Rochester Institute of Technology

RIT Digital Institutional Repository

Theses

8-2016

Modelling of Information Flow and Resource Utilization in the EDGE Distributed Web System

Bryan Thomas Meyers
btm5529@rit.edu

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Meyers, Bryan Thomas, "Modelling of Information Flow and Resource Utilization in the EDGE Distributed Web System" (2016). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Modelling of Information Flow and Resource Utilization in the EDGE Distributed Web System

by

Bryan Thomas Meyers

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of Master of Science in Computer Engineering
Department of Computer Engineering
Kate Gleason College of Engineering

Supervised by

Dr. Edward Hensel, P.E.
Associate Dean for Research and Graduate Studies
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
August 2016

Modelling of Information Flow and Resource Utilization in the EDGE Distributed Web System

by

Bryan Thomas Meyers

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science
in Computer Engineering

Supervised by

Dr. Edward Hensel, P.E.
Department of Kate Gleason College of Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York

August 2016

Approved by:

___

Dr. Edward Hensel, P.E., Associate Dean for Research and Graduate Studies
Thesis Advisor, Department of Kate Gleason College of Engineering

___

Dr. Muhammad Shaaban, Associate Professor
Committee Member, Department of Computer Engineering

___

Dr. Andres Kwasinski, Associate Professor
Committee Member, Department of Computer Engineering

Dedication

To the hundreds of open-source developers who have provided us with the tools to build newer and better software. You have spent thousands of hours building, testing, and documenting the languages, libraries, and applications that made this work possible.

We could not have done it without you.

Thank You!

Acknowledgments

I am grateful to my advisor Dr. Edward Hensel for his continued guidance and support for the entire duration of my Masters studies and research. He has been a constant source of wisdom, intuition, and motivation throughout the course of this thesis work. I could not have asked for a better advisor, mentor, and colleague in this endeavor.

I would also like to thank the remaining members of my committee, Dr. Andres Kwasinski and Dr. Muhammad Shaaban, for lending their experience and knowledge to this effort. Without the many hours they spent providing feedback and the many months they spent teaching me in the classroom, I can honestly say that this thesis would not have been possible.

Abstract

Modelling of Information Flow and Resource Utilization
in the
EDGE Distributed Web System

Bryan Thomas Meyers

The adoption of Distributed Web Systems (DWS) into modern engineering design process has dramatically increased in recent years. The Engineering Design Guide and Environment (EDGE) is one such DWS, intended to provide an integrated set of tools for use in the development of new products and services. Previous attempts to improve the efficiency and scalability of DWS focused largely on hardware utilization (*i.e.* multithreading and virtualization) and software scalability (*i.e.* load balancing and cloud services). However, these techniques are often limited to analysis of the computational complexity of the algorithms implemented.

This work seeks to improve the understanding of efficiency and scalability of DWS by modelling the dynamics of information flow and resource utilization by characterizing DWS workloads through historical usage data (*i.e.* request type, frequency, access time). The design and implementation of EDGE is described. A DWS model of an EDGE system is developed and validated against theoretical limiting cases. The DWS model is used to predict the throughput of an EDGE system given a resource allocation and workflow. Results of the simulation suggest that proposed DWS designs can be evaluated according to the usage requirements of an engineering firm, ultimately guiding an informed decision for the selection and deployment of a DWS in an enterprise environment. Recommendations for future work related to the continued development of EDGE, DWS modelling of EDGE installation environments, and the extension of DWS modelling to new product development processes are presented.

Contents

Dedication . . . . **ii**

Acknowledgments . . . . **iii**

Abstract . . . . **iv**

1 Introduction and Literature Review **1**

1.1 Product Development Process 1

1.1.1 Sequential Non-iterative Processes 1

1.1.2 Sequential Iterative Processes 3

1.1.3 Concurrent Processes 5

1.2 Information Management in Product Development 7

1.2.1 Document Management Systems. 8

1.2.2 Document Storage. 8

1.2.3 Document Change Management. 9

1.2.4 Document Version Control. 10

1.3 Distributed Web Systems for Project Management 11

1.3.1 REST 11

1.3.2 Virtualization 12

1.3.3 The Cloud 14

1.4 Parallelism in Computing 14

1.4.1 Decomposition 15

1.4.2 Partitioning 15

1.4.3 Assignment 17

1.4.4 Orchestration 17

2 Problem Statement **19**

2.1 Hypothesis 20

2.2 Statement of Work 22

2.3 Deliverables 25

2.4 Work Schedule 26

2.5 Publication Schedule 26
2.6 Required Resources 28

3 REST Development of EDGE 2.0 . . . . **30**
3.1 EDGE 1.0 System 30
3.1.1 Document Management 31
3.1.2 Project Management 32
3.1.3 Web Collaboration 33
3.1.4 EDGE 1.0 Architecture 34
3.1.5 Product Development Toolkits — FACETS 35
3.2 Design Goals for EDGE 2.0 37
3.2.1 Modern Web Technologies 37
3.2.2 Extensibility 38
3.2.3 Improved Performance 39
3.2.4 Easier Deployment 39
3.3 Evaluation of Software Tools 40
3.3.1 Languages 42
3.3.2 Frameworks 45
3.3.3 Version Control System 48
3.3.4 Persistent Relational Datastore 50
3.3.5 Templating 53
3.3.6 Mark-up Language 56
3.3.7 Summary 58
3.4 Implementation 59
3.4.1 REST Compatibility 60
3.4.2 REST Framework — Wire 62
3.4.3 REST Endpoints — Applications 64
3.4.4 Database Schema 67
3.4.5 Subversion Integration 70
3.4.6 Web Front-End 72
3.4.7 Deployment 74
3.4.8 Summary of Open-Source Software Used in EDGE 75
3.5 Extensibility 77

4 Dataflow Model: Theory and Development **82**
4.1 Assumptions and Limitations 83
4.1.1 Assumptions 83

4.1.2 Limitations . . . 85
4.2 Implementation . . . 86
4.2.1 Processing . . . 87
4.2.2 Networking . . . 88
4.2.3 Configuration . . . 91
4.2.4 Datasources . . . 95
4.2.5 Metrics . . . 99
4.3 Functional Testing . . . 103
4.3.1 Unit Testing . . . 103
4.3.2 Component Verification . . . 104
4.3.3 Simulator Integration Testing . . . 110

5 Dataflow Model: Verification Results . . . 115
5.1 Workload Characterization . . . 115
5.1.1 EDGE 1.0 Request Characterization . . . 115
5.1.2 KGCOE-Research Client Characterization . . . 117
5.1.3 KGCOE-Research Workload Timelines . . . 119
5.2 Dataflow Simulation Results . . . 123

6 Conclusions and Recommendations for Future Work . . . 128
6.1 Conclusions . . . 128
6.2 Future Work . . . 129
6.2.1 Future Research . . . 129
6.2.2 Future Development . . . 130

Bibliography . . . 131

A Code Listings . . . 135

List of Tables

3.1 FACETS Toolboxes . . . 36
3.2 Open-Source Applications . . . 75
3.3 Open-Source Libraries . . . 76
3.4 Open-Source Development Tools . . . 76

4.1 Proposed Metrics . . . 99
4.2 Example DWS Quantities . . . 100
4.3 Example Resource Throughputs . . . 102
4.4 Unit Testing Results for DWSim . . . 104
4.5 Metric Summary . . . 106

List of Figures

1.1 Waterfall Process . . . 2
1.2 Stage-Gate . . . 2
1.3 Spiral Process, reproduced from [7] . . . 4
1.4 Agile Process, reproduced from [23] . . . 4
1.5 V-Model Process, reproduced from [25] . . . 5
1.6 Concurrent Engineering Process, adapted from [8] . . . 6
1.7 Set-based Concurrent Engineering Process, adapted from [26] . . . 7
1.8 Decomposition Process . . . 16
1.9 Partition Example . . . 16
1.10 Assignment Example . . . 17
1.11 Orchestration Example . . . 18

2.1 Work Schedule . . . 27
2.2 Publication Schedule . . . 29

3.1 EDGE 1.0 Architecture . . . 35
3.2 EDGE 2.0 Architecture . . . 59
3.3 UI Examples for EDGE 1.0 (Left) and EDGE 2.0 (Right) . . . 60
3.4 Wire Architecture . . . 63
3.5 EDGE 2.0 Database Schema . . . 68
3.6 Example Wire::App Implementation . . . 78
3.7 Example Renderer Configuration . . . 79
3.8 Wire Authorization Interface . . . 81

4.1 Example Task Dependency Graph . . . 88
4.2 Example CMP CPU . . . 89
4.3 Network Model Architecture . . . 89
4.4 Network Latency . . . 91
4.5 Generated EDGE 1.0 Assignment Graph . . . 93
4.6 EDGE 1.0 Assignment Dot File . . . 94
4.7 Generated EDGE 1.0 Orchestration Graph . . . 94
4.8 EDGE 1.0 Orchestration Dot File . . . 95

4.9 Example DWSim Configuration File 96
4.10 Example of Combined Log Format for Apache HTTPd 96
4.11 ApacheLog2DB SQL Schema 97
4.12 LMS Bandwidth Approximation, reproduced from [38] 98
4.13 Exemplar Workload for CMP, colors indicate task 105
4.14 Exemplar Utilization for CPU, colors indicate task 105
4.15 Processor Utilization (ω_g): 1 Core 107
4.16 Processor Utilization (ω_g): 2 Cores 107
4.17 Processor Utilization (ω_g): 4 Cores 108
4.18 Processor Utilization (ω_g): 8 Cores 108
4.19 Processing Speedup 109
4.20 Exemplar Client Workload for Test, $\tau_n[Requests/s], \Delta t = 1[s]$, [08:41 to 08:45] 110
4.21 Annotated Heatmap Component Utilization, $\omega[Binary/s], \Delta t = 10[ms]$ . . 111
4.22 Heatmap of Component Utilization, $\omega[Binary/s]\Delta t = 10[ms]$ 112
4.23 Heatmap of Component Latency, $\lambda[s], \Delta t = 10[ms]$ 112
4.24 Heatmap of Component Throughput, $\tau_n[Binary/10ms], \Delta t = 10[ms]$ . . . 113
4.25 Heatmap of Average Component Utilization, $\bar{\omega}[-/100ms], \Delta t = 10[ms]$ . . 114

5.1 Distribution of HTTP Methods for EDGE Hosts for 2015-08-01 through 2016-06-01 116
5.2 Distribution of URI Requests for EDGE Hosts for 2015-08-01 through 2016-06-01 117
5.3 Map of IP Sources for kgcoe-research.rit.edu [2015-08-01 to 2016-01-07] . 118
5.4 Client Bandwidth Distribution ($\bar{\tau_g}$) for kgcoe-research.rit.edu [2015-08-01 to 2016-01-07] 119
5.5 Client Latency Distribution ($\bar{\lambda_g}$) for kgcoe-research.rit.edu [2015-08-01 to 2016-01-07] 120
5.6 Apache Request Throughput for kgcoe-research, $\tau_n[requests/week], \Delta t = 10[\mu s]$, [2015-08-01 to 2016-06-01] 121
5.7 Apache Request Throughput for kgcoe-research, $\tau_n[requests/day], \Delta t = 10[\mu s]$, [2015-10-01 to 2015-11-01] 121
5.8 Apache Request Throughput for kgcoe-research, $\tau_n[requests/hr], \Delta t = 10[\mu s]$, [2015-10-13 12:00 to 2015-10-14 12:00] 122
5.9 Apache Request Throughput for kgcoe-research, $\tau_n[requests/min], \Delta t = 10[\mu s]$, [00:00:00 to 00:01:00] on 2015-10-14 122

5.10 Apache Log Request Throughput for kgcoe-research, $\tau_n[request/s]$, [00:00:40 to 00:00:45] on 2015-10-14 123
5.11 DWSim Mean Utilization Heatmap for kgcoe-research, $\bar{\omega}_n[\%/s], \Delta t = 10[\mu s]$, [00:00:40 to 00:00:45] on 2015-10-14 124
5.12 Apache Client Requests vs. DWSim Throughput for kgcoe-research, $\tau_n[request/s], \Delta t = 10[\mu s]$, [00:00:40 to 00:00:45] on 2015-10-14 125
5.13 Cumulative Apache Client Requests vs. DWSim Throughput for kgcoe-research, $\Sigma\tau_n[requests], \Delta t = 10[\mu s]$, [00:00:40 to 00:00:45] on 2015-10-14 125
5.14 DWSim Client Request Mean Latency Bar Graph for kgcoe-research, $\bar{\alpha}[s], \Delta t = 10[\mu s]$, [00:00:40 to 00:00:45] on 2015-10-14 126
5.15 DWSim Client Request Mean Latency Bar Graph (Revised) for kgcoe-research, $\bar{\alpha}[s], \Delta t = 10[\mu s]$, [00:00:40 to 00:00:45] on 2015-10-14 127

Chapter 1

Introduction and Literature Review

The first section of this literature review discusses a variety of product development processes used in modern commerce, and supported by the Engineering Design Guide and Environment (EDGE). The second section defines the information management requirements common to product development processes, as well as the features of EDGE which satisfy them. The third section reviews the approaches and challenges associated with modelling and implementing EDGE as a modern Distributed Web System (DWS). The fourth section discusses the use of parallelism in computer programming.

1.1 Product Development Process

Product development processes typically define the order in which the tasks of development take place. A project manager may choose to follow a process which best suites the kind of development to be performed. If the project represents one piece of a larger design effort, this may lead to the execution of multiple development processes simultaneously. A DWS intended to support product development efforts should support a variety of processes, and allow for concurrent execution of disparate workflows. Common product development processes are described here.

1.1.1 Sequential Non-iterative Processes

Waterfall The Waterfall process represents a basic strategy for executing product development. In this process, each stage is carried out sequentially (Fig 1.1), with its influence "trickling down" to each subsequent stage of development [7]. It is criticized as being one

of many models which do not respond well to change, requiring considerable reworking of previous stages [24]. Most modern implementations of Waterfall acknowledge the presence of feedback loops between stages [7].

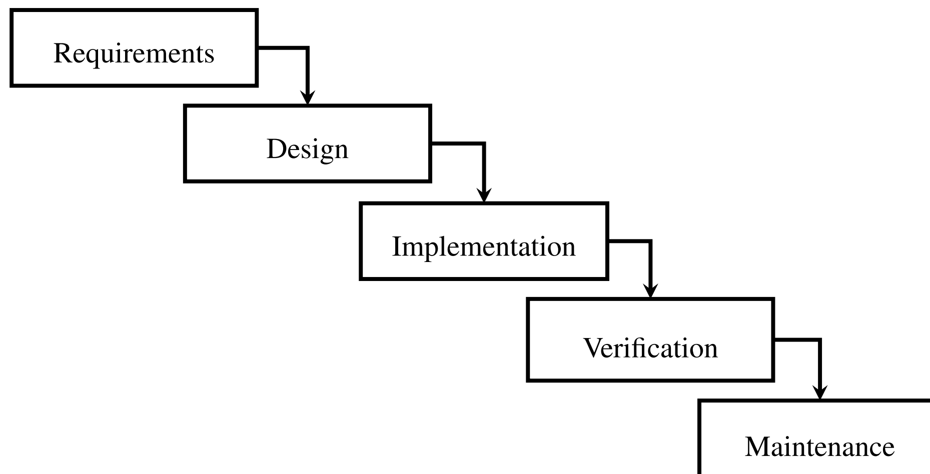

Figure 1.1: Waterfall Process

Stage-Gate The Stage-Gate approach to product development improved upon the Waterfall Model by introducing the idea of quality control checkpoints (Fig 1.2), also referred to as "gates" [11]. These gates introduce and enforce validation of design decisions and verification of expected outcomes between each stage of the process. Because each successive stage of the process is viewed as increasingly expensive, these gates provide feedback which may prevent early transition to the next stage. The Stage-Gate process is intended to reduce the overall cycle time for a given product by limiting iterations back to earlier stages of design throughout the life of a project.

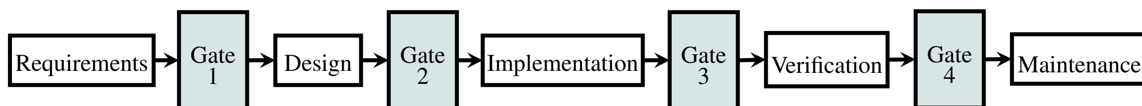

Figure 1.2: Stage-Gate

1.1.2 Sequential Iterative Processes

Spiral The Spiral Model has been developed in response to experience gained while utilizing the Waterfall Model [7]. It is depicted by polar graph with a single timeline, spiraling outward about the origin (Fig 1.3). Each quadrant corresponds with an iterative phase of the development process. The radial distance of the timeline indicates the cumulative cost of the project to date. The arc length indicates the passage of time. A completed rotation is followed by a review of all progress to date and marks the completion of an iteration cycle. Real-world examples of the spiral process may include automotive year models and the Lenovo ThinkPad line of laptops. Spiral processes focus on risk reduction, guiding toward decisions that result in only acceptable consequences. It also allows for a project to easily return to an earlier iteration or stage, which may be especially useful for "dead-end" lines of inquiry.

Agile In software development, it is common for customers to make large changes to requirements multiple times between inception and delivery. This is likely a side-effect of a customer not knowing or understanding exactly what this product should be from the start. As a result, software development firms cannot rely on processes which require detailed planning or are resistant to adaptation, as they take too long to respond to requirements changes [24]. The Agile family of development processes follows a core model in which normal development is carried out over multiple shorter iterations with periods of reflection and prioritization in-between (Fig 1.4). Each iteration is intended to produce a working product with some level of the expected functionality. The project itself may continue for as many iterations as the customer can afford or until they are satisfied that the implementation is complete. Agile processes capitalize on the individual strengths of team members, while expecting high levels of collaboration and the self-organization of task management [10].

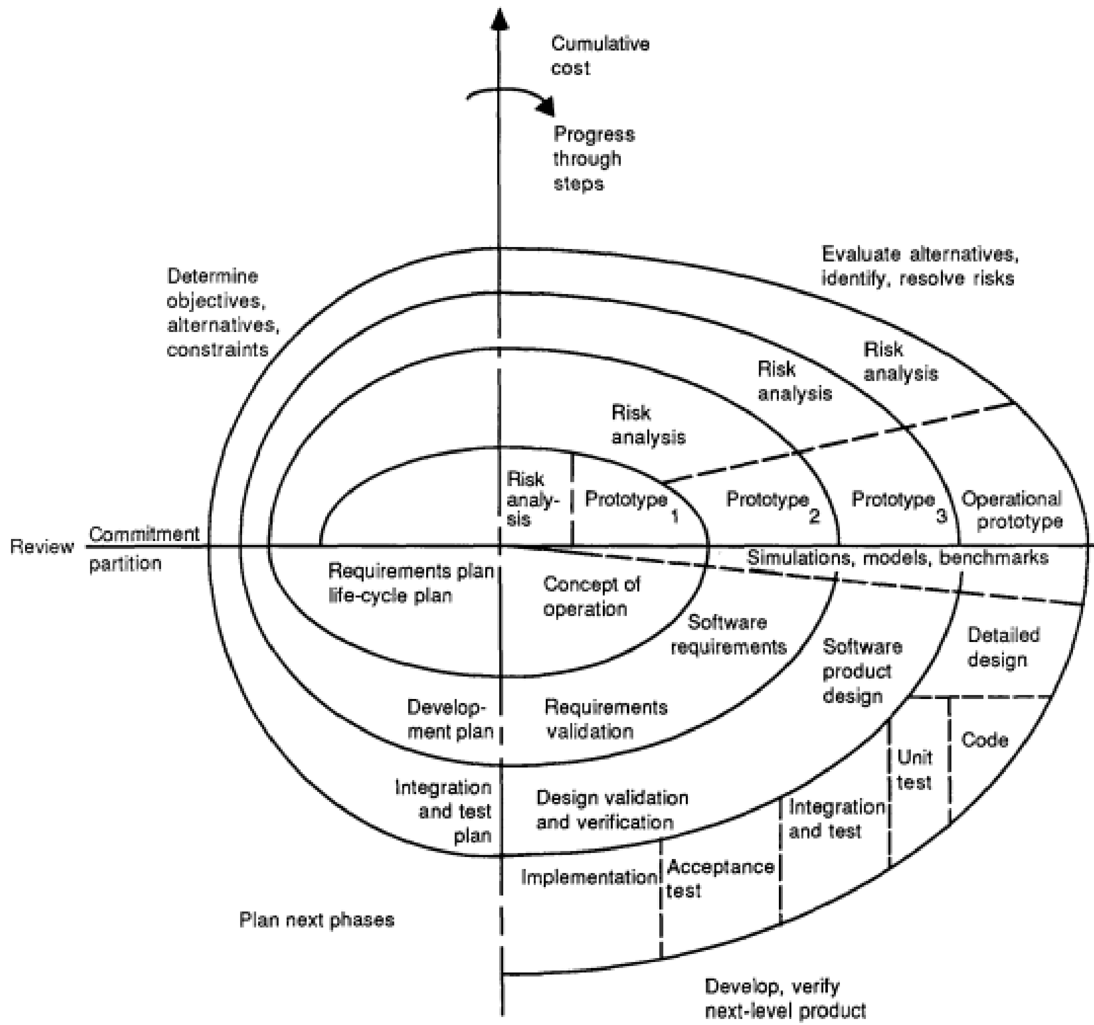

Figure 1.3: Spiral Process, reproduced from [7]

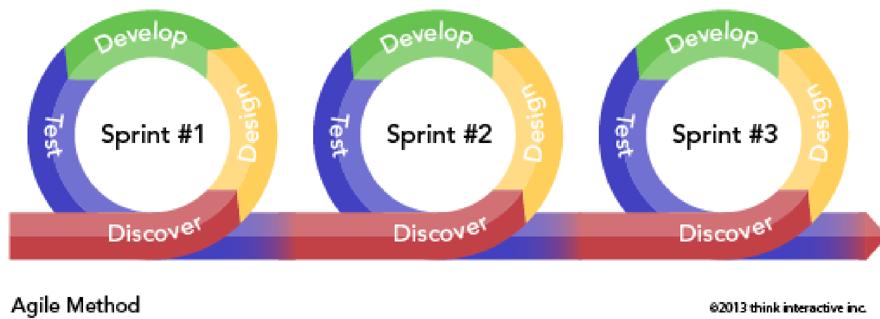

Figure 1.4: Agile Process, reproduced from [23]

Systems Engineering V-Model The Systems Engineering approach might, at first glance, be viewed as a non-iterative process (Fig 1.5). A development team first performs the Decomposition and Definition phase, followed by Implementation, and completes the work by undergoing Integration and Verification [18]. However, this makes the assumption that all tasks are performed nominally and that no problems arise in the Integration and Verification phase of development. Each phase involves tasks which may undergo multiple iterations before continuing onward (similar to Spiral). When problems occur in the Integration and Verification phase, it is necessary to return to earlier tasks to rectify the problems [18]. The later the problem occurs, the farther back it will be necessary to regress. The Systems Engineering V-Model encourages developers to continuously monitor the relationship between design and verification, in order to prevent the expensive and time consuming process of having to go back later on.

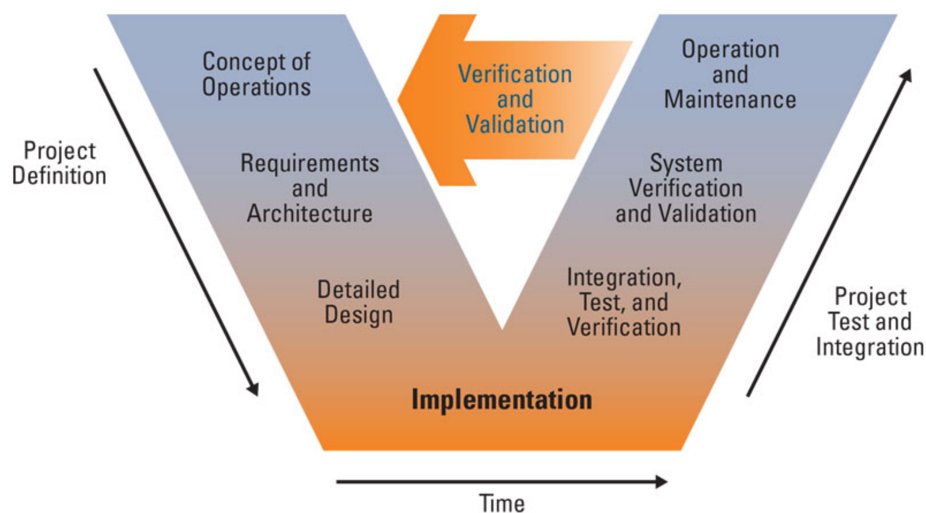

Figure 1.5: V-Model Process, reproduced from [25]

1.1.3 Concurrent Processes

Concurrent Engineering While many design processes focus on the sequence or iterations of design phases, Concurrent Engineering (CE) processes assume that most large problems

consist of multiple smaller sub-problems which may be solved simultaneously [8]. Functional, manufacturing, and structural teams should be able to work on different aspects of the same product, while communicating changes to design parameters, as illustrated in Figure 1.6. Each team may choose the solution that best meets their own sub-problem, while receiving feedback from other teams to allow for compatibility and optimization. Each team may follow a *different* design process internally, but the overarching design process is carried out concurrently. Solving these sub-problems simultaneously may reduce the overall development timeline.

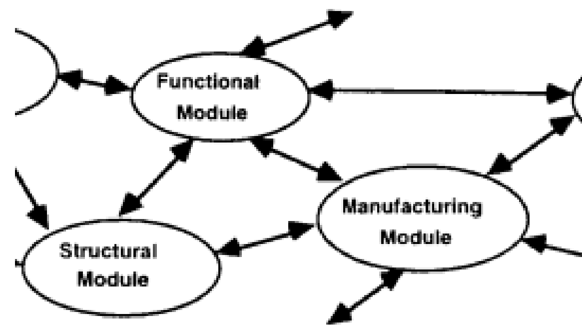

Figure 1.6: Concurrent Engineering Process, adapted from [8]

Set-based Concurrent Engineering One of the most prominent examples of Set-Based Concurrent Engineering (SBCE) is the Toyota Product Development Process [42]. Contrary to CE, SBCE allows each development team to explore a wide solution space, instead of focusing on a single proposed solution. Each sub-problem is evaluated separately, as in CE. However, the distinguishing feature of SBCE is the simultaneous exploration of multiple solutions to a given sub-problem. This allows an existing solution to be conservatively improved generation to generation, while also allowing for the development of entirely new and higher risk-bearing solutions (Fig. 1.7). At any point in time a new product can be formulated by combining sub-problem solutions. This flexibility requires that the interface between sub-problems be established ahead of time and may need to be treated as a set of

sub-problems in a separate iteration of the design process.

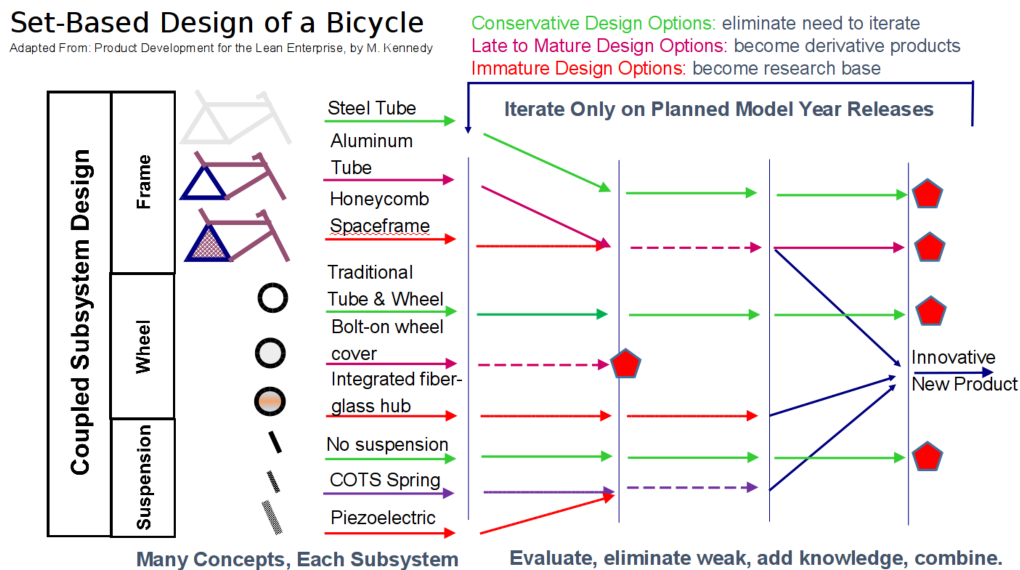

Figure 1.7: Set-based Concurrent Engineering Process, adapted from [26]

1.2 Information Management in Product Development

Much of the information presented in this section has been reproduced from the previously published conference paper [32]. It is included here in order to allow the thesis to act as a stand-alone document.

Many tools exist to facilitate collaboration for various aspects of design. Wikipedia [41] popularized a collaborative document editing environment which incorporated a simple syntax for document markup. This open source package, known as MediaWiki [31], enables developers to build upon the concept of collaborative editing by extending its core functionality with new and useful features. Google Docs [21] allows simultaneous editing of files by multiple users. Cloud file-sharing services such as Dropbox [13] and OneDrive

[35] synchronize files between collaborators. Several project management software packages such as Redmine [40] and Microsoft Project [34] permit users to perform task management and Gantt charting. CAD packages often integrate version control systems for tracking changes to solid model and drawing files [39]. Software development packages such as NetBeans [36] and Eclipse [14] provide an Integrated Development Environment (IDE) for concurrently designing and developing software packages. Version control packages such as CVS [44] and Subversion [33] permit developers to track the history of changes to documents. Few tools integrate the majority of these valuable document management and information flow capabilities into an integrated engineering design environment.

1.2.1 Document Management Systems.

Proper documentation is a critical component of any engineering design and product development work-flow. It enables contributors not only to capture the progress of a project through meeting minutes and request for change (RFC) documents, but also to organize the artifactual documents produced during the various stages of the project. Software developers have a long history of performing document management as a companion tool to a well-defined and cyclical software process [20]. Most traditional engineering disciplines employ formalized document control and change management (revision control) processes for critical documents such as design drawings. EDGE applies the concepts of document management to *all* documents generated and used by a design team. Structured document management procedures permit iterative changes to documents by multiple team members working in a collaborative environment. An effective document management system must incorporate at least three features: document storage, document change management, and document version control.

1.2.2 Document Storage.

The storage of documents is often taken for granted. However, document storage may be a complicated issue, particularly when multiple individuals must access and modify a

single set of documents. Four important issues must be addressed in an effective document storage system:

1. The **location(s)** of documents to be stored must be established. Examples of document storage locations include stacks of papers on a desk, hanging folders in a file cabinet, a formal library and archiving system, local electronic document storage, or cloud-based storage.

2. The **format(s)** in which documents are to be stored must be agreed upon. The format includes physical media such as paper, microfiche, optical disk or magnetic disks. The format also includes the logical structure of documents, such as chapters and sections, CAD file structures, and naming conventions.

3. **Access** to the storage location(s) by individuals must be authorized and authenticated. Access control is necessary to prevent viewing, theft, or modification of documents by unauthorized individuals. Authorization is the process of verifying that an individual is permitted to have certain types of access (such as read, edit, destroy), while authentication is the process of verifying that the person attempting the access is indeed who they say they are.

4. The storage system must include a mechanism for **change** to documents. This may be as simple as replacing the document entirely, or more complicated, requiring a document to be only partially modified to reflect changes.

1.2.3 Document Change Management.

Most organizations implement strict processes to be followed when altering design documents. Document change management consists of a set of tools for handling modifications to documents. Consider an example whereby Person A begins to edit a document, and she spends one hour writing. Five minutes after Person A begins editing, Person B makes several minor editorial corrections to the document and saves his changes 30 minutes after

Person A began editing. At this point, Person A has no knowledge of Person B's changes. After 60 minutes have passed, Person A finishes her work, and saves her own changes. This means that Person B's efforts will be completely written over by Person A's efforts. This is known as the "lost update" problem, and is one of the largest concerns in document management. Change management solves this problem with a few simple rules:

1. An author must begin the editing process by first making a local copy of the document to be edited.
2. The author may alter that file using the appropriate tools (word processor, CAD system, *etc.*).
3. When done making changes, the first author must verify that no intervening changes have been made by a second author to the original document.
4. If a second author has edited the document, the first author must merge the changes of the second author into their local copy of the document.
5. The first author may return their local copy of the document to the storage location.

Because this process is performed iteratively, change management tools work to automate the process, and only require user intervention for merging the two sets of changes.

1.2.4 Document Version Control.

A Version Control System (VCS) tracks the history of changes made to documents over an interval of time. Each set of changes to a document or group of documents is assigned a unique version number, in chronological order of modification. A VCS allows authorized users to view any previous version of any document. Often, a difference, or "diff", tool may simplify the side by side comparison of differences between two versions of a document. This can be useful for recovering old versions of content, or for looking at the evolution of a document over time. Such revision history also permits a thorough understanding of the

evolution of an engineering design over time. This information is particularly valuable for product failure and product liability investigations.

1.3 Distributed Web Systems for Project Management

Early web environments relied upon monolithic web servers to perform simple tasks such as email distribution or static web page serving [27]. At the time, the speed of available network connections limited the growth of these systems. The advent of broadband communications made it possible for much more information to be quickly transferred over client connections. Multimedia served as one practical use for this bandwidth. This nurtured the adoption of the internet as a place for commerce and information sharing [27]. The internet grew from approximately 23,500 websites and 44.8 million users in 1995, to an astounding 969 million websites and 2.93 billion users in 2014 [43].

It is no longer sustainable for the web to grow on the monolithic model. The inherent reliability and performance limitations of single server environments necessitate the adoption of infrastructure which relies upon thousands of servers just to provide a single service [19]. This increase of scale has led to the rapid increase in the size and number of datacenters across the globe. Concerns over the efficiency and limits of this degree of expansion are growing in light of the rise of the "Internet of Things" (IoT) and so-called "Big Data" storage networks [12]. A clear effort to model and predict the throughput, efficiency, and scalability for DWS is necessary to ensure this expansion will remain sustainable.

1.3.1 REST

In an attempt to formalize the concept of the Internet, Roy Fielding published a doctoral dissertation entitled "Architectural Styles and the Design of Network-based Software Architectures" in the year 2000 [17]. This document described the distinct features of various existing network architectures and defined a single model to describe the underlying architecture of the entire World Wide Web (WWW). He called it Representational State Transfer

(REST), in which the WWW could be seen as a stateless protocol for the transmission and translation of information across geographic and organizational boundaries.

REST relies upon three key requirements to allow the web to grow efficiently. First, all REST content must be accessible through Uniform Resource Identifiers (URIs) which specify a particular document based on its Location (URL) and its Name (URN) [6]. Second, all REST interactions with WWW content must be performed using the HyperText Transfer Protocol (HTTP). This stateless communication protocol allows requests related to a URI to be processed and handled in a standardized method. HTTP provides semantic commands to define these interactions. HTTP Methods allow clients to read (GET), create (POST), update (PUT), or remove (DELETE) content [16]. Third, a REST environment must employ the concept of "HyperMedia As The Engine Of Application State" (HATEOAS), in which any request for a URI contains all client information required to handle that request [48]. It is prohibited for a HATEOAS server to store information about a client for use in future requests. Rather, the interlinking nature of HyperMedia documents may be leveraged to assist clients in transitioning to other desirable states, often with regard to workflow or process.

1.3.2 Virtualization

Before the Personal Computer (PC), the most powerful computer systems were mainframes. A mainframe system used time-sharing to allow multiple users to use a single powerful computer [9]. Such machines were popular in academic, corporate, and governmental settings, but were cost prohibitive to consumers. The more cost-effective PC became the defacto standard for home offices and classrooms. The idea of shared resources continued to evolve and became the software mechanism known as threading. Through threading, individual programs are able to time-share the compute resources of a single system. Chip-level multiprocessing (CMP) took this to the next level by having several processors on a single chip, capable of dynamic task switching and multiprogram execution [22].

With the rise in clockspeed and core counts of recent processor designs, it has become

increasingly difficult to efficiently utilize large multicore processors [15]. It is well understood that having more cores on a single processor allows for a greater level of power efficiency and speed than having multiple sockets filled with smaller processors, which must use external buses to communicate [45]. But this leads to concerns of the safety and security of programs sharing the same memory and processing elements. Further, reliability comes in to question when a single program could cause an Operating System (OS) failure and halt the execution of other programs.

These concerns ultimately lead to the development of virtualization, a technique which allows multiple instances of an OS to execute on the same physical hardware. Early attempts at virtualization focused on emulating a computer system in software [5]. The "Guest" OS was provided a finite set of hardware resources and was unaware of being virtualized. This approach was plagued with overheads from emulation, and relied upon a "Host" OS executing in the background to guide and control the environment. One attempt to reduce these overheads involved the creation of Hypervisors, specialized operating systems intended to reduce the cost of management activities [47]. The limitations of this approach inspired CPU designers to further reduce these inefficiencies. By making the "Guest" aware of its virtual environment, it could be given direct access to the physical hardware and made aware of other "Guest" instances on the same hardware [46].

Virtualization allowed an order of magnitude improvement in server efficiency for underutilized environments, subject to certain limitations. In High Performance Compute (HPC) environments, there is little to no benefit to virtualization since the software executing in these systems is designed to fully utilize a single machine. Software for management of large scale virtual environments can be extremely expensive, and a barrier to entry for many firms. High Availability (HA) is a major concern when it comes to the uninterrupted execution of mission-critical systems. If a Host server fails, there is a significant pause in service while Guest instances are migrated and restarted on other available Host servers.

1.3.3 The Cloud

IBM coined the term *cloud computing* to describe the movement of software and virtual machines within these virtual computing environments. More recently, the *cloud* may refer several things. At the infrastructure level, *cloud* refers to the software used to manage these virtual environments, often responsible for ensuring the high availability of services and the distribution of virtual machines [30]. At the OS level, *cloud* may refer to application container environments. These software mechanisms allow multiple services to occupy the same physical or virtual server, while maintaining a certain degree of isolation from each other. At the application level, *cloud* typically refers to the frameworks used to build high performance web systems [37]. These frameworks employ a compute cluster to distribute requests to a web system, sometimes providing mechanisms for synchronizing server-side representations of client state. Today, the consumer *cloud* almost exclusively refers to data storage services hosted in these large virtual environments.

1.4 Parallelism in Computing

Prior to the 21st century, parallelism in software was achieved through clever programming tricks, compiler optimizations, or hand-coded assembly. Data parallelism could be leveraged to allow a single instruction to perform multiple simultaneous calculations or comparisons. Special instruction set architectures (ISA) and vector processors were designed to allow programmers to speed up algorithms in supercomputers. Instruction level parallelism (ILP) became possible with the advent of superscalar architectures. Many functional units work together on a superscalar processor to allow two or more instructions to execute simultaneously. Compilers could then optimize the structure of a program to "preschedule" sequences of instructions. Later, Tomasulo out-of-order execution engines would perform these this scheduling while running a program. Thread-level parallelism first made its appearance with Symmetric Multi-threading (SMT). A processor which supports SMT is able to schedule instructions, from two or more executing programs, on the the same

superscalar hardware. Eventually, chip multiprocessors (CMP) would utilize two or more processor “cores” to execute instructions at the same time. These cores may also consist of complete superscalar processors with dedicated Tomasulo engines and special instructions for data parallel operations.

These advances in computer architecture allowed for significant performance improvements in newer applications, but are only effective if the software supports thread level parallelism. Parallel programming is an essential part of this process, and consists of four major phases: Decomposition, Partitioning, Assignment, and Orchestration. The following sections provide a high-level overview of each phase.

1.4.1 Decomposition

In the Decomposition phase, an algorithm or behavioral model of an application is broken up into smaller pieces. Each piece, or Task, is a self-contained set of actions which represent the smallest meaningful units of work to be performed. If a task is too small, it may not be able to leverage instruction level parallelism. If a task is too large, it may become difficult to realize thread level parallelism in the next phase. Ultimately, it will be up to the experience of the programmer to discern the right size for these tasks.

Having decided on an appropriate set of tasks, it is necessary to define the data dependencies between tasks. Data dependencies serve to define the order of operations to be followed by the resulting software. Some tasks may be performed on multiple occasions in the same program. These temporal data dependencies can be easily resolved by replicating the tasks and using data dependencies to connect these new tasks (Fig. 1.8). The resulting task-dependency graph forms the basis of the parallel program.

1.4.2 Partitioning

In the Partitioning phase, tasks are grouped together to form processes (Fig. 1.9). Tasks should be grouped by keeping tasks which are data dependent together. This reduces the chance a task in one process will need to wait on another task in a different process to

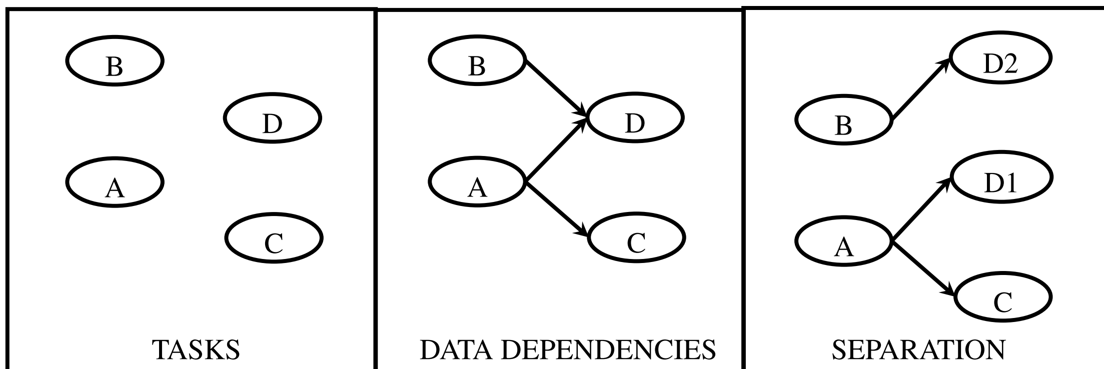

Figure 1.8: Decomposition Process

complete and communicate the result. Each of the processes can then be executed simultaneously and communicate results from one process to another less often. Simultaneous execution of the processes allows for thread level parallelism. Efficient use of a thread parallel processor can be achieved when every process takes roughly the same amount of time to execute.

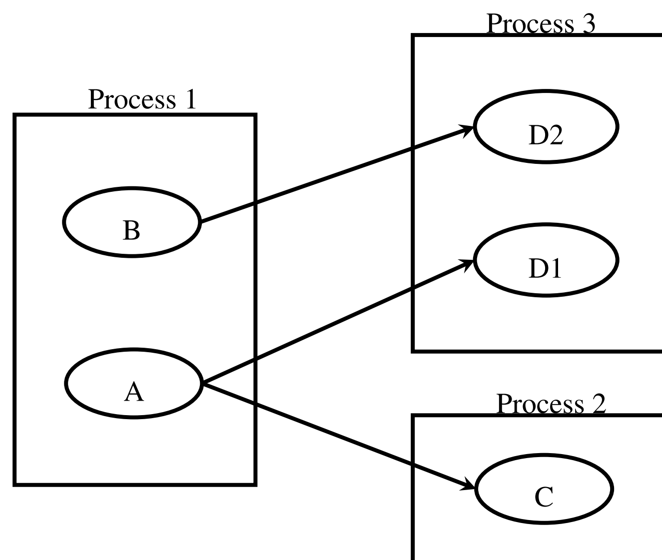

Figure 1.9: Partition Example

1.4.3 Assignment

In the Assignment phase, one or more processes are allocated to a processor for execution (Fig. 1.10). Similar to grouping tasks, processes with immediate data dependencies should be grouped closer together to reduce the need to communicate intermediate results to other process groups. Allocating too few processes will lead to idle processor threads and inefficient use of the hardware. Assigning too many processes to a single processor forces the processor and operating system to frequently switch between running threads. The overhead of switching decreases the throughput of a processor by reducing the useful computational cycles. This can be remedied by either allocating fewer processes or by creating larger group of tasks in the partitioning phase.

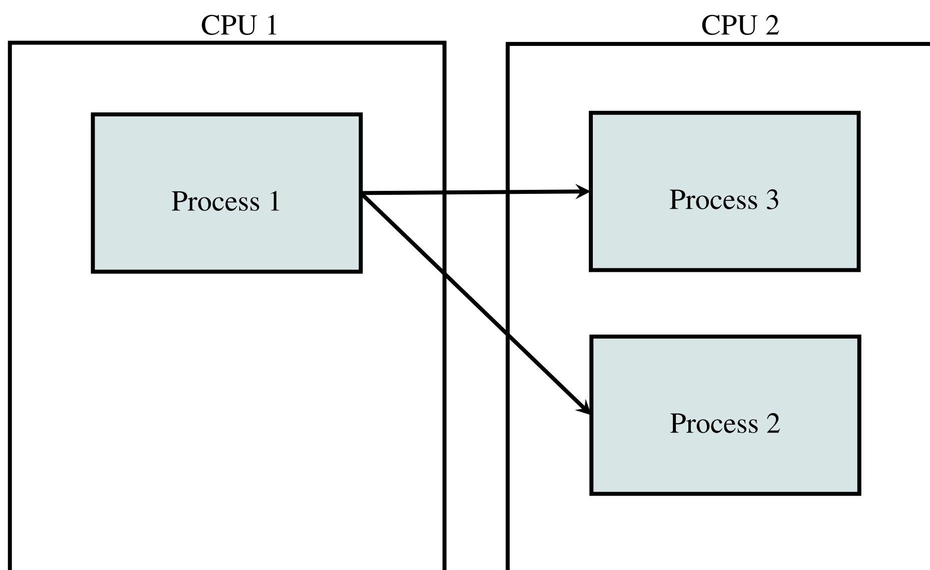

Figure 1.10: Assignment Example

1.4.4 Orchestration

In the Orchestration phase, communication between process groups is coordinated (Fig. 1.11). In the case of a library like OpenMPI, orchestration is carried out automatically and a developer need only make the program aware of the locations of different processors. Communication mechanisms may be chosen by the orchestration library, or manually specified

by the developer. For process groups in a shared memory environment (*i.e.* CMP), communication is optimally carried out by sharing pointers to memory locations. For process groups on on different computers, communication may be facilitated by specialized hardware, or by leveraging an existing computer network. Infiniband is a specialized hardware protocol which allows computers equipped with dedicated interface cards to communicate via a low-overhead protocol which copies memory from one machine to another. A TCP/IP interface may also be used to communicate between machines. This connection will have higher latency and has significantly higher overhead for the communication itself. It may also be necessary to build additional security around the protocol when communicating between machines on different subnets or in different datacenters.

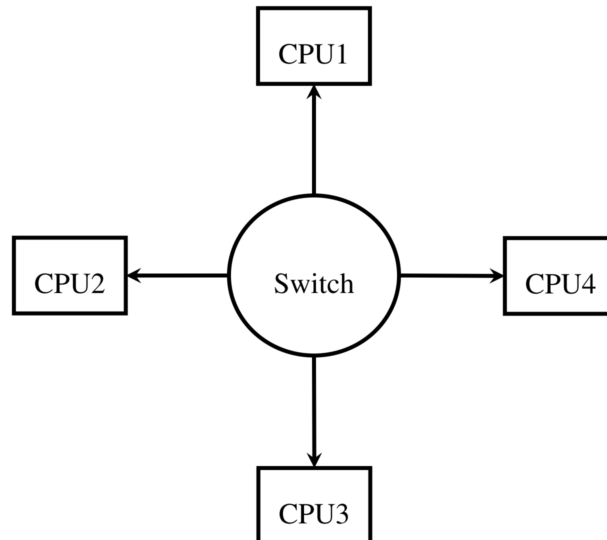

Figure 1.11: Orchestration Example

Summary With a growing reliance on the Internet for the collaboration of development teams, web-based tools are becoming a core component to modern product development processes. It is essential that these tools support a variety of development processes while also achieving high levels of parallelism and low latency communication. This work seeks to improve the understanding of performance in DWS in order to accelerate product development processes.

Chapter 2

Problem Statement

This work focuses on two parallel efforts related to EDGE development and academic research.

Development Task This work seeks to modernize the existing EDGE 1.0 DWS into an extensible REST framework in order to advance modern product development process.

Research Task This work seeks to improve our understanding of Distributed Web Systems by developing a model which accurately simulates the behavior of the EDGE DWS. This model is expected to provide feedback for the following metrics:

- **Throughput** (τ) - the measure of tasks completed in a specific time period, for a given DWS, discrete web service, or computational process,
- **Utilization** (ω) - the measure of throughput achieved relative to the ideal throughput,
- **Efficiency** (η) - the measure of throughput relative to the utilization of allocated resources,
- **Latency** (λ) - the measure of the amount of time necessary to respond to a request, and
- **Scalability** (ρ) - the measure of trade-off between latency and throughput.

2.1 Hypothesis

The research task of this thesis is motivated by a single major premise and six supporting minor premises. These motivating premises yield the research hypothesis for the proposed thesis.

Major Premise Current approaches to modelling a DWS fail to accurately predict the scalability, throughput, utilization, latency, and efficiency of realized systems.

> Administration of a DWS is typically handled in a reactive approach, or at best using a heuristic approach, for allocating resources. In reactive paradigms, a systems administrator will manually increase or decrease resources for different web services in response to observed shortcomings. This is typically performed by logging resource utilizations and responding according to a surplus or deficit of resources. Heuristic algorithms provide a degree of automation, permitting a resource to be dynamically adjusted in response to changing system load. Neither approach provides a predictive capability for DWS management.

Minor Premises

1. **Standard metrics for THROUGHPUT of DWS need to be developed**
   Throughput can be measured in a variety of manners for a DWS. Coarse measurements of requests per second fail to take into account the size of the request or its response. The measurement of the movement of data in and out of a DWS can resolve this, but hides the semantic significance of this data. A set of metrics is needed to provide relevant measures of throughput for different degrees of granularity and for different types of DWS components.

2. **Standard metrics for UTILIZATION of DWS need to be developed**
   It is necessary to identify the core aspects of a DWS for which utilization should be

assessed. By understanding how much time a resource is IDLE, decisions related to resource allocation and task assignment can be performed in a manner which reduces waste and increases availability.

3. **Standard metrics for SCALABILITY of DWS need to be developed**

   As throughput will be quantified for multiple levels of granularity, scalability must also be measured as multiple quantities. The scalability of a system will be affected by the assignment of resources, organization of program code, and the presence of processing bottlenecks.

4. **Standard metrics for EFFICIENCY of DWS need to be developed**

   Efficiency experiences a direct relationship with scalability, also requiring a number of metrics to be developed. These may include utilization of network bandwidth, CPU wait times, or queue exchange rates.

5. **Standard metrics for LATENCY of DWS need to be developed**

   Two major measures of latency are of primary interest to a DWS administrator. User-facing latency can be measured as the round-trip time required to satisfy a user's request with a server response. System latency can be measured as the time necessary for system tasks to execute from start to finish.

6. **It is necessary that DWS models be validated against empirical monitoring of production environments**

   Traditional modelling practices usually make assumptions about model components and their interactions. These assumptions may neglect the overhead of virtualization or OS functions, scheduling algorithms used to allocate tasks, or contention for non-compute resources such as network interfaces and links. Without such constraints, a model may be prevented from predicting the true behavior of a system as it is implemented. Validation of the model against an existing system may demonstrate the limitations of the theoretical model, allowing for further refinement and improved

prediction accuracies.

Hypothesis A static model of information flow and resource utilization can be used to predict the efficiency, throughput, and scalability of a static DWS, in order to guide the process of design, implementation, and deployment of a static or dynamic DWS.

2.2 Statement of Work

The objectives of this work are to:

1. **Develop a dataflow model for DWS.**

   A dataflow architecture shall be developed to model the transfer of information and processing delay of a DWS. Assumptions underlying the proposed dataflow architecture model are:

   (a) DWS elements have finite processing throughput,

   (b) DWS elements have finite communication throughput,

   (c) DWS elements have infinite energy scalability, and

   (d) DWS elements have infinite memory scalability.

   The model will be implemented in the Go programming language, which offers concurrency modelling features to simulate resource contention. It will be demonstrated that static models of DWS configurations may be used to predict 0th order performance for use in pairwise comparison.

   A static processing model will be developed to simulate the behavior of a chip-multiprocessor (CMP). Execution of processing tasks is defined by task-dependency graphs and performed by one or more processor cores. Execution time and utilization for a given task-dependency graph follows the expected performance given by Amdahl's Law and Task-Level Parallelism. The proposed model will be limited to a

pre-defined static schedule, while dynamic scheduling may be implemented in order to simulate OS scheduling algorithms (*i.e.* round-robin, preemption).

A communication model shall be developed to simulate the behavior of both a Network Interface Card (NIC) and a Level-2 switching device. The NIC will be responsible for delegation of access to a single network link for one or more concurrent processes in a given instance of the processing model. Each NIC will interface with the Level-2 switching device through a single network link. A bandwidth parameter will be used to approximate true transmission latency for data transfer across this link. The switching device will allow packets to be routed between two NIC instances. Routing will simulate link contention, allowing only a single packet to be transferred for each of the uplink and downlink directions. Message fragmentation will be performed for transfers larger than a single Maximum Transmission Unit (MTU), but will assume in-order arrival and zero packet loss.

2. **Simulate the EDGE DWS using the developed dataflow model.**

   The processing and communication models will be used to simulate one production EDGE system. Component models will be configured to reflect the properties and limitations of their physical counterparts. These components will be assembled to reflect the flow of information through a production EDGE system.

3. **Validate the dataflow model against observed performance data in three separate EDGE environments.**

   Validation of the dataflow model involves a three part approach. Part One involved collecting logged traffic data from production EDGE servers. This traffic was captured for one year duration and stored in the Apache HTTPd Combined Log format. Production access logs were captured for the following EDGE 1.0 deployments:

   - Multidisciplinary Senior Design (edge.rit.edu)
     - Purpose: collaboration on RIT Engineering student capstone projects

  - Primary Workload: web-facing design documentation and deliverables
  - Projects: 1068
  - Users: 3862
- Administration (InsideME.rit.edu)
  - Purpose: collaboration on KGCOE internal documentation and student records, by RIT Faculty and Staff
  - Primary Workload: secure document management
  - Projects: 182
  - Users: 232
- Research (kgcoe-research.rit.edu a.k.a meresearch.rit.edu)
  - Purpose: collaboration on research projects and student thesis/dissertation work
  - Primary Workload: version controlled research data and documentation
  - Projects: 307
  - Users: 418

Part Two will selectively use these logs to simulate exemplar workloads on the DWS models of an EDGE system. The logs will be translated into a set of requests which shall be used to recreate product development, administrative, and research workloads. Each type of request will be identified and decomposed into task-dependency graphs to be executed by the dataflow model of each EDGE 1.0 deployment. These workloads will then be simulated against two categories of performance metrics. First, the utilization of a component resource can be observed in the form of consumables such as CPUs, network links, RAM, disk, and energy. The second category of metrics quantify information flow in a DWS, addressing both the transaction rates and transaction sizes within a DWS.

In Part Three, the simulated workloads from Part Two will be validated against the throughput, utilization, scalability, and efficiency of an existing EDGE 1.0 server. Inaccuracies will be used to guide the refinement of model designs at both the component and architectural levels. The resulting model may be used to identify the bottlenecks, suggest alternative DWS topologies, and predict the performance impact of changes to existing environments.

2.3 Deliverables

The proposed research effort will result in several outcomes, including:

- The thesis document.
- A fully functional second generation EDGE DWS (EDGE 2.0), built on a foundation of open-source software, as a REST compliant implementation.
- A dataflow model and simulator which predict the information flow and resource utilization of a DWS.
- A conference paper was presented at the *ASME 2015 International Mechanical Engineering Congress and Exposition (IMECE)*, documenting historical usage patterns of EDGE by students in the Multi-disciplinary Senior Design (MSD) courses at RIT [32].
- A conference paper documenting the dataflow model, its use in simulating DWS, and its validation. The target conference for this paper is either *ACM STOC'17 or ACM/IEEE MODELS'17*.
- A journal paper documenting the utility, information flow, and resource utilization of the EDGE distributed web application system in academic and corporate settings. The target for this publication is one of the following journals: *Research in Engineering Design*, *Computer Aided Design*, *Journal of Computing and Information Science*

in Engineering, *Artificial Intelligence in Engineering*, *Design and Manufacturing*, or the *Journal of Mechanical Design*.

2.4 Work Schedule

Initial investigation into developing the EDGE 2.0 DWS began in the Spring of 2012 and was carried out during undergraduate study that led to the successful completion of a B.S. in Computer Engineering in May 2014. The results of that investigative period were put into practice with the development of the EDGE 2.0 DWS starting in August 2014 and continuing until the final go-live in January 2016. During this time period a paper was presented at the IMECE 2015 conference, outlining the fundamental requirements of the EDGE system for product development process, and its usage for the RIT Multidisciplinary Senior Design program (Fig. 2.2).

In January 2016, effort shifted toward development of a dataflow model for simulating Distributed Web Systems (Fig. 2.1). Preliminary work has shown the implementation of a rudimentary processing model for computational tasks which is able to accurately simulate throughput, utilization, latency, and scalability for a CMP-style microprocessor. A rudimentary model for network communication has been developed which is able to accurately simulate link contention, bandwidth sharing, packet routing, and transmission latency. Recent work focused on the integration of processing and network communication models. Validation of the integrated model against the EDGE 1.0 production logs will be presented.

2.5 Publication Schedule

Three publications will result from this work: a thesis document, conference publication, and journal publication. The conference paper was presented at IMECE in November 2015 [32]. Work on the thesis proposal was completed in February 2016 (Fig. 2.2). Work on the thesis document commenced in March 2016; a committee draft was completed in July and

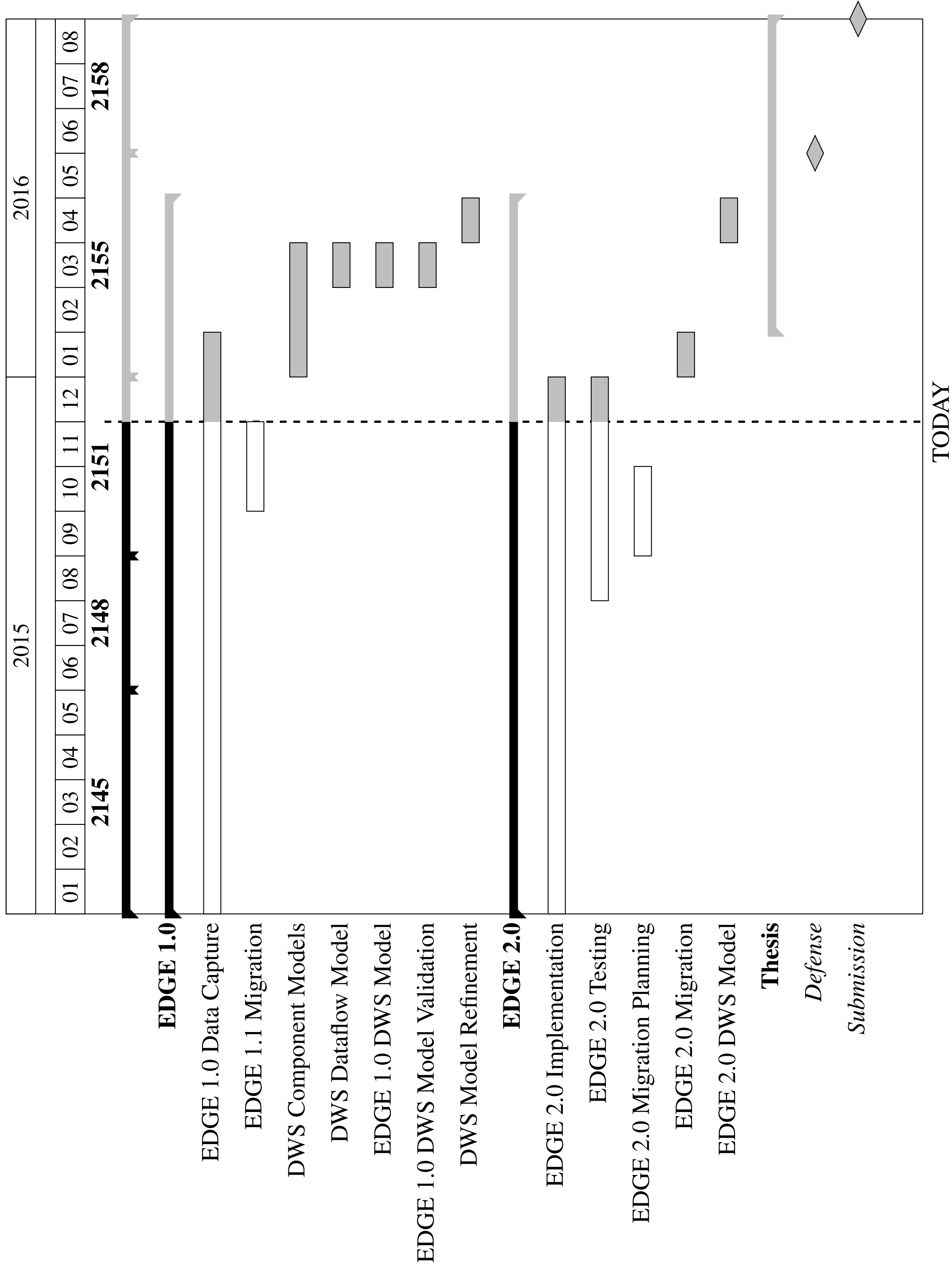

Figure 2.1: Work Schedule

submitted to ProQuest for publication in August 2016. Work on a journal publication will begin following the submission of the thesis document.

2.6 Required Resources

Production EDGE Systems Three EDGE 1.0 deployments have been in service for 5-10 years. The differing workloads from each server system may be used to validate the models developed. Access to Apache HTTPd server logs will be necessary to recreate the requests that occurred for the observation period. Capture of these logs required disk storage on the order of 100GB. It may be desirable to use a logging solution (*i.e.* Graylog, Splunk, Rsyslog) to facilitate the capture process. Further, access to the EDGE repositories will be necessary to recreate load conditions related to SVN activities.

EDGE 2.0 System Deployment of the EDGE 2.0 environment required new virtual machine resources to be allocated. This initially required a single VM with 2-4 CPU cores and 4-8 GB of RAM. However, this is expected to grow to 5-10 VMs for scalability testing.

Modelling Requirements Initial development of DWS models will rely upon the use of the Go programming language on a workstation with 4 cores and 16GB of RAM.

Data mining of the production logs to generate workloads, and execution of model simulations, may require greater resources than are available to the workstation computer. If this occurs, the Research Computing Cluster at RIT may be leveraged (at no additional cost) to meet higher computational demands.

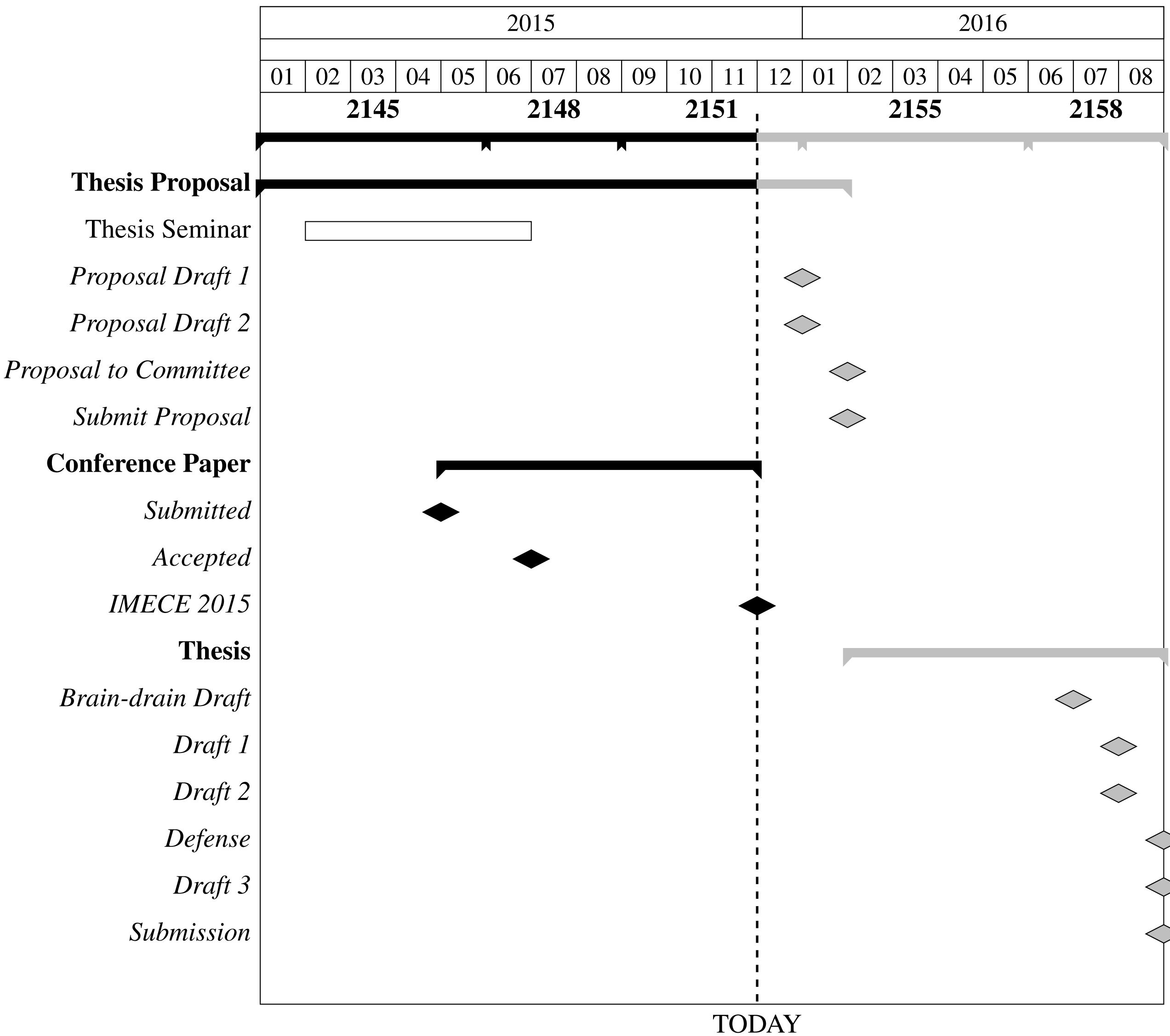

Figure 2.2: Publication Schedule

Chapter 3

REST Development of EDGE 2.0

The Engineering Design Guide and Environment (EDGE 1.0) was first developed over a decade ago by engineering student Brian Sipos. It combines a PHP web-frontend with a SQL relational database and Subversion repository system. This unique combination of software has enabled the execution of hundreds of capstone projects, provided a collaborative space for administrative procedures and curriculum development, and supported the work of over a hundred MS Thesis students. Having established such a track record, it has become increasingly important to plan for the future development of the EDGE system in order to ensure its continued growth. The following section provides a detailed discussion of the design of the EDGE system, the latest efforts in its development, and plans for future expansion of its capabilities.

3.1 EDGE 1.0 System

The primary features of the EDGE system fall into three distinct categories. Document management in EDGE provides a centralized location for the storage of product development artifacts, complete with a suite of change management and version control tools. EDGE supports project management through role-based project membership and the incorporation of project family trees. Web-based collaboration is made possible through online editing, MediaWiki support, and document rendering. These tools work in concert to support a wide variety of product development processes.

3.1.1 Document Management

It is well understood that product development efforts necessitate well-defined practices for the management of project documentation. The compilation of detailed design history can be invaluable in settling intellectual property disputes, analysis of project failure, and the assessment of future endeavours. EDGE seeks to provide a complete set of tools that may be used to properly support a product development effort through each stage of its life-cycle. At its core, EDGE facilities document storage, change management, and artifact version control.

Storage The main mechanism for information storage in the EDGE system is provided by Subversion. An SVN repository is able to handle any form of electronic information. Popular document formats include: MediaWiki, PowerPoint presentations, CAD drawings and 3D models, simulation results from MATLAB or Ansys, microcontroller source code, images of prototypes, and videos of product tests. All of this information is centrally stored for each project and may be accessed over the internet. SVN allows a snapshot of this repository to be stored on a developer's computer for offline editing. The repositories themselves can be stored on an EDGE server directly, or on a remote server. This enables flexible deployment of the EDGE system while also allowing the implementation of data backup and integrity checking to be utilized as necessary.

Version Control Subversion provides a suite of tools for handling multiple versions of the same document. First, each new version of a file is assigned a revision number. This number corresponds to every document that was submitted to the repository as part of a change set. A revision is also timestamped, with a name for the author, and an optional message indicating the work that was done. Second, Subversion can provide a historical log of revisions for every file or directory in a repository. This history can be used to examine the evolution of a file, to see who edited it last, or to find a specific revision for use in change management. Lastly, SVN also provides a set of `diff` or difference tools

which allow for comparison of different revisions of files.

Change Management EDGE employs a combination of project-based access control and Subversion commands to facilitate the change management process. Subversion provides functionality for reverting to older versions of artifacts and managing conflicting versions of documents. In the case of the lost-update problem previously discussed, SVN users must update to the newest revision before committing their changes. This forces a user to merge the changes that would have been otherwise lost. Only then are they able to update the central version of the document. Access control in EDGE requires a user to have membership to a project before being able to make changes to documents. Further, a specific role must be assigned to the user for each project before they are able to perform any modifications.

3.1.2 Project Management

In addition to the document management capabilities of EDGE, project management activities are currently supported by two main features. Roles are used to delegate permissions to specific project members. Project families can be used to relate separate projects to demonstrate the continuity of efforts, the flow of information, and roadmap style dependencies.

Roles Each role in the EDGE system delegates a unique set of permissions with respect to a project. Administration of a project is simplified through this lack of a permission hierarchy. Names for the roles were chosen to directly convey what permissions they enable.

- **Observer**

  Allows any user to become a member of a project, adding the project to a personal list of projects, but does not delegating any permissions.

- **Guest**

  Allows a user to see any non-public information in the SVN repository and permits them to export a local copy of the repository.

- **Editor**

  Allows a user to create or edit documents within the repository.

- **Curator**

  Allows a user to revert a document to an earlier revision.

- **Admin**

  Allows a user to modify project information, add or remove memberships, and to delegate roles to project members.

Project Families and Tracks Since it is common for projects to either share information or belong to the same development roadmap, EDGE facilitates these relationships through project families. Any project may be related to a parent project. This enables a parent project to possess multiple child projects. A child project may represent an evolution of the parent project or may be one of several sub-projects for a given development effort. Having these relationships between projects makes it possible to have a central set of documentation which applies to all of the child projects, while also providing a means of tracing the lineage of a particular development effort.

3.1.3 Web Collaboration

The EDGE system facilitates online collaboration through three main features. MediaWiki has been incorporated into EDGE as the defacto standard for web-facing documentation. An online editor has been written to support the editing of MediaWiki documents without a dedicated editor program. In addition to MediaWiki, EDGE supports the display of several multimedia formats inside of a web page, while also allowing for the downloading and uploading of new documentation through the website.

MediaWiki MediaWiki has a well-documented history of being used successfully for online collaboration, notably the WikiMedia Foundation. It uses a simple mark-up language

to perform document formatting and linking. This plain text format also simplifies the process of merging document changes. The EDGE version of MediaWiki has been augmented to allow for per-project namespaces, image rendering, and interwiki links. Interwiki linking is especially useful for referencing documentation in parent projects.

Online Editing In addition to support of the MediaWiki format, EDGE also features an online editor for MediaWiki documents. This allows user to make modifications to documents without a local checkout of the repository or an editing program. The editor also features a preview function that allows users to see what the modifications will look like, as they appear from the web. When a user is done editing, they may save their changes as part of a new commit, complete with a message indicating the work done. This new revision is also attributed to the author of the changes.

Document Rendering Lastly, EDGE supports the rendering of various document types to the web. MediaWiki files are translated into HTML pages which can be viewed with any web browser. Images and tables may be embedded into these pages to be presented online. HTML documents may also be viewed online. Any file may be referenced using a wiki link and then downloaded for view by the appropriate software package.

3.1.4 EDGE 1.0 Architecture

The EDGE 1.0 system consisted of a monolithic computing environment, hosting an Apache HTTPd 2.2 web server, a MySQL 5.3 database server, and a Subversion repository hosting environment (Fig. 3.1). PHP was used to coordinate the display, modification, and creation of documents within the Subversion repositories. MySQL was used to provide project and user metadata, permissions and access control, system configuration, and FACETS data storage. Mediawiki was chosen as the main document format and extended to provide media handling, namespace for projects, and inter-project links.

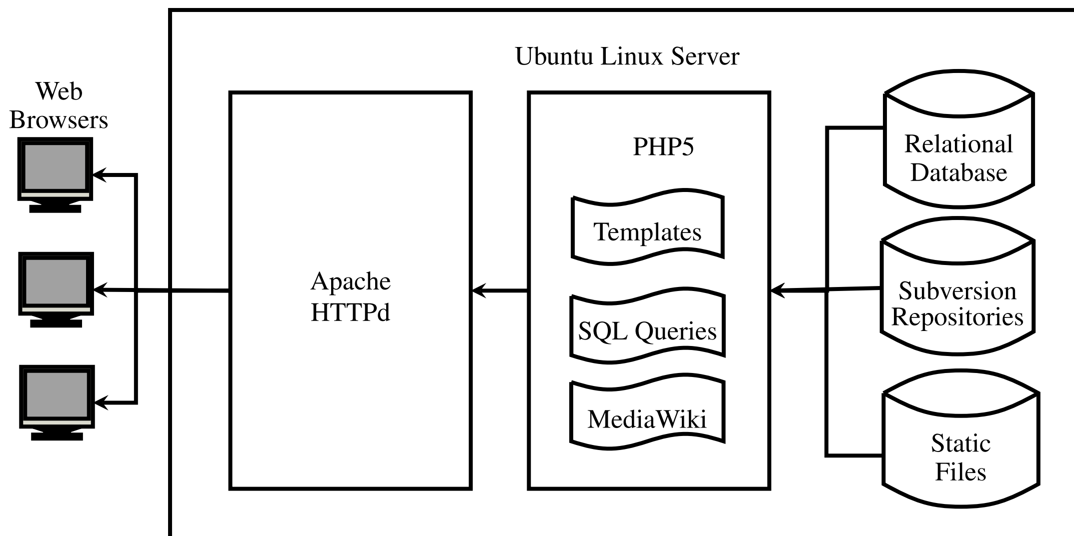

Figure 3.1: EDGE 1.0 Architecture

Pain Points for EDGE 1.0 While an EDGE system is not difficult to administer after deployment, there are several issues with the EDGE 1.0 application. First, installation of an EDGE server is performed manually and for a new system administrator can take upwards of 2-3 hours to complete. Second, resource requirements for an EDGE system are difficult to quantify. Normal EDGE operation does not require a large resource allocation. In the case of the MSD coursework, however, it has been noted that services can become sluggish or even unbearable due to user-facing latency. Third, attempts to integrate new FACETS tools into the EDGE environment has demonstrated a lack of flexibility within the existing architecture and slowed further development efforts. Lastly, open-source development of the PHP dependencies of EDGE 1.0 has either slowed down or stopped entirely. Support and patching of these libraries has seen a similar decline.

3.1.5 Product Development Toolkits — FACETS

FACETS is a collection of collaborative engineering design tools that has been developed to integrate with EDGE. Each of the FACETS Toolboxes is designed to supplement a different aspect of engineering design process (Table 3.1). FACETS Tools operate in concert by coordinating the flow of information inside the EDGE system. These tools do not enforce a

specific development process. Instead, these tools are intended to enrich and facilitate any process by supporting common aspects of design.

Table 3.1: FACETS Toolboxes

Early Design Tools	**Late Design Tools**
Needs Assessment	Engineering Models
Concept Development	DFx: Detailed Design
Feasibility Assessment	Production Planning
Engineering Analysis	Pilot Production
Tradeoff Analysis	Commercial Production
Design Synthesis	Product Stewardship

Each of the FACETS Tools utilize one or more SQL tables to store related information. Information stored in these databases is generated and displayed by interactive web applications built on top of the EDGE environment. In 2012, it was observed that even a small number of users (< 30) were able to tax an EDGE system consisting of a dual-core processor and 8GB of RAM. The concern over the performance requirements necessary to support the FACETS environment for larger numbers of users led to the need to re-evaluate the framework provided by EDGE.

Software Engineering Development Teams Development of the FACETS Tools has been carried out by four Software Engineering Senior Design projects at the Rochester Institute of Engineering. Each team performed development tasks over 22 week periods, taking place from Winter 2007 to Fall 2009. FACETS Team F1 was responsible for the initial development of support for FACETS in the EDGE environment during Winter 2007 and Sprint 2008. In addition, the team began development of the Brainball tool. FACETS Team F2 developed the Affinity Diagram, Brainstorm, and Objective Tree tools during the Winter of 2008 and Spring of 2009. FACETS Team F3 was responsible for the implementation of the House of Quality and Function Tree tools during the Winter of 2009 and Spring of 2010. FACETS Team F4 modified the existing FACETS tools to allow for the development of Android versions of the Affinity Diagram, Brainstorm, and House of Quality tools during the Summer and Fall of 2009.

3.2 Design Goals for EDGE 2.0

The EDGE system has already proven to be a successful development effort. Like any piece of software, however, there is always a need for improvement and the implementation of new functionality. In 2012, a new development effort began with the goal of designing and implementing a next-generation EDGE system. This new system would be able to support larger numbers of users, be built on modern web technologies, improve the installation and maintenance practices, and also allow for easier development of new tools to aid in product development process. The following sections discuss the goals for the EDGE 2.0 system, with respect to user feedback and evaluation of the existing code base.

3.2.1 Modern Web Technologies

When EDGE 1.0 was first developed, the predominant web technologies used for interactive web environments were Java, PHP, and SQL Relational Databases. PHP was a logical choice for generating dynamic web pages. PHPs native support for HTML templating, string manipulation, and handling form submission data provided a strong foundation for EDGE. When combined with several useful open-source libraries for PHP, it became trivial to communicate with a SQL database, Subversion repositories, and the MediaWiki language. MySQL was chosen as for the relational database due to a wealth of developer documentation, compatibility with PHP, lack of commercial licensing, and supportability on non-Unix platforms. WebDAV access to SVN repositories brought web-based access to revision histories, version differences, and file tree browsing.

In the years following the release of EDGE 1.0, much of the software development community began to focus on a web-based user experience. This led to the creation of a multitude of new tools for the creation of rich internet applications. Java gained many new libraries to move beyond Applets to full-blown server backends. Fledgling languages like Ruby and Javascript followed a similar trend as many developers sought out alternatives to

an aging PHP community, becoming the preferred choice for this new generation of web-based software. Markdown and other new mark-up languages become viable alternatives to the long-established MediaWiki. Alternatives to SQL databases were developed to handle alternative data formats, such as XML or JSON. HTML and CSS underwent sizeable upgrades while becoming HTML5 and CSS3, respectively. Even Subversion began to see a large percentage of their user base moving to the newly developed Git VCS.

The design of EDGE 2.0 required significant research to evaluate a multitude of new tools and technologies. This work began in the Spring of 2012 with the investigation of the open-source project management suite known as Redmine. Analysis of the Redmine source code revealed the reliance on a complete Ruby on Rails stack, using Ruby ERB as a web templating language, ActiveRecord for database manipulation, and jQuery for user-interaction with the web interface.

3.2.2 Extensibility

The EDGE 1.0 system has reached the maintenance phase of its development. Nearly all of the development efforts have been focused on maintaining its functionality with newer versions of server software, PHP, and its dependent libraries. Because of the majority of time being spent on maintaining a functional EDGE 1.0 environment, it has not been feasible to continue introducing new features. Further, EDGE 1.0 relies on a highly monolithic software environment. It is expected that the MySQL database, SVN repositories, and Apache HTTPd web server all exist on the same physical machine. This prohibits the modular deployment of EDGE services in a distributed environment. Services cannot be deployed to optimized hardware platforms which efficiently handle database or web-serving workloads.

EDGE 2.0 development seeks to tip the balance back towards lower maintenance and consistent introduction of both new and improved functionalities. Extensibility of the EDGE 2.0 system will be built upon the REST services model for software, in which each component provides a unique service and may be deployed anywhere in the distributed environment. Services can then be composed to perform complex workloads and may be

iteratively replaced with improved versions by maintaining a consistent API. This modularity has the added benefit of allowing services to be allocated to optimized server hardware or for tightly coupled services to exist on the same server. Reliability is gained as a side-effect of this distribution as well.

3.2.3 Improved Performance

With EDGE 1.0, performance is bottlenecked by both software and hardware. From the hardware side, disk access serves as the single largest limiting factor. Both the database and SVN repositories rely on frequent access to storage to serve user requests. While this could be solved by purchasing a faster storage backend (*e.g.* SSD), this goes against the philosophy of allowing EDGE to be flexibly deployed by any organization. Dealing with the software bottlenecks may serve to remedy the issue. For example, in EDGE 1.0 every read operation to a repository requires a local checkout by made. This checkout may persist on for a duration of time, but too many persistent checkouts will lead to high demand for local disk space. In addition, access to an EDGE repository is either largely random (web traffic) or throughput oriented (checkout). Instead of keeping local checkouts, EDGE might use the `svn cat` command to only access a single file at a time from a repository. These files may be cached to further improve access times. Instead of using the stateless WebDAV protocol for SVN client access to EDGE repositories, the stateful `svnserve` utility may be used to improve throughput for users.

3.2.4 Easier Deployment

Any web application requires a fair amount of initial setup before it can be used. Installation of an OS, software packages, and additional software libraries serve provides the foundation of the application. The application code itself is then unpacked onto the prepared system. A web server, database, and additional datastores will then need to be configured for use. Finally, the application itself may need to be configured for use. Performed manually, this process make take hours for an experienced administrator, or days for a first time

deployment.

Many tools exist to simplify this process. Scripting languages can be used to guide the installation process, possibly from start to finish. However, it may be difficult to script the installation for many different systems. A Linux package might be created for a subset of distributions, but there will likely not be a “one size fits all” approach to these packages. This process becomes further complicated when trying to support Windows servers. Design of the EDGE 2.0 system should make every effort to simplify the installation process to a few hours or even minutes.

3.3 Evaluation of Software Tools

With significant software development effort in industry being targeted at web-based applications, new tools and even whole languages have have been created to aid developers. Work on the design of EDGE 2.0 began with the evaluation of alternatives to the software and libraries in use for the EDGE 1.0 system. The following core comparison factors were instituted to limit the scope of this search:

- **Open-Source**

  Closed source tools suffer from multiple problems, including: license terms, available support, and visibility for debugging. Open-source tools are available at typically no additional cost, provide direct access to developers for support, and allow access to the source for debugging. Additionally, it also becomes possible to submit bug fixes and functionality upgrades for inclusion in later versions of the tools.

- **Well-Documented**

  Documentation is critical for the implementation of software around existing tools. Having a well-documented tool reduces the time to integration, demonstrates the quality of the software, and allows end-users to access additional support for the tools which they directly interface with.

- **Active Developer Community**

  An active developer community is critical to the future of a software tool. It ensures that bugs and security holes can be fixed, that new functionality will be developed, and that existing functionality can be improved. Lack of such a community reduces supportability of software during maintenance periods and may lead to a tool being replaced completely.

- **Developer Adoption**

  It is one thing for developers to be excited about the potential of the latest and greatest software tool, it is another thing entirely for them to adopt that tool into their applications. Strong developer adoption usually indicates that a tool works as advertised and that it was the solution to an important development problem. The adoption of an open-source tool also means a larger number of people to find and fix problems inside the tool itself.

- **Learning Curve**

  Developers should be able to quickly integrate a software tool into their application. This is only possible if the tool does not require a significant investment by the developer to not only learn how it works, but also how to leverage its functionality in practice. While some tools may require more effort than others, it should be recognized that requiring additional knowledge will mean either requiring a developer to already have experience with the tool or to invest a developer's time in training.

These factors were selected based on their importance to future development efforts for EDGE. The selected components must be able to withstand the test of time while also not requiring extensive knowledge for a developer to begin using them for new implementation efforts. This will ensure that EDGE is able to inherit these qualities.

3.3.1 Languages

Computer science has a long history of developing new languages in response to the growing needs of developers. In order to provide richer user interaction with early websites, Common Gateway Interface (CGI) scripts were used to process form information or to dynamically generate web pages. PHP naturally arose out of the CGI era, allowing for greater programmability for this type of dynamic page generation. Quickly becoming a developer favorite, PHP represents one of the first true "web" languages as it was designed from the ground up with the HTML developer in mind. Java appeared a year later, in 1996, as a language designed with a singular goal of "Write Once, Run Anywhere". Web developers saw the portability of Java programs and it quickly was adopted into web browsers in the form of "Applets". A Java Applet can be written to provide application-like functionality in the web browser. This would ultimately lead to Java Servlets which provided PHP like functionality for server-side software. Meanwhile, the Ruby language was under development in Japan as an alternative to the other object-oriented languages of the time. It was not until 2005, that Ruby gained attention in the Web community with Ruby on Rails. The Rails framework provided Ruby developers with a wide range of tools for performing the same server-side interactions as PHP and Java.

With EDGE, the choice of language was the first major topic of discussion. Now 2012, the landscape of web development had changed greatly and languages like PHP, Java, and Ruby had established thriving communities. Experiences with keeping EDGE 1.0 functional under PHP had led to questions of its usefulness in future development projects. In addition to the developer-centric factors previously mentioned, the following secondary comparison factors were adopted for evaluation of web languages.

- **Backward Compatibility**

  Programming languages are often developed with version numbers, as they are also supported by software tools (*i.e.* compilers, interpreters, debuggers, IDEs). A new

version of the language typically means improved performance, new language features, and a more comprehensive set of core libraries. However, existing source code should not suffer from compatibility issues with new versions of a language. Compilation or interpretation of the existing code should not end in failure or break existing functionality.

- **Available Libraries**

  The existence of software libraries for a language has significant ramifications on the development of new applications. A library may provide useful functionality and access to other software tools. This allows a developer to expend less effort on implementing the core functionality of a program and turn their focus to providing richer feature sets, a pleasant user experience, or greater performance and reliability.

- **Support for HTML5, CSS3, and Unicode**

  A web programming language must grow to support newer versions of core internet technologies. HTML5, CSS3, and Unicode are critical technologies to building web applications which support a wide range of multimedia, are visually appealing, and are able to be used by an international community.

PHP At first, PHP sounded like the logical choice. The existing source for EDGE 1.0 was written for PHP 5.1, and had been successfully upgraded to function with PHP 5.5 and the newer versions of libraries from the PHP Pear package repository. However, by March 2010 the PHP project had abandoned efforts to bring unicode support to PHP in version 6.0 [28]. Many developers saw this as a sign that PHP was on the decline and chose to migrate to other, more stable, languages. As a result, many of the libraries used in EDGE 1.0 saw little to no further development and ceased to function with later versions of PHP. It was not until December 2015 that the next major release of PHP (7.0) was released with support for Unicode and HTML5 [29].

Ruby Between 2006 and 2012, the Ruby language was in use by 2% of the developers surveyed by the TIOBE index [3]. This consistency serves as an indicator of the stability of the language and its community. As a language, Ruby has full support for Unicode, has advanced capabilities for handling strings, and is easily extended through dynamic programming. Domain Specific Languages (DSL) in Ruby allow developers to create reduced syntax languages which can be targeted at simplifying complicated tasks and reducing boiler-plate code. Ruby has extensive core libraries which bring support for serialized data types (*i.e.* JSON, XML, YAML), IETF communications protocols (*i.e.* HTTP, FTP, RSS), and even a built-in web server called WebBrick. The Ruby packaging format, Gem, provides developers with simple mechanisms to share specific versions of their libraries through Gem repositories (*i.e.* RubyGems.org). The Bundler tool greatly simplifies the process of requesting dependencies for a development effort by retrieving Gems and installing them. These Gems bring libraries that aid in building REST APIs (*i.e.* Sinatra), templating web pages (*i.e.* ERB, HAML, Liquid), and working with SQL relational databases (*i.e.* ActiveRecord, DataMapper, Sequel).

Java Java Applets have been used to augment the user-experience on web-pages for over a decade. However, Java has significantly more to offer. Java Server Pages (JSP) provide support for HTML templating. Server-side handling of web applications can be performed through Java Servlets. The Java Database Connectivity (JDBC) libraries provide a uniform interface to SQL relational databases. The Apache Tomcat Web Server allows a developer to tie all of these technologies together to create dynamic websites. Tomcat's Web Application Archive (WAR) file format allows an entire application to be bundled together into a single, portable container for deployment. Unicode support has existed in Java since version 1.1 [1] and was brought into Unicode 6.0 compliance with Java 1.7 in 2011 [2]. Java also has extensive support for internationalization and locale handling.

Decision Ultimately, Ruby was selected for the implementation of EDGE 2.0. With the future of PHP remaining uncertain, its development community had begun to atrophy and this could best be seen in compatibility issues with existing PHP libraries. Java has potential as a general purpose language, but suffers from very verbose application code and a lack of libraries for web-centric technologies. Development using Java would mean being forced to use the technologies provided by Oracle and following their "one-size-fits-all" mentality. Ruby, on the other hand, has continued to show a large growth in the number of tools available and even provides alternate implementations of the same technologies. Coupled with a strong developer community, wealth of developer documentation, and a terse programming grammar, Ruby represents the greatest promise and the least amount of risk.

3.3.2 Frameworks

With the decision to use the Ruby language for EDGE development, investigation shifted to the available frameworks for building web applications. A framework, in this context, is either a collection of tools that are used in concert to create an application or an existing application which may be built upon. Redmine was considered for its existing capabilities as a web-based project management tool. It provides a wiki, file storage, access to VCS repositories, and a rich task management system. Ruby on Rails serves as the framework for Redmine. By sacrificing the existing functionality of Redmine, Rails could allow for increased flexibility in the implementation of EDGE, already being built upon a robust set of Ruby libraries. Sinatra can be thought of as the "anti-Rails" by comparison. It is a Ruby Domain Specific Language (DSL) for building REST APIs and has strong support for HTML templating through the Tilt Gem.

For the framework selection process, the following new comparison factors were introduced:

- **Modular Implementation**

Modularity is a key component of reliable and bug-free software. It allows the functionality of a program to be broken up into separate systems, which may be implemented independently. A developer can then narrow their scope to a subset of features and spend significantly less time searching for the source of a bug. Modular software can be deployed flexibly according to the needs of an organization. Distributed Web Systems require modularity in order to promote scalability.

- **Minimal Boilerplate Code**

  A framework should augment the functionality of a program without dictating its implementation. It should not be the case that a framework requires a significant amount of code to access its functionality. Nor should a framework be so complete that a developer has little to no say in how their application is written.

- **HTML Templating Support**

  Templating support should be inherent in a web framework. This reduces the burden on the developer to incorporate a particular templating language into their application, increases productivity, and shortens time to market.

Redmine Redmine provides a solid core for software development efforts. It streamlines access to VCS repositories, includes a task tracking system, and facilitates workflow design for role-based design processes. Several plugins have been written to supplement Redmine's functionality. For example, the AgileDwarf plugin simplifies task management for an individual by creating a "dashboard" where they can see assigned tasks, move tasks to different stages of the workflow, and record the time spent on a task. Through this extensibility, it would be possible to implement EDGE as a set of plugins for the Redmine environment, eliminating the need to develop the infrastructure for EDGE, while focusing on user-facing functionality.

Ruby on Rails Rails was investigated due to being the framework used by the Redmine project. An application built on Rails consists of three sets of components: the Model, View, and Controller. The Model is a representation of application data described through a Ruby Object. This object itself may represent entries in a CSV file, individual files located in a folder, entries in a SQL database, or even data from an external REST service. The View consists of one or more ERB templates for displaying information found in the Model. These templates are not only used to generate HTML, but also serialized data structures like XML and JSON. The Controller ties the Model and View together, implementing the interface to Rails, while performing application logic like Authorization, handling of URL query parameters, or parsing the body of a POST request and applying changes to the Model.

Sinatra Sinatra is a Ruby DSL for implementing REST services. Unlike Rails, Sinatra does not impose any restrictions on the developer. It processes a request received by a Rack compatible web-server, parsing any query parameters and routing the request to a registered REST endpoint. Implementation of an endpoint consists of mapping an HTTP Method and URI to a specific block of code. The developer can then implement any code necessary to satisfy that request. Sinatra also provides helpful utility functions for generating error pages, rendering templates to HTML, and performing user authorization.

Decision Redmine brings a lot of functionality to the table, but suffers from a lack of internal developer documentation. This makes it fairly difficult to use the Redmine API and to incorporate new functionality into the system. Further, being built on Ruby on Rails brought along another set of difficulties. Rails forces a developer to pick from a limited subset of the available Ruby libraries when writing software. It also strongly encourages developers to write in the MVC software pattern, introducing a large percentage of boilerplate code. It was decided to develop EDGE on top of the Sinatra DSL. Building APIs and rendering content is made trivial through the built in functionality of Sinatra. Additionally,

the developers of Sinatra have made every attempt to not limit a developer's set of choices for supporting libraries.

3.3.3 Version Control System

Document storage in EDGE first relied on the Concurrent Versions System (CVS). With its popularity declining and its last version released in 2008, CVS was ultimately replaced by Subversion. This led to improved performance, a greater number of GUI clients, and continued support from the SVN developers. Around the same time, the Git Source Control Management toolchain started to gain popularity after its adoption by the Linux Kernel Project. Git's support for distributed repositories, local commits, and strong merging capabilities have made it the defacto standard for open-source software development.

The following additional comparison factors were introduced for the evaluation of VCS systems:

- **Multiple OS Support for Clients**

  Version controlled documents may need to be edited on a local workstation and possibly requiring the use of specialized software. Additionally, a user will likely have a preferred operating system. In order for a VCS to be successful for a variety of workflows, it will be necessary that client software be available for the most common operating systems (*i.e.* Windows, Mac OSX, Linux).

- **Multiple Format Support**

  A VCS should not impose a subset of document formats on a user. This only serves to prevent developers from using their preferred tools and may lead to additional software costs. Support for a wide variety of document types is necessary to avoid imposing restrictions of users.

- **Centralized Repositories**

  VCS software should always allow for a centralized repository. This creates a single location for developers to access authoratative versions of documents, while also

transferring the risk of data loss from the end-user to the service provider. Secure access to the documents is easily accomplished with a centralized system, while not being feasible to enforce when delegated to the end-user.

CVS The Concurrent Versions System (CVS) was used in the early versions of EDGE and is considered for that reason. CVS features a centralized repository with access control, and can store most data formats. Following a commit, CVS does not store the differences between versions, it stores the entire file. No metadata can be stored for a given file (*i.e.* MIME-Type, locks, owners). Active development of CVS ceased in 2008 and no official plans for newer releases exist.

Subversion Subversion (SVN) was incorporated into recent releases of EDGE as a replacement for CVS. SVN provides several important features missing in CVS. Metadata storage allows a user or application to store any additional information for a file that might be desired. In this way, SVN is also easily extended by other software. EDGE utilizes the metadata capability to store MIME-Types for files. Locks allow a user to prevent alteration of a file by other users. This is useful for preventing merge conflicts, and also for preventing unauthorized modification of files or directories in a repository. A commit only stores the differences between versions of a file. This has the benefit of reducing the storage size of a repository, at the expense of increased computational time when committing or retrieving a file. Local checkouts of a version of the repository contain all of the repository metadata, but only the most recent revision of the files.

Git Git was developed for the Linux Kernel Project as a replacement for the commercial BitKeeper SCM. Unlike SVN and CVS, Git is a distributed version control system in which every checkout contains a full history of all changes prior to the checkout. This allows any copy of the repository to act as a both a source and a backup of the repository contents. Further, users may continue to commit changes locally without sending those changes to

a centralized repository. This allows a user to continue working completely offline and while still being able to access earlier versions of a document. Once online, a user may "push" these changes to one or more repositories. In the case of strict change management, it may be necessary for an authorized user to "pull" these changes into one or more authoritative repositories. In either case, all other users may then update their local copy of the repository.

Decision With CVS development ground to a halt, this only leaves Subversion and Git as contenders for the underlying VCS of EDGE. The decision was made to continue development around SVN, while providing the extensibility for Git integration in the future. Subversion allows development efforts to preserve how EDGE interacts with version control. It provides continued support for existing repositories, without requiring migration to a new repository format. Git may be introduced in the future to provide greater flexibility in the product development process. Widespread adoption of Git for open-source software projects demonstrates the presence of developer needs not currently addressed by Subversion.

3.3.4 Persistent Relational Datastore

Persistent information in web applications has, until recently, mainly relied upon relational database models. This allows information to be broken up categorically into tables. Table entries can then be related to one another through unique identifiers. Structured Query Languages (SQL) can then we used to access and filter the information for use. Newer data formats like XML and JSON have led to the creation of many document-based persistence tools. In these systems, an entry is assigned a unique identifier which may be used to retrieve or modify its contents. Documents may also contain the identifiers of other documents, which can be used to create relationships between documents.

The following comparison factors were used to evaluate the different persistent datastores:

- **Query and Filter Support**

  With a large dataset, it is important to be able to limit the number of results that must be processed by the web application. Query and Filter mechanisms allow a developer to programmatically reduce the number of entries in a datastore that are considered, by reducing the number of entries which are returned by a request and by reducing the number of necessary requests.

- **Transactional Safety**

  Modifications to the information in a datastore should not affect simultaneous requests for information. Each request should be handled based on the currently available information and neglecting any information that was entered during the request. Further, transactional safety prevents changes from being finalized until all actions are completed successfully. A single request may involve modifying several different entries in the datastore. A transaction allows all changes to be reverted if any one change fails. This also prevents ongoing requests from accessing incomplete or incorrect information.

- **Available Ruby Libraries**

  Library support of a datastore eliminates the need for a developer to implement the code necessary to interact with it. This not only reduces development time, but allows domain experts to correctly implement a robust, reliable, and performant interface to an underlying datastore.

MySQL MySQL was first released in 1995. One of the first open-source relational database management systems (RDBMS), MySQL stores information according to a tabular data format. These tables are defined my a schema which declares the name, data-type, and properties of each column. Entries in these tables are manipulated using a Structured Query Language (SQL) dialect. A query may be used to create, read, modify, or delete rows in each table. In 2005, MySQL gained transactional safety with the release of version 5.0.

Unicode support introduced to MySQL in 2010. Numerous libraries exist for MySQL in Ruby and several of them use native C libraries for improved performance.

Postgres PostgreSQL, or Postgres, was first released in 1996. It supports all of the same features as MySQL, including transactions and unicode support. Where MySQL traditionally focused on performance, Postgres emphasizes enterprise stability and reliability. Postgres features full compliance with the SQL 2011 standard and has been shown to be fully ACID compliant for data integrity. Strong security and access control are built into the Postgres engine. Postgres authentication goes beyond local usernames and passwords, being able to integrate with external authentication sources (*i.e.* LDAP, Kerberos, RADIUS). Authorization is supported through role and group-based permission sets. These can be further customized on a per-database basis.

CouchDB CouchDB was released in 2005 and is one of many so-called "NoSQL" database engines. Unlike SQL databases, CouchDB diverges from a tabular data format and instead stores information in a document format. CouchDB utilizes JSON to represent information as nested sets of key-value pairs. Relationships are implemented through numerical identifiers or URI references to other documents. While many libraries exist for accessing CouchDB, it also has a REST compliant web API which can be accessed with an HTTP client. Access control is implemented similarly to a VCS, where authorship and group membership are used to limit permissions.

BaseX BaseX was released in 2007. Like CouchDB, BaseX is a document-based datastore. However, BaseX relies on the XML data format for document storage. These documents may be validated against an XML schema definition (*i.e.* DTD, XML Schema, RelaxNG), much like a SQL schema. Querying of documents is possible through the XPath and XQuery languages. BaseX has a native REST interface with HTTP and WebDAV

support. Administrative actions and document manipulations are performed over this interface. The modular design of BaseX allows for extension of existing functionality and is used to implement extended functionality. It is possible to configure BaseX to operate as a stand-alone web application, complete with HTML rendering functionality.

Decision The ability to quickly query a dataset has been leveraged heavily in the development of EDGE and the FACETS tools. This requirement eliminates CouchDB as an option. BaseX relies heavily on XML related technologies. This is fine for development that relies on XML, but means a developer will need to learn to define schemas and queries for XML. Postgres and MySQL share many features. The similarities between their SQL dialects makes it feasible to use them interchangeably and to plan for an eventual migration from one to the other. Postgres has promising enterprise features, but significant investigation is needed to leverage them in practice. For EDGE it was decided to use MySQL as the relational datastore. This allows design efforts to borrow aspects of the EDGE 1.0 schema design, and simplifies the migration from the EDGE 1.0 schema to a new EDGE schema.

3.3.5 Templating

Templating is a key component of dynamic websites. It allows a developer to define a consistent look and feel while allowing for a modular approach to building web pages. A templating language allows templates to go beyond "find and replace" functionality by allowing code execution inside the template itself. XSLT was originally developed to combine XML documents and to translate between XML Schema. The advent of XHTML made it possible for an XSLT to convert an XML document into an XHTML web page. ERB has a long history of being used in Rails projects to generate HTML pages. An ERB template combines static HTML elements with Ruby code to allow for conditional and loop-based generation of page content. HAML provides a reduced syntax for HTML which greatly reduces the number of characters needed to describe the elements of a document. Further, it also allows Ruby code execution to provide ERB like functionality.

The following additional criteria were used in the evaluation of templating languages:

- **Readability**

  It should be readily understood what HTML will be generated by a template. This reduces the likelihood of generating incomplete or incorrect web pages and the amount of time necessary to make modifications to existing templates.

- **Programmability**

  A template should act as more than a stencil for the layout of a web page. It must allow the developer to dynamically change what HTML is generated based on state information from a request and the specific content being rendered. This cannot be achieved without support for conditional rendering, template parameters, and iterative generation.

- **Brevity**

  Templating languages should allow for clear and concise declaration of HTML structures. Having a brief syntax which requires minimal code to generate a useful web page, allows a developer to quickly implement new templates and reduces the amount of time spent fixing rendering errors.

XSLT Paired with an XML database or XML compatible API, eXtensible Stylesheet Language Transformations (XSLT) offer a way of converting XML to other formats. An XSLT is written entirely in XML, allowing a developer to use the same language to define both a data storage format and its web representation. This is also made possible by the XML grammar for HTML, XHTML. XSLT supports conditional statements and uses a functional approach to handle arrays of data. Rendering is accomplished by applying an XSLT to XML documents. However, XSLT documents are fairly verbose compared to templates from other languages. This is largely due to the repetition present in XML tags, but is also encouraged by the functional programming paradigm.

ERB Embedded Ruby (ERB) is a templating language which allows developers to place Ruby code into an HTML document. Developers start by designing a web page using the usual HTML and CSS practices. Information is supplied to the template as variables which may be accessed and manipulated through Ruby code. Conditional and iterative rendering are supported by the same mechanism. In this way, ERB effectively extends HTML through the use of Ruby code to guide mark-up generation.

HAML The HTML Abstraction Mark-up Language (HAML) relies heavily on the “Don’t Repeat Yourself” (DRY) principle. HAML leverages a nested tree format to eliminate the need to express both starting and ending tags. These trees consist of the HAML domain specific language, which provides keywords that eliminate the verbosity necessary to describe HTML. Ruby code may be evaluated throughout the DSL to populate attribute values, provide content, and for performing conditional and iterative generation. Of the three proposed templating languages, HAML is the most terse, at the expense of not being easily read by new developers.

Decision More than a third of the existing EDGE codebase consists of HTML templates. This meant that EDGE 2.0 would require a similar amount of effort for templating and that effort should be made to improve templating speed and efficiency. ERB provides no additional benefit over raw HTML, with the exception of limited programming inside the template. XSLT suffers from the same problem, while also introducing the need to learn an entirely new programming syntax. HAML was chosen for two main reasons. First, programmability in HAML is achieved using the Ruby language, requiring no additional developer knowledge. Second, HAML significantly reduces the amount of code necessary to represent HTML in a template. These two factors serve to reduce the amount of time spent developing the structure and generation of the web pages, allowing front-end development efforts to focus on the usability, accessibility, and aesthetics of EDGE.

3.3.6 Mark-up Language

Support of a mark-up language allows EDGE users to develop web content quickly. Generation of a web-page from a mark-up language eliminates a certain degree of human error, while also providing for a consistent look and feel for the EDGE environment. Further, a mark-up language allows for comparison of documents in plaintext and without additional tools. This greatly reduces the effort of merging changes from intermediate commits.

The following additional comparison factors were used to evaluate mark-up languages:

- **Readability**

  Much like the developer requirements for template readability, a mark-up language should convey how content will be translated into HTML as intuitively as possible. This allows a document author to spend less time focusing on formatting and rendering, and more time developing quality content.

- **Extensibility**

  A mark-up language should allow for the introduction of syntax specific to where it is being used. This allows developers to incorporate new functionality into the language to assist authors in creating rich content.

- **Multimedia Support**

  The Internet relies heavily on multimedia to convey information to its consumers. A mark-up language which is used to generate HTML should provide substantial support for the inclusion of multimedia into the final document. At a minimum this should include audio, images, and streaming video.

- **Available Editors**

  Authors should be able to edit a mark-up document both offline and online. Web-based editors with integrated preview capability are necessary to support editing of a document from a web-browser. Most mark-up languages can be edited by any modern text editor. However, language specific editors allow previewing of the resulting

document, syntax highlighting and validation, and tools for generating specific document structures (*i.e.* tables and figures).

MediaWiki EDGE 1.0 utilizes MediaWiki as a mark-up language for its wide feature set, extensibility, and overall readability. It remains one of the most used mark-up languages in the world, and its functionality has continued to grow in order to support newer multimedia formats. Unfortunately, much of this new functionality relies heavily on the MediaWiki engine and not the HTML renderer. This greatly increases the difficult of extending the functionality of the language for future projects. The main online editor for MediaWiki, VisualEditor, has remained largely unchanged and provides no help with mark-up keywords and directives. There have been a few attempts to develop offline editors for MediaWiki, but none of these have gained notable popularity.

HTML While verbose, HTML is the the most flexible of the mark-up languages. An author has full control over the structure and appearance of the resulting HTML. They also have direct access to the extended multimedia capabilities of HTML5 and animation functionality of CSS3. HTML has the added benefit of many online and offline editors which provide full support for the language, previews, and even WYSIWYG editing. In practice, however, HTML can require a significant amount of knowledge on the part of the author and even experienced authors see the advantage of a simpler mark-up language like MediaWiki.

Markdown The Markdown language was first introduced in 2004 as an alternative to pre-existing solutions. Its syntax emphasizes strong readability in its textual form. A Markdown document should be equally readable in plaintext and HTML forms. Markdown supports common HTML structures like headers, lists, and links. However, it lacks a standardized syntax for tables and multimedia. This lead the Github project to adapt the syntax into so-called “Github Flavored Markdown” shortly after adopting Markdown as

their preferred document format. Despite these inadequacies, there are dozens of online and offline editors available for Markdown, and in most cases an author can be productive with a simple text editor.

Decision Of all the design decisions, mark-up language support is one of the most important to the users of the EDGE system. It is not necessary to eliminate rendering of a markup language so long as a renderer is available for the Ruby language. To this end, this decision is actually a declaration of which language will be modified to meet the needs of the EDGE system. For example, in EDGE 1.0 augmentation of MediaWiki was necessary to allow for inter-project linking and better support for the embedding of multimedia files. MediaWiki was ultimately selected as this language for EDGE 2.0. This preserves compatibility with documents written with the previous “EDGE Flavored MediaWiki”, while allowing for the continued evolution of the markup syntax. HTML and Markdown support will also exist in EDGE to provide flexibility to document authors, while sacrificing the readability and/or evolved features of the extended MediaWiki grammar.

3.3.7 Summary

EDGE will be developed using the Ruby programming language. REST functionality will be built on top of the Sinatra DSL. Document management will continue to be provided by Subversion, with the possibility for support of Git in the future. Relational data will continue to be stored in a MySQL database, utilizing an improved schema for the contained tables. HAML will be used for templating each type of page presented to an EDGE user. Rendering support for HTML, Markdown, and MediaWiki will be implemented. However, MediaWiki will be augmented to provide extended functionality that is specific to the EDGE system.

3.4 Implementation

Development of the EDGE 2.0 system began in the the Summer of 2014. EDGE 2.0 was launched in January 2016. The functionality of EDGE was divided across multiple new REST services (Fig. 3.2). These services leverage HTTP to communicate intermediate results. While no code is shared between EDGE 1.0 and 2.0, URI compatibility was a high priority for site content. Access to project content follows the EDGE 1.0 URI scheme. New URI and services were created for the administration of the environment. A new database schema was developed in order to improve system performance, while also preserving existing metadata. Visual aspects of EDGE were updated to meet the expectations of a modern look and feel, while also improving usability (Fig. 3.3). Work on EDGE 2.0 was completed on January 10th, 2016.

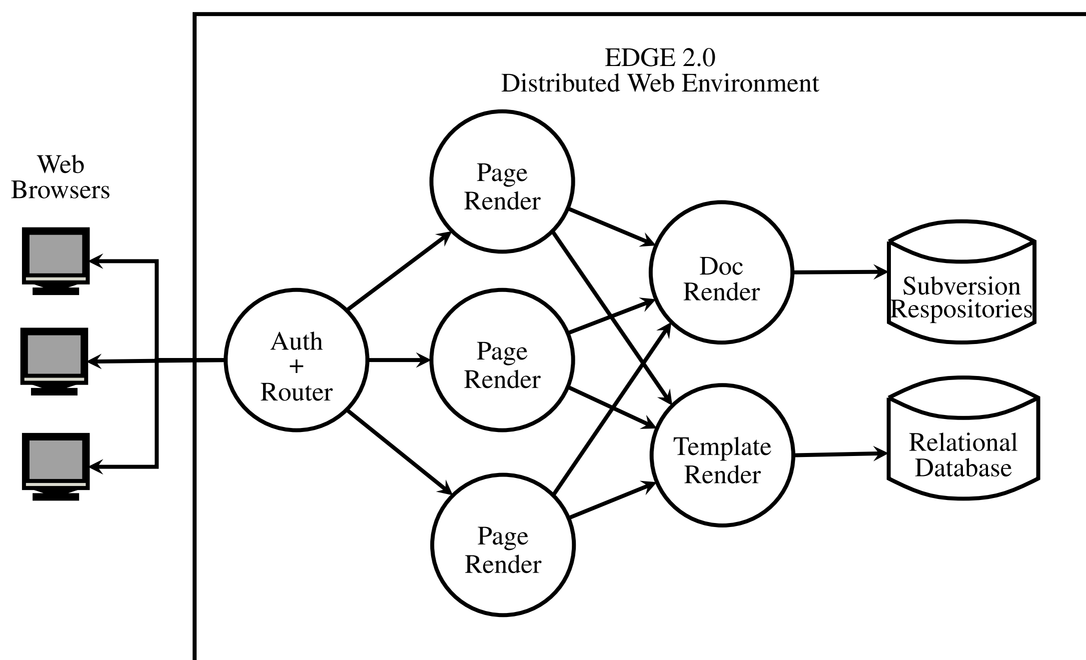

Figure 3.2: EDGE 2.0 Architecture

The following sections recount the development effort over this 18 month period. First, there will be a discussion of the efforts made to ensure REST compatibility in EDGE. This is followed by a discussion of the implementation of the core framework of EDGE, known

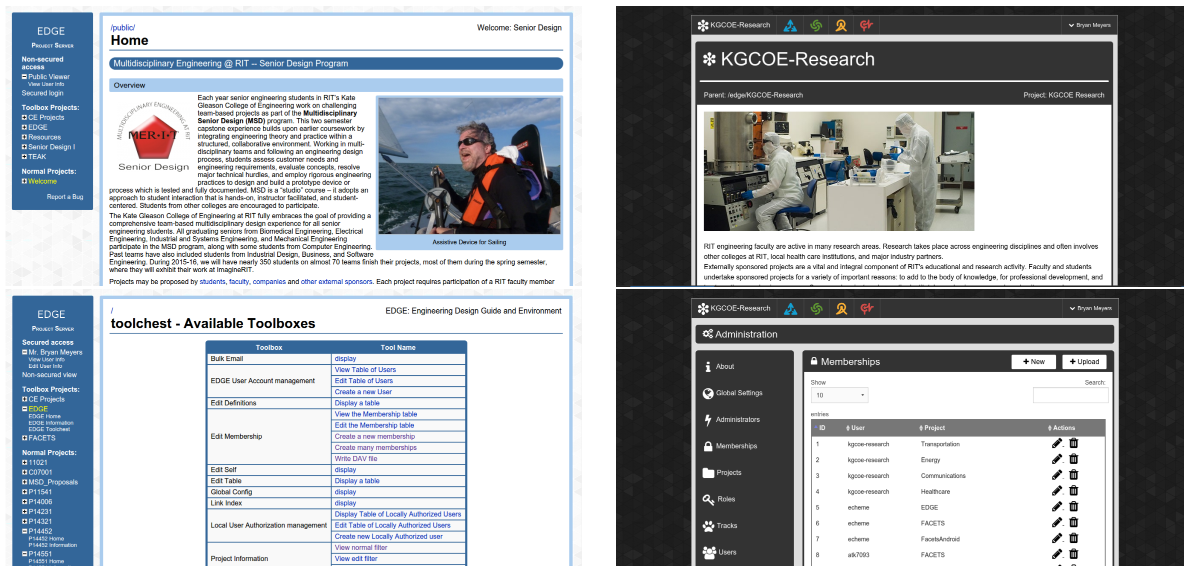

Figure 3.3: UI Examples for EDGE 1.0 (Left) and EDGE 2.0 (Right)

as Wire. The subsequent sections discuss the functionality of the REST endpoint services in EDGE, the schema for the relational database, and its integration with Subversion. Design and implementation of the web front-end is then described. Lastly, a brief overview of the open-source software used to build EDGE and the mechanism for deploying a new system are discussed.

3.4.1 REST Compatibility

In order for EDGE to be REST compatible, it must meet three main requirements. First, a REST capable protocol must be used for all high-level communications. Second, it must make proper use of Uniform Resource Identifiers. Third, it must make use of Hypermedia as the Engine of Application State (HATEOAS) the following section discusses the implementation decisions which enable EDGE to have full REST compatibility.

Protocols High-level communications inside of EDGE occur between a client and the REST endpoints, as well as between the distributed services that make up the DWS environment. These communications are performed via the HTTP protocol. HTTP is already

a REST compliant protocol, requiring no additional developer effort for these communications to be REST compliant. However, it is necessary for REST services in EDGE to use HTTP when communicating with one another. EDGE makes this possible by forcing services to use an HTTP client to send requests to each of the services. Services are required to only accept requests via HTTP. Communication with the REST endpoints already meets this requirement when using an HTTP web browser.

URI REST compliance for URIs falls into two categories. First, each REST endpoint and service within the DWS must possess a unique Uniform Resource Locator (URL). Services and endpoints in EDGE meet this requirement by being assigned a URL that describes its purpose (*i.e.* /db, /history, /edge). Second, specific data representations that are produced by a service or endpoint must be further identified by a unique Uniform Resource Name (URN). Service and endpoints in EDGE meet this requirement by assigning a URN when a representation is first created (*i.e.* /db/users/1234). Meeting these two requirements allows a REST client to request that an action be performed for a specific representation within the DWS.

HATEOAS In order to be REST compliant for HATEOAS, all client-server communications are forbidden from relying on information stored about the client by a server. For the client, this means that all information necessary to complete a request must be sent as part of the request itself. For the server, this means that it must never store information about a client for the handling of future requests. EDGE meets the client requirement by using cookies to store information about a client's current session. These cookies are stored on the client and can later be transmitted with a request. EDGE meets the server requirement through a development contract which explicitly prohibits the storage of client state server side. This constraint is enforced by other developers through code review and stated best practices.

3.4.2 REST Framework — Wire

Ruby's Sinatra framework provides the tools to quickly build APIs that meet an application's specific needs. However, functionality in EDGE is built by composing REST services on top of one another. This modularity allows the code base to be reduced significantly by reusing core functionality whenever possible. Therefore, it was decided to build a core set of reuseable services on top of the Sinatra DSL to serve as the foundation of EDGE. This framework would later be referred to as Wire.

The name Wire came about as part of an analogy of a DWS. In a DWS, multiple REST services work together to satisfy a client request. This composition of services works a lot like how a network administrator would set up a computer network. Each server is connected to the others through electrical or fiber optic "wires". Wire is a framework which networks multiple REST services together (Closet), configures those services to perform specific tasks (Applications), providing each service with a uniform request format (Context), and supplies an additional layer of security in the form of Authorization.

Closet The Closet is the single most important component in Wire. Much like an actual networking wire closet, the Closet is responsible for orchestrating the connections between REST services. Client requests are translated into a uniform request Context, routed to their respective Application based on URI, and then processed by the Application itself. Wire uses its own Ruby Domain Specific Language to configure the Closet and the Applications running inside of it. Configuration manages the dependencies between services, specific options for individual services, and also data dependencies for layout templates.

Context Ruby Rack provides a common interface which developers can use to build web services. This allows new web servers to be written for Rack without breaking compatibility for services built on top of Rack. The Closet is an implementation of the Rack interface. When receiving a request from a Rack-compatible web server, the Closet transforms the

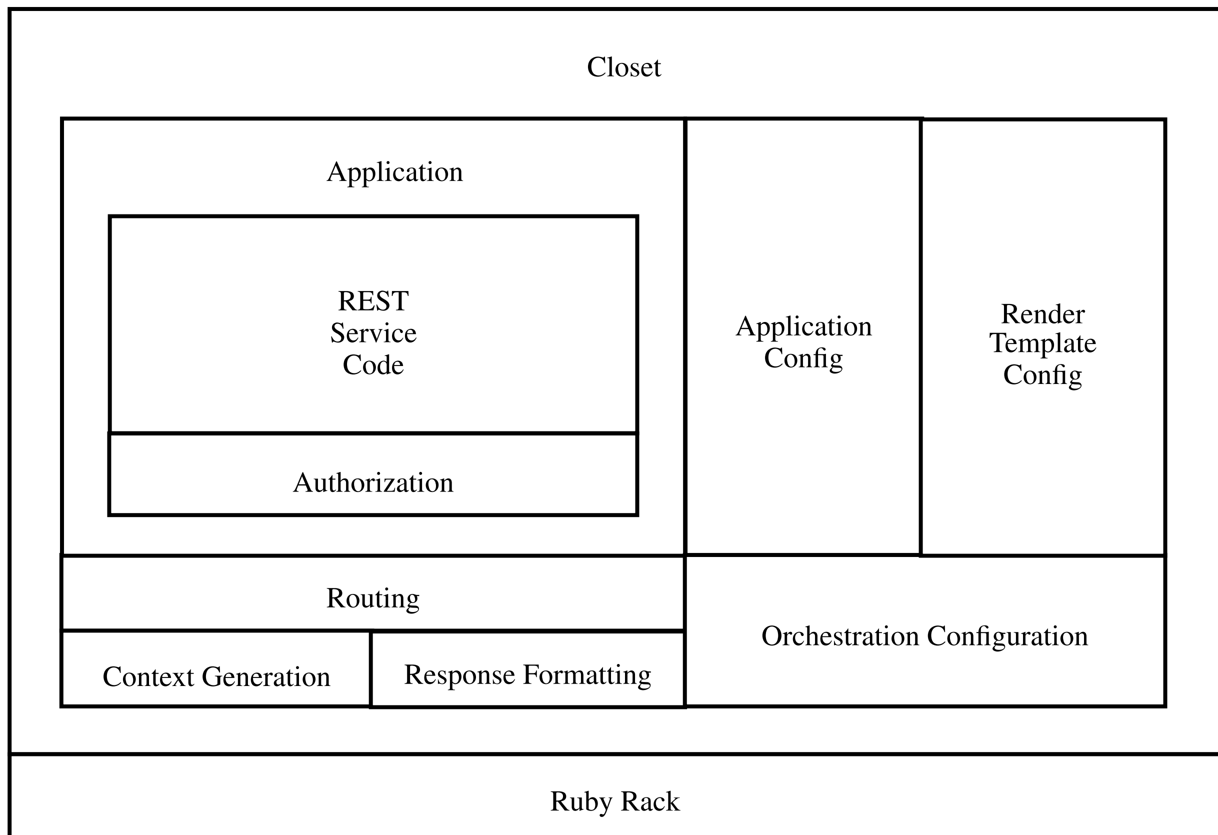

Figure 3.4: Wire Architecture

Rack request format into a easier to use internal format called a Context. A Context contains all of the information from the Rack request, as well as meta-information about the Closet's configuration and the Application that will handle the request.

Applications A Wire Application is a REST service that is accessed by URI. Applications may be instantiated as many times as necessary and may have unique configurations. Several useful core Applications exist as part of the framework to bootstrap the development process. Authorization for an Application can be performed use one of the built in mechanisms (*i.e.* any, read-only, user-specific) or by implementing a custom authorization handler. Successfully authorized requests are assigned a subset of CRUD actions which are allowed for the current request.

3.4.3 REST Endpoints — Applications

The Wire framework provides a standard set of Applications for developers to use to jump-start a new project. Each Application serves a purpose which is distinct from the others. These Applications are meant to provide basic functionality and may be replaced by newer versions or supplemented by the custom Applications that are needed for a particular DWS. The following descriptions outline the basic functionality of each of the provided applications.

File The File Application can be used to serve any file "as-is". No rendering or transformations will be applied to the file or its contents. A best-effort attempt will be made to identify the MIME Type of the file and report this in the HTTP `Content-Type` header of the response. This Application will most likely be used to handle static assets such as site logos, fonts, and JavaScript. Additionally, the File Application functions as a read-only service that will never modify the files being served.

DB The DB Application can be used to communicate with any database supported by the Ruby DataMapper library. It provides a basic CRUD interface (Create, Read, Update, Delete) to the database. Each table in the database is specified by assigning a sub-URL and a DataMapper Model. For read operations, the DB Application will return one or more results as a well-formed JSON document. For write operations, the DB Application only accepts JSON documents.

Render Render is a collection of Applications which transform between data representations. The supported CRUD actions are defined according to the functionality of the renderer. These Applications are intended to provide a complete set of tools for handling user facing content. The following Applications are available in Render:

- **Document**

  The Document renderer will request a document from another service and render the response according to the MIME Type. If no renderer is found for that MIME Type, the response will contain the unaltered document. One or more MIME Types may be assigned to a document renderer inside the Closet.

- **Editor**

  The Editor renderer will request a document from another service and place the response into an editor template based on the MIME Type. If no editor template is found for that MIME Type, the response will indicate that no editor was found. One or more MIME Types may be assigned to an editor template inside the Closet.

- **Error**

  The Error renderer relays a request to another service. If the response has a status code other than HTTP 200 Success, it will attempt to find an error template to render instead. If a template cannot be found, the response is simply forwarded. One or more error codes may be assigned to an error template inside the Closet.

- **Instant**

  The Instant renderer only accepts an HTTP Post request. Instant shares its configuration with the Document renderer. It will attempt to match the MIME Type indicated by the request URI with a document template. If a template is found, it renders the content of the request into the template and returns the response. The practical application of Instant is to provided previews for renderable documents.

- **Page**

  The Page renderer forwards a request to another service and then renderers the response into a page template. Optionally, a global template may be defined to use as a layout for a Page renderer. Use of this global layout can be enabled or disabled on a per Application basis. The Page renderer will likely be used to produce the final HTML document to be viewed by a web browser.

- **Partial**

  The Partial renderer functions similarly to the Page renderer, but is intended to perform an intermediate transformation before rendering the final page. It can be used in conjunction with a Page renderer to render information from several different services.

- **Style**

  The Style renderer transforms a SASS or SCSS stylesheet into minified CSS to be used by a web browser. A single Style renderer may serve one or more stylesheets. Styles are rendered only once, when the Closet is first initialized.

Repo Repo is a collection of Applications which provide a CRUD interface to version-controlled repositories. Currently, only Subversion is supported. A Repo Application will respond to a GET request with the latest version of a document with the MIME Type set. If the document is a directory, it will render a directory listing with links to the files and higher-level directories.

History History is a collection of Applications which provide a read-only interface to the revision logs of version-controlled repositories. Currently, only Subversion is supported. A History Application will respond to a GET request with the full history log for the repository, directory, or file specified.

Login The Login Application forces a client to temporarily redirect to another URI. This can be used to force a client to authenticate before continuing. Its only purpose is to perform this redirection.

Cache The Cache Application provides a LMDB-based cache for responses from another service. For a read operation, the Cache looks for a cached copy of the response. If none exists, it forwards the request to the service. For write operations, the Cache forwards the write request to the service. Upon completion, it requests a new version of the read response for that URI, replacing the representation in its cache and then returns the original response to the client.

3.4.4 Database Schema

Project documentation in EDGE is stored solely in a Subversion repository. Information about a project or an EDGE installation (metadata) is stored in an SQL relational database. EDGE makes use of relational data to support user-specific settings, administration, project management, and security. The core of EDGE relies on seven primary tables for operation (Fig. 3.5). EDGE interacts with the database through the Ruby Gem, DataMapper. The DataMapper Gem acts as an Object Relational Model (ORM), capable of translating between the database tables and native Ruby Objects. This greatly simplifies the task of manipulating database entries and automates the process of creating and upgrading the database schema. It is not necessary for a developer to have any SQL knowledge to use DataMapper. Further, DataMapper does not require a specific SQL database, supporting a wide range of SQL databases servers. The following sections discuss the schema chosen

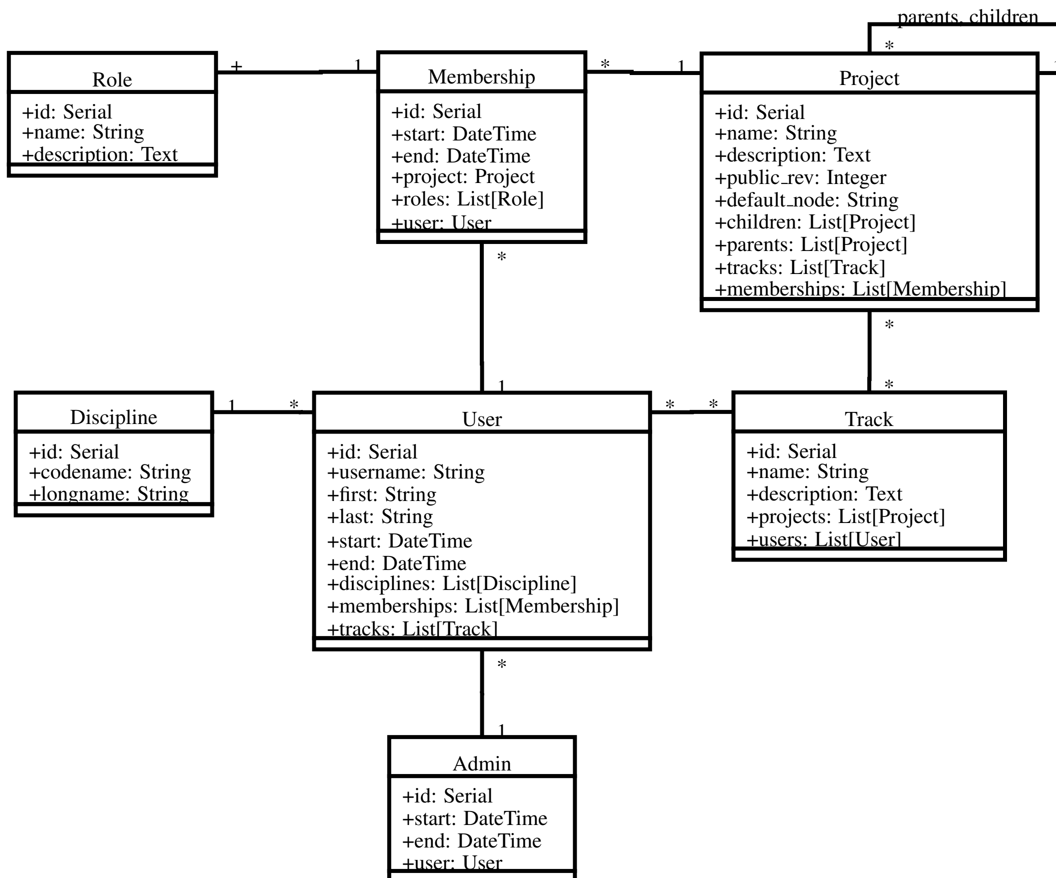

Figure 3.5: EDGE 2.0 Database Schema

for the EDGE system.

User Users are anyone who is able to log in to the EDGE system. Each user is uniquely identified by a single `username`. Optionally, a user may enter their first and last names to further clarify who they are. A start and end date are used to limit access for a user. When the end date has passed, a user is no longer able to perform EDGE tasks. At this time, the end date may be moved into the future, or a new record may be created in order to preserve an audit trail. A User may belong to one or more Disciplines, Projects, or Tracks.

Admin The Admin object is used to grant access to administrative functionality in EDGE. An Admin record specifies a user to elevate to Administrator. Each record features a start

and end date for granting this access. When the end date has passed, a user is no longer able to perform administrative tasks. At this time, the end date may be moved into the future, or a new record may be created in order to preserve an audit trail.

Discipline A Discipline is used to define an organizational unit that a user might belong to. Definition of a Discipline is performed by assigning a codename (*e.g.* ME) and a longname (*e.g.* Mechanical Engineering). In this case, users belonging to the ME Discipline might include practicing Mechanical Engineers, technicians, or support staff. Users may belong to more than one discipline.

Track Tracks are common themes or focus areas that help to group and describe development efforts. A Track is defined by a short name (*e.g.* Healthcare) and a longer description. One or more Users or Projects may be assigned to a Track based on their focus area.

Project Projects exist for any development effort carried out by EDGE Users. A Project is defined by an identifier (*e.g.* P14452), a long name (*e.g.* Dresser-Rand Compressor Wired Data Recorder), and a lengthier description. The `public_rev` field is used to indicate which revision of the repository to show for publicly viewable documents. The `default_node` field indicates which document to use as the public homepage of the project. Projects may be assigned to one or more Tracks, and have no limit to the number of Memberships. Additionally, a Project may have one or more Parent or Child Projects as part of a family tree.

Role Roles are used to assign subsets of permissions to a User for a given Project. A Role is defined by a short name (*e.g.* Editor) and a lengthier description. Delegation of EDGE permissions to a Role currently takes place in the authorization logic for EDGE and is not configurable.

Membership Memberships allow a User to be assigned to a Project, with a subset of Roles. Each Role grants the User a subset of permissions with respect to the Project. Memberships have both a start and end date which can be used to limit a User's access to the Project. When the end date has passed, a User is no longer able to perform Project tasks. At this time, the end date may be moved into the future, or a new record may be created in order to preserve an audit trail.

3.4.5 Subversion Integration

The Ruby language does not have any native client support for Subversion. In order to integrate SVN with EDGE, it was necessary to develop a library to interface with the command-line utilities `svn` and `svnadmin`. Several of the supported operations for these tools do provide XML as an alternative output format for the results of operations. This was leveraged in order to eliminate the need for string parsing or regular expression matching of command line results. The resulting SVN library was implemented to be compatible with the software interface defined by the Repo and History Applications provided by the Wire framework.

Create In order to create a new SVN repository, the `svnadmin create` command is leveraged. This command creates a new empty SVN repository whenever a new Project is first created. The command will fail gracefully if a repository already exists.

Read Reading a document from the repository is performed whenever it is rendered, edited, or downloaded by a user. In the previous version of the EDGE system, the whole repository would first need to be checked out to a temporary destination. The MIME-Type of the file was then determined, and the server would return the requested file. Periodically, a CRON task would then clean-up the repository checkouts. This workflow suffered from long checkout times for repositories with large histories or big files. It also required a sizeable amount of temporary space to handle multiple checkouts. In EDGE 2.0, this was

remedied through the use of the `svn cat` command. This command returns the binary content of the file directly, without the intermediate checkout. The `svn propget` command was then used to determine the MIME-Type for the file. This resulted in much faster read operations and a decrease in the required disk space.

Update Any time a document is created or modified through the web front-end, it is necessary to follow the same sequence of operations that a user would perform locally. First, the repository is checked out to a temporary directory using `svn checkout`. Then the file is written to the appropriate directory in the repository. If the file did not exist, the `svn add` command is used to add it to the list of tracked files. Next, the change is committed by calling `svn commit` with a change message provided by the request. Because the change was performed on the behalf of the user, it is necessary to modify the revision record to reflect the appropriate author. This action is performed using the `svn propset --revprop` command. Lastly, the temporary checkout is removed from disk.

Delete Deletion of a repository or individual file is expressly forbidden through the web front-end. This serves to prevent accidental data loss due to user error. Instead, a user must perform a file or directory deletion in a local checkout and then committing the changes. Because this action requires additional thought and effort, it mitigates the risk of a file or directory being accidentally removed while a user does not have access to a full SVN client.

History The revision history of an SVN repository may be accessed using the `svn log` command. EDGE executes this command with the `--xml` flag in order to retrieve an XML representation of the revision history. The resulting XML is then converted into a JSON representation to be consumed by other REST services. Revision history may be accessed for a single file, directory, or the entire SVN repository.

3.4.6 Web Front-End

Apart from Wire, the web front-end for EDGE represents the bulk of the software. The front-end was built around modern HTML5, CSS3, and Javascript technologies. It relies on several well-known web libraries: Foundation and SASS (CSS), Dojo and DataTables (JS), and HAML (HTML). The front-end exists as multiple distinct views which share a common layout and are served by REST service endpoints. The following is a discussion of the implementation of the EDGE web front-end, with respect to the technologies used and the functionality provided.

Technologies With web development gaining major support from the open-source software community, there are many libraries which can be used to accelerate the development process. EDGE relies on HAML for authoring layouts and views for each part of the web front-end. Styling of these views is performed by leveraging CSS3. A mobile-friendly look and feel was built on top of the Zurb Foundation CSS library. Additional styling is performed using Syntactically Awesome Stylesheets (SASS), a reduced grammar and templating language for CSS3. The Font-Awesome icon library is used as a uniform icon set across the EDGE front-end. A JavaSript library (SimplyForms) was written, using the Dojo JavaScript Toolkit, to allow HTML forms to be submitted directly to a REST endpoint using JSON as the exchange format. The DataTables JavaScript library was also used, in order to enable sorting, paging, and searching for HTML tables. Working in concert, all of these tools work together to provide a modern web experience and rich user interactivity.

Rendering and Editing Documents from an SVN repository may be rendered using three different views:

1. **/edge**

   The EDGE view renders mark-up files, audio and video players, and images from an SVN Repository to HTML and then places this HTML is the global EDGE layout.

This is the primary view for rendering documents to the web.

2. **/content**

   The Content view renders documents the same way as the EDGE view, except that it ignores the global layout. This view is especially useful for embedding rendered documents in other HTML pages or content management systems like the RIT my-Courses service.

3. **/repos**

   The Repos view performs no rendering of any kind on a document, sending the entire file to a web client. It is up to the client to either render the document or download it to a file for local viewing.

Views

- **Dashboard**

  The Dashboard provides a "one stop shop" for EDGE users. Here they can quickly navigate to their project and update their personal information. DataTables is used to allow a user to sort and search for any project where the have membership. They can also view the roles assigned to them for each of the projects. Users can modify their personal information to include a first and last name, the tracks they belong to, and contact information.

- **Project Info**

  Each EDGE Project has its own unique Project Info page. It provides a quick overview of useful information for a project including: a description, the tracks it belongs to, useful links (*i.e.* SVN Repository URL, Public Homepage), and a complete list of memberships.

- **Track Info**

Each EDGE Track has its own unique Track Info page. It provides a quick overview of useful information for a track including the description and a complete list of Projects belonging to the Track.

- **Administration**

  The Administration view allows an EDGE Admin to view information about the version of EDGE, its configuration, and each of the tables in the SQL database. Admins can use this interface to create new Projects, Memberships, Disciplines, and Tracks. Here, they can also edit these objects and set the access restriction dates for Users and Memberships.

3.4.7 Deployment

Installation of the previous EDGE system required a significant amount of time for a system administrator. First, the administrator would have to manually install an Ubuntu Server OS. Second, they would then checkout the EDGE source from SVN. Third, they would then install all OS packages necessary to support EDGE. Then they could configure MySQL and Apache, create repository locations, and configure EDGE to connect to the MySQL server. Finally, they would be able to enable the EDGE website for access. This whole process could take a half hour for an experienced administrator and upwards of three hours for a beginning administrator.

Every effort was made to simplify this process as much as possible for EDGE 2.0. Installation of the Host OS is taken care of by an automated install script. Packages, source code, and configuration are completely automated through the Ansible provisioning tool. An administrator need only initiate a network boot to start the installation process, create the Ansible variables required for provisioning, and start the provisioning process. The rest of the installation is performed entirely automatically and requires no intervention by an administrator. EDGE 2.0 also leverages DataMapper to perform the initialization of a schema and content for the SQL database. Further, EDGE 2.0 has full support for

connecting to remote database servers, eliminating the need to install one locally.

3.4.8 Summary of Open-Source Software Used in EDGE

EDGE 2.0 made extensive use of dozens of open-source software projects for its design, implementation, and deployment. The Open-Source applications used in the installation, execution, and administration of EDGE are listed in Table 3.2. The Open-Source libraries used by the EDGE application are listed in Table 3.3. The Open-Source applications used in the development and testing of EDGE are listed in Table 3.4.

Table 3.2: Open-Source Applications

Name	Version	License	Purpose	URL
Ansible	1.9.4	GPL 3.0	Provisioning	Link
Apache HTTPd	2.4.7	Apache 2.0	Web Server	Link
Bundler	1.10.6	MIT	Gem Management	Link
Gem	2.4.5.1	MIT	Gem Installation	Link
MySQL	5.5.49	GPL 2.0	Relational DB	Link
MyWebSQL	3.6	GPL 3.0	DB Administration	Link
Puma	2.15.3	BSD 3-Clause	Ruby Rack Server	Link
Ruby	2.2.4	BSD 2-Clause	Ruby Interpreter	Link
Subversion	1.8.8	Apache 2.0	Version Control	Link
Ubuntu Server	14.04	Commercial	Operating System	Link

Table 3.3: Open-Source Libraries

Name	Version	License	Purpose	URL
cLogger	2.0.2	GPL 2.1	Logging	Link
DataMapper	1.2.0	MIT	ORM	Link
DataTables	1.10.7	MIT	Interactive Tables	Link
Docile	1.1.5	MIT	Ruby DSL Builder	Link
Dojo	1.10.7	BSD 3-Clause	JavaScript Framework	Link
Font-Awesome	4.3.0	MIT/Mixed	Icon Theme	Link
HAML	4.0.7	MIT	HTML Templating	Link
jQuery	2.1.4	MIT	JavaScript Framework	Link
LMDB	0.9.8	OpenLDAP PL	Persistent Cache	Link
mysql2	0.4.3	MIT	SQL Queries	Link
Rack	1.6.4	MIT	Ruby Server Interface	Link
REST-Less	0.1.3	MIT	HTTP Client	Link
SASS	3.4.21	MIT	CSS Preprocessor	Link
Tilt	2.0.2	MIT	Document Rendering	Link
Wiki-This	0.1.4.7	MIT	MediaWiki Rendering	Link
Wire	0.1.4.26	MIT	Ruby REST Framework	Link
Zurb Foundation	5.5	MIT	CSS Framework	Link

Table 3.4: Open-Source Development Tools

Name	Version	License	Purpose	URL
Firefox	47	Mozilla PL 2.0	Validation	Link
Git	1.9.1	MIT	Source Control	Link
GitHub	n/a	n/a	Public Source Hosting	Link
GitLab	n/a	n/a	Private Source Hosting	Link
GNU nano	2.2.6	GPL 2.0	Test Editor	Link
Google Chrome	47	Commercial	Validation	Link
Graphviz	2.38.0	Eclipse PL	Documentation	Link
JetBrains Rubymine	7.0.0	Commercial	Ruby IDE	Link
YardDoc	0.8.7.6	MIT	Documentation	Link

3.5 Extensibility

Every effort was made to make EDGE as extensible as possible. The Wire framework serves as a strong foundation of tools for building new applications and orchestrating the deployment of EDGE. The design philosophies of modularity, REST, and reusability enable the introduction of new features to the EDGE DWS in the form of new web applications, support for new document formats, alternative forms of document storage, integration with external service providers, and increased security.

Applications The most important concept of DWS and REST design is the idea of composition. No one application is built upon a single monolithic program. Instead, it is built upon smaller reusable services that each perform a particular function, such as manipulating a database, rendering a document format, communicating with an external service, or even reconfiguration of the DWS. Composition of a DWS application is the act of connecting instances of reusable services together. New applications may be composed of existing services or require the creation of new services to provide non-existent functionality. This allows the development effort to center around building rich web interfaces for interacting with these services.

In the event that a new Wire Application needs to be developed a developer need only implement a single function to satisfy the interface. The `invoke()` function takes in the list of CRUD operations resulting from an authorization process and a Wire Context (Fig. 3.6). The Context contains all of the information from a REST request and information about configuration for an instance on the application. Extension of the Wire DSL allows developers to provide configuration information for the creation of a Context. The response for a Wire Application must be in the form of a Rack triplet: a three item array containing Response headers, the body of the response, and an HTTP status code. Wire Applications may perform any functionality desired, so long as this interface is followed.

```
# Proxy method used when routing
# @param [Array] actions the allowed actions for this URI
# @param [Hash] context the context for this request
# @return [Response] a Rack Response triplet, or status code
def self.invoke(actions, context)
        case context.action
                when :create
                        do_create(context)
                when :read
                        if context.uri[3]
                                do_read(context)
                        else
                                do_read_all(context)
                        end
                when :update
                        do_update(context)
                when :delete
                        do_delete(context)
                else
                        403
        end
end
```

Figure 3.6: Example Wire::App Implementation

Document Rendering EDGE document rendering is handled by Ruby libraries. Each renderer is associated with one or more MIME Types for it to handle. If a renderer gains support for a new MIME Type a simple configuration change is all that is required to enable it. New renderers may be incorporated into EDGE by configuring them for supported MIME Types. If a renderer does not exist for a particular MIME Type, it will be treated as downloadable content or it will be necessary to develop a suitable Ruby library. In the case of EDGE, it was decided to implement a custom MediaWiki renderer to allow developers to integrate new mark-up features not supported by the core MediaWiki dialect. Figure 3.7 shows how Wire handles the configuration for a renderer.

```
renderer :image do
        partial 'views/partials/image.haml'
        mime 'image/bmp'
        mime 'image/gif'
        mime 'image/jpeg'
        mime 'image/png'
        mime 'image/svg+xml'
        mime 'image/tiff'
end
```

Figure 3.7: Example Renderer Configuration

Document Storage Wire provides a common interface for adapting a repository system into a REST service through the Repo Application. This application implements all of the web-facing functionality, only requiring a developer to implement the following functions for a new repository type:

`do_create_file()`

Satisfies an HTTP POST request by creating a new repository or file.

`do_read_file()`

Satisfies an HTTP GET request when the requested item is a file.

`do_read_listing()`

Satisfies an HTTP GET request when the requested item is a directory.

`do_read_info()`

Retrieves metadata for the requested file or directory.

`do_read_mime()`

Retrieves MIME Type for the requested file or directory.

`do_update_file()`

Satisfies an HTTP PUT request by updating an existing repository or file.

A full explanation of the Repo interface can be found in Appendix A.3.

Wire also provides a common interface for accessing the history of a repository system, via a REST service, through the History Application. This application implements all of the web-facing functionality, only requiring a developer to implement a single function. The `get_log()` function retrieves the history for a specific file or directory. The History Application then utilizes a HAML template to render this information to HTML. A full explanation of the History interface can be found in Appendix A.4.

External Services One of the commonalities of most modern REST systems is the use of HTTP as an application protocol. This means that while HTTP itself may be a communication protocol, it is possible to build robust and powerful functionality around it. HTTP URIs can be used to identify specific services and functionality. HTTP methods may be used to perform semantic actions of these URI. URI may also be linked together to build complex workflows, interactive wizards, or stateful applications.

For a DWS a protocol like HTTP is the glue that holds everything together. It allows services to communicate inside of an application or, more interestingly, with external services. The Wire framework makes use of the `rest-less` HTTP client to communicate between services. Rest-less provides a single function which allows a request to be made to any service, internal or external. This makes it possible to author new Wire Applications which

serve as “abstraction layers” for external web service APIs, providing a Wire compatible interface to other services in the application. The end result being a trivial mechanism for incorporating external services into a Wire-powered DWS.

Security EDGE 2.0 does not provide an internal mechanism for authentication. While many libraries exist for Ruby that allow interaction with one or more authentication sources, it was decided that the administrator be given as much flexibility as possible when selecting an authentication source. At RIT, authentication for EDGE is handled by the Apache HTTPd web server. This allowed administrators to pick and choose from the large number of authentication modules for Apache to configure an authenticated proxy for the Puma Rack server. It is only necessary that the proxied request have the HTTP `Remote-User` header set to the authenticated user’s username.

Authorization in EDGE is handled by the Wire::Auth module. While developers may choose to use one of the existing authorization schemes, it is also possible to implement a application-specific scheme through the Wire::Auth interface. This requires a developer to implement a single method called `actions_allowed()`. This function is provided the context for the current request and is expected to return a list of the allowed CRUD operations. The list of actions is then passed to the `invoke()` method for the Wire Application. Figure 3.8 describes the authorization interface.

```ruby
# Get the actions that are possible for the current request
# @param [Wire::Context] context the context for this request
# @return [Array] the privileges for this user
def actions_allowed( context )
```

Figure 3.8: Wire Authorization Interface

Chapter 4

Dataflow Model: Theory and Development

The term *dataflow* can refer to several concepts. For this work, a *dataflow model* is a description of concurrent processes in which one process is connected to another by one or more directional links used to communicate information. A *dataflow architecture* is a physical system which operates according to the behavior of a dataflow model. Processes in a dataflow model may depend on information in order to perform their specific function. If a process cannot perform work without receiving information from another process, this communication is referred to as *blocking*. Work which can be performed without receiving information from another process is referred to as *non-blocking*.

Dataflow models can be decomposed into the following components: computational units, network communication links, network interface units, and network routing devices. Computational units perform processes which generate or transform information. Network interface units arbitrate the use of a single network communication link by one or more computational units. Network routing devices facilitate the transfer of information from one network interface unit to another. A network communication link allows information to be transferred between a network routing device and a network interface unit. Operating in concert, these computation and communication components provide services to client devices, and make it possible to model DWS as dataflow architectures.

Translating a DWS into a dataflow architecture is a two phase process. First, the computational processes must be decomposed into discrete tasks, assigned a cost based on the time required to compute a result, and associated through task-dependency graphs. Tasks requiring information produced by other tasks are referred to as dependent tasks. Second,

the task dependencies between computational devices are translated into communication tasks. Each network link will be assigned a capacity reflective of their bandwidth and will be assigned to a network routing device. A network routing device will also be assigned a capacity for its peak internal bandwidth. The resulting decomposition represents a complete DWS, statically assigned resources for each of its components.

This chapter will cover three topics. First, assumptions and limitations of the dataflow model are discussed. Second, the implementation of the dataflow simulator, DWSim, are described with respect to its computational model, networking model, configuration, datasets, and metric collection. Third, automated testing and verification of DWSim are discussed.

4.1 Assumptions and Limitations

This section describes the assumptions made in the dataflow model, an in-depth discussion of the practical limitations of what the model is capable of simulating, and the limitations imposed upon the design of DWSim.

4.1.1 Assumptions

Several assumptions underlie the dataflow model. Each assumption describes a physical limitation of real computer systems that was explicitly relaxed in the behavior of the model. These assumptions serve to limit the scope and duration of this work, but are acknowledged as non-trivial factors in real-world scenarios.

DWS elements have finite processing throughput. Processing elements employ a variety of features which impact their achievable throughput. These features may include: the scaling of operating frequency to reduce heat production or energy consumption, dynamic clock scaling is response to processing demand, “turbo” modes which allow higher throughput for a single hardware thread in the absence of other active hardware threads, and the hardware

implementation of the device. This work makes the assumption that processing throughput capacity is finite and fixed for the duration of the simulation.

DWS elements have finite communication throughput. Communication devices employ a variety of features which impact their achievable throughput. These features may include: spectrum contention for wireless networks, traffic congestion management and Quality of Service, data consumption limits, and traffic priority mechanisms. Communications networks may also experience localized failures which reduce the available throughput for a network. This work makes the assumption that communications throughput capacity is finite and fixed for the duration of the simulation.

DWS elements have infinite energy scalability. Every element of a computing environment consumes power when in use. Processors manage a limited power budget through the management of clock speed, disabling of unused processing elements, and the use of low-power idle modes. Network devices may vary their output power in response to line noise, signal attenuation, or reduced traffic flow. Real-word systems may also have to account for loss of power and the retasking of functionality among the remaining functional devices. This work makes the assumption that energy is infinitely available for the duration of the simulation.

DWS elements have infinite memory scalability. Every element of a computing environment requires the use of memory for operation. Processor performance may be impacted by the size of memory buffers, hardware caches, or the availability of RAM for storing application data. Network throughput may be impacted by the size of packet buffers in interface cards and switches. All computing elements are limited by the throughput allowable by the interfaces to memory devices. This work makes the assumption that memory devices possess infinite capacity and interface bandwidth.

4.1.2 Limitations

In addition to the aforementioned assumptions, several limitations exist in the dataflow model and DWSim application. These limitations were either imposed on the design of DWSim to limit the scope of the work or exist due to practical considerations for the use of DWSim.

Imposed The following limitations were imposed on the design of DWSim and its dataflow model:

- **Single Client Request Limit**

  At any given time, a Client may only perform a single request. If a request is currently waiting on a response, no further requests may be made by the Client. If a request is completed before the end of a timestep, the Client may not initiate another request until the following timestep.

- **Single Task Limit**

  During each timestep, a single request may be handled by each core of each CPU element. If the request is completed before the timestep is over, it will not be supplemented by other pending tasks.

- **Single Packet Limit**

  During each timestep, a network link may only process a single packet for a given link direction (*i.e.* uplink, downlink). If transmission of a packet is completed before the timestep is over, it will not be supplemented by other pending packets.

Practical The following practical limitations exist in DWSim and the dataflow model, due to a lack of information available from the previously captured Apache HTTPd web server logs:

- **Constant Execution Time**

  It is not possible to log the execution time necessary to complete a given request. As a result, computational times for the handling of a request are assumed to be constant quantities, based on the average time required to handle that kind of request.

- **Constant Message Size**

  It is not possible to log the size of both a request and a response packet. As a result, message size remains constant for every request and response related to an initial client request.

- **Request Start Times**

  It is not possible to log the time the initial request was received. As a result, DWSim uses the completion time of a request as the start time of a request during simulation.

- **Constant Link Characteristics**

  It is not possible to determine the bandwidth and latency of a network link between client and server. As a result, client bandwidth and latency were determined experimentally and assumed to be constant quantities.

4.2 Implementation

DWSim was written entirely in the Go programming language. Go was selected for its ease of use, performance as a compiled language, and powerful parallelization features. It is composed of five major subsystems: processing, networking, configuration, data input, and metric gathering. The processing subsystem is responsible for simulating the execution of arbitrary task graphs against multiple processors to simulate the handling of web-oriented computation. The networking subsystem is responsible for simulating a the infrastructure necessary to allow computational elements to communicate. The configuration subsystem is responsible for setting up the processing and networking subsystems at runtime. The

data input subsystem is responsible for replicating a captured workload during a running simulation. The metric gathering subsystem is responsible for reporting and storing the performance values observed during a simulation.

4.2.1 Processing

The DWSim processing model consists of three components: Tasks, CPUs, and Requests. Tasks are used to perform computation and communication for a particular REST service. CPUs execute Tasks in response to client Requests. A Request specifies the Tasks that a CPU must execute to complete a transaction. The following section discusses these component models.

Tasks A Task represents computation or communication to be performed by a CPU. A task may be dependent on the completion of one or more other tasks before executing. These dependencies are established through task-dependency graph in the DWSim configuration. DWSim supports three kinds of tasks. A `TimedTask` represents computation that requires a certain amount of time to complete processing. `TaskGroups` are collections of one or more tasks that must be completed one at a time before continuing. `RESTTasks` are communication tasks which require that a request be made to another service and a response to be received before continuing. Through TaskGroups and task replication it is possible to represent any task dependency graph in DWSim (Fig. 4.1).

CPU A CPU is able to receive a request and execute a task-dependency graph to create a response. DWSim supports two CPU models. `Single` is a model of a single-core CPU which is only able to handle one request at a time. `CMP` is a model of a multi-core processor in which one or more requests may be handled at a time, up to the number of available cores (Fig. 4.2). These CPU models are assigned task-dependency graphs for the REST endpoints they service by the configuration of DWSim. Each processor possesses a NIC for communication with the network model. A work queue is also available for the

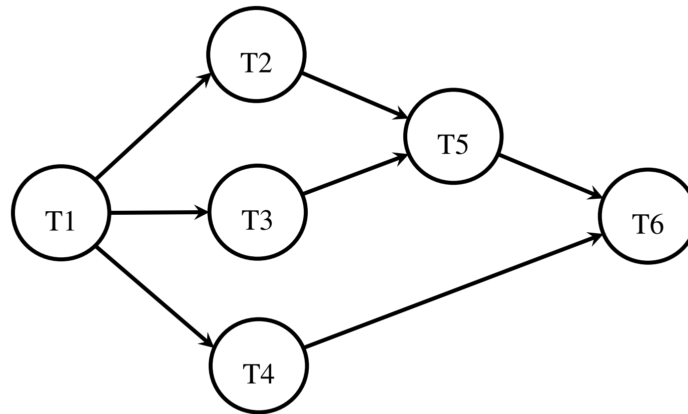

Figure 4.1: Example Task Dependency Graph

storage of pending requests.

Requests and Responses A Request is a Packet which has both a URI and destination address for a particular CPU. The URI indicates which task-dependency graph to execute to generate a response. The destination indicates which CPU should handle this request. Requests are received by the NIC and then either assigned to an idle CPU core or placed into a work queue for later processing. Upon the completion of a Request, a Response packet is generated. This packet is placed in the NIC's transmission queue to be sent to the source address of the Request. Upon being received at its destination, a Response is passed to the core which is executing the `RESTTask` with the same identifier. The `RESTTask` is then allowed to complete execution.

4.2.2 Networking

It is necessary to simulate the communication between client and server devices. The communication model focuses on four high-level components: Packets, NIC, network link, and Level-2 Switch (Fig. 4.3).

Packets A packet in DWSim serves the same purpose as its counterpart in a physical network. Each packet has a unique identifier linked with the request that created it. A

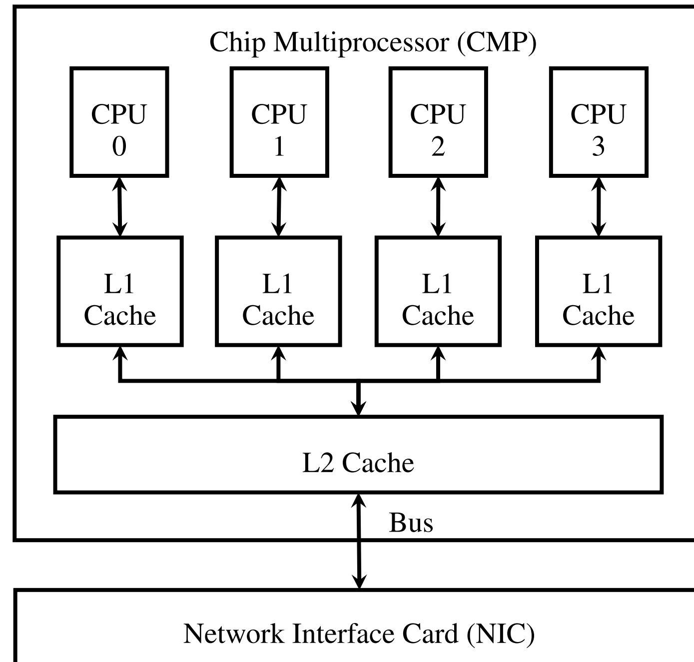

Figure 4.2: Example CMP CPU

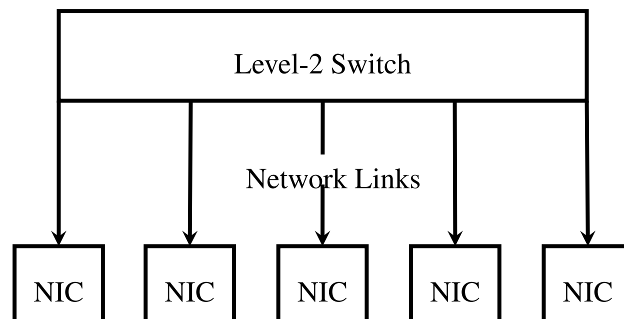

Figure 4.3: Network Model Architecture

source and destination are used to specify which hosts to route the packet between. A URI is assigned to indicate the REST endpoint that will handle the request. A size is used to indicate the amount of data the packet contains. These attributes collectively perform the same functionality as a packet in a physical system.

NIC A single NIC is allocated to each CPU to allow communication to other devices. Usage of the NIC is limited to a single executing task at a time. A single request or response may be processed during each timestep for both the downlink and uplink. Additional packets are placed in an internal FIFO to be handled in the next timestep. This allows the NIC to be utilized by other tasks while a response is being generated by either a local or remote process. Each task is given the opportunity to place a packet into the transmit queue and to read a packet from the receive queue during a timestep.

Links A network link is used to connect a NIC to a Level-2 Switch. Each link is assigned a finite bandwidth and a transmission distance. This allows for latency to be calculated according to both the amount of information transmitted (transmission delay) and the distance which that data must travel (propagation delay) (Fig. 4.4). It is important that the network link allow for more than one transmission to occur simultaneously as the propagation delay may allow further transmissions to take place. This is determined by calculating the Bandwidth-Delay Product. For example, round-trip transmission across a $1m$ copper cable takes approximately $10ns$. If that network link has a bandwidth of $100Mbps$ its Bandwidth-Delay Product is 1.0, indicating that no more than one bit exists on the line at any given moment. However, if this distance is increased to $100m$ and the bandwidth to $10Gbps$, the line may hold up to $10,000$ bits of information. With a standard MTU of 1500 bits, this would mean that at any given time up to 6.67 packets may be in transit.

Switch A Level-2 Switch allows for messages to be transferred between network links. The switch model allows for perfect cross-section bandwidth of the switch, meaning that

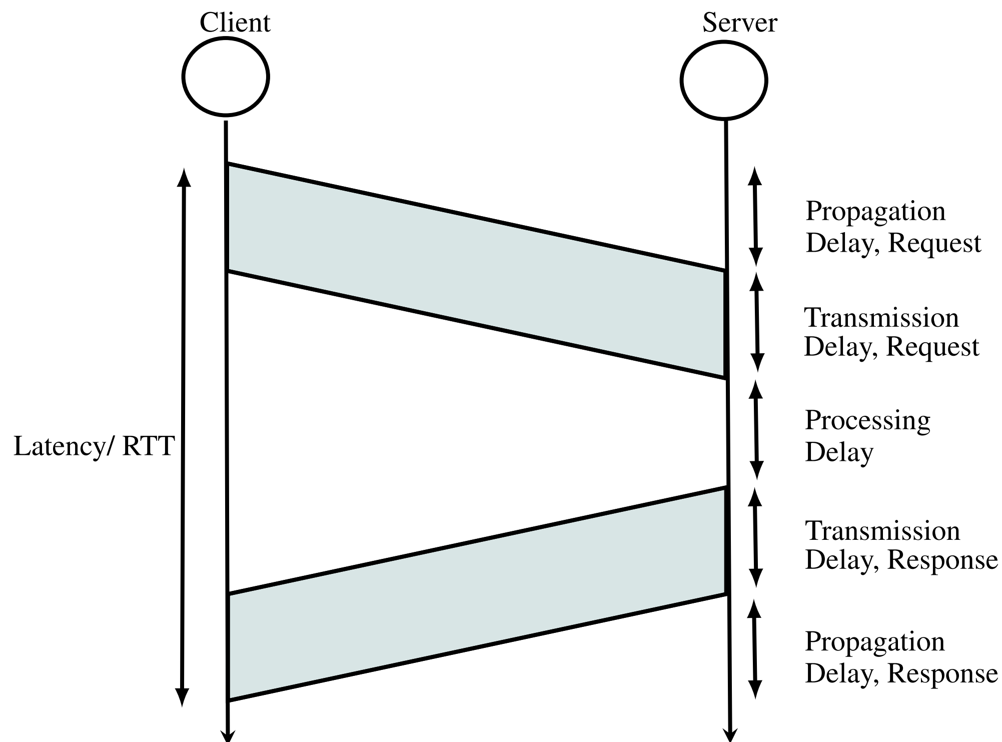

Figure 4.4: Network Latency

the switches internal capacity is equal to the total bandwidth of all of its links. Because a link is allowed to be used for transmission and reception simultaneously, it is possible to transmit and receive one packet, in either direction, during a timestep. A switch is assumed to allow an infinite internal buffer capacity for messages, to be sent in FIFO order. The switch itself is assumed to have negligible routing delay. Instead, bandwidth limitations in the network links are used to introduce realistic transmission latency in the form of packet delays.

4.2.3 Configuration

Initialization of the DWSim environment requires the use of three input files. The Orchestration file defines the physical characteristics of the processors, network links, and switch. The Assignment file defines the computational processes and their allocation to specific processors. The Configuration file is used to specify the aforementioned files, as well as

external datasources and simulation characteristics. The following section describes the usage and implementation of each of these files.

Assignment Assignment is performed through the use of the DOT graphing language. A subset of the DOT language features have been chosen to form the grammar of an assignment file. DOT subgraphs are used to represent processors. Nodes indicate the name, type, and characteristics of a computational task. Nodes declared inside of a subgraph are assigned to that processor. Dependencies between processes are indicated by DOT graph edges. Edges that cross between subgraphs indicate a remote data dependency and instruct the DWSim environment to treat this dependence as a REST operation. Use of the DOT language also allows for visual confirmation of the assignment through the generation of a graphical representation (Fig. 4.5). Lastly, a single Client node must be declared in the root graph (Fig. 4.6). Edges from the Client node indicate the REST entry-points for the DWS and will be used later to route requests for a simulation workload.

Orchestration Like the Assignment file, an Orchestration file uses the DOT graph language. As demonstrated with the Assignment file, the DOT grammar of the Orchestration file may also be used to generate a graphical representation of the system (Fig. 4.7). A different grammar is used in order to simplify orchestration of DWSim (Fig. 4.8). Nodes are used to declare the processors, as well as their type and any physical parameters. A single Switch node is used to declare the model and attributes of the Level-2 switch. Edges are used to define the network links between processors and the switch by specifying the ink bandwidth and latency. While these two files may have been combined from a technical standpoint, the process of manual verification is greatly simplified if the graphical forms are kept separate.

Configuration File The Configuration file provides all of the information necessary to begin the initialization of a DWSim environment, in the form of a YAML (Yet Another

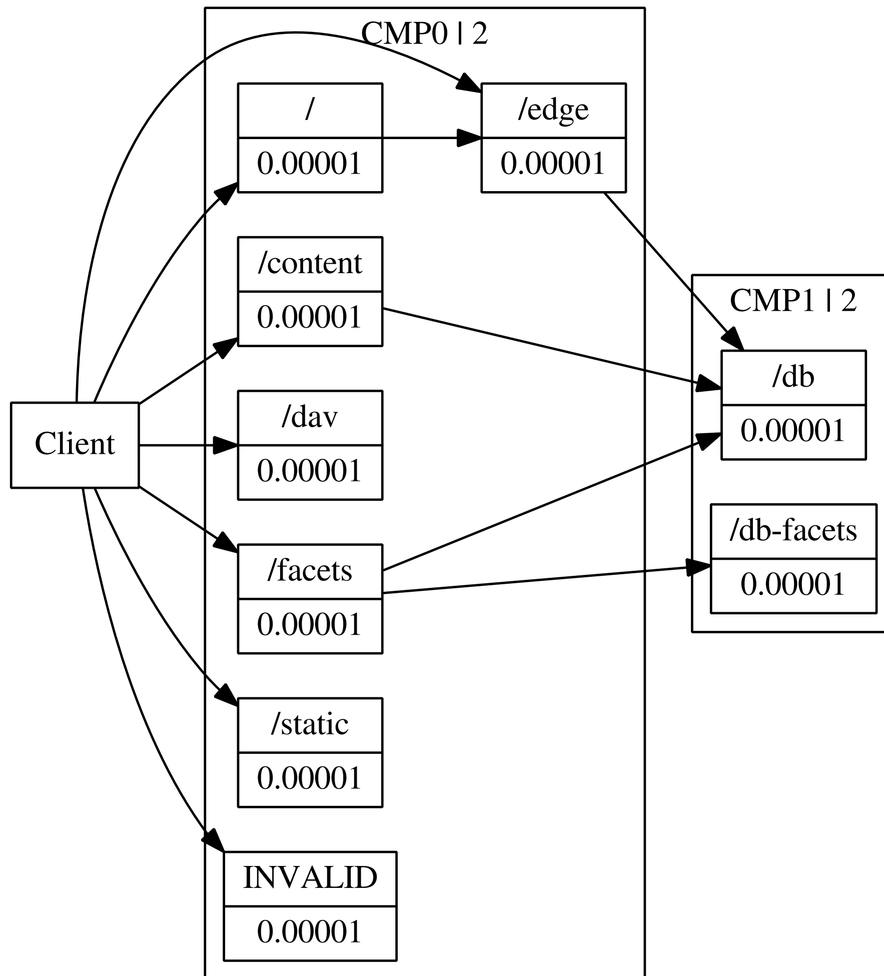

Figure 4.5: Generated EDGE 1.0 Assignment Graph

```
digraph {
    node [shape=record];
    rankdir=LR;
    subgraph cluster_Apache {
        label="CMP0 | 2";
        ""      [label="/ | 0.01",type=TimedTask];
        content [label="/content | 0.02",type=TimedTask];
        "edge"    [label="/edge | 0.02",type=TimedTask];
        facets  [label="/facets | 0.03",type=TimedTask];
        static  [label="/static | 0.02",type=TimedTask];
    }
    subgraph cluster_DB {
        label="CMP1 | 2";
        db [label="/db | 0.05", type=TimedTask];
        dbfacets [label="/db-facets | 0.07", type=TimedTask];
    }
    Client;
    Client -> "";
    Client -> content;
    Client -> "edge";
    Client -> facets;
    Client -> static;
    "" -> "edge";
    content -> db;
    "edge" -> db;
    facets -> db;
    facets -> dbfacets;
}
```

Figure 4.6: EDGE 1.0 Assignment Dot File

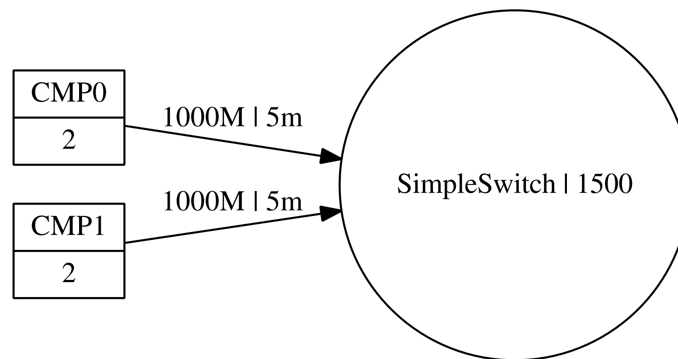

Figure 4.7: Generated EDGE 1.0 Orchestration Graph

```
digraph{
    node [shape=record];
    rankdir=LR;

    Switch [label="SimpleSwitch | 1500",shape=circle];

    Apache [label="CMP0 | 2"];
    DB     [label="CMP1 | 2"];

    Apache -> Switch [label="100M | 5m"];
    DB -> Switch [label="100M | 5m"];
}
```

Figure 4.8: EDGE 1.0 Orchestration Dot File

Markup Language) document. File paths are provided for the locations of the assignment and orchestration dot graphs, as well as the output files for simulation results (Fig. 4.9). The Results file contains a log of the completion of each client request, for verification purposes. The Latency, Throughput, and Validation files will contain the metric observations collected by DWSim. One or two databases may be used to provide client request workloads for the simulation. The ability to utilize two databases is provided mainly to eliminate the need to merge separate logs for HTTP and HTTPS traffic. Lastly the Configuration file may be used to specify the time interval between simulation steps, and the start and end timestamps for the simulation. This range of time is intended to allow inspection of multiple durations for a given workload, each potentially being assigned a different timestep for control of measurement granularity.

4.2.4 Datasources

DWSim makes use of the Apache Combined Log Format to simulate a workload for a DWS. Before these logs can be used as input, however, multiple steps must be performed. First, the logs must be combined into a single file. Second, those logs must be imported into a SQL database. Third, performance characteristics for client devices must be estimates and stored in the database. The following section describes the processes necessary to perform

```
assignment:      "./assignment.dot"
orchestration:   "./orchestration.dot"
workloadhttp:    "sqlite:///access.db"
workloadhttps:   "sqlite:///ssl_access.db"
results:         "./output/results.csv"
latency:         "./output/latency.csv"
throughput:      "./output/throughput.csv"
utilization:     "./output/utilization.csv"
timestep:        0.01
start:           "2006-01-02 15:00:00-07:00"
end:             "2006-03-05 16:30:00-07:00"
```

Figure 4.9: Example DWSim Configuration File

these steps.

Log Preprocessing Apache stores its logs into one or more files, based on a format specified by the Apache VirtualHost configuration. In order to gather the information necessary to run a simulation, the Combined log format was used to provide as much information as possible (Fig. 4.10). Additionally, the size of this log can become unmanageable over time. This led the administrator to rely on tools like Logrotate to periodically compress portions of the complete log into separate files. The first step of DWSim preprocessing involves unpacking the compressed logs. Next, these logs are combined into a single file in chronological order. The resulting file is then compressed to save disk space.

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700]
"GET /apache_pb.gif HTTP/1.0" 200 2326
"http://www.example.com/start.html"
"Mozilla/4.08 [en] (Win98; I ;Nav)"
```

Figure 4.10: Example of Combined Log Format for Apache HTTPd

Database Generation The compressed log file is then fed into `ApacheLog2DB` tool. Created specifically for DWSim, this program imports the logs into a SQL database (Fig. 4.11). It splits us each log entry into multiple pieces to consolidate duplicate information. The

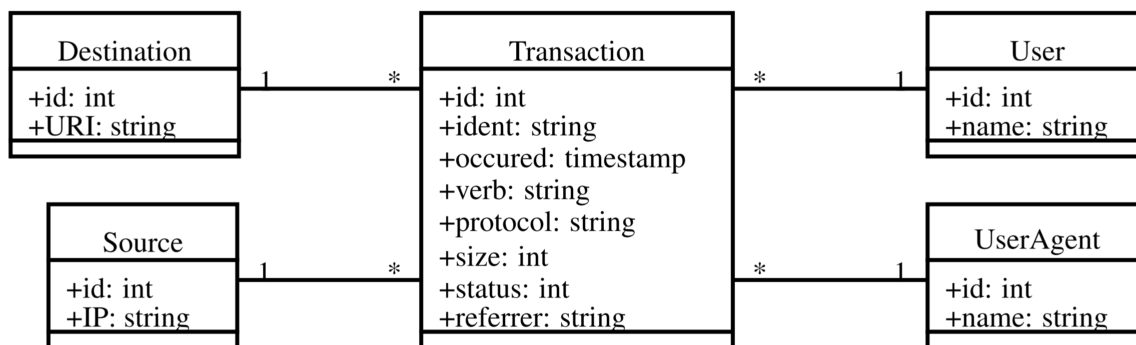

Figure 4.11: ApacheLog2DB SQL Schema

`destinations` table stores each of the URI used found in the log. The `sources` table stores each unique IP address used to access the system. The `users` table stores each of the unique usernames that were logged. The `useragents` table stores the unique User-Agent strings that were also logged. The `txns` table stores all of the transactions from the log with references to the fields found in the other tables and the remaining information from the log (*i.e.* timestamps, HTTP methods, protocol versions, *etc.*). The resulting database can later be used as the direct input for DWSim.

Network Performance With the logs successful imported, all of the unique source IP addresses have been identified. However, the logs do not store the characteristics of the network link used by each IP address. In order to recover this information, it was necessary to perform a non-invasive estimation of the bandwidth to the clients. The `ApacheLog2DB_IPStats` was written in order to measure the bandwidth and latency for each client. Estimation is performed by requesting multiple ICMP Timestamp requests to the client IP. This involved collecting 40 measurements for each packet size from 100B to 1500B in increments of 100B. A Least-Mean Squares approximation (Fig. 4.12) is used to perform a linear fit of the relationship between packet size and observed latency. While this approach originally used a linear fit to the minimum sample values and maximize estimated bandwidth [38], this work utilizes an average fit provide a more conservative approximation .The y-intercept of this line provides an estimate of the propagation delay for the connection. The

inverse of the slope of the line provides an approximation of the available bandwidth between the client and server. In cases where the client was unreachable, it was necessary to estimate these quantities based on the successful connections. This was performed by calculating the average latency and bandwidth for each unique network for the subnet masks 255.255.255.0, 255.255.0.0, and 255.0.0.0 as well as all captured values. Values were then filled in favoring the closest match to the IP address of the missing record. All of these bandwidth and latency values were then stored in a separate `ipstats` table in the log database.

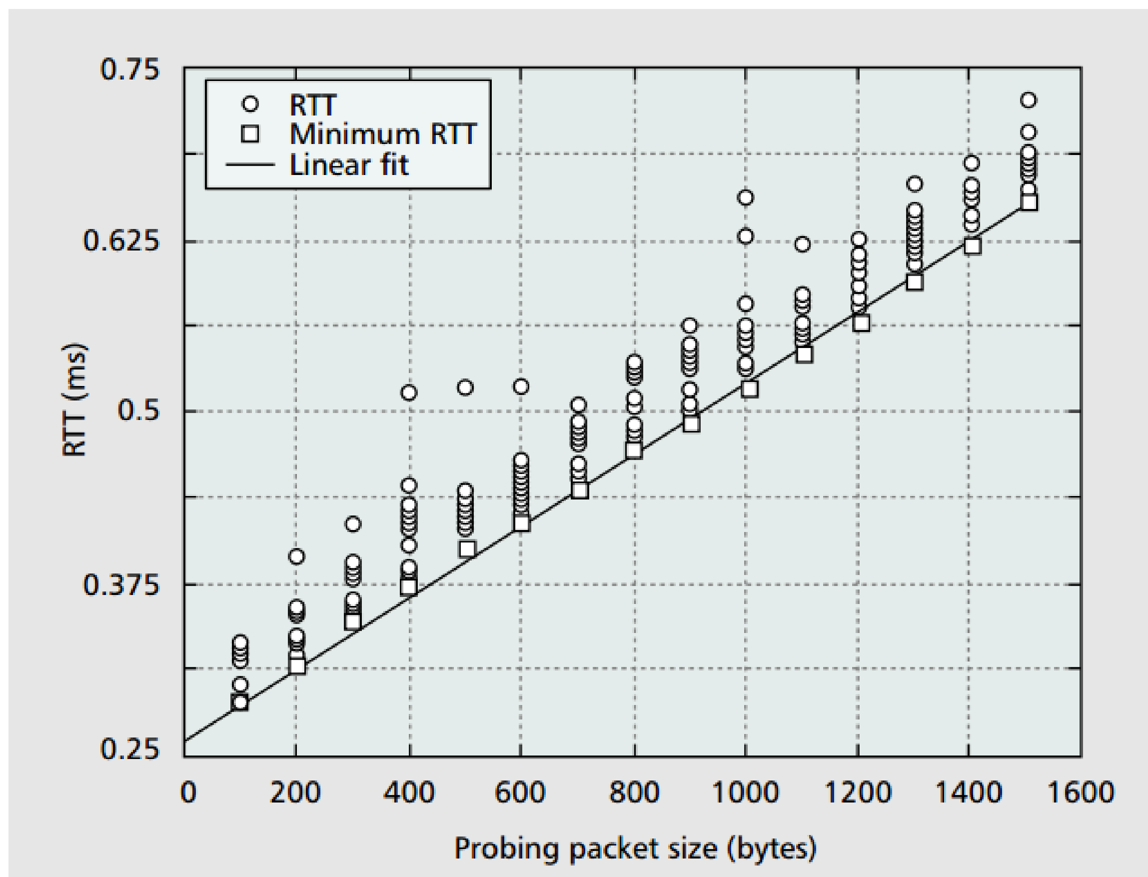

Figure 4.12: LMS Bandwidth Approximation, reproduced from [38]

4.2.5 Metrics

Multiple performance metrics can be evaluated for components of the system (Table 4.1). A Resource Unit (RU) is defined as the unit of production or consumption of a given resource for each component. The RU of a CPU could be measured in instructions, while an RU for a Network might be bytes transferred (Table 4.2). Throughput gives us the rate of production or consumption of RUs for a component. Web services exhibit a throughput which is characterized by the number of client requests handled during a discrete time interval. Utilization measures the ratio between the theoretical maximum throughput of a component and observed throughput. Efficiency characterizes a component by relating useful throughput to the total observed throughput. Latency characterizes the time from request to completion for processing of a work unit. If latency is too high, work units may no longer be useful upon completion. Extremely low latency may indicate that too many resources have been allocated and that they suffer from poor utilization. Latency is a primary concern for users, while efficiency in a primary concern for DWS managers.

Table 4.1: Proposed Metrics

Symbol	Metric	Description	Units
RU_P	Processing RU	—	$Instruction$
RU_C	Communication RU	—	$Byte$
RU_S	Memory RU	—	$Byte$
RU_E	Energy RU	—	$Joule$
τ_o	Ideal Throughput	maximum theoretical throughput of a component	RU/s
τ_g	Gross Throughput	total observed throughput of a component	RU/s
τ_n	Net Throughput	total useful throughput of a component	RU/s
ω_g	Gross Utilization	observed utilization of a component	%
ω_n	Net Utilization	total useful utilization of a component	%
η	Efficiency	ratio of useful work to observed work	%
$1-\eta$	Waste	ratio of unused work to observed work	%
λ	Instantaneous Latency	instantaneous RTT for a Request/Response	s
$\bar{\lambda}$	Mean Latency	observed average RTT for a Request/Response	s
$\tilde{\lambda}$	Median Latency	observed median RTT for a Request/Response	s
κ	Speedup	improvement in Latency for different configurations	—
γ	Sizeup	improvement in Throughput for different configurations	—
$\rho = \Delta\kappa/\Delta\gamma$	Scalability	the slope of Speedup to Sizeup	—

Other metrics enable pair-wise comparisons of component and system configurations. Speedup measures the relationship between the latency of a previous configuration to that

of a new configuration. Sizeup demonstrates the change in throughput resulting from a change in configuration. Scalability measures the ratio between a change in speedup and change in sizeup. The throughput of a DWS may increase in relation to the number of processing units. However, having too many processors may lead to an uplift in processor failure, possibly causing throughput to reduce. Similarly, having too few processors prevents throughput from meeting client demand.

Table 4.2: Example DWS Quantities

Metric	Service	Processing	Memory	Communication	Energy
RU	Request	Instruction	Byte	Byte	Joule
Throughput	Req/s	I/s	B/s	B/s	J/s (W)

Net and Gross Utilization are defined as the ratio of Net or Gross Throughput to the Ideal Throughput of a component respectively (Eqn. 4.1 & 4.2). The result is a fraction between 0 and 1.0.

$$\omega_g = \frac{\tau_g}{\tau_o} \tag{4.1}$$

$$\omega_n = \frac{\tau_n}{\tau_o} \tag{4.2}$$

Gross Utilization examines the total work performed, while Net Utilization examines the useful work performed.

Efficiency is calculated as the ratio of Net Throughput to Gross Throughput (Eqn. 4.3).

$$\eta = \frac{\tau_n}{\tau_g} \tag{4.3}$$

Mean Latency can be calculated as the average of N samples of task completion times (Eqn. 4.4).

$$\bar{\lambda} = \frac{\sum_{i=1}^{N} t_{i,f} - t_{i,s}}{N} \tag{4.4}$$

For mean user-facing latency, task completion time denotes the time from first requesting information up to the time when the information has been completely received. For mean

system-latency, task completion time denotes the time from starting a task up to the time in which the task finishes execution.

Having measures for mean latency, Speedup can be calculated as the ratio of previously observed Mean Latency to the Mean Latency observed after a system modification (Eqn. 4.5).

$$\kappa = \frac{\bar{\lambda}_{old}}{\bar{\lambda}_{new}} \tag{4.5}$$

Sizeup compares the throughput of a system, before and after a system modification (Eqn. 4.6).

$$\gamma = \frac{\tau_{o,new}}{\tau_{o,old}} \tag{4.6}$$

Scalability characterize the relationship between Latency and Throughput as the gradient of Speedup with respect to Sizeup (Eqn. 4.7).

$$\rho = \frac{\delta\kappa}{\delta\gamma} \tag{4.7}$$

Ideal Throughput may be defined as the theoretical maximum throughput of a device. However, multiple throughputs may exist for a single component (Table 4.3). It is possible for throughput to both reflect the consumption of a resource and the production of another. For example, a CPU consumes energy in order to execute instructions, which in turn consume user inputs and produce useful outputs. A network interface card (NIC) may use part of its bandwidth to receive incoming information, simultaneously using additional bandwidth to transmit information to a networked device. A server power supply converts AC electricity into DC to provide energy for the server's component parts. Table 4.3 provides examples of consumption and production throughputs for common computing devices. Throughput quantities described as infinite have no impact on the performance of the associated device in the DWSim model.

In many cases, the resources allocated to a device may not be sufficient to achieve its

Table 4.3: Example Resource Throughputs

Device	Processing (I/s)	Storage (B/s)	Communication (B/s)	Power (J/s)
CPU Worker	50	∞	∞	60
NIC	∞	∞	$10M$	10
Switch	∞	∞	$100M$	200
Harddrive	∞	$100M$	∞	15

ideal throughput. A CPU may need to reduce clockspeed, sacrificing instruction throughput, to lower power consumption, while increasing user latency. A harddrive may "spin down" to reduce idle power consumption, but this will also lead to higher latency for data transactions when returning to full power. The limited bandwidth of a network link or storage device may prevent a computational process from executing at full speed by introducing access delays. High ambient temperature may require a cooling system to reduce its thermal delta in order to avoid using large amounts of power during grid peak hours.

Instrumentation Each component within DWSim is capable of providing metric values for utilization, latency, and throughput. A uniform interface requires that each component implement three functions: `Util()`, `Latency()`, and `Throughput()`. The values for each metric are returned as an array, regardless of whether the device is capable of performing more than one action at a time. This uniformity simplifies the programming necessary to collect metric quantities from multiple devices.

Collection Each component within DWSim is capable of providing metric values for its sub-components. A CPU will collect these values for each of its cores. A Network Link will collect the values for its uplink and downlink. A Switch will collect these values for all of the links connected to it. Clients also provide insight into these metrics. The main DWSim component collects all of these values together and provides an interface for the main executable and for use in other libraries. These metrics may be accessed as raw numerical quantities at runtime or exported to CSV files for later review.

Representation Utilization is provided as a continuous quantity, and is not contingent on the assignment of work. A value between zero and one is used to indicate the percent utilization of a device relative to its ideal capacity. Throughput and latency are only reported as non-zero values when a related task is completed (*i.e.* completion of a task, reception of a packet, completion of a request). Throughput may take on a positive value greater than one to indicate the total work performed by the device. Latency is represented as a positive value signifying the time required to complete an action. Latency and throughput are always non-zero at the same time.

4.3 Functional Testing

Testing of the DWSim program was performed in three stages. In the first stage, Unit Testing of individual functions was performed to establish their correct functionality. In the second stage, the processing and networking models were tested separately against the expected performance for a real system. In the third stage, the processing and network were integrated and tested against a trivial workload. This section discusses each of these stages in depth.

4.3.1 Unit Testing

Unit Testing is the practice of evaluating the functionality of individual functions within a piece of software. This consists of devising a test which adequately explores each of the possible behaviors for a particular function, writing the software to execute this test, and then running the tests to verify correct functionality. Additionally, more sophisticated testing environments monitor the code during a test in order to determine which lines of code were evaluated. Known as a “coverage report”, this feedback allows a developer to adapt existing tests to address the conditions of execution that were not already tested.

In Go, the standard libraries possess a simple, but effective, set of functionality for handling these tests. A developer is able to create test cases by writing functions that follow

a particular naming scheme and interface. Functions must begin with the word `Test` and accept the parameter `t *testing.T` . The Go toolchain also possesses tools for automating the execution of these tests and reporting the effective Coverage. The `go test` tool searches for all test function in a given directory and executes them, reporting any failed assertions. The `-cover` flag can also be added in order to enable coverage reporting for the evaluated tests. In addition, the Go plugin for JetBrains WebStorm provides graphical feedback of these coverage results in the code editor.

Best effort was made to achieve full coverage for the DWSim code base. A few code sections were not testable as their usage of the `panic()` function does not allow the testing environment to continue execution. Table 4.4 provides a summary of the coverage reported when unit testing was performed for DWSim.

Table 4.4: Unit Testing Results for DWSim

Package	**Coverage**
kgcoe-git.rit.edu/btmeme/DWSim/config	86.5%
kgcoe-git.rit.edu/btmeme/DWSim/environment	95.8%
kgcoe-git.rit.edu/btmeme/DWSim/net	99.3%
kgcoe-git.rit.edu/btmeme/DWSim/processing/cpu	94.1%
kgcoe-git.rit.edu/btmeme/DWSim/processing/tasks	100.0%
Overall	95.0%

4.3.2 Component Verification

Processing Model The preliminary processing model was verified for an exemplar workload (Fig. 4.13). The workload consisted of 14 requests, each resulting in the execution of a single `TimedTask` with a cost of 5s. A CMP was instantiated and configured to have one to eight cores, in powers of two. Execution of the workload was performed for each of the numbers of cores, calculating the values of the following metrics: Latency, Throughput, Utilization, Efficiency, and Speedup. Figure 4.14 provides an example of the expected utilization for a quad-core CMP, denoting the separate tasks by color.

Latency was calculated according to the time necessary to complete the entire workload,

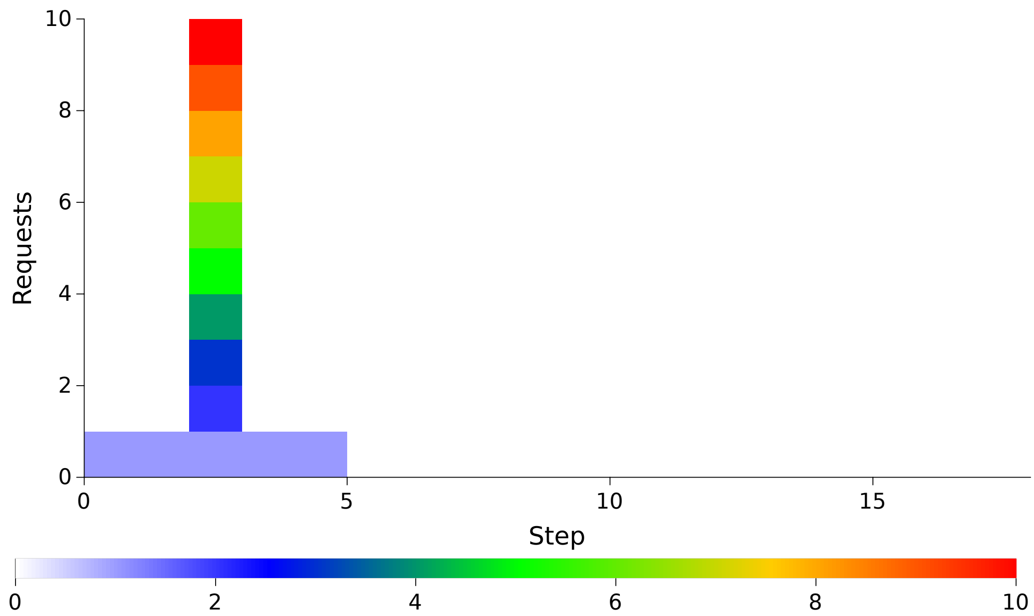

Figure 4.13: Exemplar Workload for CMP, colors indicate task

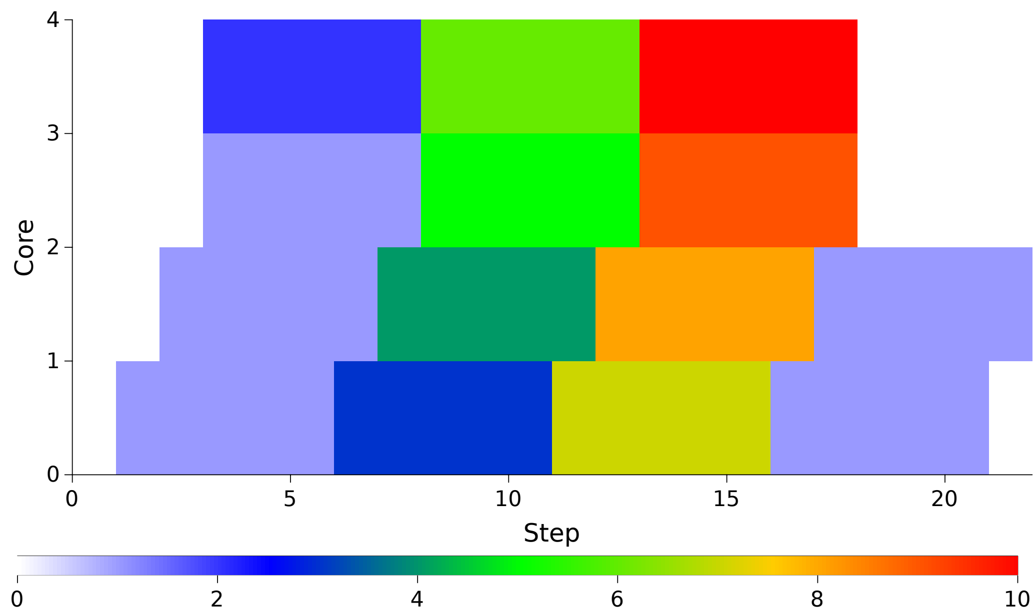

Figure 4.14: Exemplar Utilization for CPU, colors indicate task

and throughput was calculated according to the number of requests completed per second (Table 4.5). For each of the CMP configurations, latency was found to be consistent with real-world parallel execution.

Two sets of utilization metrics were captured. First, time-dependent measurements of utilization were captured for each of the CMP configurations. This allowed for verification of static scheduling, by demonstrating the change in utilization as tasks were assigned to cores. The single core results show execution for only one core at an given time (Fig. 4.15). Results for the 2, 4, and 8-core configurations indicate the correct scheduling of multiple cores, by leveraging as many cores as possible at any given time (Figs. 4.16, 4.17, 4.18). Second, average utilization was calculated for each of the configurations. Net and gross utilization were identical, as this test assumes zero overheads. Overall, average utilization was shown to decrease in response to an increase in CMP core counts (Table 4.5). This is consistent with the underutilized cores when work is not available to be executed.

Table 4.5: Metric Summary

Cores	λ	τ_o	τ_g	τ_n	ω_g	ω_n	η
1	70	0.200	0.200	0.200	1.000	1.000	1.0
2	36	0.400	0.389	0.389	0.973	0.973	1.0
4	21	0.800	0.667	0.667	0.833	0.833	1.0
8	12	1.600	1.167	1.167	0.729	0.729	1.0

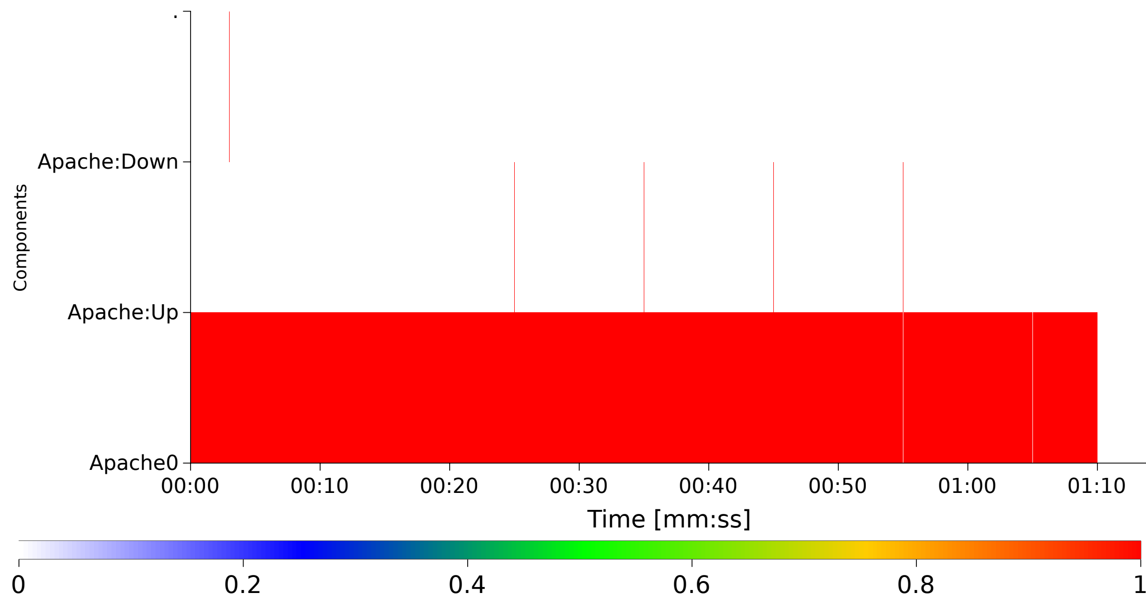

Figure 4.15: Processor Utilization (ω_g): 1 Core

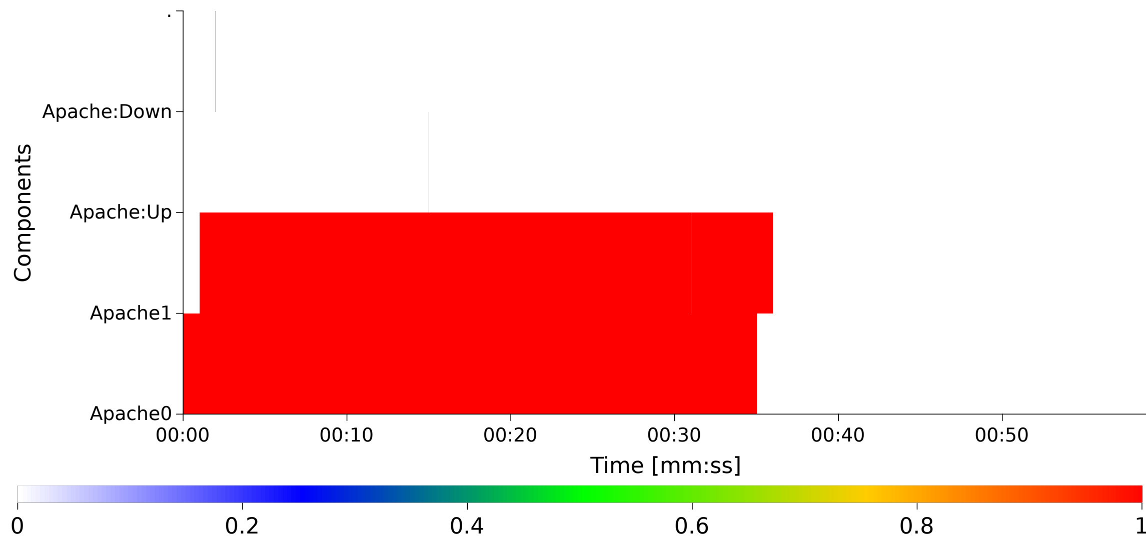

Figure 4.16: Processor Utilization (ω_g): 2 Cores

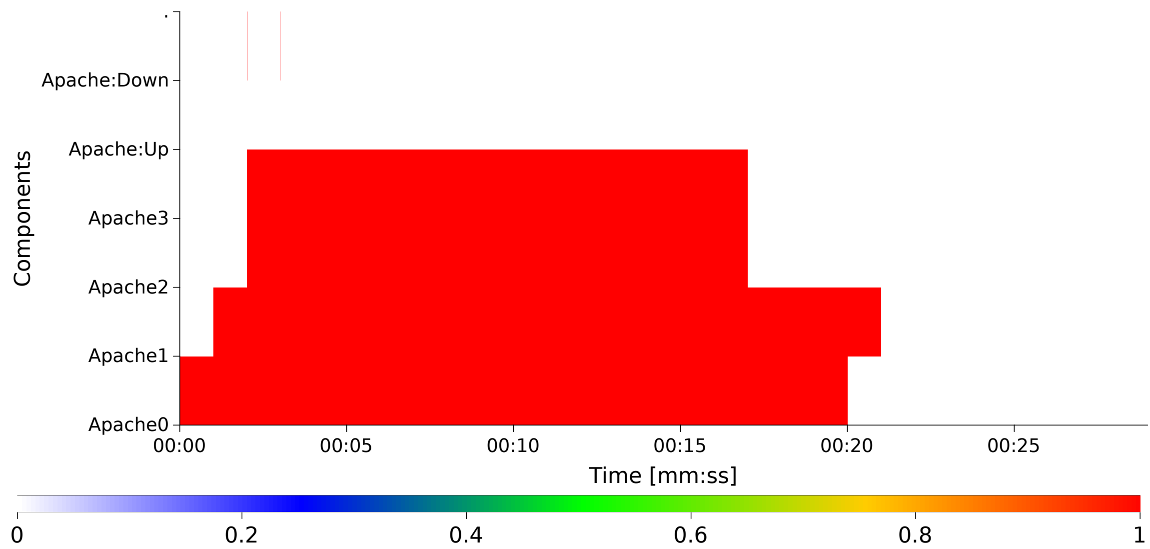

Figure 4.17: Processor Utilization (ω_g): 4 Cores

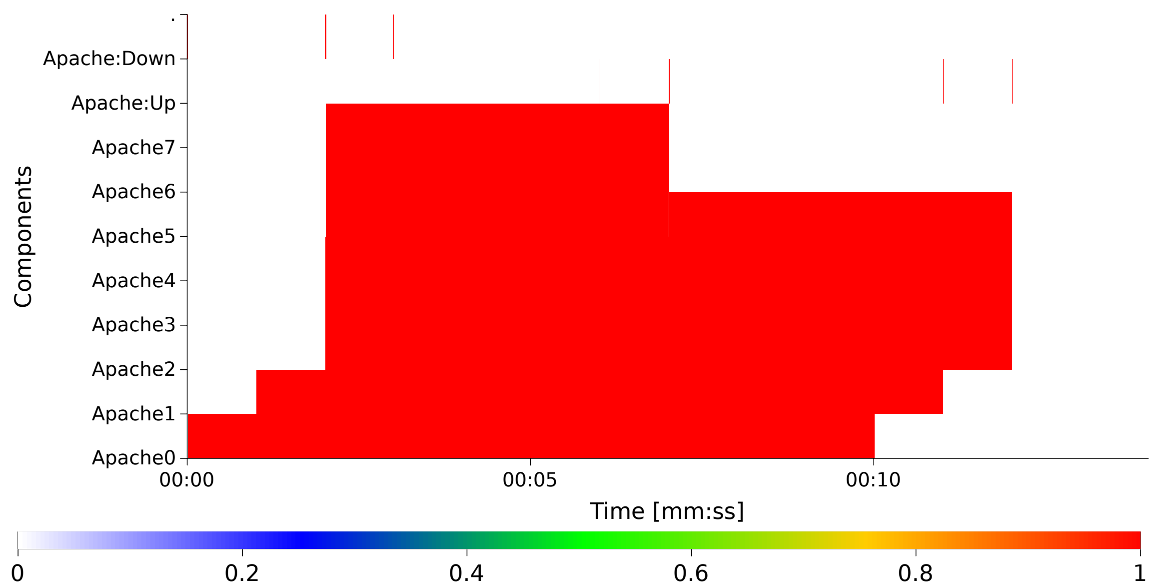

Figure 4.18: Processor Utilization (ω_g): 8 Cores

Accurate scaling of processor cores was further reinforced by the results of the speedup calculations. Figure 4.19 compares ideal speedup with the speedup exhibited by the model. Speedup was found to be consistent with the expected performance given by Amdahl's law (Eqn. 4.8) where F is the fraction of time spent executing the parallel task group, and K is the number of cores of the configured CMP [4].

$$Speedup = \frac{1}{(1 - F) + F/K} \tag{4.8}$$

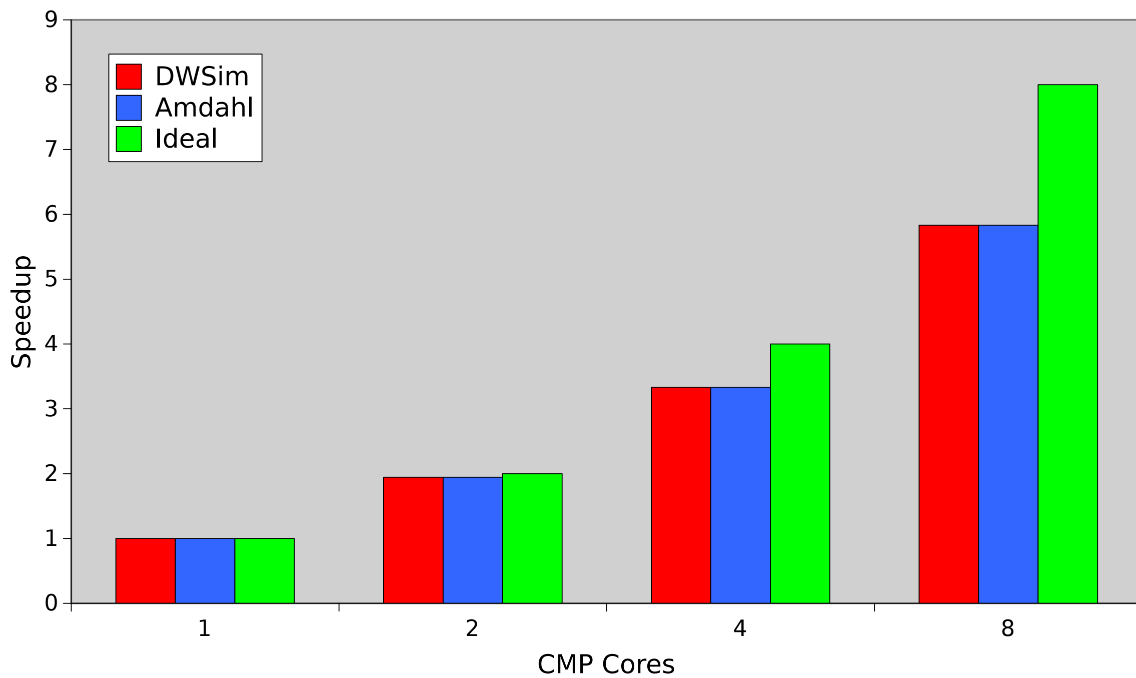

Figure 4.19: Processing Speedup

4.3.3 Simulator Integration Testing

After verifying the basic functionality of the CPU model, a small DWSim simulation was performed. This simulation utilized a single client to transmit a single request and wait for the response (Fig. 4.20). This request would require the execution of timed computation and a REST request to a database server. Through this simulation, every component of the dataflow model is utilized and the entire DWSim can be verified as functioning correctly.

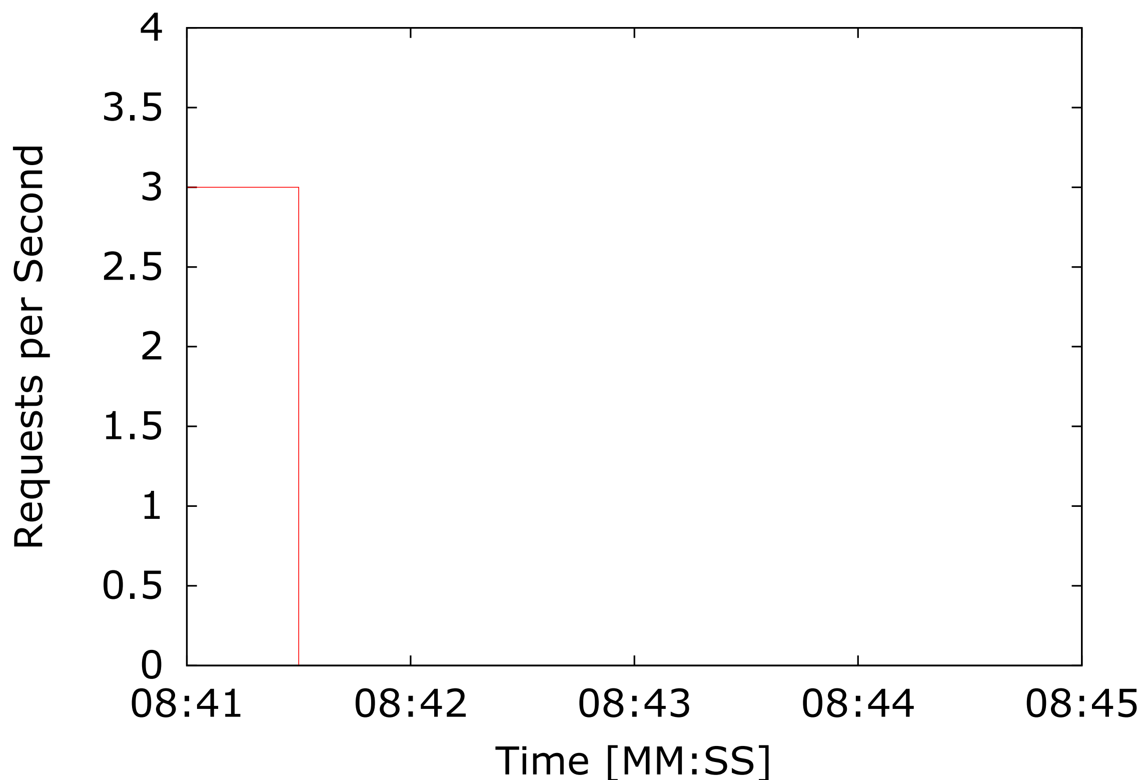

Figure 4.20: Exemplar Client Workload for Test, $\tau_n[Requests/s], \Delta t = 1[s]$, [08:41 to 08:45]

Figure 4.21 shows an annotated form of the metric results created by DWSim. A heatmap is used to show the metric values for each component over time. Network and CPU components are grouped according by processor in order to improve readability of the figure.

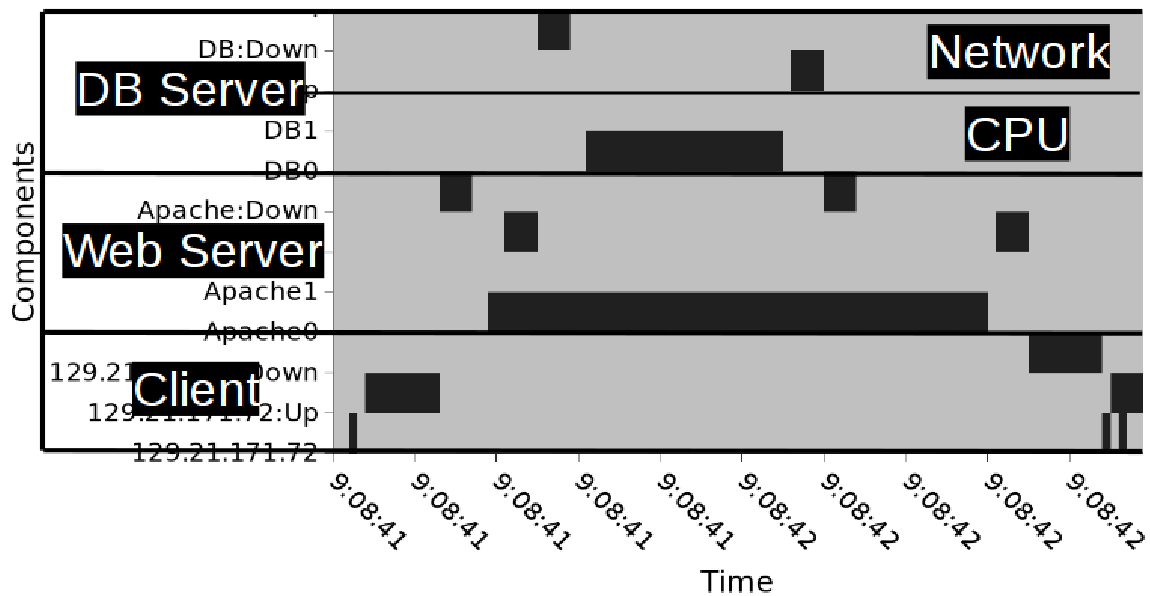

Figure 4.21: Annotated Heatmap Component Utilization, $\omega[Binary/s], \Delta t = 10[ms]$

Figure 4.22 provides a graphical representation of the utilization of each component that was instantiated for the simulation. First, the client (129.21.171.72) reads a request from the workload database and begins transmission to the Apache server. The Apache server receives the request from the client and then proceeds to transmit a request to the DB server. The DB server receives this request, performs computation, and transmits a response to the Apache server. The Apache server receives the response, performs timed computation, and then transmits the response to the client. Lastly, the client receives the response and begins the process over again with a new request. Figures 4.23 and 4.24 provide a graphical representation of the resulting latency and throughput, respectively.

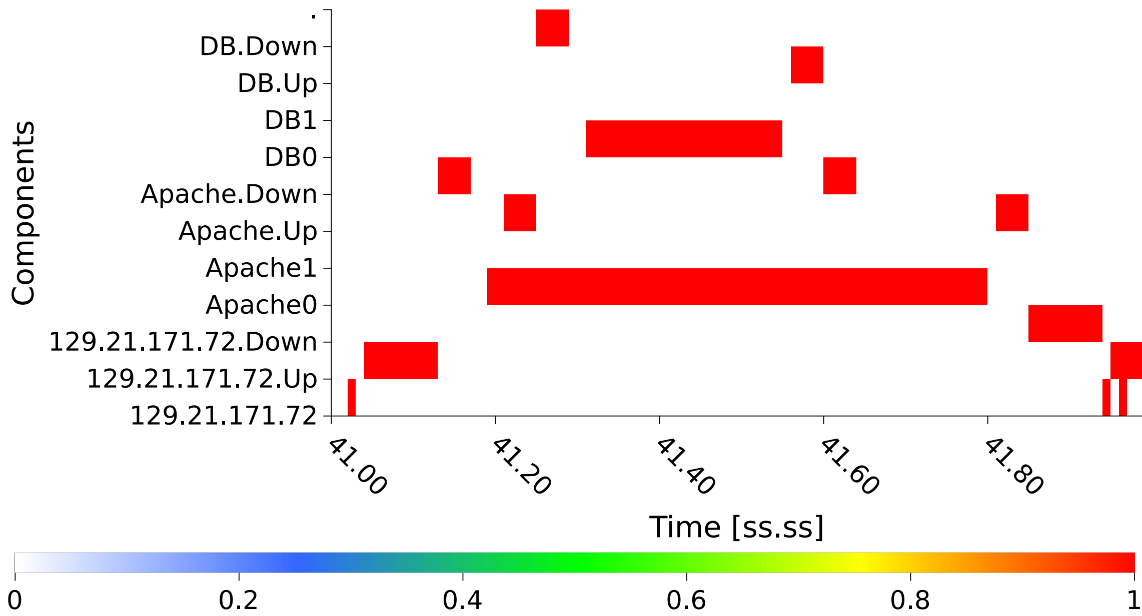

Figure 4.22: Heatmap of Component Utilization, $\omega[Binary/s]\Delta t = 10[ms]$

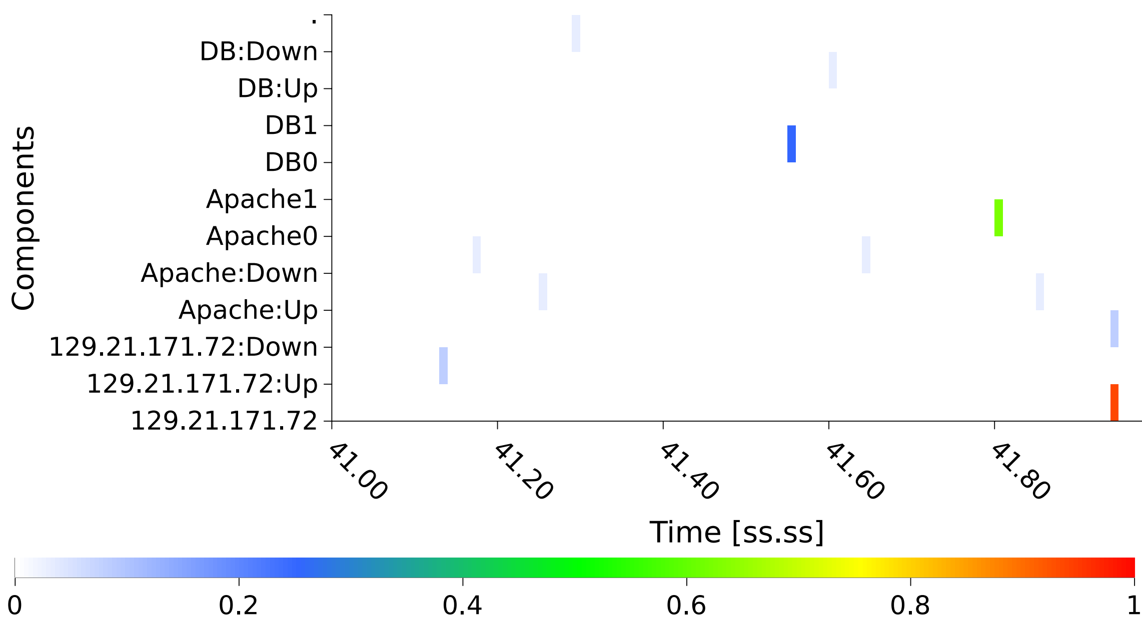

Figure 4.23: Heatmap of Component Latency, $\lambda[s]$, $\Delta t = 10[ms]$

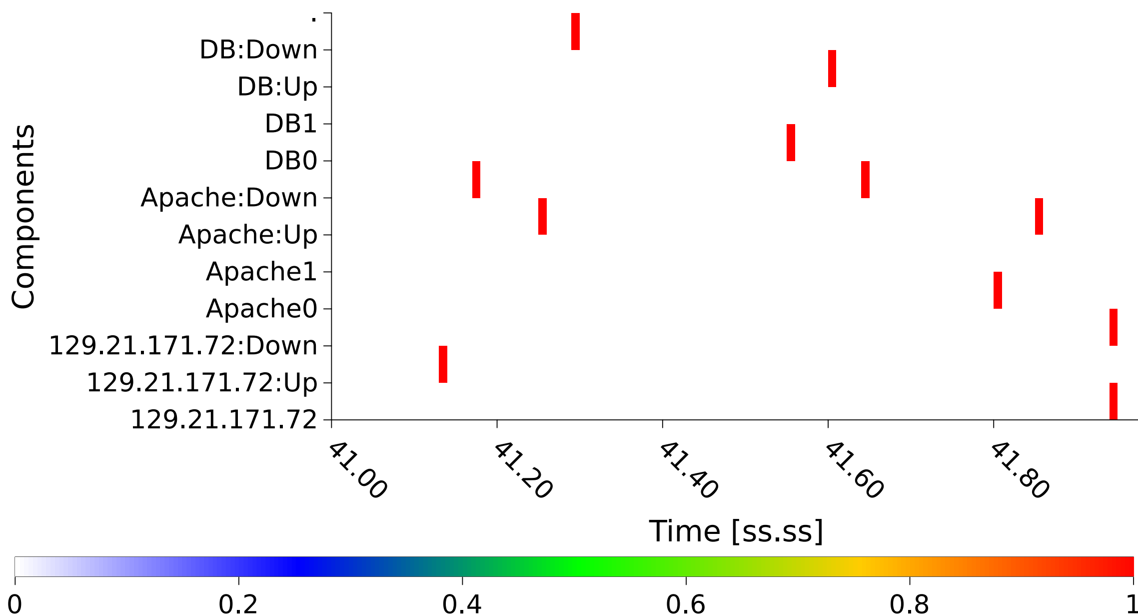

Figure 4.24: Heatmap of Component Throughput, $\tau_n[Binary/10ms]$, $\Delta t = 10[ms]$

The fine-grained detail provided by the raw form of the metric data is useful for the purposes of verification. However, a more coarse grained figure may be useful for identifying bottlenecks, poor load distribution, or under-utilized resources. By re-sampling the raw data to produce a lower resolution "overview" of a DWS, these features are more readily visible. Figure 4.25 demonstrates a re-sampling of the utilization results in Figure 4.22 to a period of $100ms$ representing the average of 10 samples

Closing Unit testing of the DWSim source demonstrated correct functionality of individual functions. Verification of the processing and networking models individually yielded the expected behavior of real-world components. Simulation of an exemplar workload produced behavior and performance consistent with an actual DWS, within the limits of the assumptions and limitations of the dataflow model and DWSim application.

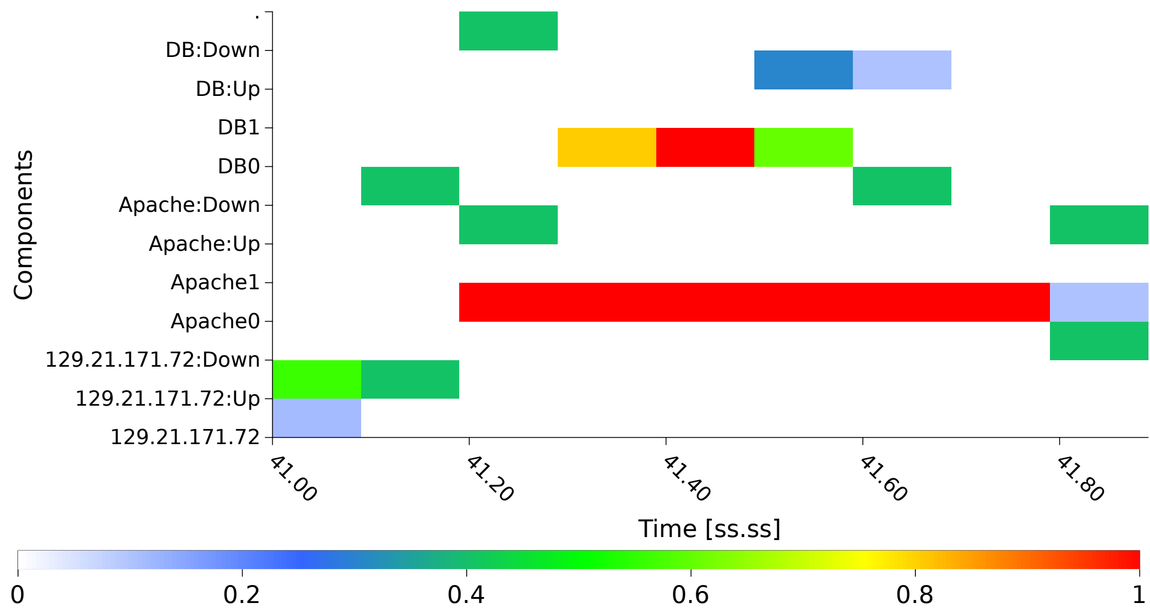

Figure 4.25: Heatmap of Average Component Utilization, $\bar{\omega}[-/100ms], \Delta t = 10[ms]$

Chapter 5

Dataflow Model: Verification Results

5.1 Workload Characterization

The workload for a DWS may consist of hundreds of thousands of client requests over an annual period. Analysis was performed for the characterization of the client requests for production EDGE systems. Characterization of client connections and observed throughput for the KGCOE-Research EDGE server. The following sections discuss the DWSim results of statistical and graphical analysis of the workloads used in simulation.

5.1.1 EDGE 1.0 Request Characterization

The following section characterizes the requests made to the production EDGE 1.0 environments at the Rochester Institute of Technology. Statistical analysis was performed for both the HTTP Methods and REST Endpoints involved in the requests received over a single academic year, beginning in August of 2015 and ending in June of 2016.

HTTP Methods HTTP Methods are a useful descriptor of the interactions at users have with a web system. Methods like `GET` are focused on the consumption of web content, while other methods focus on creating and modifying content (*e.g.* PUT, POST). Web indexing services may make use of methods like `OPTIONS` and `HEAD` to guide the process of collecting metadata. Content authors may also make use of WebDAV oriented methods when using software other than a web browser (*e.g.* TortoiseSVN). Figure 5.1 provides a distribution of these HTTP methods for the EDGE servers. For both HTTP and HTTPS traffic, `GET` is the most frequently used HTTP Method. This may either indicate that a

large percentage of client requests are focused on consumption (HTTP) or the frequent use of WebDAV clients to view and modify content (HTTPS). The high percentage of requests involving `OPTIONS`, `PUT`, and unlisted methods for HTTPS traffic, serves to demonstrate that a fair number of users make use of a WebDAV client.

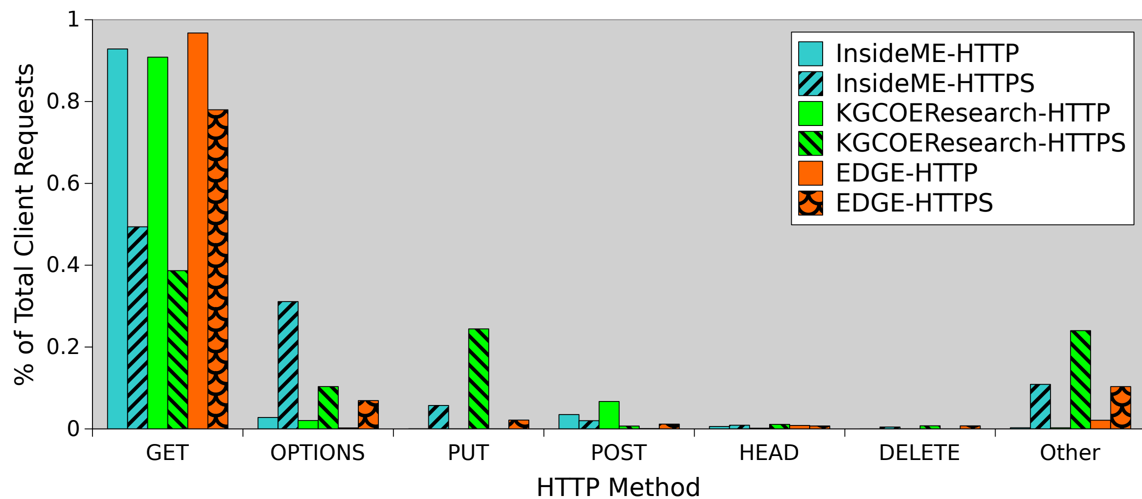

Figure 5.1: Distribution of HTTP Methods for EDGE Hosts for 2015-08-01 through 2016-06-01

REST Endpoints Client accesses to REST endpoints provide insight into which features of a DWS are used the most often and may identify which functionality is no longer worth supporting. With EDGE, REST endpoints are identified as the first "level" of a URI. Figure 5.2 shows the frequency of accesses to endpoints for each of the EDGE production systems. Overall, there are few requests to "/" which is consistent with the number of accesses to "/edge" where it redirects. The majority of non-secure HTTP requests for the EDGE systems are for "/edge" URI. This is expected as most documents are rendered to this view for public consumption. HTTPS requests for "edge.rit.edu" are frequently for this URI as project teams use this to share documents and collaborate. The "/content" endpoint serves as the secondary rendering mechanism and also experiences significant numbers of requests. The "/static" endpoint provides assets such as CSS stylesheets, JavaScript source,

and EDGE-specific logos and images. Since these files do not change often, they are very likely to be cached by a client program. The "/dav" endpoint is only accessed through HTTPS requests as it also requires user authentication. Both of the InsideME and KGCOE Research servers see the largest amount of traffic for "/dav". This is consistent with WebDAV clients being used as the primary request source for both of these servers. Changes to version-controlled documents are primarily performed through the "/dav" endpoint.

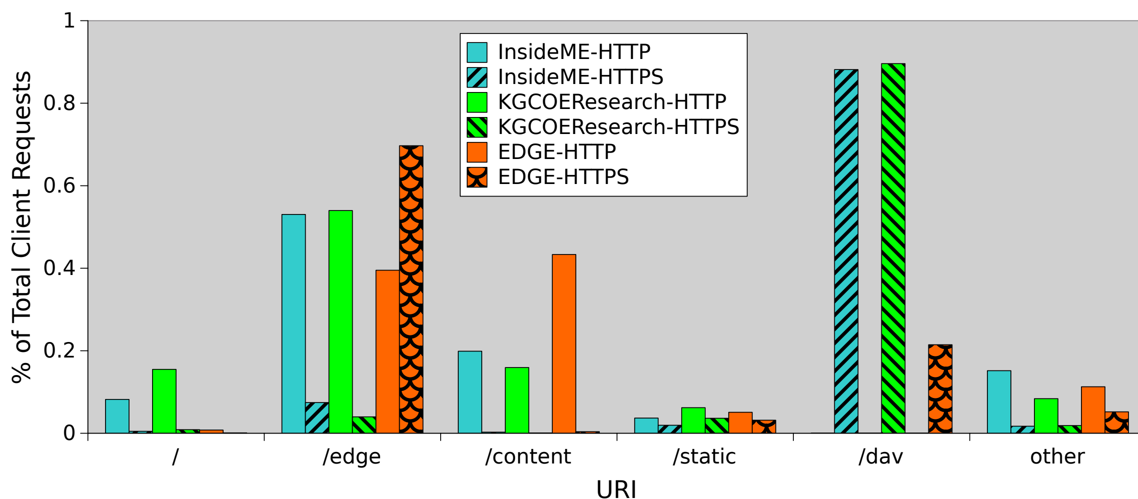

Figure 5.2: Distribution of URI Requests for EDGE Hosts for 2015-08-01 through 2016-06-01

5.1.2 KGCOE-Research Client Characterization

The following section characterizes the clients which accessed the "kgcoe-research.rit.edu" EDGE environment. Geolocation data was used to better understand the origin of client requests. The results of bandwidth and latency estimation are also presented.

Geolocation Many analytics tools for web traffic make use of geolocation data to map where client requests are coming from. This information may be used to focus efforts for content translation or to guide the topic selection for future content. In the case of EDGE geolocation data provides insight into the kinds of audience the content might have. When examining the traffic for "kgcoe-research", it is apparent that clients are largely located in

the Eastern United States and Western Europe (Fig. 5.3). However, some visitors are from as far away as Japan, South Africa, or Russia.

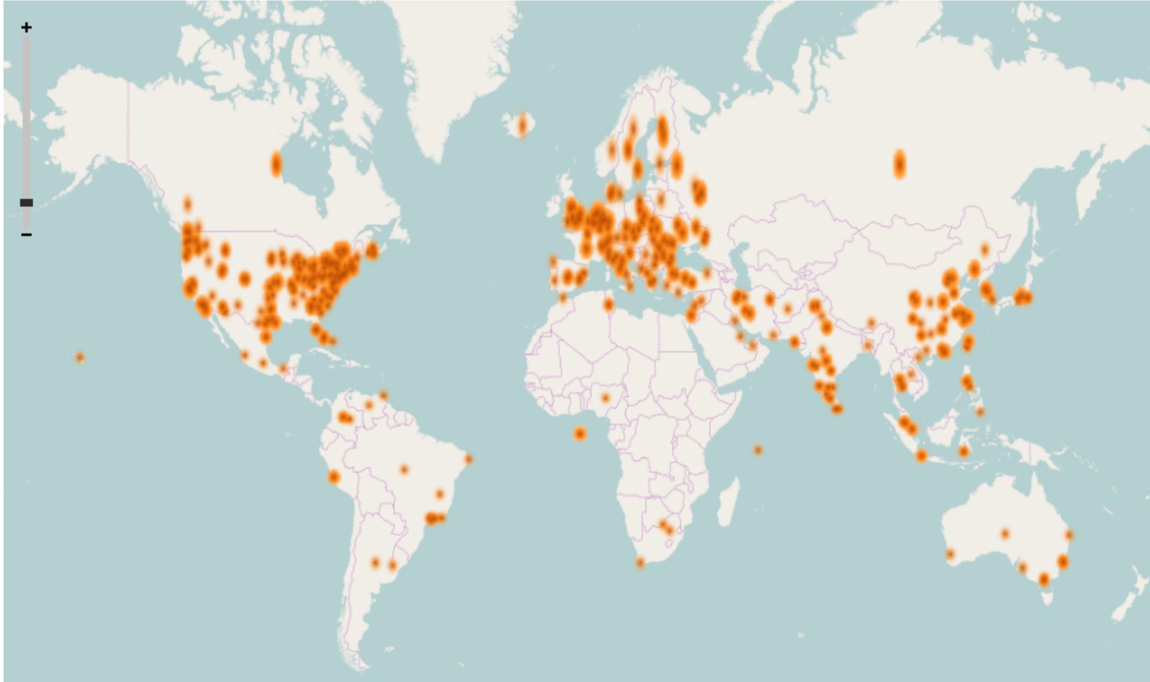

Figure 5.3: Map of IP Sources for kgcoe-research.rit.edu [2015-08-01 to 2016-01-07]

Link Characteristics The properties of the network connections used by clients may be used plan the deployment of a DWS. Measurement of client bandwidth allows an administrator to set the level of data compression and content quality for a given DWS deployment. Low speed connections benefit from higher compression and smaller file sizes, while high speed connections allow for improved quality and the opportunity to reduce compression overheads. For "kgcoe-research", the majority of the clients surveyed fell into two categories. Low-bandwidth clients were shown to have connection speeds of less than $10Mbps$, whereas high-speed clients generally fell in the 50 to $60Mbps$ range (Fig. 5.4). Client latency may indicate a need to move servers geographically closer or the use of high latency internet connections (*e.g.* satellite). From an administrative standpoint, connection latency

serves to guide the decision of how to reduce the latency of handling requests. For "kgcoe-research" the majority of clients experienced a round-trip latency of $150ms$ or less (Fig. 5.5). With such low latency it may be advisable to delay the handling of some requests in order to keep average latency low or to reduce the power consumption of servers.

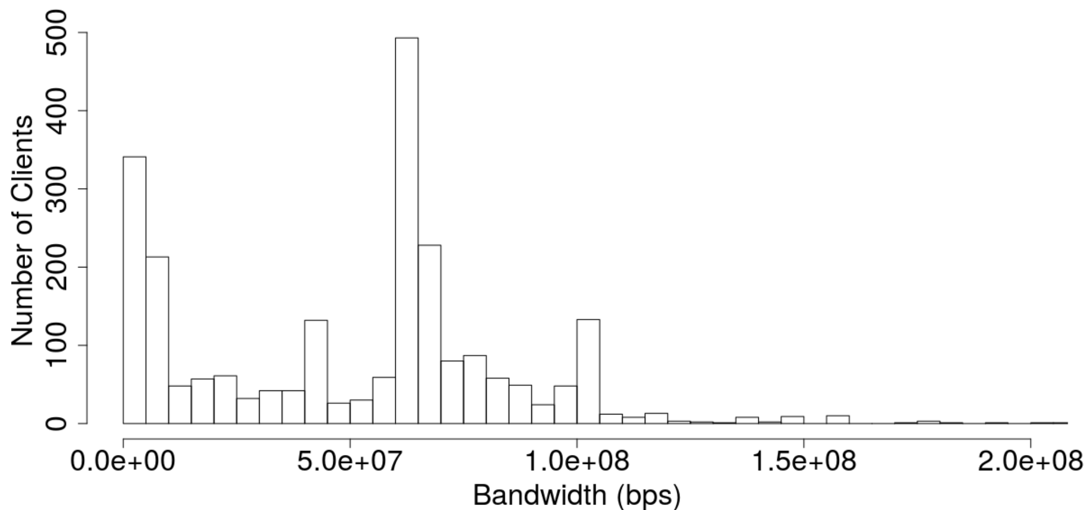

Figure 5.4: Client Bandwidth Distribution ($\bar{\tau_g}$) for kgcoe-research.rit.edu [2015-08-01 to 2016-01-07]

5.1.3 KGCOE-Research Workload Timelines

Log capture for "kgcoe-research" resulted in the collection of over 100 thousand client requests during a 5 month period beginning in August 2015 and ending in January 2016, before the migration to EDGE 2.0. Simulation periods were chosen based on graphical analysis of request timelines for multiple time-scales. Figure 5.6 shows the aggregate requests per week for the entire data capture period. The large spike in requests during October 2015 presented the possibility for studying the behavior of the dataflow model under the conditions of high load. Further investigation identified October 14th as a day of particularly high load. Figure 5.8 reveals a period of increased activity between 12:00 on the previous day and 12:00 hours, with a peak in activity between 00:00 and 01:00 hours.

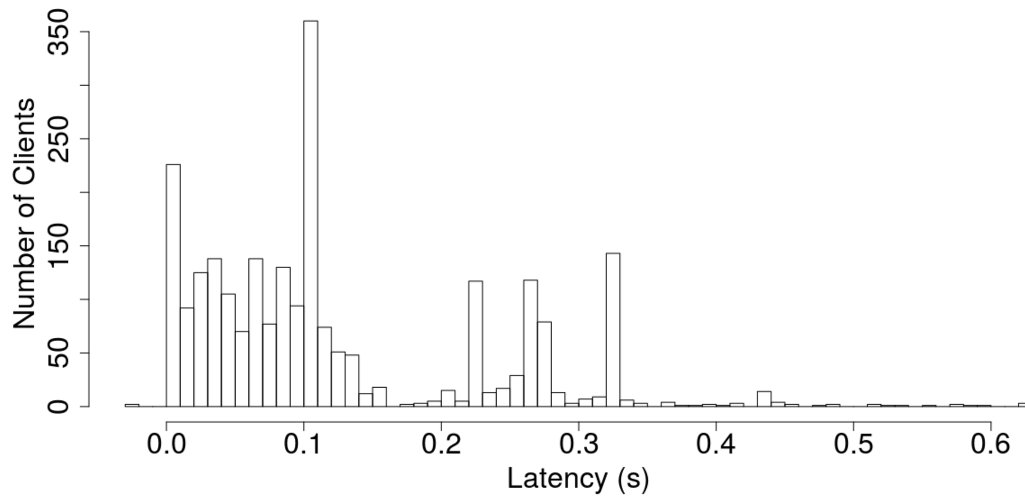

Figure 5.5: Client Latency Distribution ($\bar{\lambda_g}$) for kgcoe-research.rit.edu [2015-08-01 to 2016-01-07]

Further investigation of 00:00 to 01:00 hours revealed the presence of two major spikes of activity centered around 00:37 and 00:49 hours (Fig. 5.9). The period of time between 00:40 and 00:45 hours was selected as the simulation period to illustrate DWSim operation and results. The timestep of each simulation is specified in the caption of each figure as Δt, for example: $\Delta t = 10[\mu s]$.

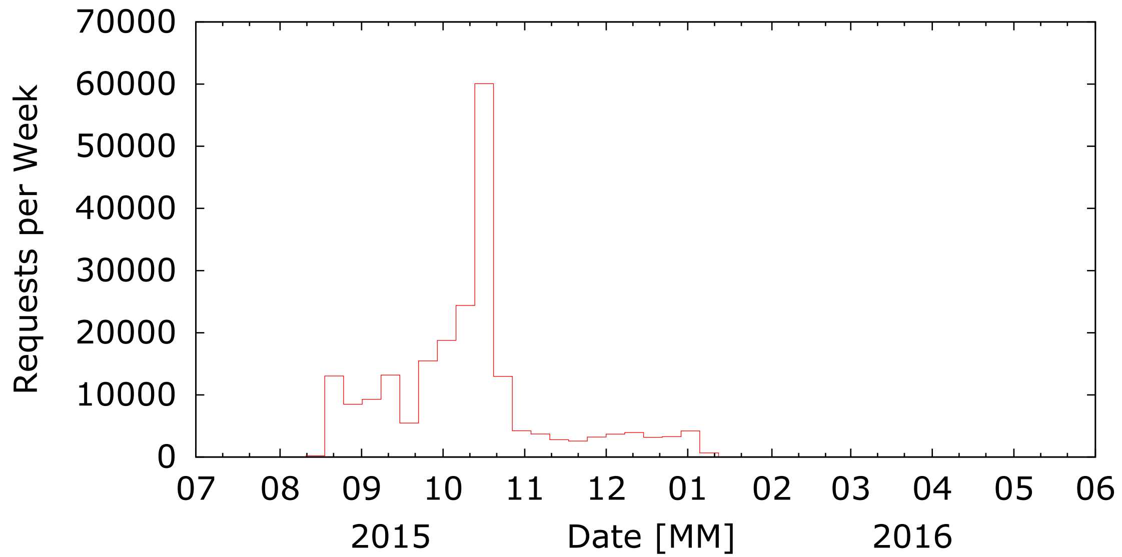

Figure 5.6: Apache Request Throughput for kgcoe-research, $\tau_n[requests/week], \Delta t = 10[\mu s]$, [2015-08-01 to 2016-06-01]

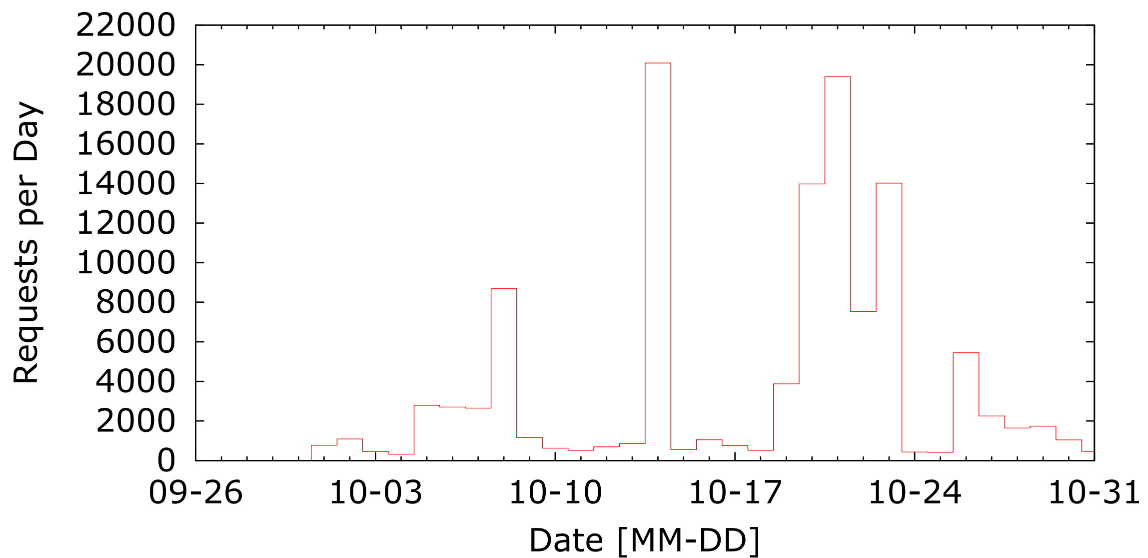

Figure 5.7: Apache Request Throughput for kgcoe-research, $\tau_n[requests/day], \Delta t = 10[\mu s]$, [2015-10-01 to 2015-11-01]

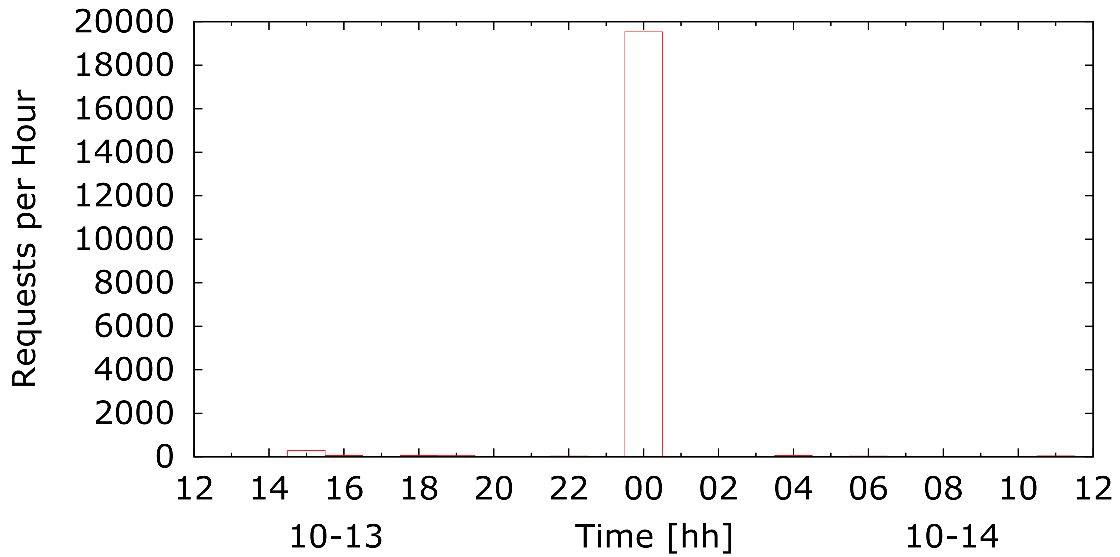

Figure 5.8: Apache Request Throughput for kgcoe-research, $\tau_n[requests/hr], \Delta t = 10[\mu s]$, [2015-10-13 12:00 to 2015-10-14 12:00]

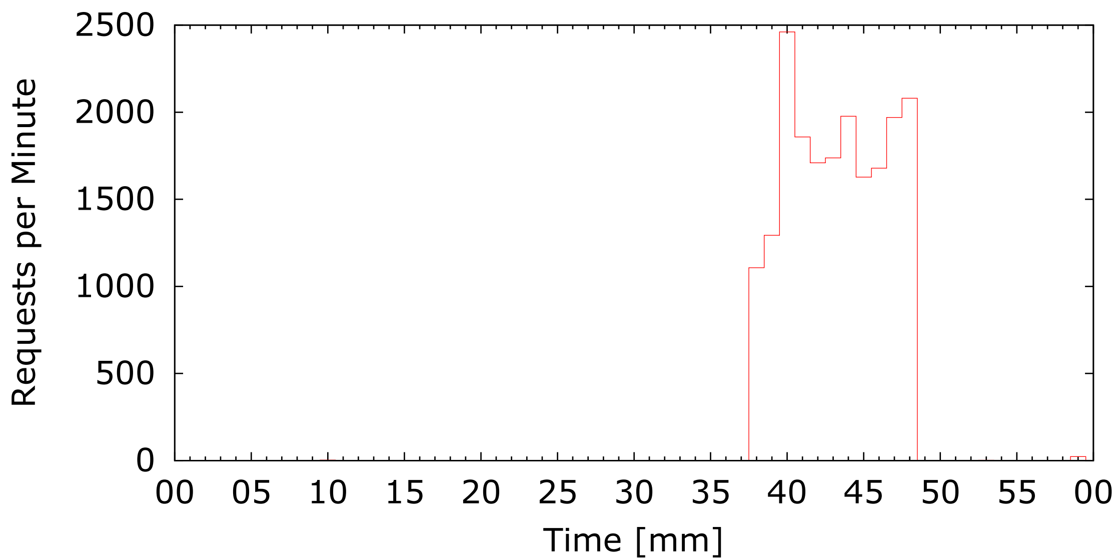

Figure 5.9: Apache Request Throughput for kgcoe-research, $\tau_n[requests/min], \Delta t = 10[\mu s]$, [00:00:00 to 00:01:00] on 2015-10-14

5.2 Dataflow Simulation Results

A simulation was performed for "kgcoe-research" over a five minute time interval from 2015-10-14 00:40:00 to 2015-10-14 00:45:00, with a $10\mu s$ timestep. During this period of time, a total of 9743 Apache requests were logged for the production system. The recorded throughput for this time period has been graphically represented in Figure 5.10. Throughput for the server sustained an average of $30 - 35$ requests per second, with intermittent periods of much higher activity.

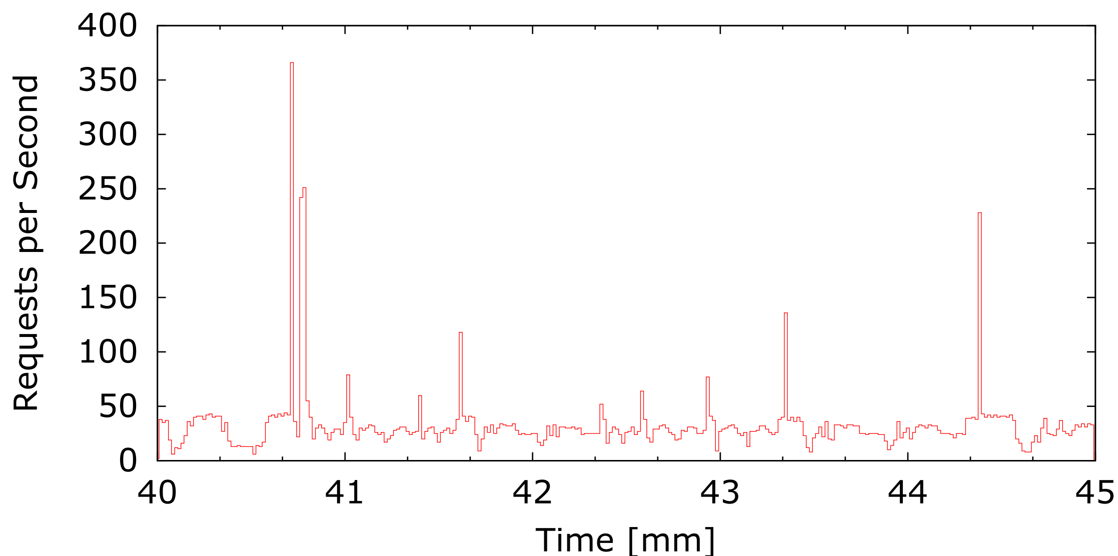

Figure 5.10: Apache Log Request Throughput for kgcoe-research, $\tau_n[request/s]$, [00:00:40 to 00:00:45] on 2015-10-14

Figure 5.11 shows a heatmap of the per component utilization as measured during simulation. The Apache HTTPd server experienced the highest levels of activity, while the DB server demonstrated much lower activity.

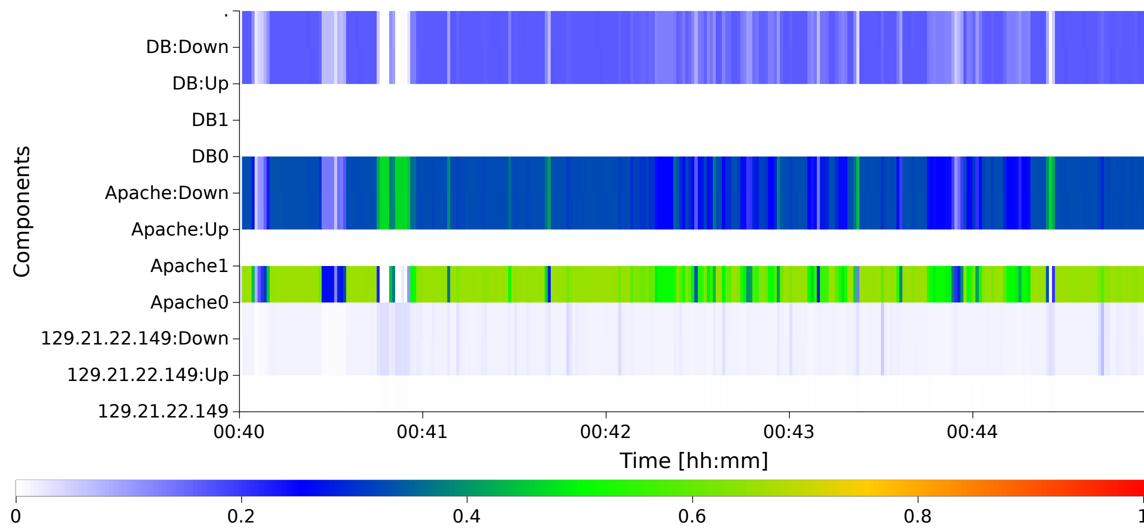

Figure 5.11: DWSim Mean Utilization Heatmap for kgcoe-research, $\bar{\omega}_n[\%/s], \Delta t = 10[\mu s]$, [00:00:40 to 00:00:45] on 2015-10-14

Figure 5.12 plots the Client request throughput observed through the Apache logs against the simulation throughput. The simulation throughput appears to be not only more stable than that of a production system, but fails to account for all of the requests made during this period. Further investigation of this trend revealed that only 9691 of the 9743 requests were actually handled during this period of time. By plotting the cumulative requests over time, it was discovered that the simulation throughput appears to lag the observed log data (Fig. 5.13). While this trend is partly caused by the constant execution time assumption, a more likely explanation would be the use of the Apache logged completion time as the transmission time for clients.

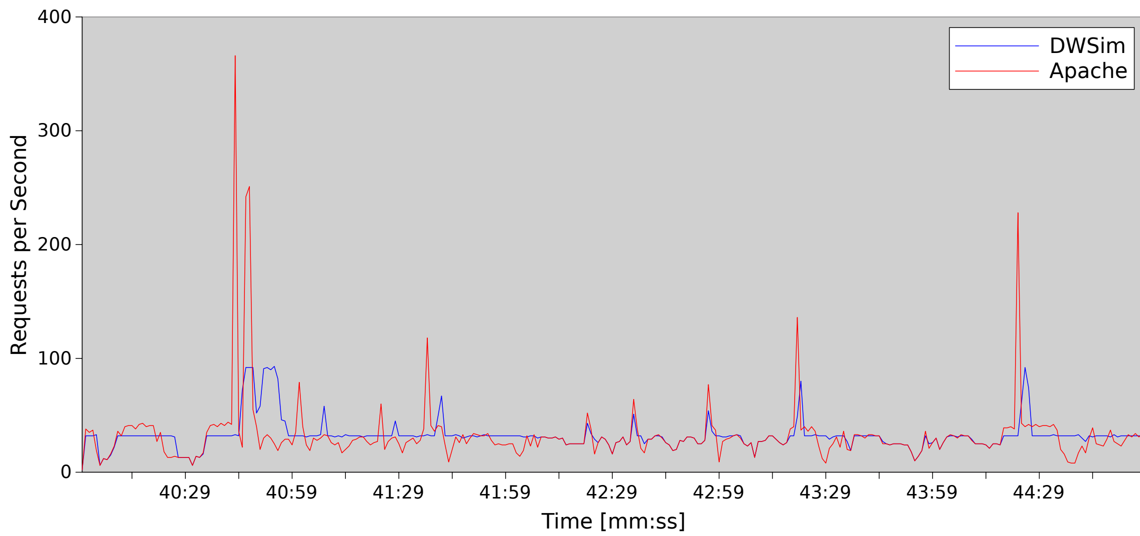

Figure 5.12: Apache Client Requests vs. DWSim Throughput for kgcoe-research, $\tau_n[request/s], \Delta t = 10[\mu s]$, [00:00:40 to 00:00:45] on 2015-10-14

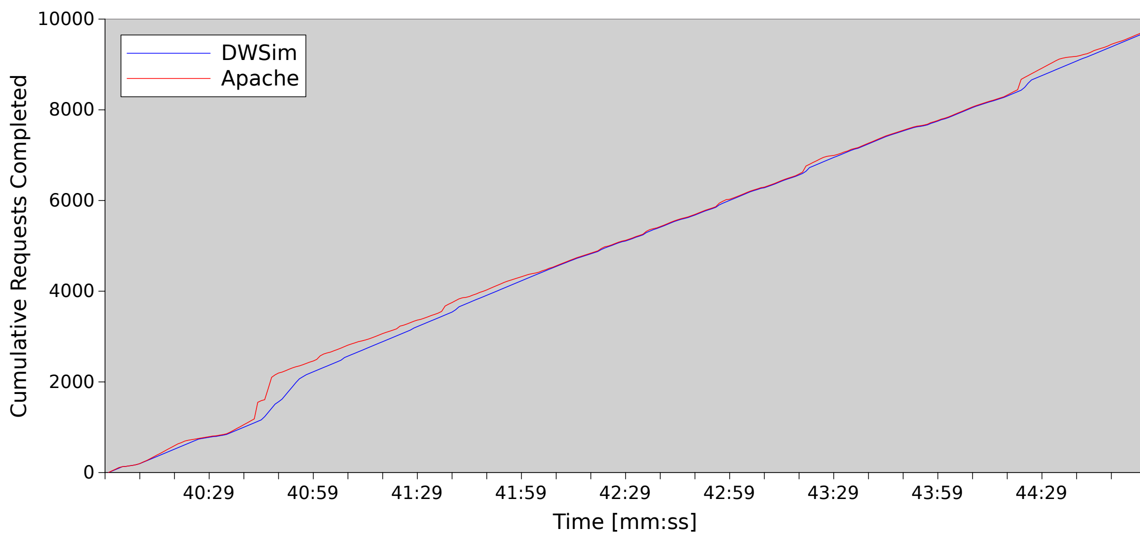

Figure 5.13: Cumulative Apache Client Requests vs. DWSim Throughput for kgcoe-research, $\Sigma\tau_n[requests], \Delta t = 10[\mu s]$, [00:00:40 to 00:00:45] on 2015-10-14

Figure 5.14 shows the mean latency for Client requests as observed during simulation. The observed latency hovers around the duration of a timestep, caused by extended periods in which latency is reported as 0s, resulting in an observed average below what was expected. A better representation of latency involved averaging only non-zero latency values, shown in Figure 5.15. This reveals two levels of latency observed during simulation: a latency of 11ms indicating requests only handled by the Apache HTTPd server and a latency of 32ms for requests requiring a second request to the DB server. The additional of 20ms between levels is consistent with the configured RTT for packets travelling between Apache and DB servers. During the two major periods of low latency starting at 00:40:45 and 00:40:50, a corresponding increase in average throughput was observed. Further investigation of the simulation workload during these periods revealed a large number of invalid requests, consistent with a minor security attack on the Apache server.

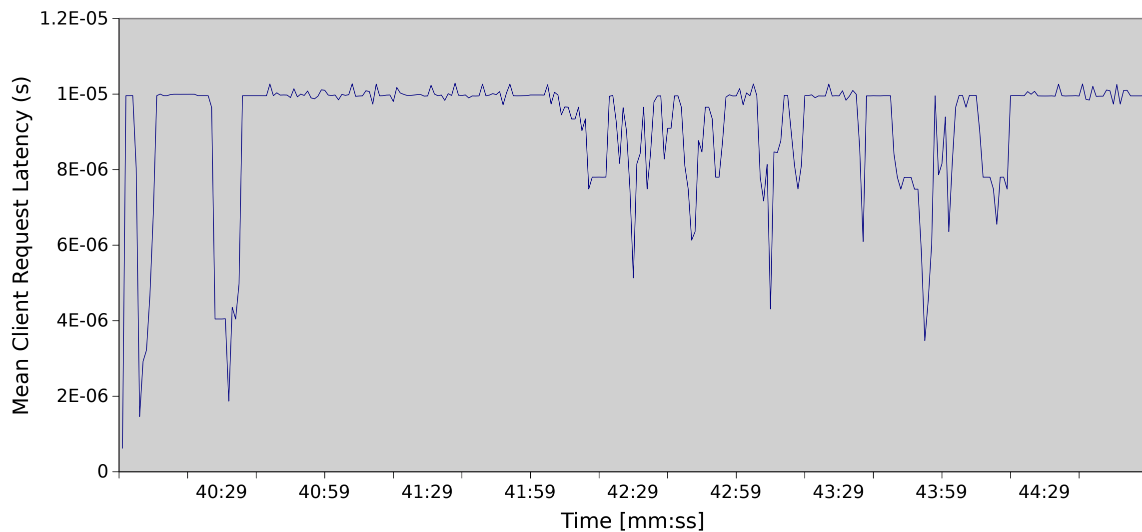

Figure 5.14: DWSim Client Request Mean Latency Bar Graph for kgcoe-research, $\bar{\alpha}[s], \Delta t = 10[\mu s]$, [00:00:40 to 00:00:45] on 2015-10-14

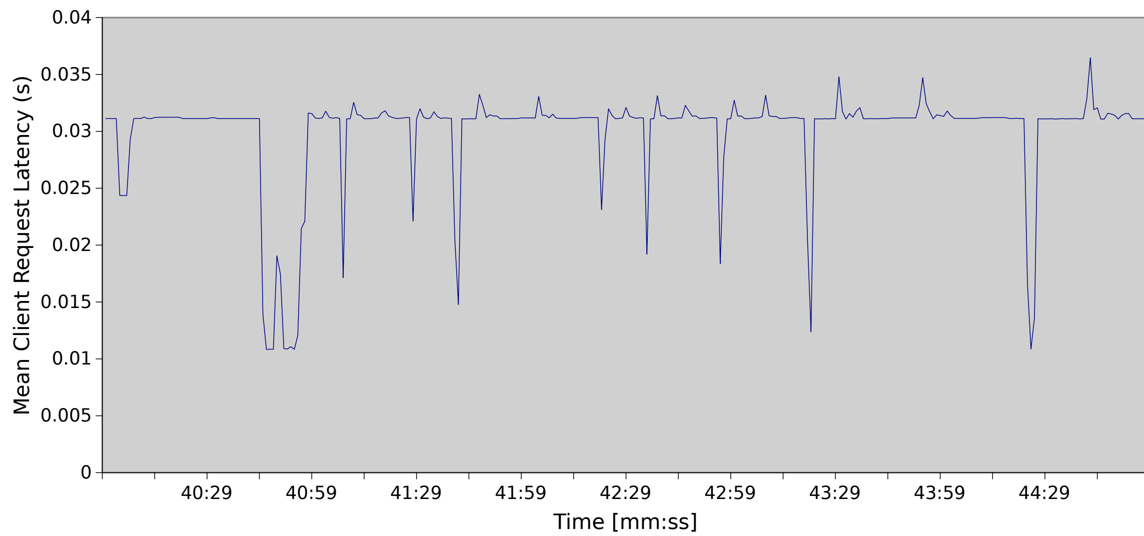

Figure 5.15: DWSim Client Request Mean Latency Bar Graph (Revised) for kgcoe-research, $\bar{\alpha}[s], \Delta t = 10[\mu s]$, [00:00:40 to 00:00:45] on 2015-10-14

Closing DWSim has been verified as producing valid simulations results, within the limitations of its implementation. The effects of constant time execution had the greatest impact on simulation results. Further improvement of the DWSim dataflow model is necessary, with initial results showing promise for its application in future work.

Chapter 6

Conclusions and Recommendations for Future Work

6.1 Conclusions

A static model of information flow and resource utilization has been demonstrated to predict the utilization, latency, and throughput of a static DWS. Further investigation is necessary to determine the efficacy of using the dataflow model to predict efficiency and scalability. Standardized metrics for DWS environments were described and then demonstrated through the use of the dataflow model. A second-generation EDGE DWS was designed and implemented successfully with the deployment of a new "kgcoe-research" server. A dataflow model for simulation of DWS environments was successfully completed, validated, and used to produce exemplar simulations of a production DWS. It has been demonstrated that the model shows promising results, which may improve with further enhancement and testing. Estimates of throughput and utilization produced by DWSim appear to show excellent agreement with the observed production system.

6.2 Future Work

6.2.1 Future Research

- **Use the dataflow model to predict and compare the performance of disparate DWS orchestrations.**

  A DWS may be configured and deployed in various ways. DWSim offers promise as a tool for determining the benefit of one DWS configuration over another. A next-generation DWSim tool may permit a-priori design of optimal or near-optimal configurations for DWS hardware resources. Further effort is necessary to evaluate the use of DWSim for comparative analysis and design of DWS configurations. Future efforts should also examine the use of DWSim to guide the optimization of a DWS configuration for factors such as hardware availability, high throughput, low latency, and low power operation.

- **Use the dataflow model to illustrate the potential of EDGE to accelerate time to market for new product development.**

  EDGE was developed to support product development efforts. The current set of tools focus on supporting the implementation of a design process and do not provide feedback to inform design process execution and evolution. The dataflow model developed for DWS makes it feasible to explore the behavior of task-dependency graphs. The similarities between manufacturing, design, and computing processes suggests that it may be possible to adapt the dataflow model to simulate the interactions between the participants in a product development process. Such a simulation could be used to provide predictions of project outcomes and time to market, as well as the scalability and efficiency of new product development teams.

6.2.2 Future Development

- **Integrate FACETS 2.0 into EDGE 2.0 to accelerate new product development**

  It will be necessary to migrate existing FACETS 1.0 tools to the new EDGE 2.0 system. These FACETS will be made to use the Wire Framework and its existing applications, and their database schemas will be improved according to the design of the EDGE 2.0 schema. Additional FACETS tools may then be developed by using these migrated tools as reference implementations. The incorporation of FACETS into the EDGE 2.0 environment presents significant opportunities to accelerate product development efforts and to enrich the documentation for the resulting projects.

Bibliography

[1] Sun ships jdk 1.1 – javabeans included. `http://web.archive.org/web/20080210044125/http://www.sun.com/smi/Press/sunflash/1997-02/sunflash.970219.0001.xml`, 1997. Accessed: 2016-07-06.

[2] Jdk 7 features. `http://openjdk.java.net/projects/jdk7/features/`, 2015. Accessed: 2016-07-06.

[3] Tiobe index. `http://www.tiobe.com/tiobe_index`, 2016. Accessed: 2016-07-06.

[4] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[6] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (INTERNET STANDARD), January 2005. Updated by RFCs 6874, 7320.

[7] Barry W Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[8] D Brissaud and O Garro. An approach to concurrent engineering using distributed design methodology. *Concurrent Engineering*, 4(3):303–311, 1996.

[9] Martin Campbell-Kelly. Historical reflections the rise, fall, and resurrection of software as a service. *Communications of the ACM*, 52(5):28–30, 2009.

[10] Alistair Cockburn and Jim Highsmith. Agile software development: The people factor. *Computer*, (11):131–133, 2001.

[11] Robert G Cooper. Stage-gate systems: a new tool for managing new products. *Business horizons*, 33(3):44–54, 1990.

[12] Sophie Curtis. Quarter of the world will be using smartphones in 2016. `http://www.telegraph.co.uk/technology/mobile-phones/11287659/Quarter-of-the-world-will-be-using-smartphones-in-2016.html`. Accessed: 2014-12-01.

[13] Dropbox. Dropbox.com. `http://dropbox.com`. Accessed: 2016-02-18.

[14] Eclipse. Eclipse ide. `http://eclipse.org`. Accessed: 2016-02-18.

[15] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. pages 365–376, 2011.

[16] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585.

[17] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[18] Kevin Forsberg and Harold Mooz Co-Principals. System engineering for faster, cheaper, better. 9(1):924–932, 1999.

[19] Eitan Frachtenberg. Holistic datacenter design in the open compute project. *Computer*, 7(45):83–85, 2012.

[20] Louis Glassy. Using version control to observe student software development processes. *Journal of Computing Sciences in Colleges*, 21(3):99–106, 2006.

[21] Google. Goodgle docs. `http://docs.google.com`. Accessed: 2016-02-18.

[22] Lance Hammond, Basem A Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *Computer*, (9):79–85, 1997.

[23] Kamila Hankiewciz. Waterfall vs. agile software development. `http://amuse.tech/waterfall-vs-agile-software-development/`, 2015. Accessed: 2015-01-18).

[24] Jim Highsmith and Alistair Cockburn. Agile software development: The business of innovation. *Computer*, 34(9):120–127, 2001.

[25] DAVE HUGHES. Is open source wireless connectivity worth the security risk? http://www.rtcmagazine.com/articles/view/103779, 2014. Accessed: 2015-01-18.

[26] Michael N Kennedy. Product development for the lean enterprise. *Richmond, VA: The*, 2003.

[27] Barry M Leiner, Vinton G Cerf, David D Clark, Robert E Kahn, Leonard Kleinrock, Daniel C Lynch, Jon Postel, Larry G Roberts, and Stephen Wolff. A brief history of the internet. *ACM SIGCOMM Computer Communication Review*, 39(5):22–31, 2009.

[28] Rasmus Lerdorf. Php 6. http://news.php.net/php.internals/47120, 2010. Accessed: 2016-07-06.

[29] Rasmus Lerdorf. Php 7.0.0 released. http://php.net/archive/2015.php#id2015-12-03-1, 2015. Accessed: 2016-07-06.

[30] Sean Marston, Zhi Li, Subhajyoti Bandyopadhyay, Juheng Zhang, and Anand Ghalsasi. Cloud computingthe business perspective. *Decision Support Systems*, 51(1):176–189, 2011.

[31] MediaWiki. Mediawiki.org. http://www.mediawiki.org. Accessed: 2016-02-18.

[32] BT. Meyers, EC. Hensel, and EA. DeBartolo. Usage of revision control tools in capstone senior design courses. In *ASME 2015 International Mechanical Engineering Congress and Exposition*, volume 15, pages V015T19A019–V015T19A028. American Society of Mechanical Engineers, 11 2015. doi:10.1115/IMECE2015-51164.

[33] C. Michael-Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. Version Control with Subversion, 2004. ISBN 0-596-00448-6.

[34] Microsoft. Microsoft project. https://products.office.com/en-us/Project/project-and-portfolio-management-software. Accessed: 2016-02-18.

[35] Microsoft. Onedrive. http://onedrive.live.com. Accessed: 2015-02-18.

[36] NetBeans. Netbeans ide. http://netbeans.org. Accessed: 2016-02-18.

[37] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. pages 124–131, 2009.

[38] Ravi Prasad, Constantinos Dovrolis, Margaret Murray, and KC Claffy. Bandwidth estimation: metrics, measurement techniques, and tools. *IEEE network*, 17(6):27–35, 2003.

[39] PTC. Windchill. `http://www.ptc.com/product-lifecycle-management/windchill`. Accessed: 2016-02-18.

[40] Redmine. Redmine.org. `http://www.redmine.org`. Accessed: 2016-02-18.

[41] Larry Sanger. The Early History of Nupedia and Wikipedia: A Memoir. *Open sources*, 2:307–338, 2005.

[42] Durward K Sobek, Allen C Ward, and Jeffrey K Liker. Toyota's principles of set-based concurrent engineering. *Sloan management review*, 40(2):67–84, 1999.

[43] Internet Live Stats. Total number of websites. `http://www.internetlivestats.com/total-number-of-websites/`, 2016. Accessed: 2016-02-18.

[44] David Thomas and Andrew Hunt. Pragmatic version control using cvs, 2003. ISBN 0-9745140-0-4.

[45] Dean M Tullsen, Susan J Eggers, and Henry M Levy. Simultaneous multithreading: Maximizing on-chip parallelism. 23(2):392–403, 1995.

[46] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando Martins, Andrew V Anderson, Steven M Bennett, Alain Kägi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[47] John Paul Walters, Vipin Chaudhary, Minsuk Cha, Salvatore Guercio Jr, and Steve Gallo. A comparison of virtualization technologies for hpc. pages 861–868, 2008.

[48] Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, Inc., 2010.

Appendix A

Code Listings

Listing A.1: ”Example Wire Configuration”

```
require 'wire'
require 'wiki-this'
require 'timbertext'
require_relative 'edge'
require_relative 'env/production'

use Rack::Session::Cookie, key: 'session', secret: 'super_secret_token'
use Rack::Deflater
Rack::Utils.key_space_limit = 10240000

use Clogger, format: :Combined ,
             path: '/var/log/rack/access.log', reentrant: true

closet = Wire::Closet.build do

        app 'admin' , Render::Page do
                auth :app do
                        handler EDGE::Auth::Admin
                end
                template 'views/admin.haml' do
                        use_layout
                end
                remote 'localhost:9292' , 'admin-partial'
        end

        app 'admin-partial' , Render::Partial do
                auth :app do
                        handler EDGE::Auth::Admin
                end
                resource 'about' do
                        all 'views/admin/about.haml'
                end
                resource 'global' do
                        all 'views/admin/global.haml'
                end
```

```
        resource 'admins' do
                use_forward
                multiple 'views/lists/admins.haml'
                single 'views/forms/admin.haml'
                extra :users, 'db-cache/users'
        end
        resource 'memberships' do
                use_forward
                multiple 'views/lists/memberships.haml'
                single 'views/forms/membership.haml'
                extra :projects, 'db-cache/projects'
                extra :roles, 'db-cache/roles'
                extra :users, 'db-cache/users'
        end
        resource 'projects' do
                use_forward
                multiple 'views/lists/projects.haml'
                single 'views/forms/project.haml'
                extra :tracks, 'db-cache/tracks'
        end
        resource 'roles' do
                use_forward
                multiple 'views/lists/roles.haml'
                single 'views/forms/role.haml'
        end
        resource 'tracks' do
                use_forward
                multiple 'views/lists/tracks.haml'
                single 'views/forms/track.haml'
        end
        resource 'users' do
                use_forward
                multiple 'views/lists/users.haml'
                single 'views/forms/user.haml'
                extra :tracks , 'db-cache/tracks'
        end
        remote 'localhost:9292' , 'db-cache'
end

app 'content-page' , Render::Page do
        auth :app do
                handler EDGE::Auth::Repo
        end
        template 'views/content.haml' do
                source :project, 'db-cache/projects' do
                        key :resource
                end
                source :tracks , 'db-cache/tracks'
```

```
                source :user ,'db-cache/users' do
                        key :user
                end
        end
        remote 'localhost:9292', 'render'
end

app 'dashboard' , Render::Page do
        auth :app do
                handler EDGE::Auth::Dashboard
        end
        template 'views/dashboard.haml' do
                use_layout
                source :memberships , 'db-cache/memberships'
                source :roles , 'db-cache/roles'
                source :tracks , 'db-cache/tracks'
        end
        remote 'localhost:9292' , 'db-cache/users'
end

app 'render' , Render::Document do
        auth :app do
                handler EDGE::Auth::Repo
        end
        remote 'localhost:9292' , 'error'
end

app 'error' , Render::Error do
        auth :app do
                handler EDGE::Auth::Repo
        end
        error 401 , 'views/errors/401.haml'
        error 404 , 'views/errors/404.haml'
        remote 'localhost:9292' , 'repos'
end

app 'db-cache' , Cache::Memory do
        auth :any
        remote 'localhost:9292' , 'db'
end

app 'db' , DB do
        auth :any
        db $environment[:database][:namespace],
           $environment[:database][:connection]
        model 'admins' , EDGE::Admin
        model 'disciplines' , EDGE::Discipline
        model 'memberships' , EDGE::Membership
```

```
        model 'projects' , EDGE::Project
        model 'roles' , EDGE::Role
        model 'tracks' , EDGE::Track
        model 'users' , EDGE::User
end

app 'edge-page' , Render::Page do
        auth :app do
                handler EDGE::Auth::Repo
        end
        template 'views/edge.haml' do
                use_layout
                source :project, 'db-cache/projects' do
                        key :resource
                end
                source :tracks , 'db-cache/tracks'
        end
        remote 'localhost:9292' , 'render'
end

app 'assets' , Static do
        auth :read_only
        local 'fonts' , './public/fonts'
        local 'js' , './public/js'
        local 'css' , './public/css'
        local 'img' , './public/img'
end

app 'instant' , Render::Instant do
        auth :any
        template 'views/instant.haml' do
                source :project, 'db-cache/projects' do
                        key :resource
                end
                source :tracks , 'db-cache/tracks'
        end
end

app 'edit' , Render::Page do
        auth :app do
                handler EDGE::Auth::Repo
        end
        template 'views/edge.haml' do
                use_layout
                source :project, 'db-cache/projects' do
                        key :resource
                end
                source :tracks , 'db-cache/tracks'
```

```
        end
        remote 'localhost:9292' , 'editors'
end

app 'login', Login do
        auth :any
end

app 'projects' , Render::Page do
        auth :read_only
        template 'views/project-info.haml' do
                use_layout
                source :memberships , 'db-cache/memberships'
                source :roles , 'db-cache/roles'
                source :tracks , 'db-cache/tracks'
        end
        remote 'localhost:9292' , 'db-cache/projects'
end

app 'tracks' , Render::Page do
        auth :read_only
        template 'views/track-info.haml' do
                use_layout
                source :projects , 'db-cache/projects'
        end
        remote 'localhost:9292' , 'db-cache/tracks'
end

app 'editors' , Render::Editor do
        auth :app do
                handler EDGE::Auth::Repo
        end
        remote 'localhost:9292' , 'repos'
end

app 'edge', EDGE::Redirect do
        auth :any
        remote 'localhost:9292' , 'edge-page'
end

app 'content', EDGE::Redirect do
        auth :any
        remote 'localhost:9292' , 'content-page'
end

app 'repos' , Repo::SVN do
        auth :app do
                handler EDGE::Auth::Repo
```

```
        end
        listing 'views/lists/dav.haml'
        repos 'testing/repos'
        web_folder 'web'
end

app 'history', Render::Page do
        auth :app do
                handler EDGE::Auth::History
        end
        template 'views/edge.haml' do
                use_layout
                source :project, 'db-cache/projects' do
                        key :resource
                end
                source :tracks , 'db-cache/tracks'
        end
        remote 'localhost:9292' , 'log'
end

app 'log' , History::SVN do
        auth :app do
                handler EDGE::Auth::History
        end
        log 'views/lists/log.haml'
        repos 'testing/repos'
        web_folder 'web'
end

app 'profiles', Render::Page do
        auth :read_only
        template 'views/profile.haml' do
                use_layout
                source :memberships , 'db-cache/memberships'
                source :tracks , 'db-cache/tracks'
        end
        remote 'localhost:9292' , 'db-cache/users'
end

app 'styles' , Render::Style do
        auth :read_only
        style 'admin' , 'views/sass/admin.sass'
        style 'kgcoe-research' , 'views/sass/admin.sass'
        style 'edge' , 'views/sass/edge2.sass'
        style 'communications' , 'views/sass/communications.sass'
        style 'energy' ,  'views/sass/energy.sass'
        style 'foundation' , 'views/sass/foundation.scss'
        style 'healthcare' , 'views/sass/healthcare.sass'
```

```
        style 'transportation' , 'views/sass/transportation.sass'
end

app :global , Render::Page do
        template 'views/layout.haml' do
                source :admins, 'db-cache/admins'
                source :project, 'db-cache/projects' do
                        key :resource
                end
                source :track, 'db-cache/tracks' do
                        key :resource
                end
                source :tracks , 'db-cache/tracks'
        source :user ,'db-cache/users' do
                        key :user
                end
        end
end

editor 'views/editors/wiki.haml' do
        mime 'text/wiki'
        mime 'text/timber'
end

renderer :audio do
        partial 'views/partials/audio.haml'
        mime 'audio/mpeg'
        mime 'audio/ogg'
        mime 'audio/wav'
        mime 'audio/x-wav'
end
renderer :image do
        partial 'views/partials/image.haml'
        mime 'image/bmp'
        mime 'image/gif'
        mime 'image/jpeg'
        mime 'image/png'
        mime 'image/svg+xml'
        mime 'image/tiff'
end
renderer :ml do
        partial 'views/partials/ml.haml'
        mime 'text/html'
        mime 'text/xhtml'
        mime 'text/xml'
        mime 'application/xml'
end
renderer :wiki do
```

```
                partial 'views/partials/wiki.haml'
                mime 'text/wiki'
                mime 'text/timber'
        end
        renderer :video do
                partial 'views/partials/video.haml'
                mime 'video/mp4'
                mime 'video/ogg'
                mime 'application/ogg'
                mime 'video/webm'
        end
end

run closet
```

Listing A.2: ”Wire Environment Fields

```
$environment = {
        host: '{host}',
        port: 9292,
        edge_user: '{user}',
        database: {
                namespace: :default,
                connection: 'mysql://{user}:{pass}@{host}/{db_name}'
        },
        dav_acl: '{svn_repos}/{svnserve_authz_db}',
        repos: '{svn_repos}',
        repos_user: '{edge_svn_user}',
        repos_password: '{edge_svn_password}'
}
```

Listing A.3: "Repo Interface"

```
module TestRepository
        extend Wire::App
        extend Wire::Resource
        extend Repo

        # Make a new repo
        # @param [String] path the path to the repositories
        # @param [String] repo the new repo name
        # @return [Integer] status code
        def self.do_create_file(path, repo)
                //code for repository creation
        end

        # Read a single file
        # @param [String] rev the revision number to access
        # @param [String] web the subdirectory for web content
        # @param [String] path the path to the repositories
        # @param [String] repo the new repo name
        # @param [String] id the relative path to the file
        # @return [String] the file
        def self.do_read_file(rev, web, path, repo, id)
                // code for reading a file
        end

        # Read a directory listing
        # @param [String] web the subdirectory for web content
        # @param [String] path the path to the repositories
        # @param [String] repo the new repo name
        # @param [String] id the relative path to the file
        # @return [Array] the directory listing
        def self.do_read_listing(web, path, repo, id = nil)
                // code for creating a directory listing
        end

        # Read Metadata for a single file
        # @param [String] rev the revision number to access
        # @param [String] web the subdirectory for web content
        # @param [String] path the path to the repositories
        # @param [String] repo the new repo name
        # @param [String] id the relative path to the file
        # @return [Hash] the metadata
        def self.do_read_info(rev, web, path, repo, id)
                // code for collecting metadata
        end

        # Get a file's MIME type
        # @param [String] rev the revision number to access
```

```
        # @param [String] web the subdirectory for web content
        # @param [String] path the path to the repositories
        # @param [String] repo the new repo name
        # @param [String] id the relative path to the file
        # @return [String] the MIME type
        def self.do_read_mime(rev, web, path, repo, id)
                // code for retrieving a MIME-Type
        end

        # Update a single file
        # @param [String] web the subdirectory for web content
        # @param [String] path the path to the repositories
        # @param [String] repo the new repo name
        # @param [String] id the relative path to the file
        # @param [String] content the updated file
        # @param [String] message the commit message
        # @param [String] mime the mime-type to set
        # @param [String] user the Author of this change
        # @return [Integer] status code
        def self.do_update_file(web, path, repo, id, content,
                                message, mime, user)
                // code for modifying a file
        end

end
```

Listing A.4: ”History Interface”

```
module TestHistory
        extend Wire::App
        extend Wire::Resource
        extend History

        # Get the log information for any part of a Repo
        # @param [String] web the web path of the repo
        # @param [String] repo the name of the repository to access
        # @param [String] id the sub-URI of the item to access
        # @return [Hash] the history entries
        def self.get_log(web, repo, id = nil)
                // code for retrieving the log
        end
end
```