Seattle: A Platform for Educational Cloud Computing*

Justin Cappos Ivan Beschastnikh Arvind Krishnamurthy Tom Anderson

Department of Computer Science and Engineering, University of Washington
Seattle, WA 98105, U.S.A.
{justinc, ivan, arvind, tom}@cs.washington.edu

ABSTRACT

Cloud computing is rapidly increasing in popularity. Companies such as RedHat, Microsoft, Amazon, Google, and IBM are increasingly funding cloud computing infrastructure and research, making it important for students to gain the necessary skills to work with cloud-based resources. This paper presents a free, educational research platform called Seattle that is community-driven, a common denominator for diverse platform types, and is broadly deployed.

Seattle is community-driven — universities donate available compute resources on multi-user machines to the platform. These donations can come from systems with a wide variety of operating systems and architectures, removing the need for a dedicated infrastructure.

Seattle is also surprisingly flexible and supports a variety of pedagogical uses because as a platform it represents a common denominator for cloud computing, grid computing, peer-to-peer networking, distributed systems, and networking. Seattle programs are portable. Students' code can run across different operating systems and architectures without change, while the Seattle programming language is expressive enough for experimentation at a fine-grained level. Our current deployment of Seattle consists of about one thousand computers that are distributed around the world. We invite the computer science education community to employ Seattle in their courses.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer science education; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*client/server, distributed applications*; C.4 [**Computer Systems Organization**]: Performance of Systems—*design studies, measurement techniques*

General Terms

Experimentation, Measurement, Performance, Security

Keywords

Cloud computing, peer-to-peer computing, cluster computing, distributed computing

*This work was partially supported by NSF Grant CNS-0834243

SIGCSE'09, March 3–7, 2009, Chattanooga, Tennessee, USA.

1. INTRODUCTION

Cloud computing is rapidly increasing in popularity as many organizations outsource hardware and maintenance and instead focus on software [3, 4, 11, 12, 14, 8, 16]. However, despite the attention, there is a lot of disparity in what cloud computing means. RedHat / Amazon's EC2 [3] provides cloud computing as a collection of Linux machines with storage functionality [4]. Google's platform for cloud computing hides locality and scalability issues from the programmer who writes programs to a custom programming API [11]. Microsoft views it as a virtualization layer between the hardware and the OS and is releasing a developer toolkit for providing the user with "software plus service." [16]

We provide an educational platform called Seattle that is a common denominator for a wide range of these definitions. Seattle's simple to learn programming language, a subset of the Python language, is expressive enough to allow students to build algorithms for inter-machine interaction (like a global store or a DHT). As a result, Seattle is useful in many pedagogical contexts ranging from courses in cloud computing, networking, and distributed systems, to parallel programming, grid computing, and peer-to-peer computing.

Seattle is a community-driven effort that depends on resources donated by users of the software (and as such is free to use). A user can install Seattle onto their personal computer to enable Seattle programs to run using a portion of the computer's resources. Seattle programs are sandboxed and securely isolated from other programs running on the same computer. Seattle provides hard resource guarantees that an erroneous or malicious program cannot circumvent.

In addition, Seattle runs on a variety of different operating systems and architectures including Windows, Mac OS-X, Linux, FreeBSD, and even portable devices like Nokia N800s and jail broken iPhones. Code written for Seattle is automatically (and transparently) portable to different architectures and runs the same across all systems.

Seattle has a preexisting base of installed computers and is already widely deployed on almost one thousand computers that are spread across hundreds of universities worldwide. Seattle users can run their programs on computers span-

ning the Internet – a feature that is currently being used by several classes at major universities.

This paper describes the architecture of the Seattle cloud computing platform (Section 2) including the programming API (Section 2.1), the sandboxing mechanism (Section 2.2), the control of sandboxes on a host computer (Section 2.3), and the way in which students manage their running programs (Section 2.5). Following this, we describe the computational resources available to classes using Seattle and how we expect this platform to grow in the future (Section 3). Next, we provide some example assignments to demonstrate how Seattle can be used in courses (Section 4). We then discuss related work (Section 5) and conclude (Section 6).

2. ARCHITECTURE

To use Seattle, the instructor creates an account on our website and obtains an installer. The machines that run the installer (such as computers in the universities computer lab) donate resources that are credited to the instructor. The instructor can then obtain resources on machines around the world. As of December 1st, 2008 the current policy is that for each donation, the instructor receives resources on ten other computers. However, the instructor can delegate those resources either directly to students or to TAs who do more fine-grained delegation. Students and TAs download a toolkit and then experiment with their resources.

Seattle's architecture is comprised of several components. At the lowest level the *sandbox* component guarantees security and resource control for an individual program. Programs are written to the Seattle API in a subset of the Python programming language. This API provides portable access to low level operations (like opening files and sending messages). At a higher layer, the *node manager* determines which sandboxed programs get to run on the local computer. A public key infrastructure is used to authorize control over sandboxed programs. Lastly, the *experiment manager* lets students control their program instances across computers.

2.1 Seattle API

Seattle provides a programming API for low-level operations (like writing to files or sending network messages) and maintains program portability using an abstraction layer. Platform specific code below this abstraction layer handles non-portable operations enabling unmodified programs to run on a wide variety of platforms.

The API consists of five categories: file, network, timer, locking, and miscellaneous. The file API calls enable limited access to the local computer's persistent storage (interacting only with files in a single directory). The network API calls provide the local IP address, perform a DNS lookup, enable sending and receiving of UDP messages, and managing of and communicating over TCP connections. The timer API calls enable the programmer to put the current thread to sleep and to schedule functions to be called at later times. For example, the programmer can register an event to periodically send a heartbeat message to another computer. The locking functions allow the programmer to handle concurrency in their program (as common state may be accessed and modified by multiple threads at the same time). The miscellaneous API calls allow the programmer to exit the program, to generate random numbers, and to provide the amount of time the program has been running.

2.2 The Sandbox

The sandbox's primary goal is to securely execute user code. There are two aspects to this — preventing insecure actions and limiting resource consumption. To prevent insecure actions the sandbox hooks into the Python parser and reads the program's parse tree. Only actions that the sandbox can verify as safe may execute.

To control resource consumption on the host the sandbox interposes on all API calls made by a program. The sandbox monitors the overall use of resources like CPU, memory, and disk space to ensure the program does not exceed its bounds. Each API call that uses a monitored resource is evaluated before being granted or denied the resource. The sandbox also restricts the rate at which API calls are performed.

The restrictions and resource limits of the sandbox are configurable and may restrict different programs in different ways. For example, one program may only be allowed to receive UDP packets on port 11111, while another program may be restricted to receiving UDP packets on port 22222. This enables multiple sandboxes on the same computer to host programs controlled by different users.

2.3 Node Manager

While useful in itself, the sandbox is part of a larger ecosystem. The sandbox isolates a specific running program on a host computer, but does not address how that program is started, which programs are run, and who has permission to run a program. Such functionality is provided by the node manager, which manages sandboxed running programs as part of what we call *vessels*. The node manager stores information about the vessels it controls and allows vessels to be started, stopped, combined, split, and changed.

2.3.1 Vessels

A vessel is a controlled environment for running code (implemented using the Seattle sandbox). Intuitively, a vessel includes the program's sandbox and the node manager state (such as the resources and restrictions assigned to the program). Vessels have well defined boundaries that prevent them from interfering with one another (for example, different vessels have their own disjoint set of network ports). Each vessel has associated with it a restrictions file, a stop file, and a log. The restrictions file lists what the vessel can and cannot do (such as the network ports that can be used) along with the amount of each resource the vessel may use. The stop file enables the node manager to stop the vessel (by creating a file with that name). The log is a circular buffer written by the vessel to communicate useful information to the vessel owner. The log helps the programmer to diagnose failures and to capture program state for off-line analysis.

A common scenario is for a student to obtain a vessel from their instructor. The student then decides the program they want to run in their vessel. To do this, she directs the experiment manager to install the program on a node. The experiment manager uploads the program to the student's vessel (along with any data files) and executes the program in the vessel. The student also can easily perform this action on groups of vessels spread across many different nodes. The student can then monitor the status of her program by looking at a status indicator provided by querying the node manager (coarse-grained monitoring) or by downloading information about the program from its circular log buffer (fine-grained monitoring). The user can also stop the vessel

while retaining all of the state so that she can examine data files and logs.

In a more complex example an instructor splits a single vessel on a node into multiple vessels, and assigns each vessel to a student in the class. Vessels may also be combined for flexibility. For example, the students may be allowed to work in groups. Once the groups are formed, some of the students may decide to combine their vessels so as to get more resources in a single vessel.

2.4 Locating Seattle Nodes

It is important to note that there is nothing parallel or distributed about the node manager. The node manager only manages the vessels on the local system. To facilitate global location of resources, the node manager inserts a key/value pair into two different public DHTs (OpenDHT [19]) every five minutes. The inserted key is the owner's public key and the value is the local computer's IP address. This allows a user to lookup their public key to find the nodes with vessels they control without needing to keep track of these nodes on their own.

2.5 Experiment Manager

The experiment manager is the main tool in the toolkit that students use to interact with Seattle. The experiment manager transparently handles discovery of vessels the user controls by querying the DHT, and communicates with remote node managers to perform actions on the user's behalf.

The experiment manager provides the user with a shell interface (similar to PLuSH [1]) in which the user can issue commands. For example, users can install software in their vessels – the experiment manager uploads a program into the vessels the user specifies. Users can also start and stop vessels, or report on the status of a vessel. A vessel's status can be fresh (has never run a program), started (is running a program), stopped (was requested to stop), or terminated (terminated due to a normal exit or an unhandled program exception). When a vessel has failed (perhaps due to a bug in the student's code), exception information with a stack trace of the fault is logged. The student can use the experiment manager to find the failed program instances to inspect their logs or to see an exception's stack trace.

3. DEPLOYMENT

Since it is unclear what the future of cloud computing will be, we are interested in providing students with the most diverse set of resources possible. Seattle runs on Linux, FreeBSD, Mac OS-X, XO (one laptop per child)[1], and Windows platforms. Seattle also runs on mobile devices like the iPhone[1] (if jail broken) and Nokia N800[1]. We are interested in adding support for other platforms as users express interest in running Seattle on new platforms.

In addition to platform diversity, network connection diversity is also important. Seattle is already widely deployed at universities around the world. An instance of Seattle is running on each PlanetLab [18] node, giving a presence on close to one thousand nodes at hundreds of universities.

PlanetLab provides access to computers at a large number of well connected locations, although most only have two computers per site available. As we are now publicly releasing Seattle for educational use, we expect to see an increase in resource diversity. We anticipate that universities will install Seattle on most of the computers in their lab. Such deployments will provide a distributed environment that effectively emulates cluster computing. We also anticipate that many students will load Seattle on their home computers. This will allow for emulation of peer-to-peer computing as home computers will typically have connectivity characteristics representative of the average Internet user.

We believe that the diversity of platforms and network connections enables a wide range of pedagogical uses. Students can experiment with cluster computing, grid computing, peer-to-peer, and cloud computing by simply varying where their program is deployed. Additionally, the same program will run in any of these environments. Of course, the characteristics of the environment will determine the efficiency and scalability of the employed distributed algorithm — a crucial distributed systems lesson for students.

[1]Some threading libraries and operating systems do not provide accurate CPU and/or memory information. As a result, certain resources cannot be effectively sandboxed

4. SEATTLE IN THE CLASSROOM

This section describes educational resources available to the student and instructor. The resources are divided into two areas. There is a student portal which contains resources for students who are learning the basics of Seattle. There is also an educator portal that contains resources for educators to help them use Seattle in the classroom.

4.1 Student Resources

To aid students learning Seattle, we provide a student portal with tutorials describing how to program and use Seattle, documentation for the API, a resources and restrictions guide, and other documentation. The tutorials provide code snippets demonstrating the API, and explains how to use the experiment manager to perform different tasks. Our experience has shown that students can quickly learn the Python programming language [20, 6] and quickly learn to program in Seattle. Our experience is that undergraduates who had no previous networking experience can implement programs like overlay multicast and TCP forwarding with a few hours of effort after completing the tutorial.

To illustrate how easy it is to program Seattle, A popular first project for networking students is the Echo client/server. The two *complete* Seattle programs are both concise and simple. The echo client is just 6 lines of code:

```
# Handle an incoming message
def got_reply(srcip, srcport, mess, ch):
  print 'received:',mess,"from",srcip,srcport

if callfunc == 'initialize':
  # when a message arrives on my IP, port 43210,
  # start an event to call the function 'got_reply'
  recvmess(getmyip(), 43210, got_reply)
  # send a hello message to my IP, port 54321
  sendmess(getmyip(), 54321, 'hello', getmyip(), 43210)
  # exit in one second
  settimer(1,exitall,())
```

The echo server consists of just 4 lines of code:

```
# Handle an incoming message
def got_message(srcip, srcport, mess, ch):
  sendmess(srcip,srcport,mess)

if callfunc == 'initialize':
  recvmess(getmyip(), 54321, got_message)
```

Furthermore, a first project for many students in distributed systems is to measure the connectivity characteristics between pairs of computers [2]. The Appendix lists a complete 30 line Seattle program that performs an all-pairs-ping and displays its results in a webpage when contacted.

4.2 Educator Resources

To aid instructors in integrating Seattle into their existing curriculum, we also provide an educator portal. The first item on the educator portal is a description of instructor experiences with Seattle. After Seattle has been used, we have asked the students to fill out a survey outlining how they felt the platform impacted their experience. We recorded the results on the web page.

Besides information about experiences with Seattle, the educator portal also contains course materials such as handouts and example assignments. One special-purpose assignment called the TakeHomeAssignment requires no programming and takes about 1 hour to do. The purpose of the TakeHomeAssignment is to show the student or instructor the practical effects of non-transitive connectivity and NATs on the Internet today, while introducing them to Seattle.

In addition, the educator portal provides a collection of ready-to-go programming assignments for use with Seattle. The assignments include implementing a reliable protocol on top of UDP, performing overlay routing using link-state routing on the Internet, building application level services like a webserver, and understand layering of services by creating a chat server that operates over HTTP.

The educator portal also provides assignment ideas that are appropriate for course long projects or graduate assignments. For example, students may implement a DHT (such as Chord [22]) to better understand non-transitive connectivity. A simple implementation will work well on LAN environments but will fail horribly on the Internet due to non-transitive connectivity and other network effects. After measuring these effects and then understanding the reason behind Chord's poor performance, the students can discuss solutions to these problems. Students can then implement and test these solutions to achieve better performance and reliability. This assignment emphasizes good software engineering practices (since code is reused), that test and deployment environments may differ significantly, and encourages students to come up with unique solutions to the problem yet is easily evaluated using a small set of metrics.

5. RELATED WORK

There are many different cloud computing platforms in use today. Amazon runs a cloud of RedHat servers to provide computing resources [3], which are similar in purpose to Seattle but provide a virtual machine instead of a programming language instance. This leads to better performance, but is not flexible to support donated resources and is not free. Their storage back end is functionally similar to the global data store proposed as an assignment on Seattle's educator portal.

Microsoft proposes a software plus services [16] architecture where the cloud is used as an auxiliary to enhance the capabilities of local software. While Microsoft has announced the pending availability of a developer toolkit, to the best of our knowledge it is not available at this time.

Google provides a cloud computing-like infrastructure with AppEngine [11], which executes programs written in a constrained version of the Python language and supports high level abstractions (like a global database). While useful for building locality-oblivious web applications, its transparent handling of scalability and locality makes it unsuitable for teaching these fundamental distributed systems topics.

IBM has announced plans for a cloud computing product called "Blue Cloud" [8], which supports OS images using Xen and PowerVM Linux. Blue Cloud also supports Hadoop [13] for MapReduce-type query processing and is intended to support high performance computing. Hadoop has also been used to teach cluster computing for large data processing [15]. The MapReduce [9] paradigm used by Hadoop simplifies distributed data processing. However, this simplifying abstraction also limits the scope of systems concepts that may be taught with Hadoop. We believe that a more general platform should be used to teach the system concepts that power Hadoop's implementation. These concepts may then be applied more broadly to other distributed computing abstractions, and cloud computing more generally.

In addition to cloud computing, there are a variety of grid computing and volunteer computing platforms. Globus [10] is a popular Grid toolkit, which has been used to build a variety of service-oriented applications. BOINC [5] is a volunteer computing platform supporting the SETI@Home and Folding@Home projects. BOINC leverages donated CPU cycles for computation, particularly spare resources on home machines. Globus and BOINC both target distributed computation and strive to hide locality and similar information from the programmer. Seattle is lighter-weight software that exposes locality and is therefore suited for students in distributed systems courses. Seattle also comes with a widely accessible ready-to-use platform of thousands of machines. We are not aware of any Globus- or BOINC-powered testbeds available for educational use.

Limited compute resources have been a key constraint in teaching distributed systems [21]. Seattle is architected to do this safely and efficiently. Recent efforts engaged undergraduates in distributed computing with simple-to-use platforms designed for cluster computing. The DCEZ platform offers a simple interface that students can use without any prior knowledge of distributed computing [17]. We have likewise endeavored to make Seattle simple to use, but target teaching of distributed systems issues that arise at Internet scales. Lastly, because Seattle runs on a variety of embedded platforms with limited resources, such as cellular phones and PDAs, our platform complements prior work on platforms for teaching ubiquitous computing [7].

6. CONCLUSION AND FUTURE WORK

This work presents the educational networking platform Seattle. Seattle is a free, portable, and lightweight platform using donated computational resources. Seattle enables students to learn the concepts of networking and distributed systems on computers spread around the Internet. Seattle can also emulate cloud computing, peer-to-peer computing, and cluster computing within a simple framework. Computers running Seattle are protected from malicious and misbehaving code, making it safe to contribute resources from multi-use computers. Seattle has resources available for students to use on about a thousand computers worldwide.

We are currently working to extend Seattle to improve Seattle's NAT traversal and to provide better aggregate restrictions on Seattle traffic.

7. REFERENCES

[1] J. Albrecht, R. Braud, D. Dao, N. Topilski, C. Tuttle, A. C. Snoeren, and A. Vahdat. Remote Control: Distributed Application Configuration, Management, and Visualization with Plush. In *Proc. 21th Systems Administration Conference (LISA '07)*, Dallas, TX, Nov 2007.

[2] PlanetLab All-Pairs Ping Data. `http://ping.ececs.uc.edu/ping/`.

[3] Amazon EC2 - Amazon Web Services @ Amazon.com. `http://aws.amazon.com/ec2`.

[4] Amazon S3 - Amazon Web Services @ Amazon.com. `http://aws.amazon.com/s3`.

[5] BOINC. `http://boinc.berkeley.edu/`.

[6] J. Cappos and J. Hartman. Why It Is Hard to Build a Long Running Service on Planetlab. In *Proc. of the 2nd Workshop on Real, Large Distributed Systems*, San Francisco, CA, Dec 2005.

[7] S. Chandra. Beacond: A Peer-to-Peer System to Teach Ubiquitous Computing. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 257–261, New York, NY, USA, 2003. ACM.

[8] M. Darcy. IBM Press room - 2007-11-15 IBM Introduces Ready-to-Use Cloud Computing. `http://www-03.ibm.com/press/us/en/pressrelease/22613.wss`.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[10] I. T. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In H. Jin, D. A. Reed, and W. Jiang, editors, *NPC*, volume 3779 of *Lecture Notes in Computer Science*, pages 2–13. Springer, 2005.

[11] Google App Engine - Google Code. `http://code.google.com/appengine/`.

[12] Welcome to Google Docs. `http://docs.google.com/`.

[13] Welcome to Hadoop! `http://hadoop.apache.org/core/`.

[14] HP Press Kit: HP, Intel and Yahoo! Create Global Cloud Computing Research Test Bed. `http://www.hp.com/hpinfo/newsroom/press_kits/2008/cloudresearch/index.html`.

[15] A. Kimball, S. Michels-Slettvet, and C. Bisciglia. Cluster Computing for Web-Scale Data Processing. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 116–120, New York, NY, USA, 2008. ACM.

[16] M. O'Gara. "Cloud Computing Is the Plan" – Ballmer Memo. `http://wireless.sys-con.com/node/618924`.

[17] C. Pheatt. An Easy to Use Distributed Computing Framework. *SIGCSE Bull.*, 39(1):571–575, 2007.

[18] PlanetLab. `http://www.planet-lab.org`.

[19] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A Public DHT Service and Its Uses. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 73–84, New York, NY, USA, 2005. ACM.

[20] C. Shannon. Another breadth-first approach to CS I using python. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 248–251, New York, NY, USA, 2003. ACM.

[21] C. Stewart. Distributed Systems in the Undergraduate Curriculum. *SIGCSE Bull.*, 26(4):17–20, 1994.

[22] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Peer-to-Peer Lookup Service for Internet Applications. In *Proc. SIGCOMM 2001*, pages 149–160, San Diego, CA, Aug 2001.

APPENDIX

The following Seattle program measures network latency to a list of IP addresses and displays a webpage showing the latency to each node. This program has 30 lines of code, with 11 lines of comments. Without the webpage functionality the program is only 21 lines of code. This is a complete Seattle program — no code is omitted or abbreviated.

```
# send a probe message to each neighbor
def probe_neighbors(port):
  for neighborip in mycontext["neighborlist"]:
    mycontext['sendtime'][neighborip] = getruntime()
    sendmess(neighborip, port, 'ping',getmyip(),port)
  # probe again in 10 seconds
  settimer(10,probe_neighbors,(port,))

# handle an incoming message
def got_message(srcip,srcport,mess,ch):
  if mess == 'ping':
    sendmess(srcip,srcport,'pong')
  elif mess == 'pong':
    # elapsed time is now - time when I sent the ping
    mycontext['latency'][srcip] = getruntime() - \
                          mycontext['sendtime'][srcip]

# display a web page with the latency information
def show_status(srcip,srcport,connobj, ch, mainch):
  connobj.send("<html><head><title>Latency Information</title>"+ \
           "</head><body><h1>Latency information from "+ \
           getmyip()+' </h1><table border="1">')
  # list a row for each node we are talking to
  for neighborip in mycontext['neighborlist']:
    if neighborip in mycontext['latency']:
      connobj.send("<tr><td>"+neighborip+"</td><td>"+ \
           str(mycontext['latency'][neighborip])+"<td></tr>")
    else:
      connobj.send("<tr><td>"+neighborip+"</td><td>Unknown<td></tr>")
  connobj.send("</table></html>")
  connobj.close()

if callfunc == 'initialize':
  # this holds the response information (i.e. when nodes responded)
  mycontext['latency'] = {}
  # this remembers when we sent a probe
  mycontext['sendtime'] = {}
  # get the nodes to probe
  mycontext['neighborlist'] = []
  for line in file("neighboriplist.txt"):
    mycontext['neighborlist'].append(line.strip())
  # call gotmessage whenever receiving a message
  pingport = int(callargs[0])
  recvmess(getmyip(),pingport,got_message)

  probe_neighbors(pingport)

  # register a function to show a status webpage
  pageport = int(callargs[1])
  waitforconn(getmyip(),pageport,show_status)
```