**RESOURCE FOR TEACHING NETWORKING AND DISTRIBUTED SYSTEMS WITH SEATTLE**

**ABSTRACT**

In this paper we describe why an instructor might use Seattle [4] to teach a course on networks or distributed systems. We give an overview of the Seattle educational testbed and describe assignments and resources available to educators planning to use Seattle in the classroom. Finally, we discuss the applicability of Seattle to other types of distributed systems paradigms such as cloud computing, peer-to-peer networking, and ubiquitous computing.

**INTRODUCTION**

Many educators teaching networks and distributed systems courses have their students write networking code that they run over the LAN in a computer lab. This gives students practical experience with writing networking code. Many instructors also try to add some realism to these environments, such as simulating network failures by killing processes on LAN nodes. Unfortunately, even an altered LAN network behaves nothing like the real Internet. It lacks highly variable network bandwidths and latencies, non-transitive routing, and other features such as middleware boxes that are fundamental to today’s Internet architecture. Such programming assignments leave students unprepared for the challenges they will face when writing code that must work between arbitrary hosts on the Internet.

An orthogonal issue facing instructors teaching networks and distributed systems is to maintain the relevance of their courses to the potential career paths of their students. Cloud computing and peer-to-peer applications are becoming increasingly popular. Numerous corporations now look for applicants that possess the skills necessary to manage large infrastructure and to write applications that can utilize resources in the cloud. A short sample of these includes companies such as Microsoft, Amazon, Google, and IBM. The SIGCSE community has not missed this trend, and efforts such as in [1] and in [2] are underway to make cloud computing accessible to students in the classroom. It is important for students to gain first-hand experience with these technologies to prepare students for the changing technology trends.

The purpose of the Seattle [3] educational testbed is to solve both of the above challenges. Seattle is a free, educational testbed that is comprised of resources donated from universities. Today, Seattle consists of resources on about 1000 computers at over 100 universities on 6 continents [4]. This means that students using Seattle receive invaluable experience with running their code on computers across the Internet. Additionally, Seattle’s support for heterogeneous devices can be leveraged by instructors teaching topics such as ubiquitous computing. And as new types devices are added to the Internet, Seattle can naturally incorporate them into the testbed in the form of new donations. As a result, the testbed’s network and networked devices are representative of the Internet at large even as Internet technologies undergo change.

**WHAT IS SEATTLE?**

Seattle is a community-driven testbed of computers from around the world. It is completely free for anyone to use. Universities and individuals donate available computer resources on multi-user machines, such as the machines in a computer laboratory. Resources are donated by installing Seattle software, which runs in the background while the computer is being used for other tasks. Seattle uses incentives to drive adoption of the testbed. As of March 2009, for every computer running the donation software, the testbed provides resources on ten other computers to the person making the donation.

Seattle can be installed in a restricted user account, but has protection built-in, so many users simply install Seattle under a normal user account. Seattle provides protection through a sandboxed environment called a *vessel*. A vessel provides strict isolation guarantees and resource restrictions. This prevents a student program running inside a vessel from impacting the security or performance of the donating computer. Seattle programs are written in a specific language (a restricted version of Python) which makes it possible to ensure the safety of the programs.

Students who install Seattle on a computer on which they have an account can run programs manually in a vessel. However, students are unlikely to have an account on all of the computers at different universities where they want to run Seattle programs. To help with this, every donating computer also runs a *node manager*. The node manager allows students to run Seattle programs on remote computers. It also ensures that the total amount of resources consumed by all vessels is kept below 10%. As a result, the performance impact of Seattle on other users of the donated computer is minimal.

To run programs on remote computers, students use the Seattle shell (called *seash*). Seash allows students to quickly and easily find vessels they control, upload and download data files, run code, and retrieve debugging information about their program.

The Seattle software requires no maintenance or overhead for the university's technical support staff. Seattle software comes with a software updater, which automatically keeps the Seattle software up-to-date without any user or administrator intervention. As a result, once the user installs the Seattle software, they need not do anything else to donate to the educational testbed or to keep their software current.

There is also a SeattleGENI website that facilitates sharing of resources [6]. This web site allows instructors to share the donation credits they receive with students taking their courses. It also allows students to request resources on computers with specific network characteristics, such as LAN or WAN.

**SEATTLE WEB RESOURCES**

The Seattle website [4] has a host of resources available to instructors and students. The two main points of interest to an instructor are the *student portal* and the *educator portal*. The student portal contains tutorials that introduce students to Python, teach the restricted Python language used to write Seattle programs, explain how to run a Seattle vessel locally, and describe how to use seash for remote execution.

The educator portal contains a set of assignments and resources for instructors. These assignments are fully specified and ready to hand out to students. There are also instructor-only solutions that can be requested by an instructor if they want to see a solution to an assignment without implementing it. Here is a description of these assignments:

* The **Take-Home Assignment** introduces seash and teaches about connectivity on the Internet. The goal is to teach student about non-transitive connectivity and NATs (Network Address Translation) boxes. The student locates non-transitive connectivity and sets up one-hop routing to avoid it. The student also determines if their home computer is behind a NAT. This is an ideal "first assignment" for a class because it only takes an hour and requires no programming.
* The **Stop and Wait Assignment** has students implement a reliable messaging protocol on top of UDP. This assignment introduces students to Seattle. It is simple to code (the provided solution is just 50 lines of code), but teaches students about the non-trivial topic of reliable transmission on the Internet. Students can also use their solutions to take useful measurements of Internet traffic, such as latency, bandwidth, and jitter.
* The **Sliding Window Assignment** is an advanced extension to the Stop and Wait Assignment that introduces students to an important optimization used by transport protocols like TCP. This exercise in pipeline packets prepares students for more advanced transport layer topics such as flow control, and congestion control.
* In the **Link State Routing Assignment** students implement a link state routing protocol that runs between Seattle nodes. Students implement Dijkstra's shortest path routing and then apply their code to route packets with minimal latency between nodes.
* The **Web Server Assignment** introduces students to layering of network services and protocols (HTTP). Students implement a high throughput server that can handle hundreds of simultaneous connections. To make the project tractable, students implement only rudimentary parsing of the HTTP request header. The assignment mandates that student solutions support the Firefox browser. This teaches students about interoperability because student code must work with software they did not write.
* The **Chat Server Assignment** is a fun extension to the Web Server Assignment. Students implement simple POST-method processing in their web-server and use it to create a web chat forum. This assignment solidifies students' understanding of protocol layering and further unravels HTTP.

Details for all of these assignments can be found on the Seattle project website[4]. Besides regular one to two week assignments, Seattle can also be used for long-term class projects and for undergraduate research. On-going projects that we are aware of include constructing a peer-to-peer distributed hash table and writing an implementation of MapReduce based on Hadoop [1].

**SEATTLE PROGRAMMING LANGUAGE : REPY (RESTRICTED PYTHON)**

Seattle programs are written in a programming language that is a subset of Python called *Repy*. Repy exposes an API that simplifies network programming as much as possible. In our experience, we have found that students who know another programming language can learn Repy after a single pass through the tutorial.

One of the first networking programs a student will write is a program that sends UDP pings. The UDP ping client sends a UDP ping message to the UDP ping server and waits for a response. The server responds to all messages by sending a UDP pong message back to the client. This is a common first assignment in many networking classes because the client and server are each 20-30 LOC (lines of code) in most programming languages. In Repy, the implementations are much shorter (6 LOC for the client, and 4 LOC for the server), and look like this:

#####################################

# UDP PING CLIENT:

# Run this program with server IP, and server port args

# handle the reply from the server

def **got\_reply**(srcip, srcport, mess, ch):

print "received message: '" + mess + "' from" + \

srcip + ":" + str(srcport)

**if callfunc == 'initialize':**

# register a callback to handle the server’s reply

recvmess(getmyip(), int(callargs[1]), got\_reply)

# send a UDP ping message to the server

sendmess(callargs[0], int(callargs[1]), "ping", \

getmyip(), int(callargs[1]))

# exit in two seconds no matter what

settimer(2, exitall, ())

#####################################

# UDP PING SERVER:

# respond to a client’s UDP ping with a UDP pong

def **got\_ping**(srcip,srcport,mess,ch):

sendmess(srcip,srcport,"pong",getmyip(), \

int(callargs[0]))

**if callfunc == 'initialize':**

# register a callback to handle the client’s ping

recvmess(getmyip(), int(callargs[0]), got\_ping)

The Repy tutorial also has short examples, such as how to get the time from an NTP (Network Time Protocol) server and how to fetch pages from a web server. Similarly, programs that previously would have been considered a significant assignment (like UDP overlay multicast), can now fit on a slide, and be discussed by instructors in a lecture about multicast. The resources for learning Repy in the student portal demonstrate that even complex networking functionality is easy to express in Repy.

**CONCLUSION**

Instructors teaching networking and distributed systems courses often have few resources at their disposal. The Seattle testbed is intended to support such instructors by providing a large scale testbed with valuable network diversity, and a collection of ready-to-use assignments. This paper overviewed some of the basic Seattle feature: the Seattle testbed is composed of donated resources. Students use seash (Seattle shell) to manage a large ensemble of Seattle vessels, and write Seattle programs in Repy (a subset of Python). Seattle programs run in resource constrained and secure sandboxes called vessels, which make it safe and easy to donate resources to the testbed. We hope that instructors will find the Seattle testbed to be a useful tool in their courses.

**BIBLIOGRAPHY**

[1] R. A. Brown. "Hadoop at home: large-scale computing at a small college," *in 40th SIGCSE technical symposium on Computer science education*, Chattanooga, TN, USA, 2009, pp. 106 - 110.

[2] A. Kimball, S. Michels-Slettvet, and C. Bisciglia. "Cluster Computing for Web-Scale Data Processing," *in 39th SIGCSE technical symposium on Computer science education*, Portland, OR, USA, 2008, pp. 116 - 120.

[3] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. "Seattle: a platform for educational cloud computing," *in 40th SIGCSE technical symposium on Computer science education*, Chattanooga, TN, USA, 2009, pp. 111 - 115.

[4] "Seattle: The Internet as a Testbed." Accesssed March 12th, 2009 at https://seattle.cs.washington.edu/

[5] "GENI | Global Environment for Network Innovations using Seattle." Accessed March 12, 2009 at https://seattle.cs.washington.edu/geni/