Intelligent Simulation of Worldwide Application Distribution for OnLive's Server Network

Sam Jaffe, Thinh Nguyen, Brendan Stephen

March 6, 2014

A Major Qualifying Project Report:
submitted to the Faculty of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
by

Sam Jaffe

Thinh Nguyen

Brendan Stephen

Date: March 2014

Approved:

Professor David Finkel, Advisor

Professor Mark Claypool, Advisor

Abstract

OnLive is a cloud-based video game streaming service. As part of their service, OnLive must distribute game content to their servers, but lacks the ability to judge the effectiveness of the way they have distributed the content. We seek to build a simulation framework with which OnLive can evaluate the effectiveness of different application distribution strategies. In order to do this, we built a model of OnLive's service and re-implemented their Intelligent App Distribution algorithm into our simulation. Using a genetic algorithm, we were able to programmatically construct new application distribution ratios that met demand equally as well as OnLive's current ratios did while saving a large amount of disk space. Through our simulation, we were able to determine that OnLive is able to meet current user demand while reducing the average amount of space used on their servers by over 80%, and can handle much higher levels of demand with lower disk space usage as well.

Acknowledgements

The completion of this project would not have been possible without the assistance of individuals who dedicated their time and talents in making this project a success. First, we would thank OnLive for presenting us with this opportunity to learn and interact in a real business environment. We are thankful to Arnold de Leon, John Speck, Rosemarie Carbone, Corey Ziegler Hunts, Paulo Ang, and the many other wonderful people at OnLive we were able to work with as a part of this project. We would also like to thank our advisors, Professors David Finkel and Mark Claypool, for their outstanding support throughout the project.

Contents

Table of Figures

Table of Tables

1 Introduction

This section gives a short introduction about OnLive and their internal infrastructure. Included are our project's goals, motivations, and objectives.

1.1 OnLive Overview

OnLive, the sponsor of this project, is a company in the cloud-computing field of technology. They offer a number of products including: OnLive Game Service, a video game streaming service, and OnLive Desktop, a desktop running Windows (Bode, 2010). OnLive also introduced two new services on March 5th: CloudLift, a complimentary gaming service that allows one to upload their downloaded games to the cloud to play anywhere from services such as Steam, and SLGo, a platform for players of Second Life to play entirely in the cloud and on all their devices connected to the internet.

The OnLive Game Service is the company's flagship service. The concept is to allow gamers to play computer games entirely in the cloud eliminating the need to upgrade personal hardware components. OnLive installs all of the game content onto their servers and provides software to stream games onto users' devices. Customers of the service download only the OnLive application and play games of interest without downloading them. OnLive serves desktops, laptops, tablets, and other mobile devices in a cross-platform experience for the user. One is able to play games across all of their devices and pick up where they last left off. All a user needs is an Internet connection.

In addition to their cloud gaming service, the company provides OnLive Desktop, a service that allows users to run Windows on their mobile device. Similar to gaming service, the operating system content is stored on the cloud and delivered to users' devices through OnLive

Desktop software. Currently, the software is available on iPad and Android tablets. The Windows operating system version is Windows Server 2008.

1.2 OnLive Network Infrastructure / Environment

To service their customers, OnLive has built a backend infrastructure around the world. The network infrastructure at OnLive begins with the following terminologies: region, site, segment, and apphosts (which are equivalent to a server), as depicted in Figure 1. A region is a cluster of sites based on geographical location. OnLive currently has two regions, one in the United States, and one in the United Kingdom. A site is the location of each server farm within a region. OnLive currently has five sites in the United States region: in California, Texas, Virginia, Illinois and Georgia (Grant, 2009). OnLive recommends that users be physically located within 1,000 miles of a site in order to receive the best possible user experience (Goldman, 2009). At each site, the servers are separated into segments, which are a cluster of apphosts. Finally, at the smallest size, an apphost is a dedicated server capable of serving a single user at any given time. Each apphost contains a common application named the GSP (Game Service Portal) that is in charge of serving the welcome screen when users first connect to the gaming service. Each apphost contains a different subset of games and applications as will be explained why later.

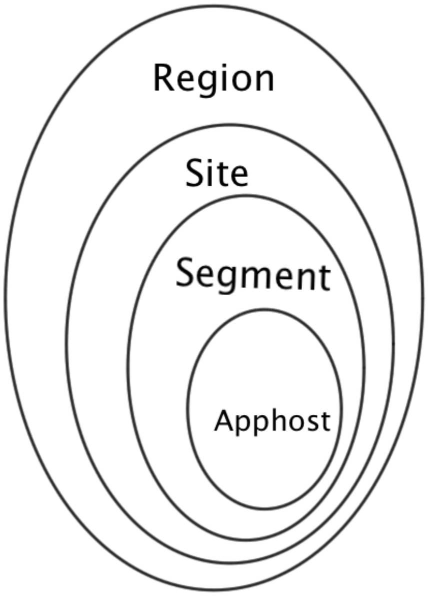


Figure 1: Hierarchy of OnLive's Server Network

The usual scenario when a user first connects to the OnLive service can be depicted as the interactions between a region, site, and apphost. When a user opens the OnLive software, the user is connected to the closest site, as shown in Figure 2. Then, the user will be directed to an apphost with a welcome application named Game Service Portal (GSP), which presents a game menu. Once a user connects to it, the apphost becomes busy and is unavailable to anyone else for use. After a user chooses a game to play, the system will identify an apphost containing the requested game and switch the session to that apphost without revealing the switch to the user. This switch happens seamlessly. After the user is done playing the game, the user can choose to request another game, or quit the service.

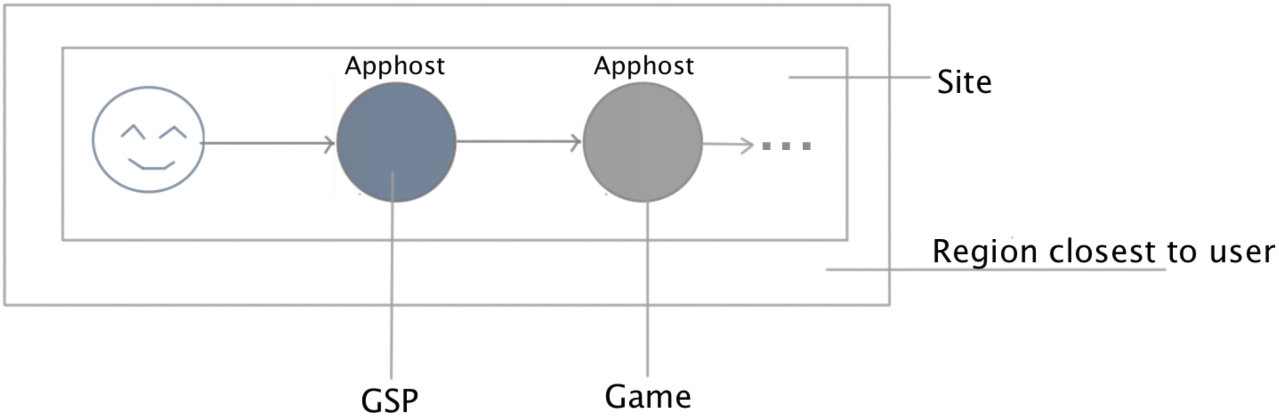


Figure 2: A successful user game request

There is also a failure state that can occur that does not allow a user to play a requested game. As stated, a user chooses a game and the system switches to another apphost having the game. However, if all apphosts capable of serving the chosen game title are currently being used, then the user will not obtain an apphost and the user will be notified to try again later, shown in Figure 3.

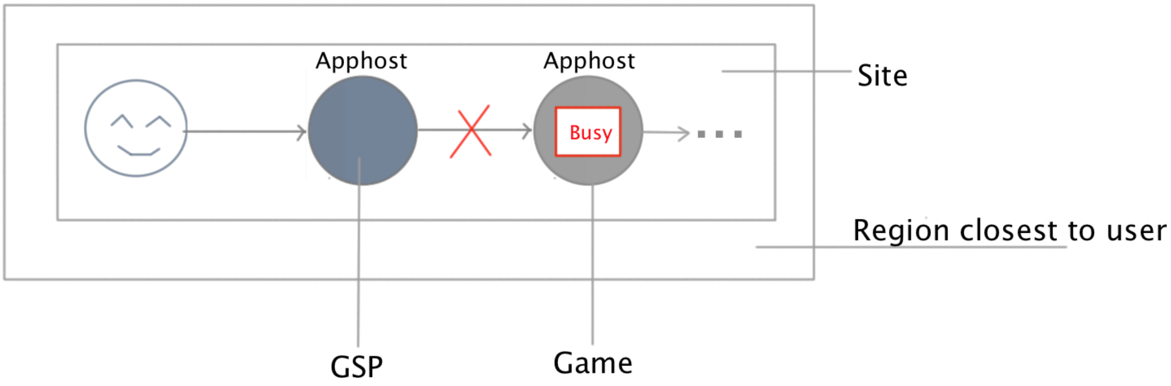


Figure 3: An unsuccessful user game request

1.3 Project Motivation

Distributing applications onto the apphosts requires the use of an OnLive specific Intelligent Application Distribution (IAD) script. This script uses an algorithm that creates a distribution of game content for a given site. This distribution says which apphosts get which games and applications then deploys this distribution to that site.

One of the largest issues with the IAD is the fact that its input, a file that says how prevalent each application should be on the apphosts must be calculated manually. Every app is given a distribution ratio that designates how much it should be present on the system. The IAD assigns games and applications to apphosts based on a target distribution ratio and distribution priority. However, the IAD cannot automatically calculate these distribution ratios. Therefore, every time OnLive needs to change the distribution of games on their apphosts, someone needs to determine the changes by hand. As a way to make the IAD simpler for human comprehension, OnLive uses a ratio system that only uses three levels (low, medium, and high). It is difficult to manually configure more degrees of granularity between ratios but would be more desirable.

Furthermore, OnLive does not have a metric to judge how successfully the IAD creates content distributions. That means that they must depend on factors like the judgment of whoever is calculating the distribution ratios and on the number of user complaints to determine if a game has too much or too little a presence. OnLive would like to have an optimization scheme that deciphers if a given distribution is more successful than another before deployment.

1.4 Objectives

Our project addressed the limitation of the current application distribution system at OnLive and provided statistical analysis for it. To do this, we built a simulation framework that calculated success of the IAD's different content distributions. This simulation framework granted the ability to test success of different distribution ratios without making changes to the actual network infrastructure. We also needed to accurately simulate application distributions on OnLive's apphosts and model users' application usage. Our simulator must behave in a comparable manner to the real-world server network. We needed to show where users were able to successfully access the game or application they desired and where the system failed to deliver content. We also intended to change content distributions by factors such as the size of the catalog available, the total disk space on the apphosts, the server's region and the time of day that the access is attempted.

Our simulator needed to present output statistics that reveal differences in configuration states. The data our simulator generated needed to be meaningful and include well-defined success criteria. The success criteria included game requests received and if all apphosts fulfilled every game request. A successful distribution will minimize the number of instances where a user was unable to use the requested application. Also, if an app is requested more often than others, it should have a higher distribution ratio and thus be loaded onto more apphosts to improve success. Optimizing the granularity between distribution ratios allowed for a more sensitive configuration that prevented too many or too few instances of a game on the servers.

Another goal for the simulation framework was to generate reports that can be provided to the Network Operations Center at OnLive. These reports should include analytics of the simulator's runs as well as the distribution(s) used. They should provide suggested modifications to actual distribution of content in production and ideally provide a function to either automatically or require only a single button press to implement changes on the actual network infrastructure.

1.5 Deliverables

Over the course of our project, we created several deliverables. At first we created the simulation framework. This included not only the code of the simulator, but also our verification tests. Included with the simulator was a graphical user interface that a user could interact with and run simulations under different parameters. We also tested all of our code in a separate test directory, and all of those tests were included as well.

With the simulator, we developed analytics to describe success of different distributions of content. We found that successful distribution of content to be ones where all users received their requested application on their first try, and all content was distributed to just enough disk space on the apphosts. Also, the number of apphosts necessary for our simulation matched how much demand the system was under. The final deliverable was the documentation for our work. Our code is well documented as are our tests. To use our software we developed several readme guides for future developers.

1.6 Report Roadmap

We discuss our preliminary research for our project by looking at what OnLive currently employs for content distribution. We also research past projects that have solved similar issues to gain insight into a proper process to be successful. Our process for development discusses how our team divided work as well as details the many milestones of the project from start to completion. The main results of our project lists information about our simulator as well as the verification tests performed on it. We also compare results from our new distribution of content against the old distribution currently employed at OnLive.

2 Background Research

We investigated the tools, algorithms, and processes that our project required before starting. To understand how OnLive currently distributes content, we researched the tools and software used as well as the algorithms employed. To decide how our group was to complete this project, we researched related work that solved a problem similar to ours in the past.

2.1 Intelligent Application Distribution (IAD)

The Intelligent Application Distribution script is written in Python and is the main distribution system OnLive employs. All games and applications are distributed based on a distribution ratio that says how prevalent that game or application should be on the apphosts.

2.1.1 How the IAD works

Every app host has its own disk space and hardware type. Disk space on any given apphost is less than the space needed to store all the game content in OnLive's catalog. Therefore, each apphost can only store a limited number of games. Two potential solutions to this problem are to use common storage for all the app hosts or an app distribution system that divides out the games onto each apphost. When evaluating the cost of both solutions, OnLive decided to go with the second solution, and they called it the Intelligent Application Distribution (IAD).

To explain how the IAD works, one thinks of an apphost as a bin and every game or application as a package. Each package is different in size and each bin has its own capacity. The IAD's mission is to distribute all the packages efficiently into the every bin. However, at the same time, the IAD must carefully consider its distribution mechanism. Since when a user connects to an apphost, another user cannot obtain it. If the application a user needs is only on an occupied apphost, that user cannot connect to their requested game or application. This scenario is problematic because it is possible that app hosts are available, but this user cannot use them because the app itself is unavailable.

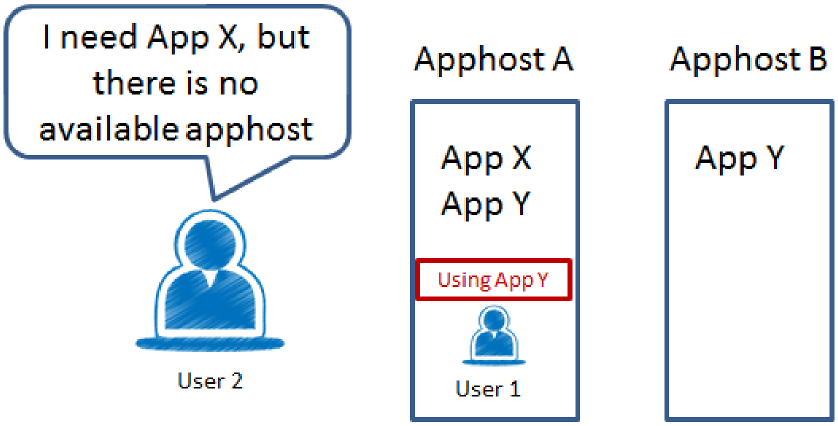


Figure 4: Example of a current IAD limitation

In Figure 4, User 1first requests to play App Y and was given the first apphost with that app at Apphost A. When User 2 then connects to OnLive wishing to play App X, he is unable to complete his request. Even though Apphost B is available, User 2 cannot be served because the distribution of applications put App X only onto Apphost A. To solve this issue, one can argue that optimizing the separate mechanism that assigns users to appropriate apphosts to be the best course of action. However, that option is unlikely to succeed because that would not solve the issue of assigning enough applications to apphosts based on popularity. In this way, the issue in Figure 4 would not happen because App Y would be distributed to enough servers to handle the demand and a user would statistically never request an app when a server does not have it available. Therefore, our project focuses on investigating how a wise mechanism to distribute games and applications can help solve this problem.

When the IAD creates a content distribution, it takes into account attributes such as the distribution ratio assigned to a game, the disk space it uses, different variants of the title, as well as the hardware type of each apphost. Based on these attributes, the distribution ratios will adjust to minimize the number games and applications unable to fit on the apphosts. Unfortunately, the IAD cannot adjust these fields automatically; the distribution ratios are input manually by the network operations team at OnLive.

2.1.2 The IAD's Bin Packing Algorithm

The way the IAD works and how OnLive needs to be able to distribute applications to its apphosts is similar to a bin-packing problem. A bin-packing problem's goal is to find a way to pack every object in to a number of bins of limited size. Each object to be placed in the bins has a set size. While the bin-packing problem itself is NP-hard, there exist algorithms to compute close to optimal solutions in O(n·log(n)) time (Johnson, 1973, page 2).

However, the IAD has a number of constraints on it that make it distinct from the bin-packing problem. One factor is that our goal is not to place every application on an apphost, but rather to minimize the number of applications that need to be distributed out to meet demand. Additionally, an apphost can only take a given application once, and there may be as many copies of an object as there are apphosts to distribute to, whereas there is no such constraint in the bin-packing algorithm.

2.2 Discrete Event Simulation

We use discrete event simulation as the primary technology for our project at OnLive. We chose to use discrete event simulation in part because it would model the target system in a discrete manner, where each event happens 'instantly' at a specified time. This will allow us to observe user and apphost behavior to make predictions and adjust the model.

2.2.1 Discrete Event Simulation Description

Discrete Event Simulations are programs that are designed to emulate the behavior of a system under study. They allow us to study discrete-event dynamic systems in which delay is an intrinsic feature (Fishman, 2001). These systems can range from individual CPUs to large-scale networks (Jacob, 2013). A simulation will keep track of the state of the simulated system, allowing data to be measured. One of the primary features of discrete event simulation is that all changes in state occur ‘instantaneously’ within the simulation. An event is at this instance in time where the state changes.

While Discrete Event Simulation concepts and designs have existed for around 50 years, the rapid growth of the PC industry has enabled many simulation techniques to rapidly move from concept to practical use (Fishman, 2001).

In general, every discrete event system expresses several concepts (Fishman, 2001):

- Work, which denotes the objects that enter the system in need of services
- Resources, which represent objects that can provide the services that are needed
- Routing, which delineates the collection of required services, the resources that will provide them, and the order in which those services are performed
- Buffers or queues of work, depending on the model, that have an infinite or a finite capacity. Finite buffers require rules for what to do if work arrives but the buffer is full
- Scheduling is the pattern of availability of resources
- Sequencing represents the order that resources provide services to remaining work

2.2.2 Purpose of Discrete Event Simulation in our Project

Before we put discrete-event simulation into context as a tool of analysis, we describe how it relates to modeling in general. To study a system, we first accumulate knowledge to build a model. A model can be a formal representation based on theory or a detailed account based on empirical observation. According to Fishman (Fishman, 2001), it enables an investigator to organize his/her theoretical beliefs and empirical observations about a system and to deduce the logical implications of this organization. This brings into perspective the need for detail and relevance to the system that leads to improved system understanding. It is also easier to manipulate a model instead of the main system because a model expedites the speed with which an analysis can be accomplished. It also provides a framework for testing the desirability of system modifications and permits control over more sources of variation than direct study of a system allows.

Discrete-event simulation presents techniques that can approximate the values of performance descriptors to within a remarkably small error. The approximations come from running the simulation on the model of interest and analyzing data based on sample paths generated during the execution. Often, discrete-simulation is used to study the alternatives of the system. Simulation-generated data can sharpen an investigator's understanding of the system. Thereby, he can offer the alternative configurations that lead to the system's best performance with regard to explicitly stated criteria.

Discrete-event simulation can benefit the development of optimal or near-optimal policies for system management. In many situations, one builds an analytical model of a system in order to determine an optimal policy for managing the system, with regard to a specified criterion. However, these analytical models may leave out some fundamental properties of the system and end up displaying inaccurate system performance. Therefore, the analyst may have a hard time to keep track and maintain important characteristics when creating a model. Discrete-event simulation can help to resolve this issue. It allows one to build a simulation model that contains the features in question. A discrete event simulator can launch under the analytically optimal policy, and then execute under institutional management policies as well. Comparing the result helps determine the degree in which the analytically optimal policy prevails over the institutionally based policies. This joint use of analytical and institutional methods considerably enhances the value of discrete event simulators.

2.2.3 Network Discrete Event Simulation Algorithm

Network simulators are useful in the design and configuration of a computer communication network. As we build a simulator for OnLive's cloud gaming service, this algorithm was helpful to us. A network simulation consists of servers with different state variables and events that must be executed for given amounts of time in the order of a queue. After locating the state variables and the events based on both our knowledge of the system under study and the objectives of the study, we can apply the discrete event simulation algorithm to our simulation program, shown in Figure 5.

NETWORK DISCRETE-EVENT-SIMULATION ALGORITHM

```
1. Initialize the server state variables
2. Initialize the 'collection of pending events'
3. Initialize the simulation clock
4. While there are pending events to be handled:
    a. Remove the pending event (E) with the smallest timestamp (t)
    b. Set simulation clock to that time t
    c. Execute the event handler for event E
```

Figure 5: Network Discrete-Event Simulation Algorithm

As each event occurs instantly with regards to the simulation clock, the simulation can be run on any system and time delays can be introduced with state variables and rescheduling.

2.3 Related Work

We researched similar projects and studies, so that we can broaden our view and find and process for development that solved problems like this in the past. The study “Aware Virtual Machine Placement for Cloud Games” aimed to maximize the cloud gaming provider’s total profit while achieving just-good-enough Quality-of-Experience (Hong, 2013). Similar to our project, the authors built a simulation to simulate the user, server structure in cloud gaming service companies. Then, they conducted and implemented their own algorithms to study the fields of interest. Since the simulator that they built had almost identical properties as ours, we inherited some properties that they used. For example, we included properties such as gamer inter-arrival time, gamer session length, and number of servers. Furthermore, we implemented trace-driven simulation techniques mentioned in their paperwork as well for our heuristic algorithm.

3. Methodology

This chapter details the various work environment setups, tools, and practices we employed in order to complete the project. Work environment setups include descriptions of the different types of software for use on the project as well as for documentation. Tools include descriptions of software necessary to produce the code for the project. Practices employed include details regarding the development process used by the group. It also entails the timeline for the project as well as distinguished roles for group members.

3.1 Work Environment and Tools

This section depicts the various tools in our work environment as well as the many Java libraries necessary to write our software. There are software applications used by both OnLive and our project team that we document here.

3.1.1 OnLive Work Environment Setup

We used the existing technologies at the company as a starting point for our work environment. This section describes the available tools from OnLive that aided our development process.

3.1.1.1 GitHub Repository

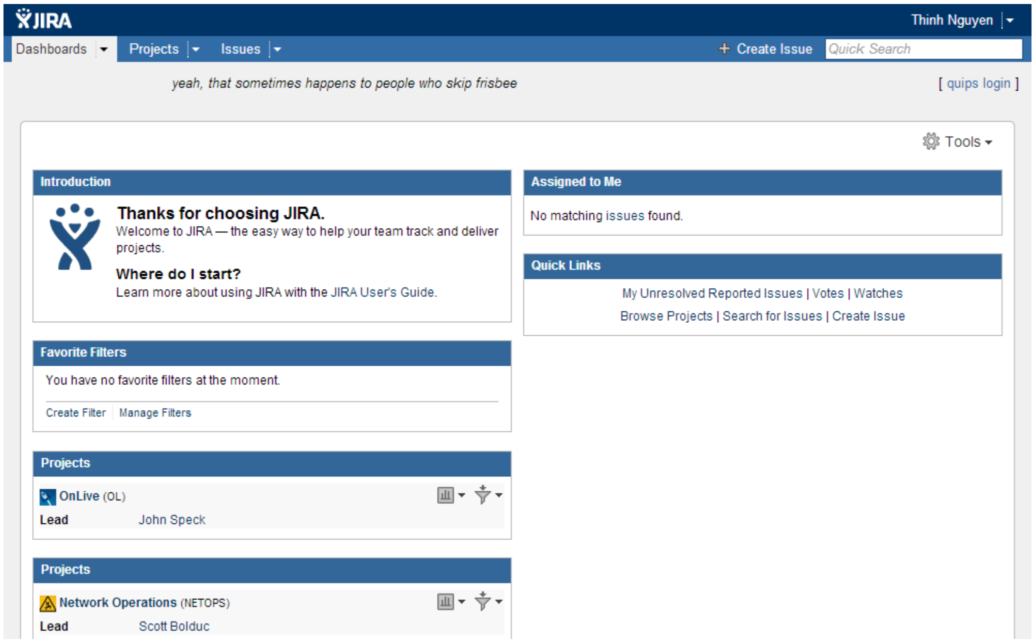
OnLive has its own version control system through GitHub (source: https://github.com/). To accommodate this environment, we developed all of our code and used Git as our repository management system. OnLive has a corporate license with GitHub through the domain name onlive.github.com, which is accessible through the company's intranet. For our project, we created a repository name "toy_sim" on that domain and used version control to push and pull from it.

Additionally, OnLive gave us full access on all of the repositories on their domain. This gave us access to the Intelligent App Distribution (IAD) code and all of the development libraries necessary for the implementation of their software. We used much of the code available to help us build our simulation software.

3.1.1.2 Jira, Pidgin, Wiki

Besides using the OnLive GitHub utility, we employed three other tools, namely Jira, Pidgin and Wiki, in our development process. These three tools helped us collect data as well as enhance our connection to other departments within OnLive.

OnLive uses a task based ticketing system called Jira (source https://www.atlassian.com/software/jira), shown in Figure 6, that tracks current development bugs and tasks for all departments. It is built on a web based framework so all company departments have easy access. For example, a department posts a status (a ticket) of a bug on Jira, and assigns the resolution to the corresponding department. The ticket is sent to the department where they can post comments and updates. When a bug is resolved, the ticket is closed and stored in the system's database with information about how it was resolved. In this way, all tasks and resolutions are documented. Our team used Jira to submit requests for data collection from other departments.



yeah, that sometimes happens to people who skip frisbee

Introduction

Thanks for choosing JIRA.

Welcome to JIRA — the easy way to help your team track and deliver projects.

Where do I start?

Learn more about using JIRA with the JIRA User's Guide.

Assigned to Me

No matching issues found.

Quick Links

My Unresolved Reported Issues | Votes | Watches

Browse Projects | Search for Issues | Create Issue

Favorite Filters

You have no favorite filters at the moment.

Create Filter | Manage Filters

Projects

OnLive (OL)

Lead John Speck

Projects

Network Operations (NETOPS)

Lead Scott Bolduc

Figure 6: Jira ticket management system

Another tool that we used at OnLive was a chat service named Pidgin (source, https://www.pidgin.im/), which is a Jabber instant messaging client, as shown in Figure 7. Pidgin is supplied through jabber.onlive.com. OnLive employees use Pidgin to form chat forums, which serve specific purposes such as answering specific topic for a given department. Seeing the benefit of this chat service, we created a chat room where our team could ask questions of the group as well as post statuses of our development process.

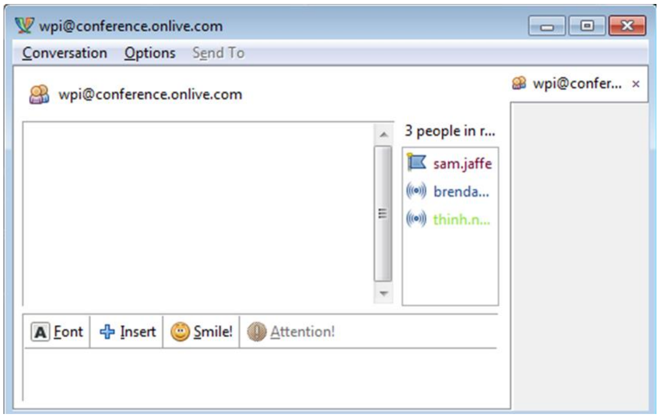
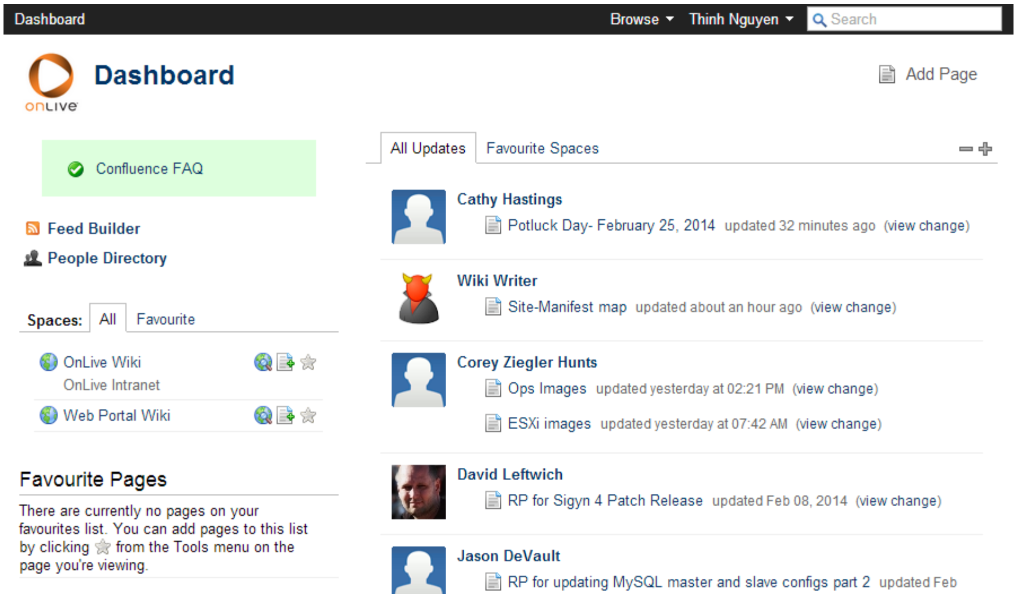


Figure 7: Pidgin chat client

Lastly, OnLive has a wiki website located at wiki.onlive.com, as shown in Figure 8. This website contains all the documentation for various procedures and software created at OnLive. Every employee at OnLive has a profile and account where they can post documentation and processes they used as well as add to posts made by others. Information about all the previous projects at OnLive, including information about the IAD, could be found on the wiki. Our team found information about the internal infrastructure of OnLive's systems as well as background information for our project.



Dashboard

OnLive Wiki
OnLive Intranet

Web Portal Wiki

Favourite Pages

There are currently no pages on your favourites list. You can add pages to this list by clicking ☆ from the Tools menu on the page you're viewing.

Cathy Hastings
Potluck Day- February 25, 2014 updated 32 minutes ago (view change)

Wiki Writer
Site-Manifest map updated about an hour ago (view change)

Corey Ziegler Hunts
Ops Images updated yesterday at 02:21 PM (view change)
ESXi images updated yesterday at 07:42 AM (view change)

David Leftwich
RP for Sigyn 4 Patch Release updated Feb 08, 2014 (view change)

Jason DeVault
RP for updating MySQL master and slave configs part 2 updated Feb

Figure 8: OnLive Wiki

3.1.2 Our Work Environment Setup

We found many other software tools necessary for our project that we included besides the environment tools OnLive set up for us. This section lists various software tools, languages, and code libraries we used throughout our project.

3.1.2.1 Simulation Library and Coding Language Choice

Preliminary research for this project focused on different software and languages that are specifically designed to handle discrete event simulation. There are a few different options that we investigated that could possibly fulfill our needs to model the OnLive network.

Siman – This general purposes simulation language incorporates special purpose features for modeling manufacturing systems. These features simplify and enhance the modeling of each component of a manufacturing system. The advantage of this language is that it works best for general purpose modeling as compared to other languages. The disadvantages of this languages use is the initial limited knowledge our project team has with it, as we have better experience with other languages. If we choose to use this simulation language, we need to purchase it for use.

Mason – (source, http://cs.gmu.edu/~eclab/projects/mason/) this is a fast discrete event multi-agent simulation library core in Java. The advantages of using Mason are that it is free, our group has better experience with Java, and its models are self-contained that can run inside other Java frameworks. The disadvantages of using Mason are that it is not specific to discrete event simulation and not optimized for it.

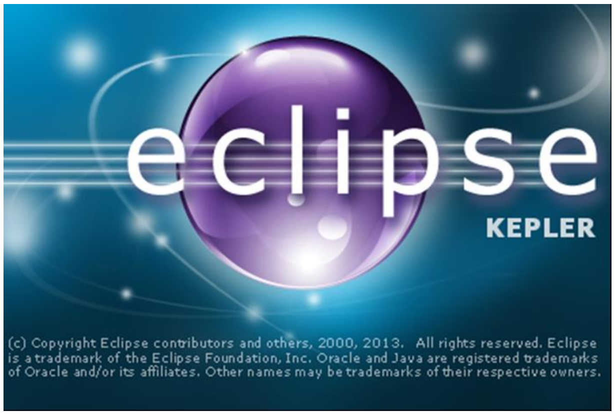
SimEvents / Simulink – (source, http://www.mathworks.com/products/simevents/) this software provides a discrete event simulation engine that manages and processes sequences of asynchronous events. These events help model mode changes and trigger state transitions within time based systems. The advantage of using SimEvents is that it offers the most robust tools and variety of options when modeling the network. These tools are all internal to the software and will not require large amounts of code. It also offers large amounts of internal analytics options that can be built with Matlab. The disadvantage of this software is that the learning curve is much greater when compared with the other two that we researched. If we choose to use this simulation software, we need to purchase it for use.

After evaluation, our team decided to choose the Mason Java library because of its extensibility and breadth of user guides and documentation for its use. Moreover, because the team's members all have extensive experience in Java, using Mason library would benefit us in development.

3.1.2.2 Integrated Development Environment (IDE) and Version Control

After selecting the Mason Simulation Library and the Java coding language for development, we found tools that would best setup a development environment.

Eclipse is an IDE tool built specifically for Java development (source, https://www.eclipse.org/). Its assistance in code correction, library importing, and project build automation, assist programmers with minor programming issues. The IDE software also has an extensive library of plug-ins making it more powerful and extendable than other IDEs. We chose to use Eclipse, shown in Figure 9, because of its features that specifically support Java development.

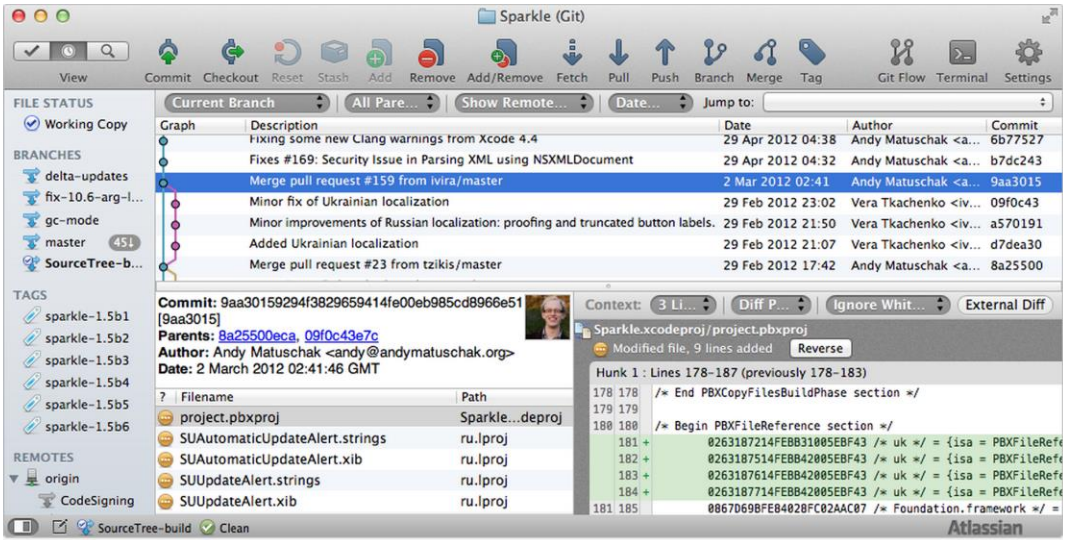


(c) Copyright Eclipse contributors and others, 2000, 2013. All rights reserved. Eclipse is a trademark of the Eclipse Foundation, Inc. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Figure 9: Eclipse splash screen

Eclipse has a built in version control feature, but we decided to use another tool that we have had more experience with handling Git repositories. We chose to use free software called

SourceTree (source, http://www.sourcetreeapp.com/), shown in Figure 10. SourceTree supports GitHub repository management as well as provides a graphical user interface for tracking development tasks and resolving conflicts. Additionally, SourceTree easily creates branches that allow programmers to create their individual coding environments that do not impede on other programmers. Using this feature, each of us worked simultaneously on our branches before merging into the master branch. This method helped us avoid accidentally changing the code others were working on.



Graph	Description	Date	Author	Commit
	Fixing some new Clang warnings from Xcode 4.4	29 Apr 2012 04:38	Andy Matuschak <a...	6b77527
	Fixes #169: Security Issue in Parsing XML using NSXMLDocument	29 Apr 2012 04:32	Andy Matuschak <a...	b7dc243
	Merge pull request #159 from ivira/master	2 Mar 2012 02:41	Andy Matuschak <a...	9aa3015
	Minor fix of Ukrainian localization	29 Feb 2012 23:02	Vera Tkachenko <iv...	09f0c43
	Minor improvements of Russian localization: proofing and truncated button labels.	29 Feb 2012 21:50	Vera Tkachenko <iv...	a570191
	Added Ukrainian localization	29 Feb 2012 21:07	Vera Tkachenko <iv...	d7dea30
	Merge pull request #23 from tzikis/master	29 Feb 2012 17:42	Andy Matuschak <a...	8a25500

Commit: 9aa30159294f3829659414fe00eb985cd8966e51 [9aa3015]
Parents: 8a25500eca, 09f0c43e7c
Author: Andy Matuschak <andy@andymatuschak.org>
Date: 2 March 2012 02:41:46 GMT

?	Filename	Path
	project.pbxproj	Sparkle...deproj
	SUAutomaticUpdateAlert.strings	ru.lproj
	SUAutomaticUpdateAlert.xib	ru.lproj
	SUUpdateAlert.strings	ru.lproj
	SUUpdateAlert.xib	ru.lproj

Sparkle.xcodeproj/project.pbxproj
Modified file, 9 lines added

```
Hunk 1 : Lines 178-187 (previously 178-183)
178 178   /* End PBXCopyFilesBuildPhase section */
179 179
180 180   /* Begin PBXFileReference section */
    181 +         0263187214FEBB31005EBF43 /* uk */ = {isa = PBXFileRefe
    182 +         0263187514FEBB42005EBF43 /* uk */ = {isa = PBXFileRefe
    183 +         0263187614FEBB42005EBF43 /* uk */ = {isa = PBXFileRefe
    184 +         0263187714FEBB42005EBF43 /* uk */ = {isa = PBXFileRefe
181 185           0867D69BFE84028FC02AAC07 /* Foundation.framework */ =
```

Figure 10: SourceTree graphical user interface

3.1.2.3 Documentation

Thorough documentation of our project is essential to hand over and transition our software development project after we leave. Having well documented software not only assists OnLive to understand the code better but also allows them to extend it later on. To document our code, we used Javadoc because it supports internal comments and builds a website from the comments. To show the Java classes' relationship, we used software called Visual Paradigm for UML (source, http://www.visual-paradigm.com/product/vpuml/), shown in Figure 11. This software scans our code and produces class diagrams, sequence diagrams, and package diagrams. Visual paradigm will help others at OnLive understand our project structure and behavior.

Build Quality Applications Faster, Better and Cheaper
Visual Paradigm
for UML
Standard Edition
Initializing Environment...
Ver. 11.0(20140110)
Copyright © 1999 - 2013 Visual Paradigm International. All rights reserved.
Visual Paradigm

Figure 11: Visual Paradigm for UML splash screen

3.1.2.4 Statistical Modeling Software

During development we needed the ability to estimate the best statistical distribution to a given set of data. We needed professional statistical estimation software to ensure our distributions that describe session length, inter-arrival time, and popularity of applications was the best distributions we could find. This was a crucial step in our verification process. We chose to use free-to-try software called EasyFit (source, http://www.mathwave.com/). This software from MathWave Technologies took a sample set of 5000 points of data and compared its distribution to over 60 different distribution types. It also gave a summary of its calculated best distribution parameters based on goodness of fit tests. The goodness of fit tests included a Kolmogorov Smirnov, Anderson Darling, and Chi-Squared tests that ranked the distributions that best fit these three tests.

3.1.3 Java Libraries

Several of the Java libraries that we used in order to be able to complete the project are libraries that OnLive itself uses in development. The Jackson library (source, https://github.com/FasterXML/jackson) is a JSON parsing library that is designed with efficiency in mind. JSON, which stands for JavaScript Object Notation, is a low-overhead datainterchange format. OnLive' s Application Distribution Manifest files are written in JSON formats, so in order to be able to copy the IAD's behavior we needed to be able to parse them. Snakeyaml (source, http://code.google.com/p/snakeyaml/) is a YAML (YAML Ain't Markup Language) parser, which provides human-readable data serialization. The primary configuration files for the IAD are written in a YAML format. Additionally, the IAD needs to be able to connect to a MySQL database to query information such as the list of available apphosts and app variants on a site. Log4J is a Java logging API produced by the Apache Software Foundation that makes it easy to track where our program is and where any errors originate.

Other libraries include JSAT (source, http://code.google.com/p/java-statistical-analysis-tool/), or Java Statistical Analysis Tool, which allowed us to compute the distributions of our data efficiently. We also used GNU Trove (source, http://trove.starlight-systems.com), which provides list and map objects for operating with primitive types, such as int or double, directly, which is not a feature that is available in Java's Collection or Map classes.

3.2 Division of Labor and Software Engineering Practices

After arriving at OnLive, we began developing a basic event simulator; we called it "toy_sim". This basic simulator allowed us to delve into the Mason library and test its features before starting official development on the full simulator. To do this, we needed to define roles each team member can fulfill. Based on our individual backgrounds and experience, we created three roles: a main programmer, a main tester, and a project manager. Each role fulfilled a different piece of project development. We organized ourselves into a development process similar to the V-Model of software development. The V-model is a commonly used management practice employed by software development program managers as shown in Figure 12 (source, http://istqbexamcertification.com/what-is-v-model-advantages-disadvantages-and-when-to-use-it/). This software development model separated development and testing of a piece of code into two separate paths. While true implementation of the V-Model places the entire project on this course, we implemented the V-Model on a per feature level. This means that each feature built into the simulator would have an independent V-Model implementation where programming, development, and verification were handled separately. This process allowed our group to be agile and adapt to change. The main programmer was responsible for the development leg, the main tester was responsible for the testing leg, and the project manager ensured the proper levels of verification at all levels.

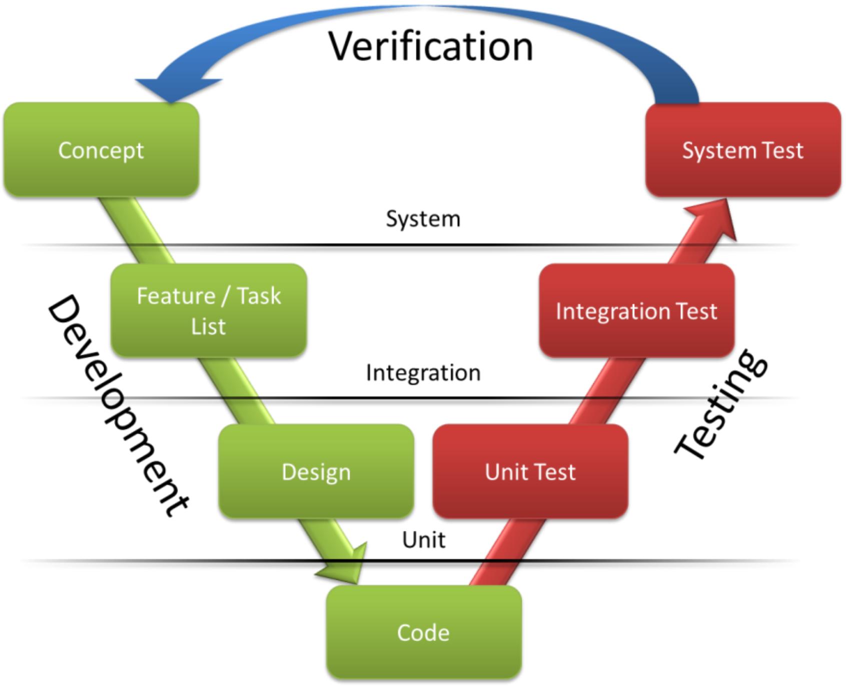


Figure 12: V-Model for software development

3.2.1 Main Programmer Role

The Main Programmer is responsible for the development of code. He is best suited for the responsibilities involved with the production of our software. He has an extensive background with best practices in Java development and is able to implement robust program structure quickly.

3.2.1.1 Development Process

As mentioned above, our development process resembles that of the V-Model development process with a focus on working on individual features or components at a time. As part of this, we would first determine the feature of our simulator that we most needed to work on next, and then applied the V-Model process to that singular part of our overall objective. Having determined our target feature, we would determine the necessary behaviors or sub-features that would need to be implemented in order to complete the development. Once we had determined the necessary steps to complete the chosen feature, we would then code it and begin testing. Because of the division of roles that we had, other features could be developed while the completed feature underwent testing.

Because of our feature-driven V-Model process, changes in the direction of the project would only cause minor inconveniences. This is primarily because one of our goals with this development pattern was to enforce a high level of modularity on the system. This allowed us to easily replace behaviors without having to directly modify any of the actual code of the simulation body. For instance, in one case, our target moved from writing a simulator that would leverage OnLive's IAD system to generate accurate distributions to implementing and improving our own copy of the IAD in Java. Because we had designed our code to be highly modular, once the IAD code had been ported into Java, linking it to our simulation was trivial.

3.2.2 Main Tester Role

The Main Tester is responsible for the tests that evaluate the code produced by the Programmer. He has knowledge with testing procedures and implements testing processes to ensure our software is robust. The tester process includes the following steps; code coverage, unit test, and integration test. Code coverage ensures that all of the code has been run at least once in the program. Unit Test verifies all possible input and output variables for a given Java class does not break the code. Integration Test evaluates that behaviors between all classes and objects are correct.

3.2.2.1 Testing Process

The testing practice that we used followed the guidelines from Osherove's, *The Art of Unit Testing* where it is recommends to creating a test class for each class of interest (Osherove, 2009, Chapter 2). Also, we aimed to cover each individual logic statement such as "if" and "while" to make sure the logic statement serves its intended purpose. Additionally, this person tested the interaction between classes or interfaces. To make sure the interaction behaves correctly, we wrote a separate test class to test the interaction. This kind of testing is called integration test. We used stub and mock objects to fake the behavior of real objects by plugging them into the test object of interest. While the stub objects usually contain trivial operations, the mock object may carry class fields to see how it interacts with other class fields (Osherove, 2009, Chapter 2).

Because test cases must be maintainable and extendable, we implemented design pattern principles into our test cases. We employed three designs patterns that commonly appear in the testing world: an abstract test infrastructure class pattern, template test class pattern, and abstract test driver class pattern as described in Figure 13 and Figure 14. Using these patterns, we effectively conserved development time, maintained the code, and enhanced its readability (Osherove, 2009, Chapter 6).

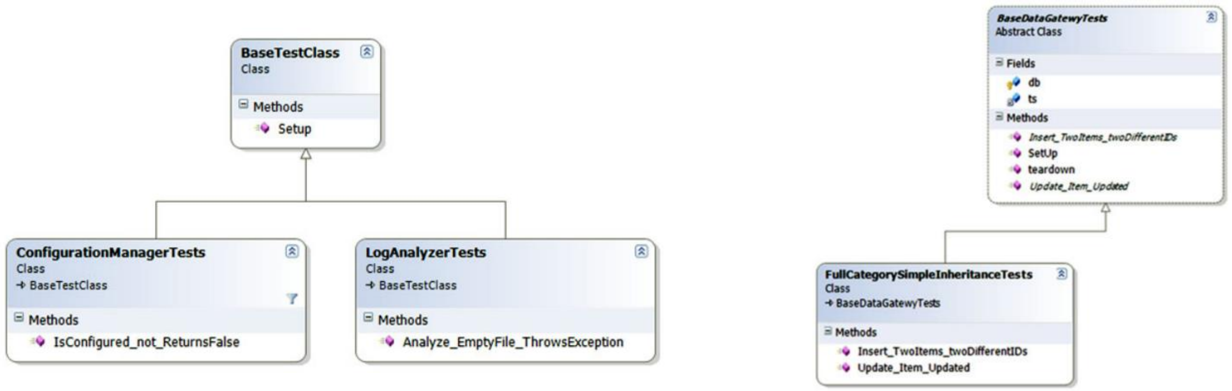


Figure 13: Abstract test infrastructure pattern (left) and template test class pattern (right)

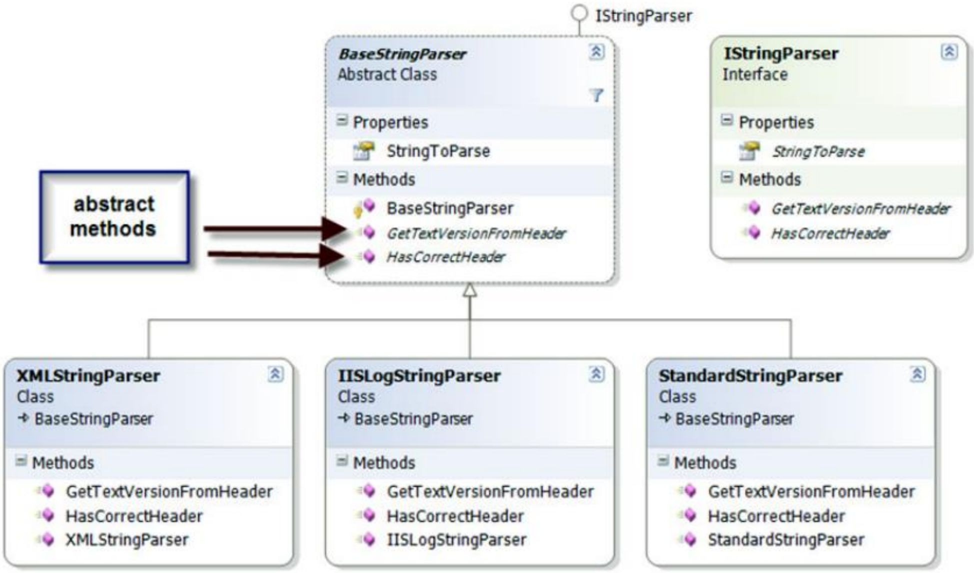


Figure 14: Example of an abstract test driver class pattern

3.2.3 Program Manager Role

The Program Manager is responsible for the continued progression of the project and makes the decisions for when and how next steps of the project are implemented. This individual was the main contact person for others in OnLive, set up meetings, writes agendas and emails, as well as developed presentations and report documentation.

As a program manager, he ensures that both the Programmer and Tester have all the required materials and the help they need to ensure that there are no tasks taking too much time to complete. To accomplish this, he develops tracking documentation that measures the current status of the project at all times.

3.2.3.1 Verification Process for Development

Tracking documentation for the program development begins with a list of tasks or requirements that the Programmer must complete to finish the project. A full listing of the tasks generated can be found at Appendix B: Software Development Tracking. These tasks are created in conjunction with the Programmer to outline the entire project from start to finish. If any task is out of the project scope, but may be included with an increase in scope, it becomes project growth and does not factor into the current project projection. Every task is then assigned a difficulty weight or work unit from one to five, with one being extremely easy and completed in a short amount of time and a five being extremely difficult and taking a large amount of time. Each task is then set to “Not Started”. As the programmer codes, he selects a task to accomplish, assigns himself to it, writes the start date of the task, and changes the state of the task to in progress. After the programmer finishes a task, he writes the completion date of the task and changes the state of the task to complete.

The tracking document takes the raw information provided by tasks and creates analytics that follows the progression of development. The weightings of all tasks are added up and divided by the number of days the project will run. This creates an expected progress rate of a certain amount of work to be completed each day. Each day this unit of work is deducted from the total number of units and creates a burn down rate. This rate is compared to the actual rate of completion for each day. When a task is completed, the amount of work units it had is deducted from the total number of remaining units on the project. This comparison can be plotted on a burn down chart. Figure 15 is an example of our burn down chart.

The burn down chart can be used to visualize current development trends and predict when a project will end. The expected progress (or the total number of weightings distributed equally each day to the end of the project, in green below), is the target development rate. To ensure a project is on course, it is important that actual progress be relatively close to expected progress. If the actual progress goes above the expected progress line too much, it means the project is too difficult and either more people need to work on it, or tasks need to be removed. Conversely, if the actual progress is too far below the expected progress line, the project is ahead of schedule and growth tasks need to be added or reduce the amount of people on the project.

To track how much work is done each day, the blue bars represent the cumulative amount of tasks completed at a given date. This data describes when there are lulls or increases in productivity.

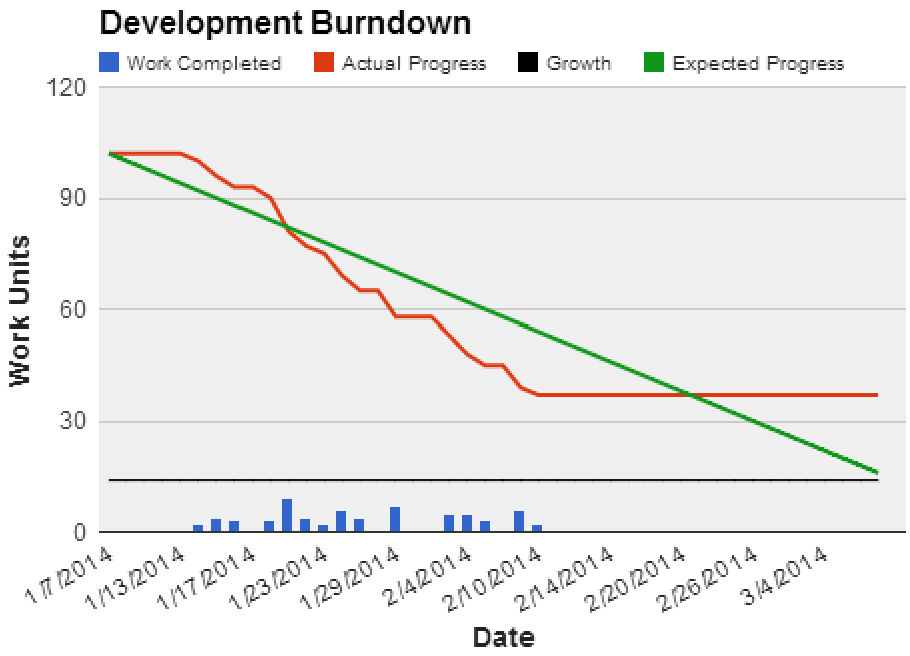


Figure 15: Burn down chart example

3.2.3.2 Verification Process for Testing

Tracking documentation for testing progress is similar to development tracking. Instead of a burn down chart, we chose to use a chart that tracks up how much code has been covered. To do this, the Tester develops a list of Java classes or modules that need to be tested. A full list of these classes and the test classes used can be found at Appendix C: Test Coverage Tracking. These modules are listed with a current percent coverage of 0%. To start testing, the tester assigns him to the modules he wishes to test and inputs a start date. As he develops JUnit test cases for the module, he attaches the JUnit test names to the module and increases the code coverage on the entire module that the JUnit test covers. In this way, the percent covered each day goes up.

To track progress, a goal code coverage percentage for all code must be selected. Our team decided on 90% code coverage. This goal code coverage value is the goal coverage for other software development projects of relative scale. This is a good choice of coverage as on average, the percentage of code that is untestable on a project of this scale is 10%. To get 90% coverage over the project's duration, there is an expected coverage percentage per day that must be attained. This completion per day can be plotted against the actual code coverage completion each day on a graph like in Figure 16. The blue bars show how much work was completed each day of the testing progress. This shows overall productivity each day.

It is important to keep the actual progress and expected progress lines close together. If the actual progress line goes above the expected line, then one is ahead of schedule. Conversely, if under the expected line, the actual progress is behind schedule. We used this method to predict our completion dates as well as be to increase awareness of our current testing rate.

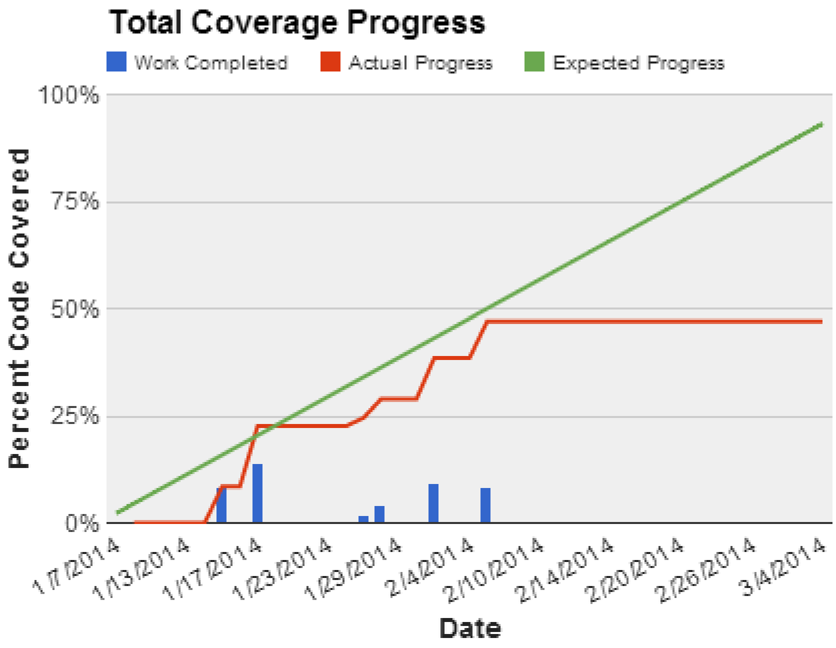


Figure 16: Coverage progress chart example

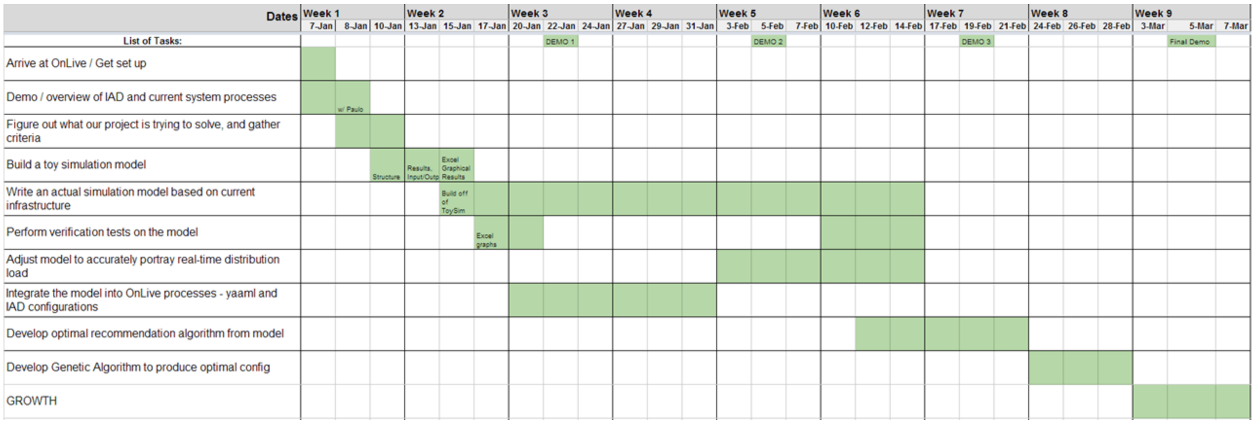
3.2.3.3 Error Logging

As development progressed, both the tester and the program manager found errors in the code. To resolve bugs in the software, a document to track errors was created. The document requested information about the bug to help the program developer resolve issues. Information required was the module and method with the error, which (if applicable) test found the error, what exactly the expected result was vs. the actual result, and what was necessary to be changed. This information could be filled out at any time and it will be submitted to the program developer where he can resolve at his own pace. After an error was resolved, the program manager and tester would verify the change was successful.

3.3 Timetable for Project

The project timetable was developed to establish project checkpoints and important dates in production. The group produced a Gant Chart that outlined when a task should be started and completed for the entire length of the project. This Gant Chart, shown in Figure 17, includes several project demos that provided opportunities for individuals invested in the project at OnLive to see our progress and provide feedback early in development. Also included in the timetable was space for growth. Growth is a segment of time that can be taken if more tasks are added to the project without affecting the deadline of the project. We used this time just in case there were any issues along the way in development.

In each row of the first column in the Gant Chart in Figure 17 is a task from the task listing. Each column represents a date of the project from start on the left, to end on the right. When a task must be worked on at a given date, the cell intersection is colored green. These cells show over the course of the project timeline what tasks are being worked on.



Dates	Week 1			Week 2			Week 3			Week 4			Week 5			Week 6			Week 7			Week 8			Week 9		
	7-Jan	8-Jan	10-Jan	13-Jan	15-Jan	17-Jan	20-Jan	22-Jan	24-Jan	27-Jan	29-Jan	31-Jan	3-Feb	5-Feb	7-Feb	10-Feb	12-Feb	14-Feb	17-Feb	19-Feb	21-Feb	24-Feb	26-Feb	28-Feb	3-Mar	5-Mar	7-Mar
List of Tasks:								DEMO 1						DEMO 2						DEMO 3						Final Demo	
Arrive at OnLive / Get set up	🟩																										
Demo / overview of IAD and current system processes	🟩	w/ Paulo																									
Figure out what our project is trying to solve, and gather criteria		🟩	🟩																								
Build a toy simulation model			Structure	Results, Input/Outp	Excel Graphical Results																						
Write an actual simulation model based on current infrastructure					Build off of ToySim	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩									
Perform verification tests on the model						Excel graphs	🟩									🟩	🟩	🟩									
Adjust model to accurately portray real-time distribution load													🟩	🟩	🟩	🟩	🟩	🟩									
Integrate the model into OnLive processes - yaaml and IAD configurations							🟩	🟩	🟩	🟩	🟩	🟩															
Develop optimal recommendation algorithm from model																	🟩	🟩	🟩	🟩	🟩						
Develop Genetic Algorithm to produce optimal config																						🟩	🟩	🟩			
GROWTH																									🟩	🟩	🟩

Figure 17: Gant Chart schedule example

4. Results

This chapter details the results of our project and explains our findings. Our results include a description of the simulator and models built to represent OnLive's server network and system usage. It discusses how we verified the accuracy of our software through various tests. A section on our graphical user interface depicts how we translate our simulator and modules into a graphical representation. We also present a comparison between the old Intelligent App Distribution and the new simulated version through calculated metrics.

4.1 Modeling

Identifying the entities and how the entities interact with each other was crucial because it acted as the skeleton for our later work. Moreover, identifying complications in the model early helped remove unnecessary entities that could have led to an over complicated system.

We identified four major entities that interact within the OnLive system: users, sites, apphosts (servers), and appvariants (applications). Each entity has different parameters that alter the behavior of the system and interacts with other entities in different ways. A single user entity will arrive discretely in time to a site that has multiple apphosts. The user will then request an appvariant. The site will identify a free apphost having the application, then route the user to it. If there is no apphost available, the user will retry a set number of times before giving up.

We combined this knowledge of the OnLive system and our knowledge of discrete event simulation to create a basic model that simulates this minimal behavior.

4.2 Our Simulator

4.2.1 Class Hierarchy

As we implement the basic model into our simulation, we created a class hierarchy in Figure 18 and a description of the different entities in Table 1. We use the class UserSession to describe a user in OnLive's network because we make certain assumptions about a user's behavior that is more apply described as a user engaging in a single session on OnLive's service, or user-session.

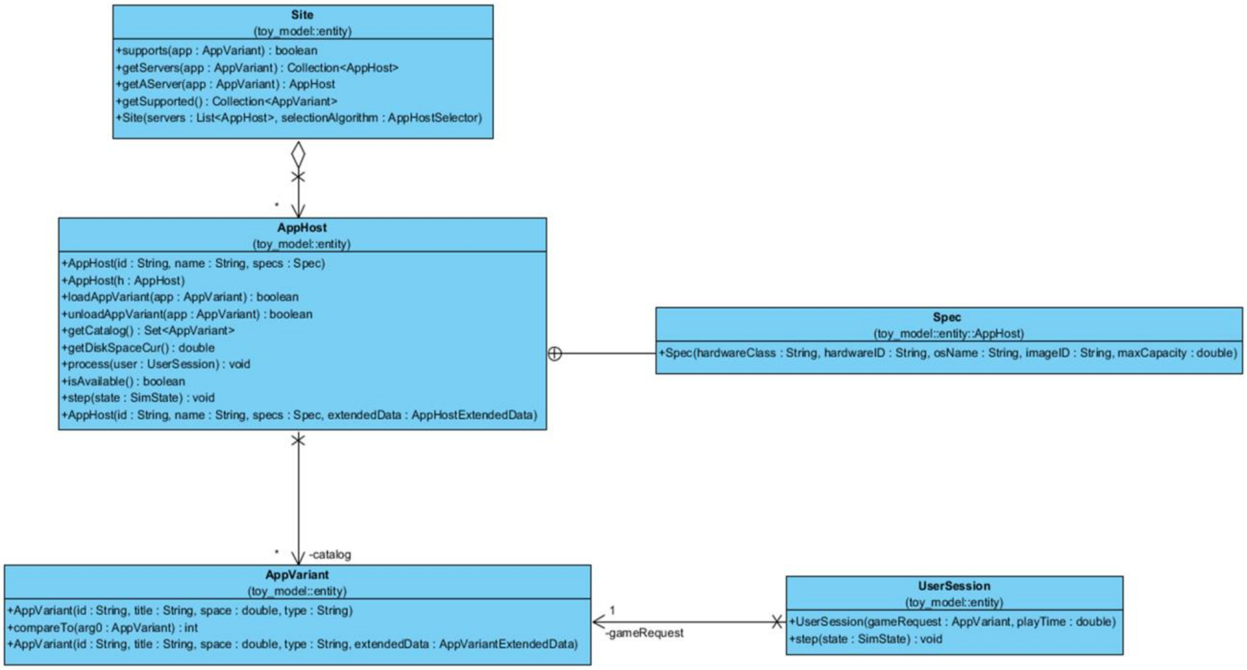

Site
(toy_model::entity)
+supports(app : AppVariant) : boolean
+getServers(app : AppVariant) : Collection<AppHost>
+getAServer(app : AppVariant) : AppHost
+getSupported() : Collection<AppVariant>
+Site(servers : List<AppHost>, selectionAlgorithm : AppHostSelector)
*
AppHost
(toy_model::entity)
+AppHost(id : String, name : String, specs : Spec)
+AppHost(h : AppHost)
+loadAppVariant(app : AppVariant) : boolean
+unloadAppVariant(app : AppVariant) : boolean
+getCatalog() : Set<AppVariant>
+getDiskSpaceCur() : double
+process(user : UserSession) : void
+isAvailable() : boolean
+step(state : SimState) : void
+AppHost(id : String, name : String, specs : Spec, extendedData : AppHostExtendedData)
Spec
(toy_model::entity::AppHost)
+Spec(hardwareClass : String, hardwareID : String, osName : String, imageID : String, maxCapacity : double)
*
-catalog
AppVariant
(toy_model::entity)
+AppVariant(id : String, title : String, space : double, type : String)
+compareTo(arg0 : AppVariant) : int
+AppVariant(id : String, title : String, space : double, type : String, extendedData : AppVariantExtendedData)
1
-gameRequest
UserSession
(toy_model::entity)
+UserSession(gameRequest : AppVariant, playTime : double)
+step(state : SimState) : void

Figure 18: Entity class hierarchy

Class	Description
Users	This is the main controller of the simulation
UserSession	Contains information such as play duration, name, number of retry, etc.
AppHost	Contain all the fields related to the server hardware and catalog.
AppHost.Spec	The hardware and operating system specifications of an AppHost, encapsulated. Represents that apphosts could be considered identical if only looking at their specs.
AppVariant	Stores application information.
Site	An aggregate of apphosts corresponding to a 'site' in OnLive's network.

Table 1: Entity class hierarchy descriptions

4.2.2 Modularity for Dependency Injection

In addition to these basic entity classes, we expanded the scope of our model, compartmentalizing different behaviors into modules, which allowed us to use a dependency injection approach to constructing the strategies in our model. This will allow OnLive to easily extend the strategies we have created or to design their own.

These modules range from simple behaviors such as the SessionLength module, which provides a distribution for the amount of time a user will be playing a specified application, to more sophisticated ones, such as the AppHostCatalogProvider, which assigns catalogs of appvariants to the apphosts. Our java implementation of the IAD is an example of an AppHostCatalogProvider module type. Information about module types is in Table 2, and for a complete list of implementation of each type please see Appendix A: List of Module Types and Implementations.

Module	Purpose
AppHostCatalogProvider	Controls how appvariants are distributed to apphosts
IADArguments	Reads command-line arguments that would normally be passed to the iad.py file from a configuration file.
Summarizer	Summarizes the IAD run, includes all features
DistributionRuleProvider	Provides YAML configuration data for dist ratios
AppHostGroupAssigner	Performs the assignment step of the IAD algorithm
TargetAppHostSelector	Selects the best-suited apphost to be assigned an appvariant group in a step of AppHostGroupAssigner
SurplusUnloader	Unloads (selects) surplus apphosts from an appvaraint group
AppSelectionProvider	Determines which appvariant a user wants to play
ArrivalInterval	Determines when a user will arrive
SessionLength	Determines how long the user will play for
AppHostSelector	Determines which apphost (if any) will serve the user
WeakestLinkFinder	Finds the weakest link in a generation
MetricCalculator	Computes the objective function of a simulation run
InitialConfiguration	Computes the initial distribution ratios for the DistributionRuleProvider
MetricComparer	Compares metrics together to determine if the results are better than before or are within certain thresholds

Table 2: List of current module types and their purpose

4.2.3 Scheduling

The Mason simulation library manages the simulation process. It holds a structured scheduler that builds a stack of events and sets the events to occur at a chosen point in time. The initial schedule is generated before the simulation starts. During the simulation, the scheduler triggers the behavior of the user-session objects in its queue according to time. As the simulation runs, objects can add themselves to the queue, and in the case that a user is unable to obtain an apphost, the user-session will reschedule itself to be run again later. After the session successfully completes, the instance will be disposed, and the system calls the next event in the scheduler.

To execute, a user session needs to obtain an available apphost in the system. A user session may not receive an apphost when requested if the application is not present or if other users occupy all other apphosts that can serve the app. When a user session cannot acquire an apphost, it will go to sleep, register its wake up time to the scheduler, and request the application again when getting up. Moreover, if the user session cannot obtain an apphost after a certain amount of requests, it will give up and not subscribe itself in the scheduler again. Figure 19 depicts the lifecycle of a user session

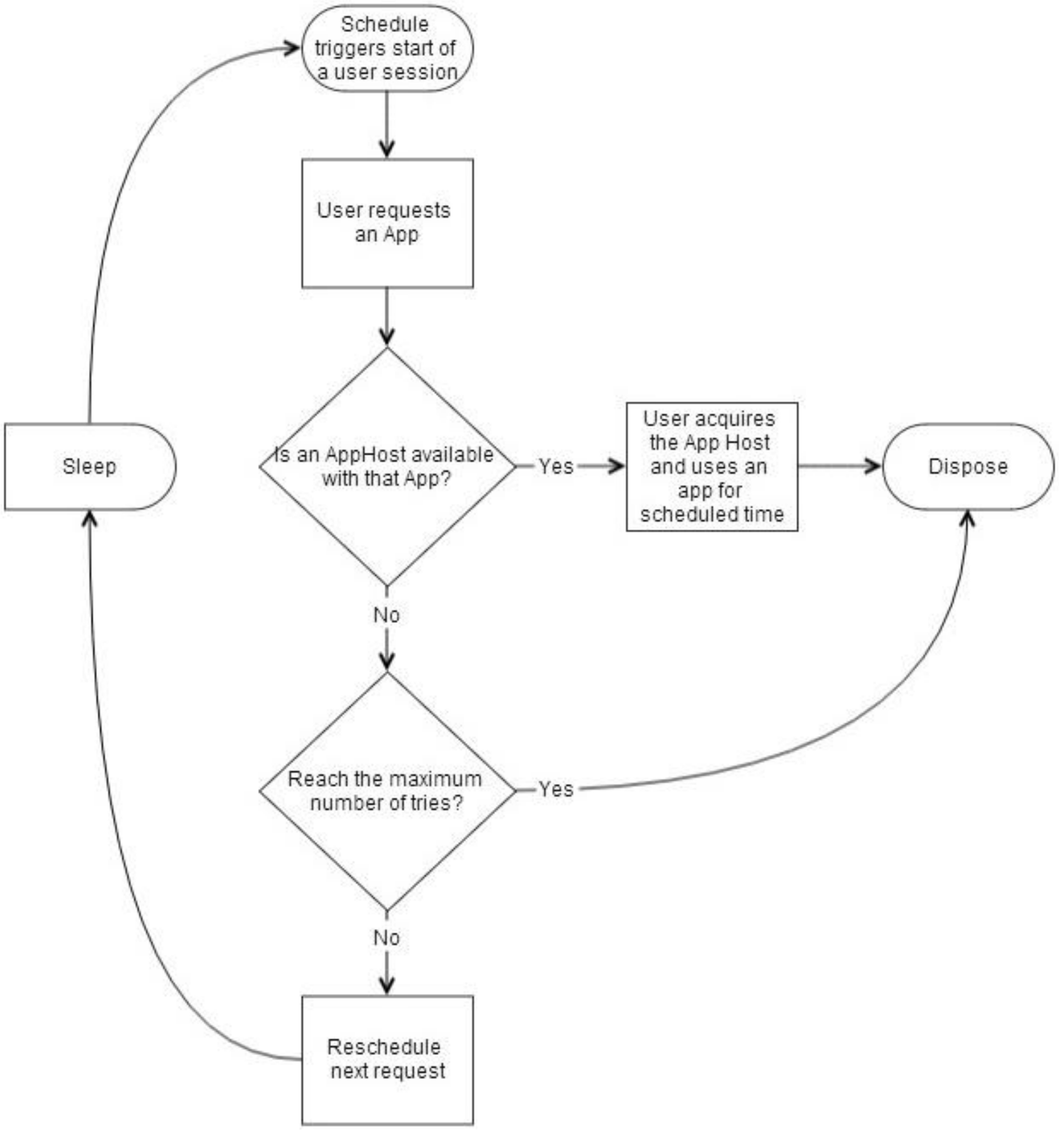

Schedule triggers start of a user session
User requests an App
Is an AppHost available with that App?
Yes
User acquires the App Host and uses an app for scheduled time
Dispose
No
Reach the maximum number of tries?
Yes
No
Reschedule next request
Sleep

Figure 19: Lifecycle of a single user session

4.3 Finding the right distribution

Defining parameters to a user session to make it behave like a real user session requires a distribution analysis of real data. We identified three parameters that required modeling in the simulation: the length of a user-session (how long one user uses one application), the time between users starting sessions, and the user's choice of application.

In order to determine the appropriate values for each of these, we applied several different analysis techniques to a data set of approximately 1.6 million user sessions provided to us by OnLive. A small subset of this data is in Table 3. This data listed the start and end times, the site the user was on and the appvariant that the user requested. The simplest to compute was the distribution of the application users are interested in. We used this data directly in order to form a popularity distribution to assign the popularity of different applications. This allows us to model the preferences of users for what applications they use based on real world data. Additionally, these ratios can be adjusted to correspond to real-world changes in popularity of applications.

site	start_time	end_time	duration	application_id
eac	11/30/2013 19:13	12/1/2013 0:44	19874515000	aSDzfpSSLfEQkOhAKAwOIc
ead	11/30/2013 19:21	12/1/2013 2:12	24611251000	alRFS9y5vizAv8CE6fWtGM
lab	11/30/2013 19:31	12/1/2013 1:06	20132353000	aHiDtVs0LkjB6XcYijEEM7
dae	11/30/2013 20:05	12/1/2013 0:26	15650868000	ayBFDBBs5nw44xfVFepFq5
ead	11/30/2013 20:12	12/1/2013 0:36	15844387000	af-0cnUOHmAy1L9fI5YjV9

Table 3: Example data from OnLive

Exponential distributions are often used for modeling the inter-arrival times in a process such as connecting to OnLive's apphosts. As such, we parsed the data from OnLive and applied estimation techniques to compute the best fitting exponential distributions for each application on OnLive's apphosts.

Computing the duration of user sessions required a more in-depth approach. One of the most-used distributions to describe connection duration to a site is the gamma distribution, but we found that the gamma distribution would poorly fit our data with the use of the professional statistical fitting software. We used EasyFit, and found that a Weibull distribution to be a superior distribution. The Weibull distribution is used to model processes where there exists a certain failure rate, which can increase, decrease or remain constant over time, depending on one of the distribution's parameters. An example comparison between actual data from OnLive for a single game title and the Weibull distribution estimated from that trace data is shown in Figure 20. For this data set, we obtained a p-value of 0.018 using the Chi-Squared test, discarding the noisy first few minutes of data. This statistic was generated comparing approximately 1500 data points on session length. As the Chi-Squared test statistic tends to increase as the comparison sample size increases, even a well-suited distribution will receive a very small p-value, so we consider this to be a well-suited distribution.

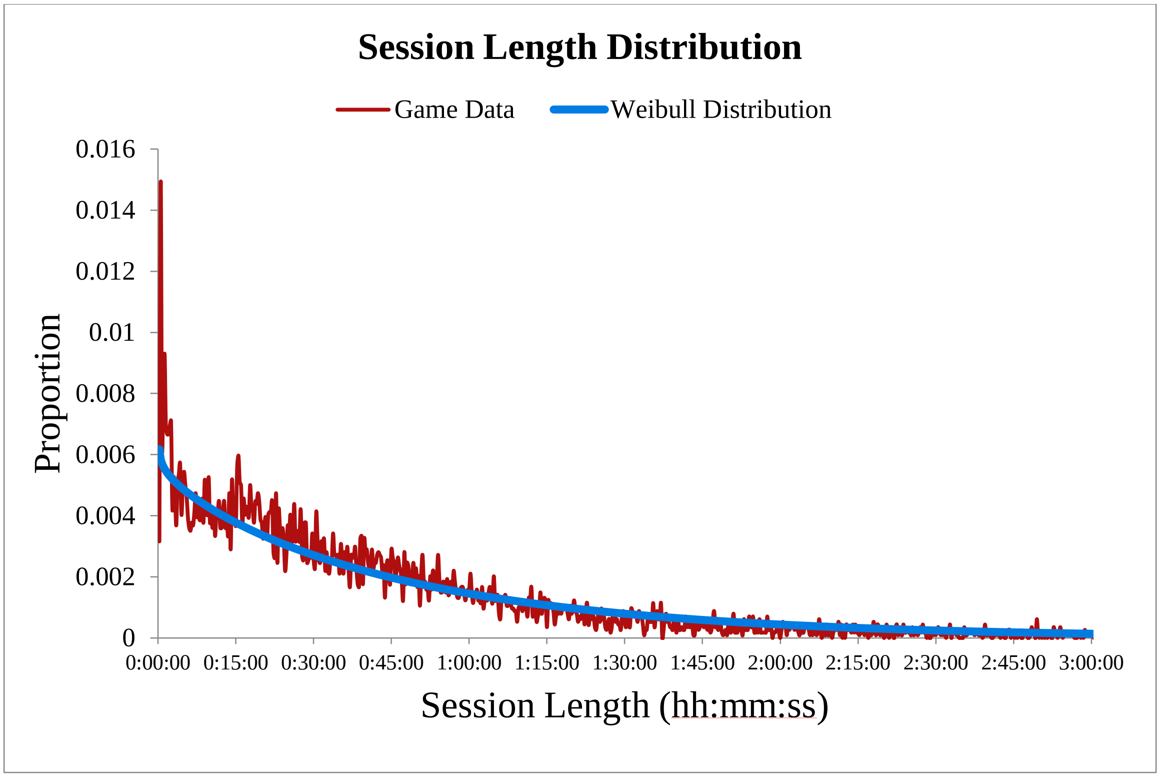


Figure 20: Example Weibull Distribution fitted to OnLive session length data

Using the EasyFit statistical analysis software, we found that a four-parameter generalized gamma distribution often fit our session data better than the Weibull distribution, with a statistic approximately two-thirds that of the Weibull distribution on the Kolmogorov-Smirnov test (K-S test). With the K-S test, the null hypothesis states that there is not a statistically significant difference between the expected values and the empirical values, so failing to reject the null hypothesis is indicative that our expected distribution is well suited. For example, on the game Civilization V, the four-parameter generalized gamma distribution failed to reject the null hypothesis at a confidence level of α=0.05, where we rejected the null hypothesis with the fitted Weibull distribution. We decided that the accuracy of the Weibull distribution was sufficient for our purposes and would be much easier to compute through our own software.

4.4 Simulation Model Validation

With the models and distributions calculated by our model, we needed to verify that our calculations were accurate. This is one of the most important steps in the creation of our simulation as it validates our model to be an accurate representation of the real world. We first verified basic factors, such as mimicking the number of apphosts and applications in OnLive's network. We also verified the estimated statistical distributions that described session length, inter-arrival time, and popularity of applications. Finally, we put our simulation through verification checks to verify that it behaved in a predictable manner when parameters such as the average session length or number of apphosts present were adjusted while holding all other parameters constant.

4.4.1 Basic Verification Test

The first verification check that we applied to our model was a 'base case' type of check. We placed the model under strict conditions such that the expected results could be computed by hand. To do this, we reduced the number of users to three, the number of apphosts and applications to one, and provided fixed arrival times and session lengths to each user.

A user would arrive at times of zero, 50 and 100 seconds, with a session length of 100 seconds. If no apphosts were available for the user, then the user would wait for 60 seconds before trying again. As seen in Figure 21, this is the expected state of the apphost and behavior of the users. User #2, despite arriving second at a time of 50 seconds, would end up failing to get the requested session at times of 50, 110 and 170 seconds, before finally acquiring the apphost at a time of 230 seconds.

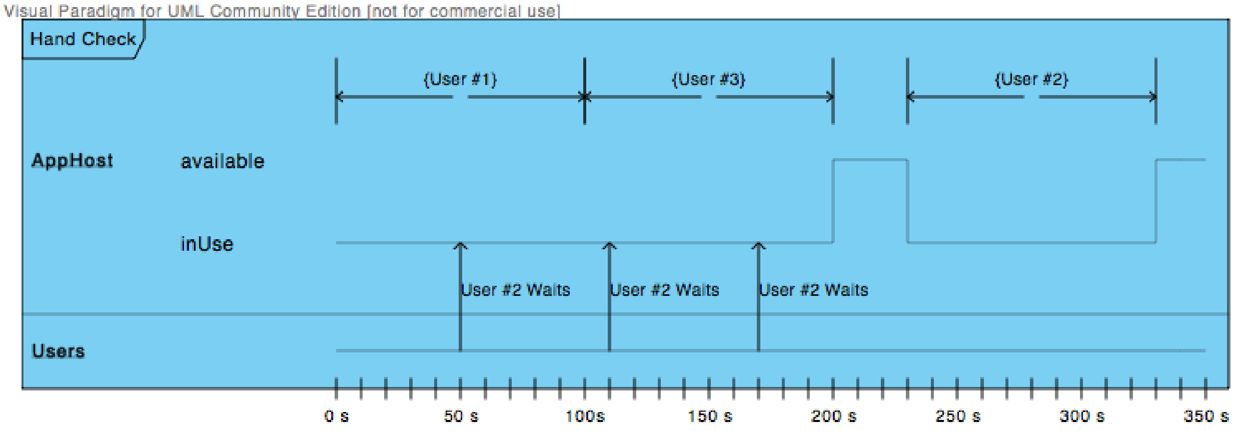


Figure 21: A timing diagram of the basic simulation

Using these input parameters, we ran our model and found that we received the expected results in output in our simulation's log file.

4.4.2 Model Verification Tests

In addition to a verification of our model that could be checked by hand, we also ran several simulation runs in which we modulated one input parameter while keeping all other parameters constant. The different parameters that we changed were average user session length, average user inter-arrival time and number of apphosts at two different average session lengths. The session lengths were normally distributed with a standard deviation of 10 seconds and the inter-arrival time was distributed according to an exponential distribution. Because the apphost catalogs were assigned at random, we ran each simulation several times in order to reduce noise caused by the distribution of applications among apphosts.

One of our checks was to show that the failure rate would increase as the session length is increased. In order to see the effect of only session length, shown in Figure 22, we fixed the number of apphosts and the average arrival interval as shown in Table 4. We can see that in this simulation case the number of failures begins to increase rapidly after the average session length reaches approximately 300 seconds and continues to rise until it approaches a failure rate of 1.0 with an average session time of 800 seconds. While these numbers do not correspond to real world data, they show that there will points exist where large session lengths can completely overwhelm a site's ability to service users.

Number of AppHosts	15
Average Session Length	**variable**
Average Inter-arrival Time	52 seconds

Table 4: Variable for Session Length Verification Check

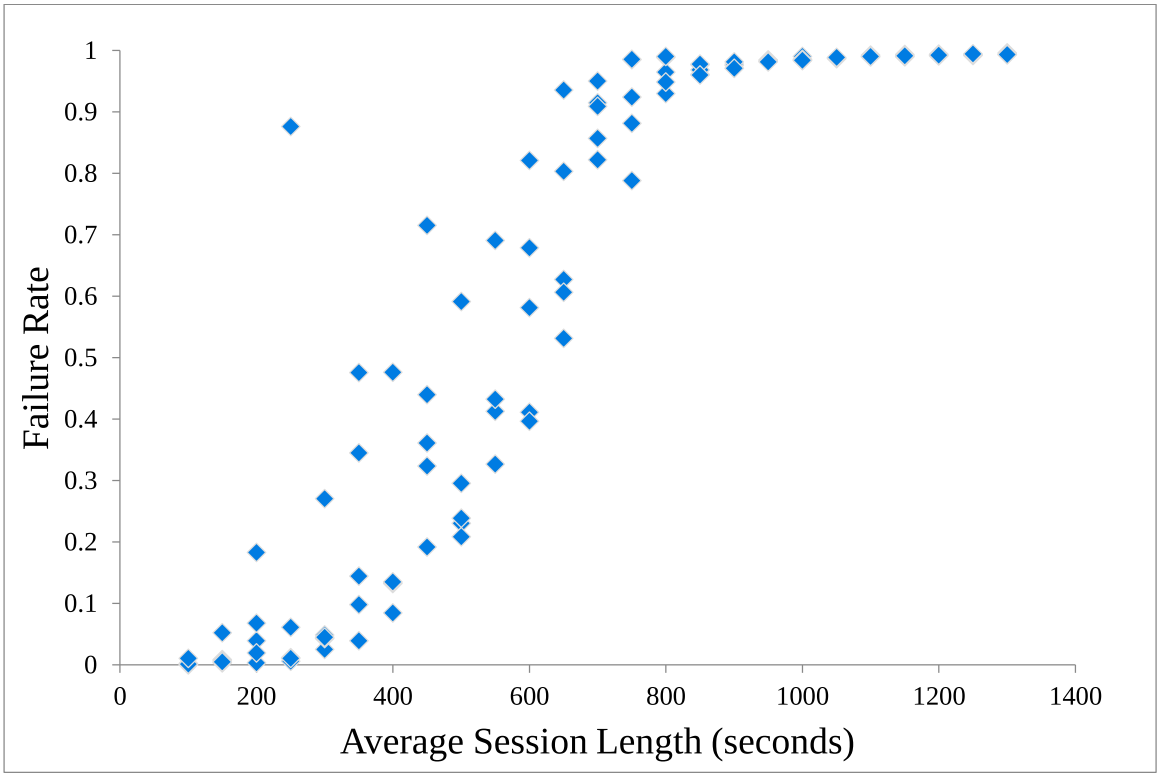


Figure 22: Failure Rate vs. Session Length

Our second such check made the inter-arrival time the variable and held the number of apphosts and session length constant, as seen in Table 5. By reducing the inter-arrival time we can see that the failure rate is increased in Figure 23. Though there is a large amount of noise, we observe that the failure rate decreases at a diminishing pace as we increase the value of the average inter-arrival time. This follows an exponential distribution as shown in the trendline.

Number of AppHosts	15
Average Session Length	300 seconds
Average Inter-arrival Time	**variable**

Table 5: Variable for Inter-Arrival Time Verification Check

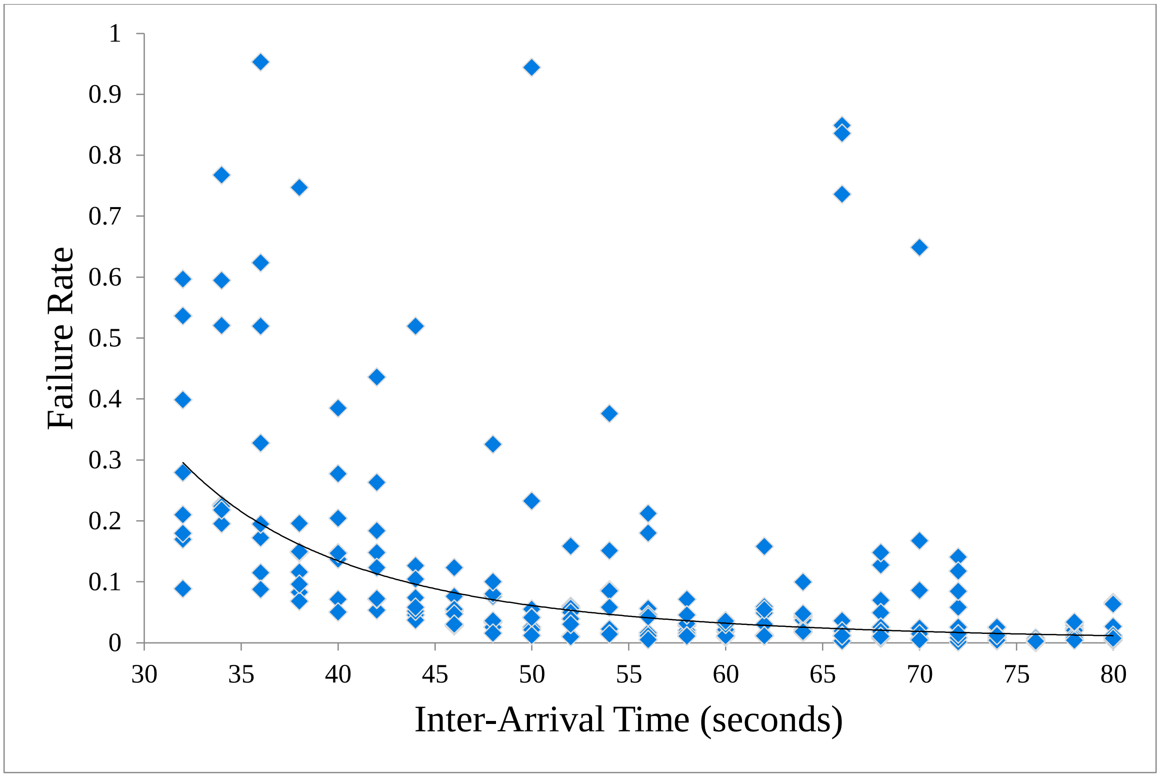


Figure 23: Failure Rate vs. Inter-Arrival Time

We then explored the change in failure rate as a function of the number of apphosts. We had two different testing configurations. Our first investigation had an average session length of 500 seconds (Table 6). We observe that as the number of apphosts increases, the failure rate rapidly approaches zero in Figure 24. With an average session length of 500 seconds, there exists a threshold at around 20 apphosts present on a single site where failure rate drops to nearly zero.

Number of AppHosts	**variable**
Average Session Length	500 seconds
Average Inter-arrival Time	52 seconds

Table 6: Variable for AppHosts Verification Check with 500 second average session

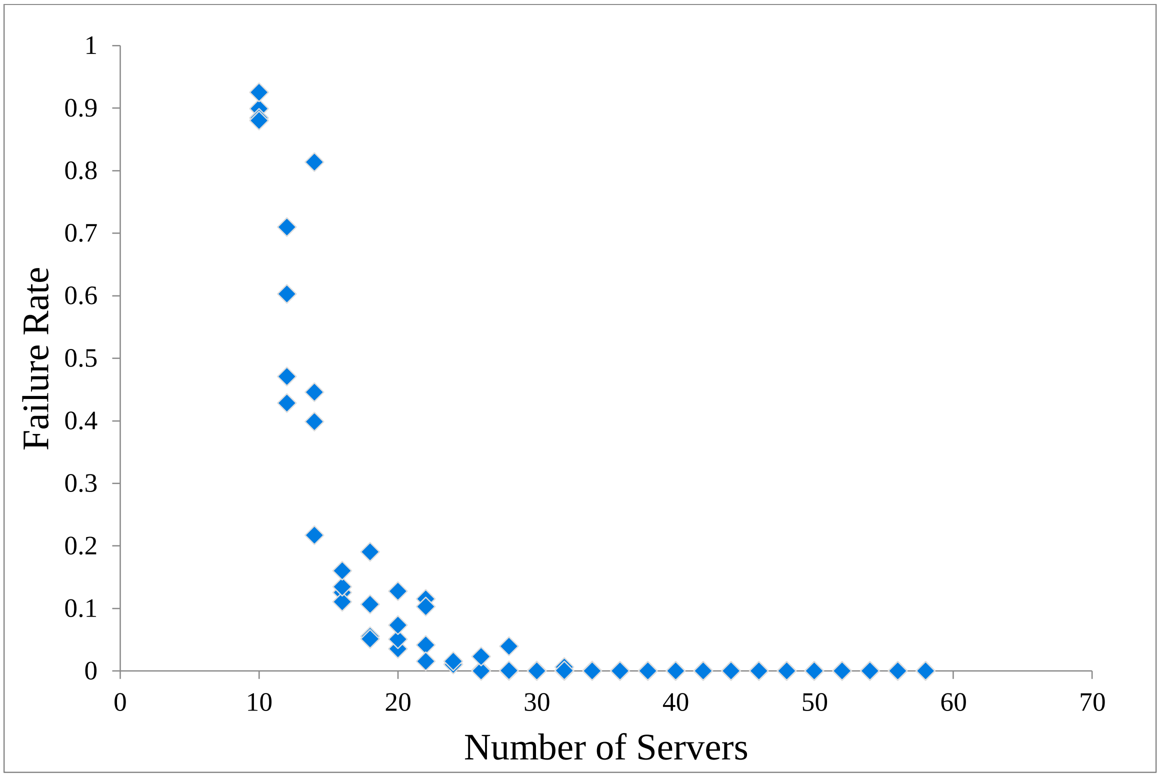


Figure 24: Failure Rate vs. Number of AppHosts with 500 second average session length

Similar to the 500-second average session length graphs, we also tested at an average of 1000 seconds in Table 7. Similar in pattern to the 500-second average session test, the 1000-second test has a threshold of approximately 30 apphosts present in the site in Figure 25. The shapes of the failure rate plots are similar as well noting a consistent behavior just a shift in the number of required apphosts on a site.

Number of AppHosts	**variable**
Average Session Length	1000 seconds
Average Inter-arrival Time	52 seconds

Table 7: Variable for AppHosts Verification Check with 1000 second average session length

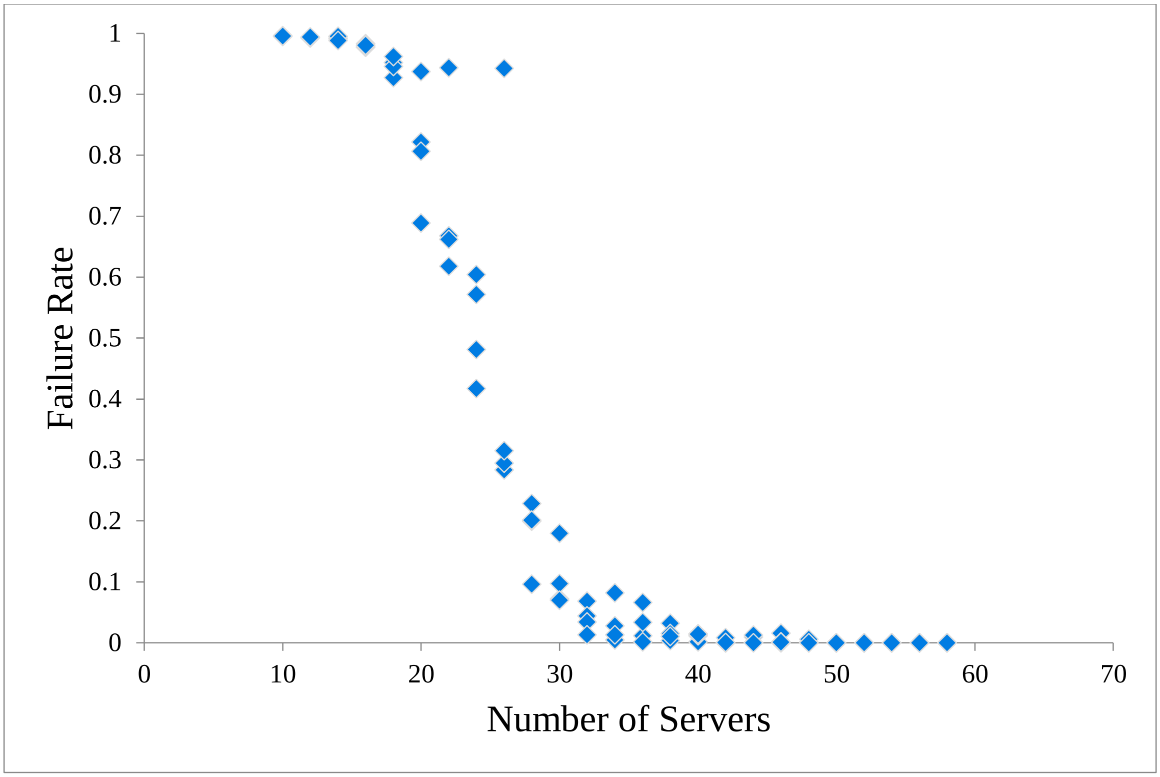


Figure 25: Failure Rate vs. Number of AppHosts with 1000 second average session length

OnLive has indicated that they are currently able to meet user demand because they simply have more apphosts available on every site than the minimum required. Through our simulator verification tests, we observe a similar behavior. After crossing a certain threshold of apphosts, we are able to meet every user request essentially without fail.

These verification checks demonstrate that our model is capable of accurately modeling changes in the input parameters. From this, we are able to deduce that given the appropriate configuration parameters and strategies, we would be able to properly model the state of one of OnLive's sites.

4.4.3 IAD Verification

To allow for accurate bin packing of apps onto apphosts we converted the Python implementation of the IAD into Java for use in our simulation. The Python implementation is what OnLive uses to put applications onto their apphosts with a bin-packing algorithm. As this

Python script is essential to OnLive's production process, we needed to ensure our IAD in Java behaved identically on our simulator. To do this, we pulled a YAML (YAML Ain't Markup Language) configuration file currently in use on the production IAD code and ran it on our simulated IAD. The YAML file contains distribution ratio and ordering rules for different appvariants and apphosts. We then compared the logged outputs, and found by hand that the distribution ratios were identical. With our results validated, we moved to optimizing the IAD code.

4.4.3.1 IAD Complexity Improvements

In order to understand the current operation of the IAD, a few new terms must be introduced. An apphost is the term OnLive uses to refer to a single server that streams content to a user. Appvariant refers to a single version of a single application/game title; there are separate appvariants for things such as a game and its associated demo. An appvariant group is a group of appvariants. It has the property that, as far as disk space is concerned, all appvariants in a group have a large amount of overlap in their code. An example of this is Borderlands, which has a game, a game demo and several DLC content packs, each of which is an appvariant. Looking at all of these appvariants as a single, cohesive group saves a large amount of space. While a single apphost can hold a very large number of appvariants, and thus appvariant groups, it can only serve one user one appvariant at a time. We use the term configuration rule to refer to a line in the YAML configuration file. A configuration rule can have fields such as 'app' or 'os', which provide it with filters for which appvariant-apphost pairs to apply the rule to and values like 'dist_ratio', which tell the IAD what proportion of apphosts to try and distribute the given appvariant to. Another important term is work queue, which represents a heap of actions that the

IAD takes as part of its distribution process. Each entry in the work queue associates an appvariant group with a list of apphosts that are being targeted for distribution.

As part of reverse engineering the IAD code into Java, we profiled parts of it in order to get a better understanding of how it worked. The IAD distribution process can be divided into two steps: the bin-packing step and an assignment step. In the bin-packing step, each pairing of apphost and appvariant is run through a series of precondition checks before being assigned to an appvariant group (several appvariants that share code content). We determined several important factors in this step of the code: the number of apphosts (h), and the number of appvariants (v) being immediately visible. After going through all of the preconditions, all of the apphost-appvariant pairs' distribution ratios are computed from the IAD's YAML configuration file input. We found that with each pair that reached this part, another (l) steps would be taken to search through the list of configuration rules for applicable ones. Some rules may define an "apps" field, which lists several appvariants at once, having an average number of (n) apps listed in a rule. With this, the bin-packing step would take $O(v \cdot h \cdot l \cdot n)$ time.

We found that the number of rules defined in a standard configuration file to be roughly proportional to the number of appvariants and number of appvariants in each apps list. As such, we can reduce our first analysis of the bin-packing step to officially take place in $O(v^2 \cdot h)$ time. We had noticed that this sort of complexity was present when first porting the IAD code. With this analysis, we refactored the way rules were handled. We changed the code to precompile the rules into a map object. This meant that we no longer needed to search a long list for values, reducing our time complexity to $O(v \cdot h)$. This process went from taking about 45 seconds to about one second for a single simulation run over 400 apphosts and 700 appvariants.

The next part of the IAD algorithm is the assignment phase. The assignment algorithm first creates a series of work queues that contain all of the appvariant groups. This takes $O(h \cdot v \cdot \log(v))$ time to generate the work queues. This is because we need to insert each group into its position in the queue and search for incorrectly configured apphosts in each group. Additionally, with each step of the distribution, the first work item in the queue is popped and may or may not be replaced. We then search all target apphosts to find the optimum target at an additional $O(h^2 \cdot v)$ time to assign appvariants to apphosts. This gives us a total time complexity of $O(h \cdot v + h \cdot v \cdot \log(v) + h^2 \cdot v)$ time, which reduces to $O(h \cdot v \cdot (h + \log(v)))$. With our optimizations, a single 2.67GHz processor core computes the distribution for the 400 apphosts and 700 appvariants in a time of around two seconds.

4.5 Genetic Algorithm

Our primary goal was to build a framework for OnLive to evaluate the effectiveness of different distributions. Unfortunately, as OnLive creates distributions by hand, it would be difficult for them to take proper advantage of the benefits that our simulation allows. As such, a method to programmatically generate a better distribution is needed to provide better configurations.

In order to do this, we implemented a genetic algorithm that runs our simulation, analyzes the results and uses those results to create new configuration parameters for the next generation, as seen in Figure 26 below. The heuristic used while running the simulation subtracts points from an appvariant for a miss (where a user was unable to get the app requested) and for a give up (after a user tries too many times to get the app requested). At the end of the simulation the scores for all appvariants are compared. The appvariant with the worst score is what the genetic algorithms alters next. The worst app variant's distribution ratio is then increased incrementally in several child generations. Each child generation is then run through the IAD simulation again generating new tallied metrics. From the children, the one with the best score survives and is used as the next generation’s starting point. This process runs until a certain success ratio is reached for all appvariants.

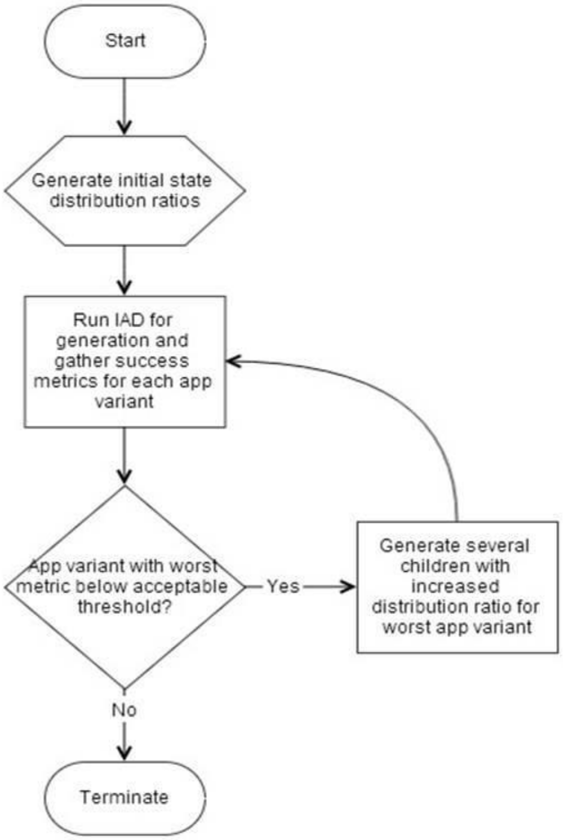

Start
Generate initial state distribution ratios
Run IAD for generation and gather success metrics for each app variant
App variant with worst metric below acceptable threshold?
Yes
Generate several children with increased distribution ratio for worst app variant
No
Terminate

Figure 26: Genetic Algorithm flow chart

To allow OnLive maximum flexibility in further development of the genetic algorithm, factors such as the method by which the simulation metric is calculated and how a generation is created are placed in modular sections. This way, OnLive may replace their functionality as they develop a better idea of what success criteria looks like. Our current metric takes the sum of the number of failures and one hundred times the number of times a user gives up, and is printed in Figure 27 as a pair of those counts. An example of what the genetic algorithm outputs for each generation are shown in Figure 27. In it, we see the genetic algorithm attempting to improve the distribution of the game NBA 2011, trying increasing distribution ratios to determine is an improvement can be made.

```
Testing configuration parameter {dist_ratio=0.810, app=NA-2K-NBA2K11-10059/01.003/18706} for iteration 2
The score for configuration {dist_ratio=0.810, app=NA-2K-NBA2K11-10059/01.003/18706} was (948, 49)
Testing configuration parameter {dist_ratio=0.905, app=NA-2K-NBA2K11-10059/01.003/18706} for iteration 2
The score for configuration {dist_ratio=0.905, app=NA-2K-NBA2K11-10059/01.003/18706} was (925, 51)
Testing configuration parameter {dist_ratio=1.000, app=NA-2K-NBA2K11-10059/01.003/18706} for iteration 2
The score for configuration {dist_ratio=1.000, app=NA-2K-NBA2K11-10059/01.003/18706} was (667, 32)
We have successfully improved the configuration with {dist_ratio=0.240, app=NA-2K-NBA2K11-10059/01.003/18706},
causing an increase of 1450.0 points
```

Figure 27: Sample genetic algorithm output per generation

4.5.1 Initial State

One of the major objectives in developing our genetic algorithm was to construct an initial state that is well suited to usage data. Such a starting point serves two purposes. First, it allows us to be closer to our target solution without having to run the genetic algorithm for too many iterations. Secondly, a poorly suited distribution scheme will drastically increase completion time for each simulation run. By analyzing popularity data extracted from session data provided by OnLive, we can construct a starting distribution that reflects the balance of popularity of applications in OnLive’s services. By scaling the distribution ratios off of the most popular title, we were able to create a starting distribution that was both well-suited to the system and allowed us to meet our target distributions with a large amount of disk space remaining on the apphosts.

The initial state has two different implementations. The first is to create a new state based only off of popularity data. This implementation will be used when migrating from the old IAD code to our new simulated IAD code in production. In future use of our IAD, one uses the second form of implementation for the start state. This form takes in a given YAML configuration and assigns the initial state ratios to what is specified in the file. This means that in production our simulator can take a previously run YAML configuration and test its success on new system load levels and popularity changes. The second implementation will only change distributions that have changed. This means when our new IAD is run from month to month, for example, changes on the live system will only update the app variants that require change.

4.6 Simulation Graphical User Interface

The graphical user interface for our simulator was based on altering the modular system. Being able to select what modules to run for a given simulation is crucial functionality to express with a GUI. To get this functionality, we employed the Reflections Java library. This library is a Java runtime metadata analyzer that scans the classpath, indexes the data, and makes it queryable at runtime (source: https://code.google.com/p/reflections/). With Reflections we can generate the list of modules in our software and gather dependencies between different modules. In this way, if OnLive produces a new module for the simulation, the GUI will automatically pull its information and present it without ever touching the code.

When the GUI is first opened, a window named “Module Loader” appears and lets a user select which modules to run for a simulation. At first, all the modules are unselected and have a red alert sign letting the user know seen in Figure 28. As the user selects which modules to use, there is a possibility that a module requires the selection of sub-modules. When this happens, the required selection fields and a yellow alert icon appear as shown in Figure 29. The yellow alert icon also has a tooltip attached that informs the user what sub-modules must be included. After all modules and sub-module have been selected, the alert icons change to green checkmarks and the “Load Modules” button becomes active, as shown in Figure 30.

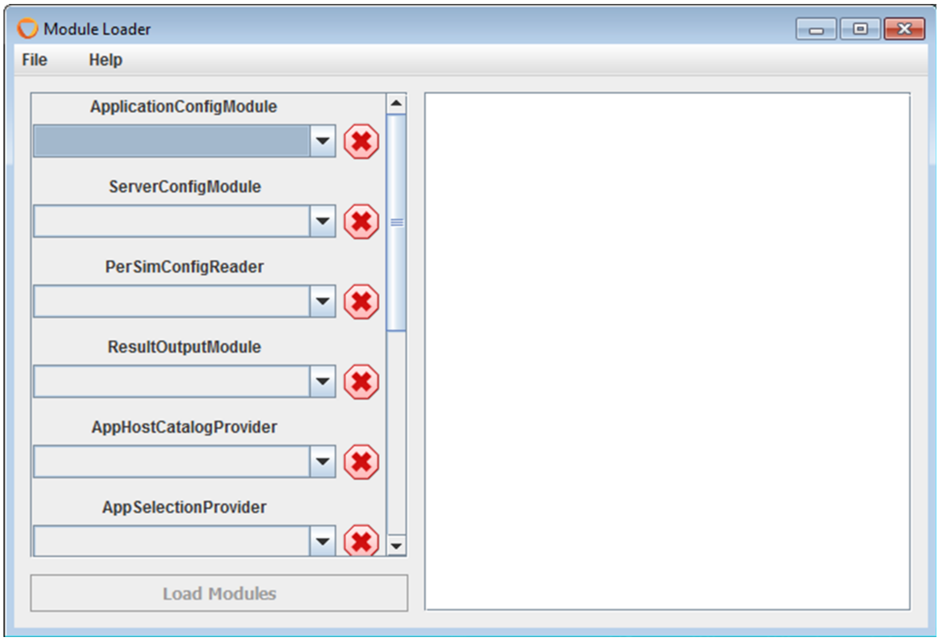
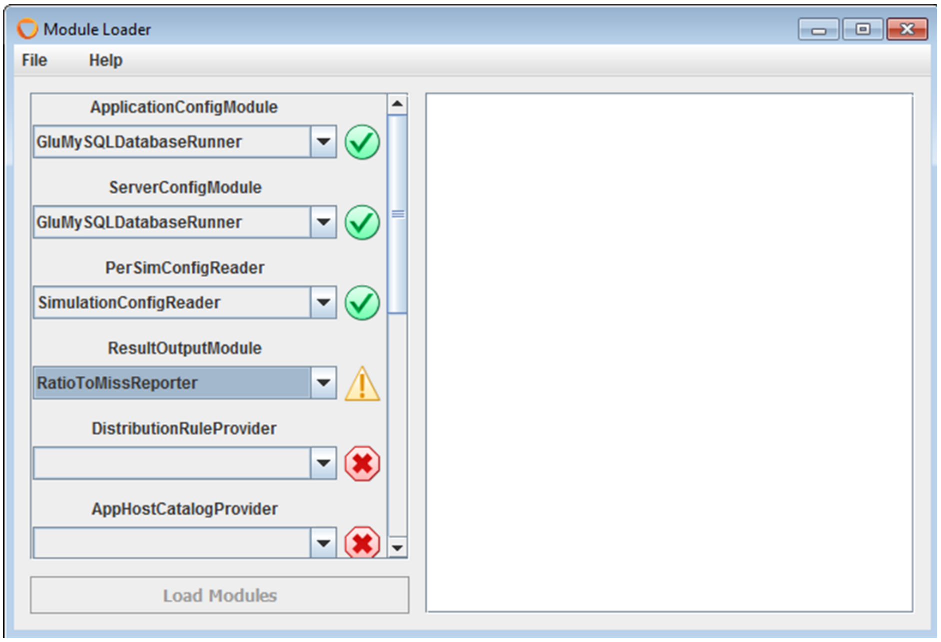
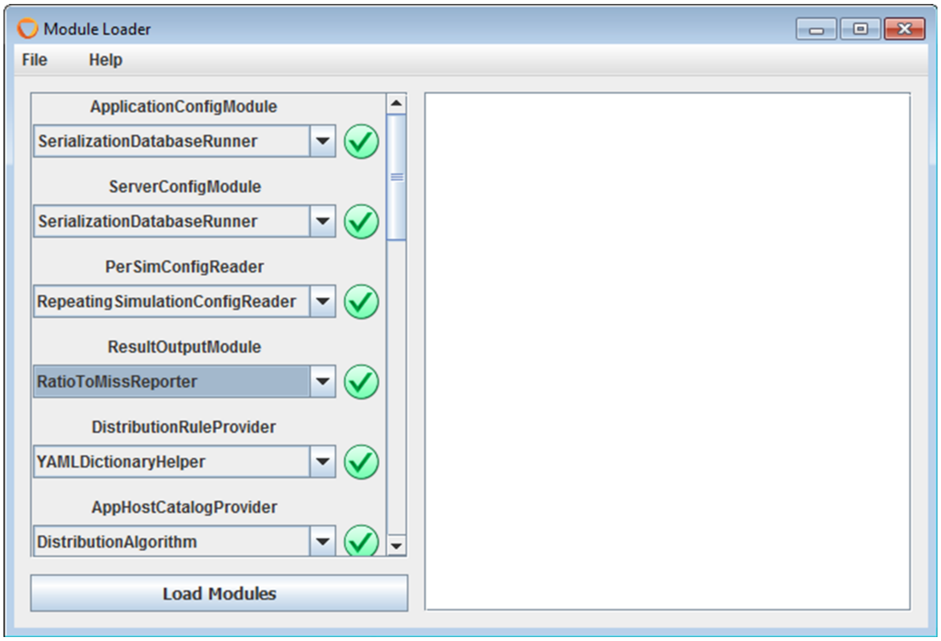


Figure 28: No modules selected when GUI first opens



ApplicationConfigModule: GluMySQLDatabaseRunner ✓
ServerConfigModule: GluMySQLDatabaseRunner ✓
PerSimConfigReader: SimulationConfigReader ✓
ResultOutputModule: RatioToMissReporter ⚠

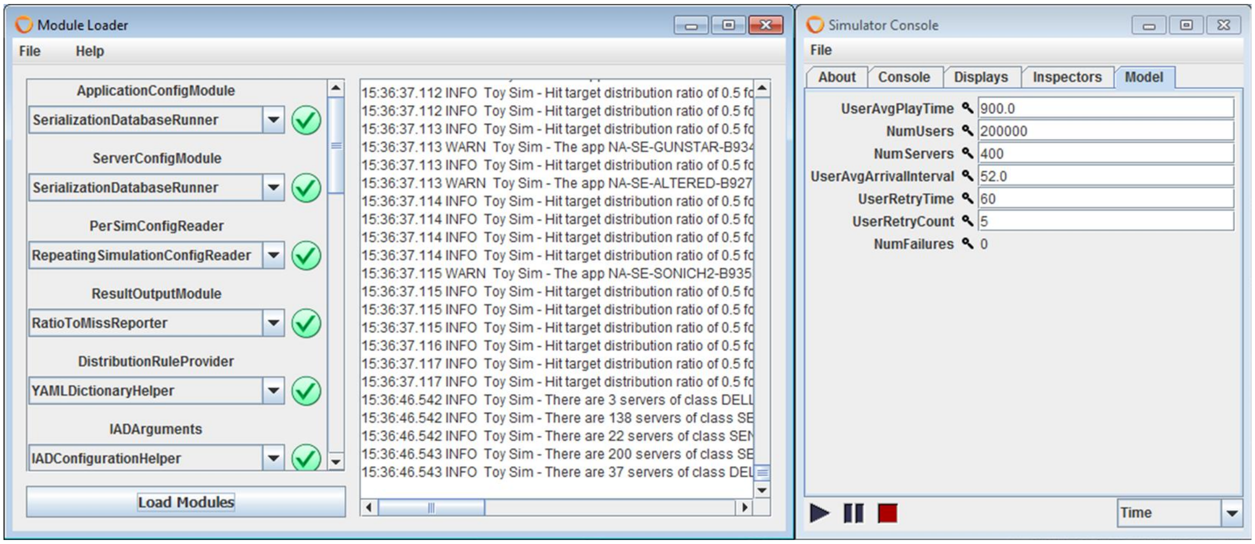
Figure 29: Sub-Module requirements create a yellow alert icon



ApplicationConfigModule: SerializationDatabaseRunner ✓
ServerConfigModule: SerializationDatabaseRunner ✓
PerSimConfigReader: RepeatingSimulationConfigReader ✓
ResultOutputModule: RatioToMissReporter ✓
DistributionRuleProvider: YAMLDictionaryHelper ✓
AppHostCatalogProvider: DistributionAlgorithm ✓

Figure 30: All modules selected and the "Load Module"s button is active

When the "Load Modules" button is pressed, the simulator builds the configuration requested. The console window prints out a list of all the modules loaded and the initial startup of the simulator. Once the modules have successfully loaded, a new window opens up called the "Simulator Console" as shown in Figure 31. This console window is from the Mason Library and it controls the simulator (play, pause, and stop). It also allows the option to change parameters like number of users, number of servers, inter-arrival time and retry counts for uses between simulation runs. In this way OnLive can easily analyze the behavior of different simulation states with a single configuration. The simulation itself is normally only run once, however if a user selects the use of the genetic algorithm, it will run until the termination state.



ApplicationConfigModule: SerializationDatabaseRunner ✓
ServerConfigModule: SerializationDatabaseRunner ✓
PerSimConfigReader: RepeatingSimulationConfigReader ✓
ResultOutputModule: RatioToMissReporter ✓
DistributionRuleProvider: YAMLDictionaryHelper ✓
IADArguments: IADConfigurationHelper ✓

```
15:36:37.112 INFO  Toy Sim - Hit target distribution ratio of 0.5 fc
15:36:37.112 INFO  Toy Sim - Hit target distribution ratio of 0.5 fc
15:36:37.113 INFO  Toy Sim - Hit target distribution ratio of 0.5 fc
15:36:37.113 WARN  Toy Sim - The app NA-SE-GUNSTAR-B934
15:36:37.113 INFO  Toy Sim - Hit target distribution ratio of 0.5 fc
15:36:37.113 WARN  Toy Sim - The app NA-SE-ALTERED-B927
15:36:37.114 INFO  Toy Sim - Hit target distribution ratio of 0.5 fc
15:36:37.114 INFO  Toy Sim - Hit target distribution ratio of 0.5 fc
15:36:37.114 INFO  Toy Sim - Hit target distribution ratio of 0.5 fc
15:36:37.114 INFO  Toy Sim - Hit target distribution ratio of 0.5 fc
15:36:37.115 WARN  Toy Sim - The app NA-SE-SONICH2-B935
15:36:37.115 INFO  Toy Sim - Hit target distribution ratio of 0.5 fc
15:36:37.115 INFO  Toy Sim - Hit target distribution ratio of 0.5 fc
15:36:37.115 INFO  Toy Sim - Hit target distribution ratio of 0.5 fc
15:36:37.116 INFO  Toy Sim - Hit target distribution ratio of 0.5 fc
15:36:37.117 INFO  Toy Sim - Hit target distribution ratio of 0.5 fc
15:36:37.117 INFO  Toy Sim - Hit target distribution ratio of 0.5 fc
15:36:46.542 INFO  Toy Sim - There are 3 servers of class DELL
15:36:46.542 INFO  Toy Sim - There are 138 servers of class SE
15:36:46.542 INFO  Toy Sim - There are 22 servers of class SEN
15:36:46.543 INFO  Toy Sim - There are 200 servers of class SE
15:36:46.543 INFO  Toy Sim - There are 37 servers of class DEL
```

UserAvgPlayTime: 900.0
NumUsers: 200000
NumServers: 400
UserAvgArrivalInterval: 52.0
UserRetryTime: 60
UserRetryCount: 5
NumFailures: 0

Figure 31: Fully Initialized GUI, ready to start the simulation

4.7 Behavior Scenario Tests

With the simulator built, we tested different behavioral scenarios with it. Among the tests included how well current distributions on the OnLive server network compare to what our genetic algorithm created and how well we can optimize the distribution of applications.

4.7.1 Production vs. Simulation Results

We created a baseline test that took a current YAML configuration from production and generated metrics for its configuration of app variants. This allowed us to generate a base case for how successful a current distribution is on the network. The success metric was designed with a maximum possible score of zero points, with negative points awarded for each time a user requests an appvariant but fails to receive an apphost (-1) and for each time a user gives up (-100). The reason for this heuristic was because a user that retries and successfully receives an apphost is much a more forgivable failure than a retry that fails so many times the user gives up. Next we ran our genetic algorithm to create a new distribution. The genetic algorithm used the success metric as a heuristic to alter generation distributions. After the genetic algorithm reached a termination state where it performed as best it could, a new game content distribution was generated.

We took the old content distribution from production and compared it to the new content distribution generated from our genetic algorithm by running the simulation on each with increasing usage densities. Usage density is how much demand a site incurs based on user inter-arrival time. The smaller the inter-arrival time, the more users are accessing the site at any given time. We compared densities from zero to 15 times the current demand density. The density stress tests were then plotted to observe performance of the distribution compared to demand. We compared the current content distribution used in production on one of the sites against the initial and final state of our genetic algorithm. We plotted both the number of failures, shown in Figure 32, and the number of users giving up in Figure 33. The figures show where a distribution begins to fail and how much it fails.

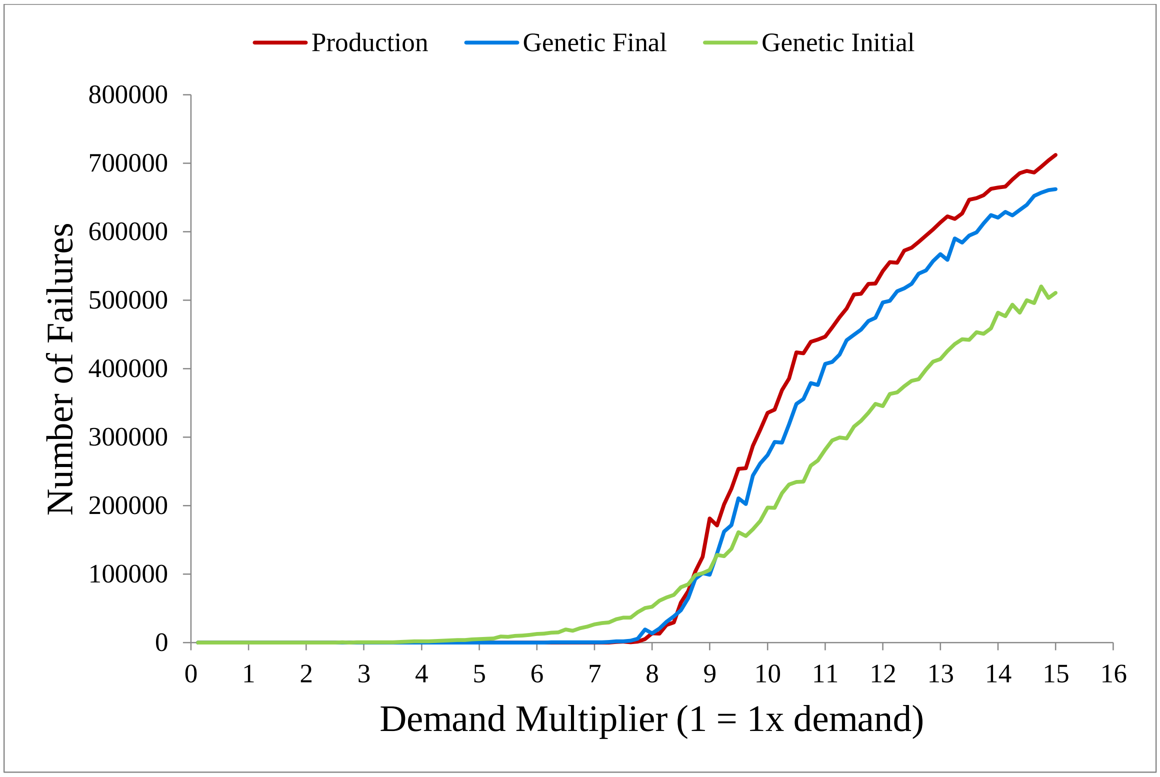


Figure 32: Total failures vs. demand for in production and genetic content distributions

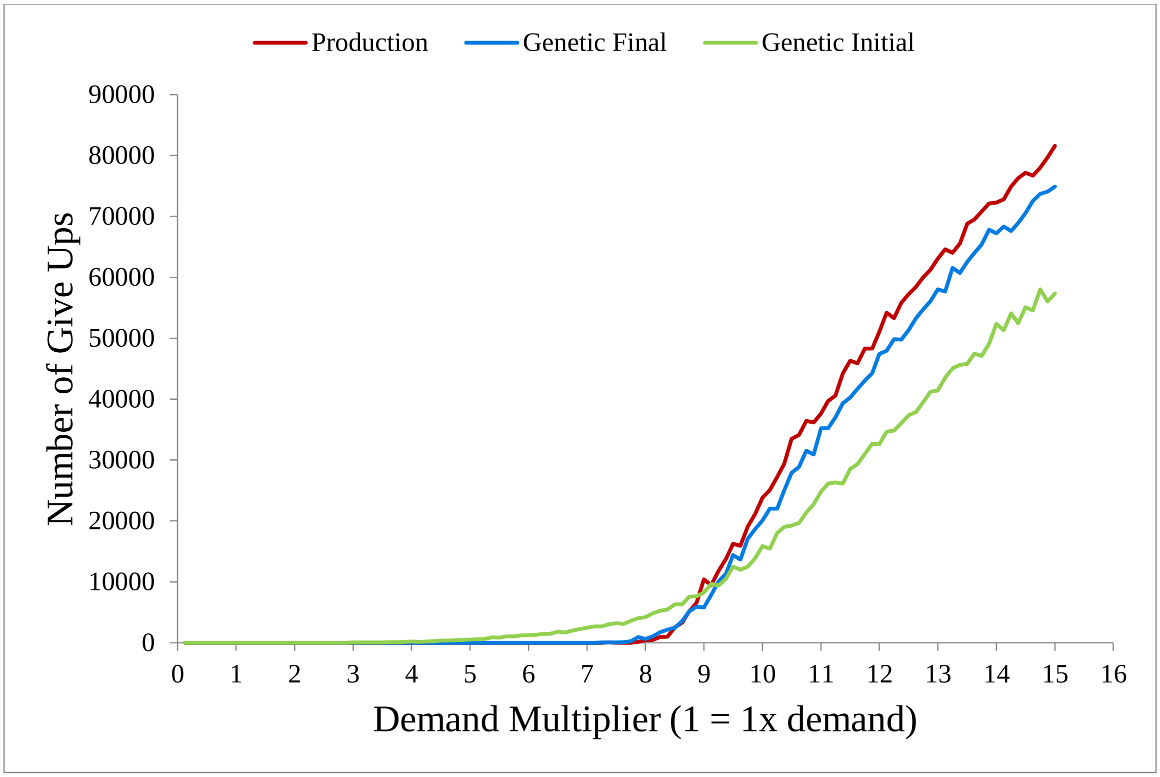


Figure 33: Total user give ups vs. demand for in production and genetic content distributions

From our metrics, it is important to note that with actual site density, 1x demand, there were no failures incurred by either the in production distribution or the genetic algorithm's distribution. This means that these two distributions can handle current demand for the site under test. At five times the density however, the initial state of the genetic algorithm begins to incur both users failing to receive and apphost and users giving up. At around 8x demand density, both the in production distribution and the genetic algorithm final distribution begin to incur failures and give ups. Also from the two figures, at 15x the demand density we see that the genetic algorithm at both the initial state and final state perform better than the current in production. At extreme usage demand increases, we see that the in production distribution to be inferior to what distributions we create with our simulator. At current demand levels, it is important to note that our distribution performs just as well as the now in production distribution.

Knowing that both distributions can handle current demand, we then observed what differences the distributions had. We found that the biggest difference between the distributions was the amount of space the distributions required. Because the in production distribution was not altered between density runs, the average free space per apphost was the same, at an average use of 620 GB. In contrast, the average amount of space used on an apphost for our genetic algorithm's final distribution was only 87 GB as shown in Table 8. This means that our simulated distribution saves space on apphosts without causing any stress to the system's performance. The amount of space saved for each apphosts is significant. This increase in space can be used as space where OnLive can grow their catalog of games and applications without purchasing any new hardware. We see that even as we increase demand, we continue to save space with our distribution. This means that our simulations distribute applications at a much lower ratio while maintaining OnLive's ability to handle every user's request up to a much higher demand level.

	Actual Site Density	5x Site Density	10x Site Density
In Production Distribution	620 GB	620 GB	620 GB
Genetic Algorithm's Distribution	87 GB	186 GB	480 GB
Reduced Used Space By:	**533 GB (86%)**	**434 GB (70%)**	**140 GB (23%)**

Table 8: Total space used per AppHost

4.7.2 Minimum Number of AppHosts

After analyzing our data from the simulation, we saw the average number of apphosts in use increased proportionally to a change in the arrival density. We then tested what the minimum number of apphosts for a site would be given user demand. In Figure 34, one can see that as density increases so does the number of servers in use. This data shows that the number of apphosts necessary to accommodate a given demand correlates linearly. Five times the density means five times the number of apphosts are necessary so the site is not overwhelmed.

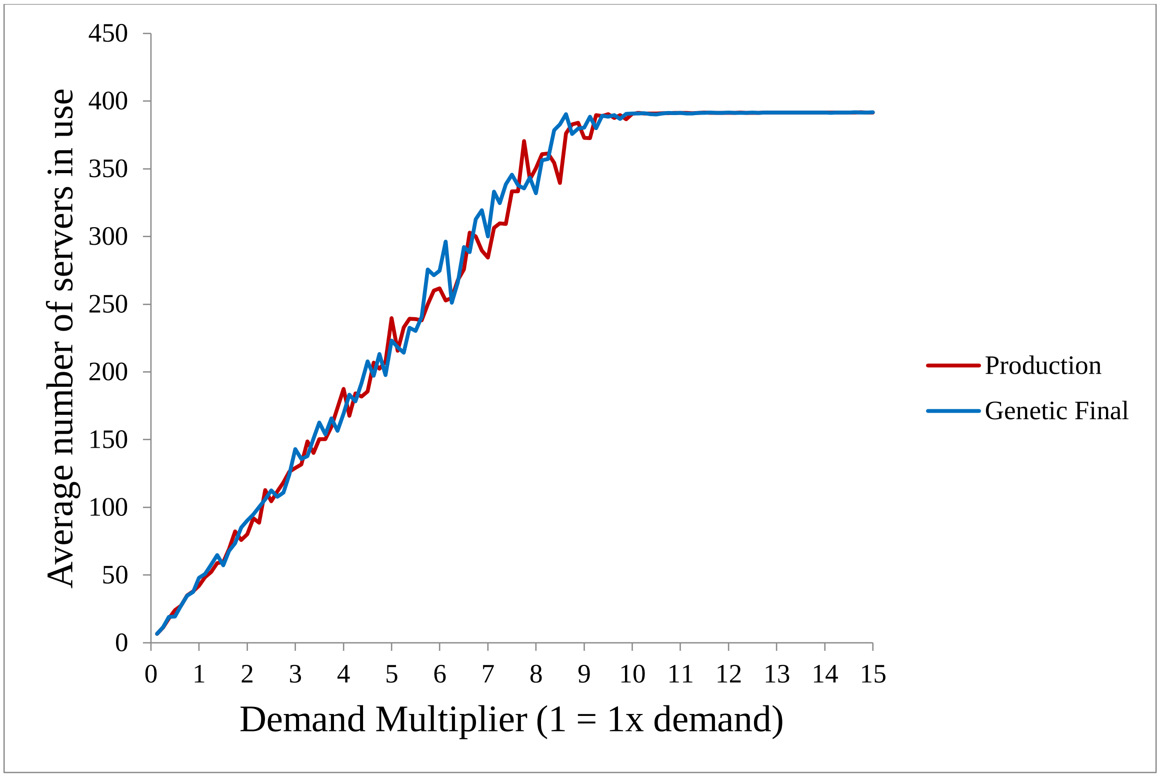


Figure 34: Server activity vs. demand for in production and genetic distributions

5 Conclusion

OnLive needed a process to calculate the success of their content distributions. Our project addressed the limitation of their current distribution system and provided a tool to calculate success.

Our team successfully built a simulator that accomplished this goal as well as implemented a new distribution algorithm. To accomplish this, we integrated a model of the OnLive network with the Mason simulation library. We developed distribution-matching tools for the simulator to fit data to statistical distributions. We then carefully performed verification checks to confirm the simulation output data matched the real life data OnLive gave us. We developed a genetic algorithm that automates the process to find a better distribution by repeatedly changing distribution ratios between generations. In addition, we successfully defined success criteria for distributions that were not known before the project. Moreover, we build graphical and console interfaces for the project that allows the user to load different modules and modify different properties of the simulation. We ensured all the code was well documented and testing coverage was more than 90%.

We found that our simulator better distributes content to OnLive's worldwide server network. We conclude, after running the simulation, that OnLive currently uses more space on their apphosts than is minimally required to meet demand. The average free space per apphost is large in our new distribution, and our application configuration behaves as successfully as the current distribution on the network. We provided OnLive a tool that can better estimate system behavior without making changes to the live network until desired. Our simulator can test different behavioral scenarios and distribute content when a better distribution is found.

5.1 Future Work

There remain opportunities for growth, and this section outlines various growth initiatives for our software. These future work opportunities are optional functionality that proved attractive during development and can be implemented after the conclusion of our project.

5.1.1 Time Dependent Distribution Changes

There exists a cost when making a distribution change on live apphosts. This cost occurs when content must be added or removed from the disk space on the apphost. When a change is being made, the apphost enters a busy state and cannot be used by customers until the change is complete. This means that some apphosts will be taken down whenever a change is made at the expense of the users. The amount of time that an apphost remains inactive is based on how much must be added or removed. More data that must be changed means more time that the apphost will not be in use. This time may be predicted by our simulator at the end of the distribution change as well. This can even be an argument in the genetic algorithm. The more data that must be added or removed the worse a generation can look in the genetic algorithm. In this way, for future development, add to the success metric in the genetic algorithm a parameter for how much space / time a distribution change will make. In this way, the genetic algorithm will work towards finding not only an optimal distribution, but one that take the least amount of time to make a distribution change. Implementing this feature could solve the minimum-cost flow problem.

5.1.2 Multi-Site Implementation

Our current simulator only calculates better distributions for one site at a time. While we have functionality to observe and analyze global data, our simulator only alters distribution configuration on a site basis. To improve the functionality of the simulator, it is possible to allow for multiple site alterations at once. This is possible by specifying which sites need to be altered before starting the simulation. The software can then run the simulation and genetic algorithm simultaneously for each site and output separate configuration files for each of these sites. As the usage data can be input from a data dump file of global data, our simulator can sift through that data and pull out only relevant usage statistics for the site in the simulation. Implementing this feature will allow the functionality to update all sites in production at once.

5.1.2.1 Site Switching

It is possible to extend the software further after implementing multiple site optimizations. If we allow for multiple sites to be interpreted at once, one can find distributions that work across sites. In the future, if a user is unable to get an apphost at the site they are currently at because the app requested is unavailable, perhaps it is possible to switch to a site that does have that app available. This process of site switching changes the user behavior and app session behavior. This is a possible optimization for region usage data and for when the app catalog becomes too large to fit entirely on a single site.

5.1.3 Latency Based on IP Address

OnLive catalogs user data including the IP address and where a user is connected to connect a user to a site they are closest to geographically. The reason for this is because the further a user is from the site they are connected to, the greater the latency of content. In this way, the users in our simulation can add a parameter that specifies where they are located in the world in relation to the sites currently in operation. We can analyze real world data stored in OnLive databases and trace where users are located so that the simulator matches actual worldwide geographical behavior. With this parameter, our simulator can calculate which sites and users suffer from the greatest amount of latency. With this information, our simulator may even be extended to suggest a location in the world to place a new site to best satisfy usage geographically.

5.1.4 Improve the Genetic Algorithm

The genetic algorithm functions well enough to converge to a better distribution than what existed previously, but there are improvements that can be made. This section outlines a few changes that may be applied in the future.

5.1.4.1 Initial State Improvement

One way to make the simulation run faster is to improve the genetic algorithm starting point. Our current implementation of the genetic algorithm creates an initial state solely based on popularity. This is a good enough starting point to push forward in our development, however it can be improved to also factor in session length. If an app is not very popular but has a very long average session length, then it needs to be distributed more. Calculating session length's influence on distribution ratio can be done through multiple runs of the current genetic algorithm to find how apps with the largest session lengths were altered. Then the initial state generator algorithm can account for this relationship to front load computations that would have been sorted out in more time with the genetic algorithm

5.1.4.2 Change Multiple Distribution Ratios at Each Generation

Another improvement to the genetic algorithm is how it currently changes distribution ratios between generations. The genetic algorithm, as it works in our implementation, only changes the worst distribution ratio for a single app and tests various ratios for that one app. This functionality can change to alter multiple app distribution ratios at once. To do this, rather than changing the one worst app, the genetic algorithm can change all distributions that had errors in the last run with multiple permutations. If done right, the genetic algorithm will take less time to find a better distribution.

5.1.4.3 Scale-Back Algorithm

The genetic algorithm works to find a distribution with the most misses and then increases that app's distribution ratio. However, there is no way to go about reducing a distribution ratio if it has no misses and is distributed too much. One way to go about this would be to reduce all application distribution ratios that did not see any misses by a given number, perhaps 5%. The genetic algorithm will then continually reduce the distribution until it records an app's first miss, then it will set the app's distribution ratio to the last ratio with no misses. This will ensure that no app is overly distributed. It is important to add this feature, as it will make sure that apps are distributed just enough and not any more. This saves space on the apphosts as well as make sure distribution do not just grow and grow over time.

Bibliography

Bode, Karl. (June, 2010). "Broadband Streaming Game Platform OnLive Launches". DSLreports.com. http://www.dslreports.com/shownews/Broadband-Streaming-Game-Platform-OnLive-Launches-108955

Fishman, George. (2001). *Discrete Event Simulation - Modeling, Programming, and Analysis.* New York: Springer-Verlag.

Grant, Christopher. (April 2009). "GDC09 interview: OnLive founder Steve Perlman wants you to be skeptical". Joystiq. http://www.joystiq.com/2009/04/01/gdc09-interview-onlive-founder-steve-perlman-wants-you-to-be-sk

Goldman, Tom. (December 2009). "OnLive Fully Detailed in Columbia University Presentation". The Escapist. http://www.escapistmagazine.com/news/view/97182-OnLive-Fully-Detailed-in-Columbia-University-Presentation

Hua-Jun Hong, De-Yu Chen, Chun-Ying Huang, Kuan-Ta Chen, and Cheng-Hsin Hsu. (October 2013). “QoE - Aware Virtual Machine Placement for Cloud Games”. Department of Computer Science, National Tsing Hua University. http://mmnet.iis.sinica.edu.tw/pub/hong13_cloud_game_vm.pdf

Jacob, Matthew. (January 2013). Discrete Event Simulation. *Resonance, Volume 18, Issue 1*, Pages 78 - 86.

Johnson, David S. (1973). Near-optimal bin packing algorithms. Massachusetts Institute of Technology. Dept. of Mathematics. Massachusetts Institute of Technology. http://hdl.handle.net/1721.1/57819

Leadbetter, Richard. (January 2010). *In Theory: Is this how OnLive works?*. http://www.eurogamer.net/articles/digitalfoundry-onlive-beta-article

Mason. (2013). George Mason University, Evolutionary Computation Laboratory, GMU Center for Social Complexity. http://cs.gmu.edu/~eclab/projects/mason/

Mathworks. (2013). *Discrete-Event Simulation in Simulink*. MathWorks. http://www.mathworks.com/products/simevents/description2.html

Osherove, Roy. (2009). *The Art of Unit Testing: With Examples in .Net*. New York: Manning Publications.

Pegden, Dennis. (1985). *Introduction to Siman*. Pennsylvania State University. http://informs-sim.org/wsc85papers/1985_0013.pdf

Appendices

Appendix A: List of Module Types and Implementations

Module Types

Module	**Purpose**
ApplicationConfigModule	Configures appvariant data
ResultOutputModule	Outputs statistics about the simulation
ServerConfigModule	Configures apphost data
PerSimConfigReader	Sets configuration parameters such as number of users on a per-sim basis
AppHostCatalogProvider	Controls how appvariants are distributed to apphosts
IADArguments	Reads command-line arguments that would normally be passed to the iad.py file from a configuration file.
Summarizer	Summarizes the IAD run, includes all features
DistributionRuleProvider	Provides YAML configuration data for dist ratios
AppHostGroupAssigner	Performs the assignment step of the IAD algorithm
TargetAppHostSelector	Selects the best-suited apphost to be assigned an appvariant group in a step of AppHostGroupAssigner
SurplusUnloader	Unloads (selects) surplus apphosts from an appvaraint group
AppSelectionProvider	Determines which appvariant a user wants to play
ArrivalInterval	Determines when a user will arrive
SessionLength	Determines how long the user will play for
AppHostSelector	Determines which apphost (if any) will serve the user
WeakestLinkFinder	Finds the weakest link in a generation
MetricCalculator	Computes the objective function of a simulation run
InitialConfiguration	Computes the initial distribution ratios for the DistributionRuleProvider
MetricComparer	Compares metrics together to determine if the results are better than before or are within certain thresholds

Implementations by Module Type

ApplicationConfigModule	**Purpose**
ApplicationConfigReader	Reads application.config
IADAppVariantReader	Calls a modified iad.py to get the appvariant list
TraceAppConfigReader	Gets the applications from a trace file
SerializationDatabaseRunner	De-serializes data
DatabaseDumpRunner	Reads a dump of database queries matching GLuMySQLDatabaseRunner
GluMySQLDatabaseRunner	Queries the GluDB, just like the IAD does

ResultOutputModule	Purpose
NullReporter	Does not output anything
ExcelReporter	(not included by default) Outputs to an excel file, uses a lot of memory
RatioToMissReporter	Reports the product-key, app type, (target) distribution ratio and number of g16s
ServerInUseCountReporter	A time map of how many servers are in use

ServerConfigModule	Purpose
ServerConfigReader	Reads server.config for server prototype data
SerializationDatabaseRunner	De-serializes data
DatabaseDumpRunner	Reads a dump of database queries matching GLuMySQLDatabaseRunner
GluMySQLDatabaseRunner	Queries the GluDB, just like the IAD does

PerSimConfigReader	Purpose
SimulationConfigReader	Applies each line from simulation.config once
RepeatingSimulationConfigReader	Applies a single line from simulation.config endlessly

AppHostCatalogProvider	Purpose
RandomCatalogProvider	Randomly assigns a random number of Apps to a server
IADCatalogProvider	Calls a modified iad.py
DistribtuionAlgorithm	An IAD Skeleton in java

IADArguments	Purpose
IADConfigurationHelper	Reads from iad.config arguments that would be passed in on command-line to the IAD

Summarizer	Purpose
IADSummarizer	Summarizes the IAD results in the same way the IAD does now

DistributionRuleProvider	Purpose
YAMLDictionaryHelper	Reads from iad_config.yaml the YAML configuration rules for the IAD distribution

AppHostGroupAssigner	Purpose
IADDistributionAssigner	Generates the IAD’s work queues and executes them

TargetAppHostSelectior	Purpose
IADTargetMostSpace	Selects the “host-est with the most-est”

SurplusUnloader	Purpose
RandomSurplusUnloader	Unloads apphosts above target at random

AppSelectionProvider	Purpose
PopularitySelectionProvider	Weights appvariants by popularity
RandomSelectionProvider	Rolls a fair die
TraceAppRunner$TraceAppSelectionProvider	Reads a line from a session trace

ArrivalInterval	Purpose
ExponentialAnalysisArrivalInterval	An exponential distribution
GenericExponentialArrivalInterval	An exponential distribution estimated from a session trace
TraceAppRunner$TraceAppDistribution	Reads a line from a session trace

SessionLength	Purpose
GenericGammaSessionLength	A gamma distribution
GenericLevySessionLength	A Lévy distribution
GenericNormalSessionLength	A normal distribution
GenericWeibullSessionLength	A Weibull distribution
WeibullAnalysisSessionLength	A Weibull distribution estimated from a session trace
TraceAppRunner$TraceAppDistribution	Reads a line from a session trace

AppHostSelector	Purpose
FirstFitAppHostSelector	Grabs the first available apphost
LeastPopularAppHostSelection	Grabs the apphost with the least popular catalog
LeastWasteAppHostSelector	Grabs the apphost with the fewest appvariants
RandomAppHostSelector	Grabs a random available apphost

WeakestLinkFinder	Purpose
MostMissesLinkFinder	The weakest link is the appvariant that had the most failures, but is not at 1.0 distribution

MetricCalculator	Purpose

FailureCounter	Sums up all failures and one hundred times all giveups

InitialConfiguration	Purpose
DefaultInitialConfiguration	Uses iad_config.yaml as a starting point
GeneticSeedConfiguration	Computes a starting point from the propularity.txt trace data

MetricComparer	Purpose
FailureCounter$FailureMetricComparer	Allows -50% growth to be considered an improvement.

Appendix B: Software Development Tracking

Tasks List	Description of Task	Difficulty?	Progress
GUI config	GUI interface that changes configuration and runs simulation (no display)	3	Completed
Better data recording	Improve the efficacy of the ExcelReporter by recording data like failures and sessions	2	Completed
Configuration file for servers	Create a configuation scheme to define different server types and the ratios at which they appear on the site	1	Completed
Configuration file for Apps	Create a configuration scheme to define the applications that will be placed on servers for the simulation run	1	Completed
Configuration for content distribution	Create a configuration scheme that allows the customization of how Apps are distributed to the servers	2	Completed
Configuration for content selection	Create a configuration scheme that distributes how users choose the application that they wish to play	2	Completed
Model user play time accurately	Improve the accuracy of the calculation of how long a user will play the application that they have requested	3	Completed
Copy IAD process	Create a CatatlogProvider that is able to mirror the distribution process of the IAD on a given system to a reasonable degree of accuracy	5	Completed
Generate list of applications such that it mirrors OnLive's catalog	By having our App catalog mirror the catatlog used by OnLive, we can more closely model the reality of OnLive's services	2	Completed

Generate server config such that it mirrors a target site	By mirroring the server distributions of an OnLive Site, we can show that our model can simulate a real-world environemnt successfully	2	Completed
Sanity Check model	Create a case that can be worked out by hand to show that the model behaves as expected	1	Completed
GUI Tracking Failures Chart	A graph that displays Failures over time	2	Completed
GUI Tracking Graph	Track Failures on a real-time graph through the GUI	3	Completed
Integrate IAD - lightweight	Import an our project to the site whose applist we wish to simulate.	4	Completed
Make the clock distibutions modular	We want to make modules that can be loaded without touching the code that change the way things like session length are distributed	2	Completed
Console output for the GUI	When the GUI opens, we also have a console window that prints out everything like the Console in Eclipse	2	Completed
Server Selection as module	The method by which the server is selected may change over time, so we wish to make a convenient way to change the way servers are selected to serve a user	2	Completed
Construct a trace example of onlive's services	Being able to run our simulation using a trace of real world data allows us to check our model against the real world to prove its accuracy	3	Completed
Add real-time analytics to simulation	In order to improve our simulation's usefulness, we need to be able to analyse factors like server load in real time so as to best be able to improve our model.	4	Completed
Modular System	Easy to integrate modules that configure the simulation	4	Completed
Re-write IAD from Python into Java	Re-writing the IAD into Java will allow better integration with our model and allow us to build on the functionality at our will	5	Completed
Set up Window Builder	Get the Window Builder Library and set it up for the actual GUI implementation	2	Completed
Pull from Site Database	Glu commands that pull usage data for list of apps from a live site	3	Completed
Analyze Site Data and configure the model	changes # of sessions/users/play time based on popularities in the current system state	3	Completed

Validate Trace to server configuration that mimics onlive's systems	In order to demonstrate that a trace simulation is a true representation of OnLives systems, validate our output with live servers	3	Completed
Serialize site data	Serialize site data as a way to run the simulation without pulling from the GLU db every time	2	Completed
Model user app selection accurately	Construct a SelectionProvider that is able to provide application selections that mirror real-world conditions	3	Completed
GUI for configuring Module use	Drop Down or check boxes that let you select which modules you would like to run for the simulation, back end	3	Completed
GUI Load and Save Config	Load and Save Configurations from the GUI	2	Completed
GUI User Experience	Make the GUI more user friendly	3	Completed
Develop structure for Genetic Algorithm Implementation	A Genetic Algorithm will run the simulation until a better distribution is found. Based on various parameters. Requires a metric and Records	4	Completed
Develop Metrics for use in Genetic Algorithm	The metric must differentiate different runs of the simulator. A more successful distribution is a stronger metric.	3	Completed
Develop better initial state for genetic algorithm	In order to improve the efficiency of our genetic algorithm, it should be able to compute a good starting point to iterate from	2	Completed
Genetic Algorithm Feedback	Able to output a new config and use as input for the GA. A refactor	3	Completed
GUI Treading Issue	Resolve an issue where the GUI becomes inert when running a simulation. Need to extract the threading procedure from MASON code.	4	Completed
General GUI Improvements	GUI improvements to improve usability.	2	Completed
(Growth) Time dependent distribution changes	There is a cost of time for each change to the distribution that puts some app hosts in a busy state when updating their catalog. Predict this time. Minimum-cost flow problem	4	Growth
(Growth) Multi-Site Implementation	Able to run multiple sites in a single simulation (pull site specific data and run the simulation for ever site with one button)	3	Growth
(Growth) Latency based on IP address	Calculate latency for users based on distance from site	5	Growth

(Growth) Site Switching	How does the affect of user site switching change behavior (when a user gets g16, does changing to another site help?)	2	Growth
(Growth) Develop better algorithm for generations	Our genetic algorithm should be able to alter more than one parameter per generation and do so without adversely affecting other app variants	4	Growth

Appendix C: Test Coverage Tracking

Java Classes tested for Code Coverage	Name of Tests that test the class	% Code Coverage
toy_model\FilePaths.java	A Class that keeps constant String	98.20%

toy_model\ServerGenerator.java	generateServers_numServerInConfigGreaterThanZero_ReturnListOfServerNotEmpty();
	generateServers_numSeversInConfigIsZero_ReturnEmptyServerList()
		95.70%

toy_model\analysis\Analyzer.java	analyze_PassInTraceDataWithEmptyRawData_ReturnNull();
	analyze_PassValidTraceData_ReturnExpectedTraceAnalysis();
		100.00%

toy_model\analysis\DistributionData.java	see toy_model\analysis\Modeler.java	100.00%
toy_model\analysis\Modeler.java	This class generate distribution data and was tested visually	70.40%
toy_model\analysis\TraceData.java	see toy_model\analysis\TraceDataReader.java	77.80%

toy_model\analysis\TraceDataReader.java	computeTrace_parseFile_getASetOfTraceDataMatchesTheTrace();
	computeTrace_parseFileTraceExisted_addSession()
		82.60%

toy_model\analysis\TraceToDist.java	testComputeTraceData()
	testWritePopularityData()
	testWriteArrivalData()
	testWriteSessionData()
		46.40%

toy_model\config\AbstractConfigReader.java <Abstract Class>	contructor_EmptyFile_emptyConfigReader(); constructor_TwoLinesFile_ListOfTwoString(); getNextLine_EmptyFile_returnEmptyString(); getNextLine_TwoLinesFile_returnFirstLine(); getNextLine_LinesWithEmptyLineOnTop_returnNull()	75.70%

toy_model\config\ApplicationConfigReader.java	testConfigure_SetsApps()
	testConfigure_HasAppAsExpected()
		85.90%

toy_model\config\Configuration.java	intialize(); hasNextUserConfiguration_perSimisNotNull_returnTrue();
	hasNextUserConfiguration_perSimisNull_returnFalse();
	hasNextUserConfiguration_perSimhasOneConfigFilebutTheFileIsEmpty_returnFalse();
	hasNextUserConfiguration_perSimhasTwoConfigFilesbutBothFileIsEmpty_returnFalse();
	hasNextUserConfiguration_perSimhasThreeConfigFilesButOnlyOnehasContentInIt_returnTrue();
	getNextConfiguration_perSimisNull_returnDEFAULT_CONFIG();
	getNextConfiguration_perSimhasThreeConfigFilesButOnlyOnehasContentInIt_returnConfiguration()
		93.60%

toy_model\config\IADAppVarientReader.java	testInternalExtractAppData_HasNineApps()
	testInternalExtractAppData_HasGooButHasNoGSP()
		50.30%

toy_model\config\RepeatingSimulationConfigReader.java	configure_testFile_configureTheSameNumberFromTheTestFile()	98.80%

toy_model\config\ServerConfigReader.java	testConfigure_HasThreePrototypes()
	testConfigure_HasExpectedRatio()
		100.00%

toy_model\config\SimulationConfigReader.java	testConfigure_HasExpectedValues()	100.00%

toy_model\config\TraceAppConfigReader.java	testConfigure_HasThreeApplications()
	testConfigure_HasGSPAlienMoW()
		100.00%

toy_model\config\entity\ExtendedServerPrototype.java	create_listOfsiteHostIsEmpty_instanceVarIsZero()
	create_listOfsiteHostIsEmpty_returnDefaultExtendedProtoAppHost()
	create_ListOf2AppHostCallCreate2Time_returnTheSecondAppHost()
	create_ListOf1ElementCallCreate2Times_returntheDefaultAppHost()
	create_addToListofElement_checkifInstanceCountIsCorrect()
		100.00%

toy_model\config\entity\ServerPrototype.java	getInstanceCount_TestServerTypeName_ReturnInstanceCountEqualToSize();
	create_TestCreateServerFromOneServerPrototype_ReturnServerWithCorrectServerType()
		100.00%

toy_model\config\module\ModuleConfiguration.java	This class actually coverage is 96.3%; the other 8% cannot be covered	96.30%

toy_model\config\module\ModuleConfigurator.java	search_moduleNotExistInModuleList_returnFalse()
	search_moduleExistInModuleList_returnFalse()
	checkDependencies_theRequiresModuleClassDoesnotImplementModule_ThrowException()
	checkDependencies_DontHaveTheDependencyClass_ThrowException()
	validateAllModules_theRequiresModuleClassDoesnotImplementModule_ThrowException()
		95.10%

toy_model\database\DatabseChunkConfigurator.java	testDatabaseChunkConfigurator()
	testConstructPrototypeList()
	testExtractAppCerts()
	testExtractAppHost_IsNotNull()
	testExtractAppHost_HasExpectedIdentity()
	testExtractAppVariant_IsNotNull()
	testExtractAppVariant_HasExpectedIdentity()
	testExtractAppVariant_HasNoNodes()
	testExtractAppVariant_AfterInitNodes_HasNodes()
	testExtractContainerNodeData()
	testExtractPreloadData()
		100.00%

toy_model\database\DatabaseDumpRunner.java	testGetAppVariants_HasSizeThree()
	testGetAppVariants_HasAmnesiaAsSecondEntry()
	testGetAppHosts_HasSizeTwo()
	testGetAppHosts_IsFiftyFifty()
		100.00%

toy_model\entity\AppVariant.java	equals_IfAppAHasIdenticalIDAsAppB_ReturnTrue();
	equals_IfOneAppDoesnotHasIdenticalIDAsTheOtherApp_ReturnFalse();
	equals_IfParamIsNotAnApp_ReturnFalse(); equals_IfParamIsNull_ReturnNull();
	equals_IfParamIsNull_ReturnFalse()
		100.00%

toy_model\entity\AppHost.java	addApplication_AppSpaceLargerThanAppHostSpaceAvailable_ReturnFalse();
	addApplication_AppSpaceLowerThanAppHostSpaceAvailable_ReturnTrue();
	addApplication_AppSpaceEqualAppHostSpaceAvailable_ReturnTrue();
	addApplication_AppSpaceisNegative_ReturnFalse()
		97.60%

toy_model\entity\Site.java	supports_appInSite_ReturnTrue(); supports_siteDontHaveApp_ReturnFalse();
	supports_appIsNull_ReturnFalse(); getServers_InvalidApp_ReturnEmptyListOfServer();
	getServers_ValidApp_ReturnListOfServer(); getServers_NullApp_ReturnEmptyListOfServer();
	getAServer_AppIsValidAndAllServerAvailable_AppHost();
	getAServer_AppIsValidAndAllServerNotAvailable_ReturnNull();
	getAAppHost_AppIsNull_ReturnNull(); getAServer_AppIsInvalid_ReturnNull()
		100.00%

toy_model\entity\distribution\FirstFitAppHostSelector.java	select_IfListOfServerIsEmpty_ReturnNull() select_IfAllServerNotAvail_ReturnNull()
	select_ListNotEmptyAndAllServerAvail_ReturnServer()
	select_ListHasOneElement_ReturnThatElement()
	generateServers_numSeversInConfigIsZero_ReturnEmptyServerList()
	generateServers_numServerInConfigGreaterThanZero_ReturnListOfServerNotEmpty()
		100.00%

toy_model\entity\distribution\IADCatalogProvider.java	assignCatalog_testIfTheIADCatalogProvidergivePopulateCorrectAppliCationList_returnVoid()	81.30%

toy_model\entity\distribution\LeastPopularAppHostSelector.java	toPopularity_passInCollectionOfAppVariant_returnTotalPopularity()
	select_IfListOfServerIsEmpty_ReturnNull() select_IfAllServerNotAvail_ReturnNull()
	select_ListNotEmptyAndAllServerAvail_ReturnServer()
	select_ListHasOneElement_ReturnThatElement()
	generateServers_numSeversInConfigIsZero_ReturnEmptyServerList()
	generateServers_numServerInConfigGreaterThanZero_ReturnListOfServerNotEmpty()
		100.00%

toy_model\entity\distribution\LeastWasteAppHostSelector.java	select_ListOfTwoAppHost_ReturnAppHostWithTheLeastWaste()
	select_IfListOfServerIsEmpty_ReturnNull() select_IfAllServerNotAvail_ReturnNull()
	select_ListNotEmptyAndAllServerAvail_ReturnServer()
	select_ListHasOneElement_ReturnThatElement()
	generateServers_numSeversInConfigIsZero_ReturnEmptyServerList()
	generateServers_numServerInConfigGreaterThanZero_ReturnListOfServerNotEmpty()
		100.00%

toy_model\entity\distribution\PopularitySelectionProvider.java	excecute_putInListOfPopularity_returnExpectedValue()
	selectApp_RandomIsZeroPoint4_returnAppVariant()
	selectApp_RandomIsZeroPoint9_returnAppVariant() selectApp_RandomIs1_returnAppVariant()
		94.80%

toy_model\entity\distribution\RandomAppHostSelector.java	select_IfListOfServerIsEmpty_ReturnNull() select_IfAllServerNotAvail_ReturnNull()
	select_ListNotEmptyAndAllServerAvail_ReturnServer()
	select_ListHasOneElement_ReturnThatElement()
	generateServers_numSeversInConfigIsZero_ReturnEmptyServerList()
	generateServers_numServerInConfigGreaterThanZero_ReturnListOfServerNotEmpty()
		100.00%

toy_model\entity\distribution\RandomCatalogProvider.java	getApplications_checkIfApplicationPassesInCorrectly_returnListOfApplication()
	assignCatalog_10AppVariantAndChooseRandomly8OfThem()
	assignAllCatalog_10AppVariantAndChooseRandomly8OfThemForEachAppHost()
		94.20%

toy_model\entity\distribution\RandomSelectionProvider.java	select_IfListOfServerIsEmpty_ReturnNull(); select_IfAllServerNotAvail_ReturnNull();
	select_ListNotEmptyAndAllServerAvail_ReturnServer();
	select_ListHasOneElement_ReturnThatElement();
	generateServers_numSeversInConfigIsZero_ReturnEmptyServerList();
	generateServers_numServerInConfigGreaterThanZero_ReturnListOfServerNotEmpty();
	generateServers_NumServerIsNegative_ReturnEmptyServerList()
		100.00%

toy_model\entity\distribution\TraceAppRunner.java	testTraceAppSelectionProvider()
	testSelectApp()
	testNextSLength_ThrowsWithoutInitedTraceAppSelectionProvider()
	testNextArrival_ThrowsWithoutInitedTraceAppSelectionProvider()
	testNextSLength_ThrowsWithoutMatchingTraceAppSelectionProviderCall()
	testNextArrival_ThrowsWithoutMatchingTraceAppSelectionProviderCall()
	testNextSLength_WorksWithMatchingTraceAppSelectionProviderCall()
	testNextArrival_WorksWithMatchingTraceAppSelectionProviderCall()
		91.70%

toy_model\entity\distribution\iad\DistributionAlgorithm.java	equals_twoKeyDoubleisIdenticalInSomeSpecificField_returnTrue()
	equals_twoKeyarenotIdentical_returnFalse() equals_oneKeyTupleisNull_returnFalse()
	equals_comparetoNullKey_returnFalse()
	equals_twoGroupHaveAreIdenticalInKeyTuples_returnTrue()
	equals_twoGroupHaveKeyTuplesThoseAreNotIdenticalToEachOther_returnFalse()
	equals_compareToNullGroup_returnFalse() equals_compareToObject_returnFalse()
		100.00%

toy_model\entity\distribution\iad\IADConfigurationHelper.java	testConstructor_passInFile_checkIfReadCorrectly()	100.00%

toy_model\entity\distribution\iad\IADDistributionAssigner.java	equals_twoWorkItemIdentical_returnTrue() equals_twoWorkItemNotIdentical_returnFalse()
	equals_compareWithNullObject_returnFalse() equals_ObjectIsNotWorkItemTuple_returnFalse()
	compareTo_TwoWorkKeyTupleIsIdentical_return0()
	compareTo_SurplusRatioEqualButRandomIsGreater_ReturnValueGreaterThan0()
	compareTo_SurplusRatioEqualButRandomIsSmaller_ReturnValueGreaterThan0()
	compareTo_SurplusRatioSmaller_ReturnValueGreaterThan0()
	compareTo_SurplusRatioGreater_ReturnValueGreaterThan0()
		100.00%

toy_model\entity\distribution\iad\IADPreconditions.java	testIADPreconditions_NullArgument() testIADPreconditions_NotNullArgument()
	testIsAppHostAllowed_AcceptsWpi() testIsAppHostAllowed_RejectsNotWpi()
	testMeetsServiceReqs_MatchesInclusive_Accept()
	testMeetsServiceReqs_MatchesExclusive_Reject()
	testMeetsServiceReqs_DoesntMatchInclusive_Reject()
	testMeetsServiceReqs_DoesntMatchExclusive_Accept()
	testDeferAppVariant_DeferIfNoContainers() testDeferAppVariant_DoNotDeferIfContainers()
	testIsHostOperational_DeniesInoperativeHost() testIsHostOperational_AllowsOperativeHost()
	testIsAppEnabled_AcceptsEnabledAppVariant() testIsAppEnabled_DeniesUnenabledAppVariant()
		100.00%

toy_model\entity\distribution\iad\IADSummarizer.java	just print methods	100.00%

toy_model\entity\distribution\iad\IADTargetMostSpace.java	compareTo_return0() compareTo_returnNegativeNumber() fitInCache_notEnoughASpace_ReturnNull() fitInCache_enoughSpace_returnNotNull() popBestTargetAppHost_appHostListIsEmpty_returnNull() popBestTargetAppHost_appHostListHas1ElementButThatElementCannotFit_returnNull() popBestTargetAppHost_appHostListHas1ElementButThatElementIsFit_returnThatElement()	96.00%
toy_model\entity\distribution\iad\LoadState.java	This is an enumerator class	100.00%
toy_model\entity\distribution\iad\RandomSurplusUnloader.java	selectSurplusAppHostsToUnload_returnListOfAppHost()	100.00%
toy_model\entity\distribution\iad\RuleHelper.java	parseBounds_FirstAndLastCanBeParseIntoInteger_returnInteger() parseBound_FirstAndLastCannotBeParseIntoInteger_throwException() parseBound_dataIsEmpty_returnDefault() evalSliceFiler_sliceIsNotOfCorrectFormat_returnNull() evalSliceFiler_sliceIsOfCorrectFormat_returnAPredicate() evalSliceFiler_sliceIsOfCorrectFormatButNotInteger_returnNull() evalSliceFiler_sliceValueDontHaveDash_returnAPredicate() testPredicate_ValidSliceRange_returnTrue() testPredicate_InvalidSliceRange_returnFalse() testPredicate_passInSingleValue_returnTrueIfEquals() testPredicate_passInSingleValue_returnFalseIfNotEquals()	100.00%
toy_model\entity\distribution\iad\SliceFilterHelper.java	testEvalSliceFilter_SingleArgAcceptsSelf() testEvalSliceFilter_SingleArgDoesntAcceptsOther() testEvalSliceFilter_TwoArgAcceptsSelfs() testEvalSliceFilter_RangeArgAcceptsRange() testEvalSliceFilter_RangeAndSingleArgAcceptsRange() testEvalSliceFilter_SkipsBadFormat() testParseBounds_RequiresAtLeastTwoArgs() testParseBounds_RequiresNumericalArguments() testParseBounds_RequiresIntegerArguments() testParseBounds_CanTakeEmptyArguments() testParseBounds_DoesParseArgs()	100.00%
toy_model\entity\distribution\iad\YAMLDisctionaryHelper.java		80.00%
toy_model\entity\extended\AppHostExtendedData.java	execute_loadNewKeyValueToEmptyList_returnTrueAndListHas1Element() execute_loadNewKeyValueofEmptyList_checkReturnValueField() execute_unloadEmptyList_ListStillEmpty() execute_unloadListHas1Element_ListBecomeEmpty() execute_load2ElementThenunload1Element_ListContain1ElementLeft() execute_unloadListHas2Element_CheckreturnValue() execute_loadKeyAlreadyEsist_returnValueDontIncrease() loadAppVariant_containerNodesContain2Elements_returnValueDesired() unloadAppVariant_containerNodesContain2Elements_returnValueDesired()	97.80%
toy_model\entity\extended\AppVariantExtendedData.java	equals_TwoAppVariantExtendedDataHasTheSameID_returnTrue() equals_TwoAppVariantExtendedDataHasDiffID_ReturnFalse() equals_OneAppVariantExtendedDataIsNull_ReturnFalse()	100.00%
toy_model\genetic\DefaultInitialConfiguration.java	testGetInitialState()	100.00%

toy_model\genetic\FailureCounter.java	testComputeMetricFor()
	testComputeMetricFor_HasNSurrenders()
		100.00%

toy_model\genetic\GeneticSeedConfiguration.java	testGetInitialState_AssignsDistRatioOneToMostLikedApp()
	testGeneratePopularityMap_NotNull()
	testGeneratePopularityMap_NoGSP()
	testGeneratePopularityMap_HomeFrontIsMax()
		87.90%

toy_model\genetic\GeneticUsers.java	GeneticUsers.GenerationRuleProvider:
	testGenerationRuleProvider_NotNull()
	testGetDirectives_IsNull()
	testOffer()
	testSetDirectives()
	
	GeneticUsers: This class' actual coverage is 8%, the other 92% belongs to the simulation code and cannot be tested
		8.00%

toy_model\genetic\MostMissesLinkFinder.java	testComputeWeakestLink()
	testComputeWeakestLink_NotSame()
	testComputeWeakestLink_TryImproveNotNull()
	testGetWeakestApp()
		88.00%

toy_model\genetic\RatioToMissReporter.java	testWriteData()
	testComputeRatios()
		61.70%

toy_model\genetic\ServerInUserCounterReporter.java	testWriteData()	56.70%

toy_model\records\ExcelReporter.java	testAsExcelTime()
	testGetOrCreateSheet_NewSheetNotNull()
	testGetOrCreateSheet_RepeatCallIsSame()
	testWriteArray()
	testWriteConfig()
	testWriteFailureData()
	testWriteFailures()
	testWriteServerData()
	testWriteServerLoadData()
	testWriteSessionData()
	testCreateAndSetWithValueXSSFSheetXSSFRowCollectionOfAppVariantTObjectIntMapOfAppVariant()
	testCreateAndSetWithValueXSSFSheetXSSFRowIntDouble()
	testCreateAndSetWithValueXSSFSheetXSSFRowIntInt()
	testCreateAndSetWithValueXSSFSheetXSSFRowIntString()
	testGetFirstEmptyRow_NotNull()
	testGetFirstEmptyRow_StartsAtZero()
	testGetFirstEmptyRow_RepeatCallIsSame()
	testGetFirstEmptyRow_NullCellIsEmptyRow()
	testGetFirstEmptyRow_EmptyStringIsNotEmptyRow()
	testGetFirstEmptyRow_NonEmptyStringNotEmptyRow()
	testGetFirstEmptyRow_NonEmptyStringNotEmptyRow_Index()
	testNextRow_NotNull()
	testNextRow_IsIndexPlusOne()
	testNextRow_RepeatCallIsSame()
		80.90%

toy_model\records\Records.java	testRecords()
	testRecordsListOfAppHost()
	testRecordsListOfAppHostHasMatchingList()
	testGetAllServers_startsEmpty()
	testAddServer()
	testGetAllFailures_startsEmpty()
	testAddFailure()
	testGetAllSessions_startsEmpty()
	testAddSession()
	testGetFailuresPerApp()
	testSetServersPerApp()
	testSetUsersPerApp()
		100.00%

toy_model\records\Session.java	testSession()
	testGetDuration()
	testGetFinishTime()
	testGetStartTime()
		90.90%

toy_model\util\CollectionHelper.java	union_AunionB_returnUnionSet() intersection_AintersectB_returnIntersectSet()
	intersection_AandBDoNotIntersect_returnEmptyList() intersection_AcontainB_returnB()
	difference_AandB_returnTheDifference() difference_AContainB_returnElementInABultNotB()
	difference_AContainB_returnEmptyList() containsAny_thereExistIntersection_ReturnTrue()
	containAny_thereIsNoIntersection_returnFalse()
		100.00%

toy_model\util\ConstructorHelper.java	See toy_model\config\module\ModuleConfigurator.java	57.10%

toy_model\util\Pair.java	equals_twoPairisIdentical_returnTrue() equals_TwoPairIsNotIdentical_returnFalse()
	equals_FirstFieldMatchButNotTheSecond_ReturnFalse()
	equals_SecondFieldMatchButNotTheFirst_ReturnFalse() equals_ParameterIsNull_ReturnFalse()
	equals_ParameterIsNotAPairObject_ReturnFalse() toString_returnDesireName()
		95.30%

toy_model\util\Predicate.java	both_bothPredicateIsTrue_returnTrue() both_onePredicateisFalse_returnFalse()
	both_firstPredicateisNull_returnValueOfSecondPredicate()
	both_secondPredicateisNull_returnValueOfFirstPredicate() not_predicateIsTrue_returnFalse()
		95.90%

toy_model\util\TraceAnalysisDataReader.java	extractAllTraceData_passedInTestFile_checkparams()	93.60%