki/EducationalAssignments/TakeHome.
[19] "Seattle Events," https://seattle.poly.edu/wiki/SeattleTalks.
[20] "ACM SIGCOMM Educational Resources," http://edusigcomm.info.ucl.ac.be/.
[21] J. Kurose and K. Ross, *Computer Networks: A Top Down Approach Featuring the Internet*. Addison Wesley, 2006.
[22] "Building a reference monitor that implements access control," https://seattle.cs.washington.edu/wiki/EducationalAssignments/SecurityLayerPartOne.
[23] "Access control testing and penetration," https://seattle.cs.washington.edu/wiki/EducationalAssignments/SecurityLayerPartTwo.
[24] "C. Heffner. How to hack millions of routers," http://www.youtube.com/watch?v=Zazk0plSoQg.
[25] S. Wallace, M. Muhammad, J. Mache, and J. Cappos, "Hands-on internet with seattle and computers from across the globe," *Journal of Computing Sciences in Colleges*, 2011.
[26] "C. Barsan and J. Cappos. ContainmentInSeattle," https://seattle.cs.washington.edu/wiki/ContainmentInSeattle.
[27] D. Schwerdel, D. Hock, D. Günther, B. Reuther, P. Tran-Gia, and P. Müller, "Tomato-a network experimentation tool," 2011.
[28] "Google native client," http://code.google.com/p/nativeclient/.
[29] R. Faris and N. Villeneuve, "Measuring global internet filtering," *Access denied: The practice and policy of global Internet filtering*, 2008.
[30] C. Zhang, C. Huang, K. Ross, D. Maltz, and J. Li, "Inflight modifications of content: who are the culprits?" in *Workshop of Large-Scale Exploits and Emerging Threats*, 2011.
[31] D. Choffnes, F. Bustamante, and Z. Ge, "Crowdsourcing service-level network event monitoring," in *ACM SIGCOMM Computer Communication Review*. ACM, 2010.
[32] A. Binzenhöfer, K. Tutschku, B. Graben, M. Fiedler, and P. Arlos, "A P2P-based Framework for Distributed Network Management," in *New Trends in Network Architectures and Services, LNCS 3883*, 2006.
[33] D. Anderson, "Boinc: A system for public-resource computing and storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, 2004.
[34] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@ home: an experiment in public-resource computing," *Communications of the ACM*, 2002.
[35] C. Simpson, D. Reddy, and G. Riley, "Empirical models of TCP and UDP end-user network traffic from NETI@ home data analysis," in *20th Workshop on Principles of Advanced and Distributed Simulation*, 2006.
[36] M. Dhawan, J. Samuel, R. Teixeira, C. Kreibich, M. Allman, N. Weaver, and V. Paxson, "Fathom: A browser-based network measurement platform," 2012.
[37] S. Sundaresan, W. de Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescape, "Broadband Internet Performance: A View From the Gateway," in *In Proceedings of SIGCOMM*, 2011.