Improving Service Level Agreements for a Job Scheduler by Visualizing Simulations

by

Dina M. Betser

S.B., Massachusetts Institute of Technology (2011)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

Author . . . .
Department of Electrical Engineering and Computer Science
May 1, 2012

Certified by . . . .
Dr. John Wilkes, Principal Software Engineer, Google, Inc.
VI-A Company Thesis Supervisor
Thesis Supervisor

Certified by . . . .
Asuman Ozdaglar, Associate Professor
MIT Thesis Supervisor
Thesis Supervisor

Accepted by . . . .
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

Improving Service Level Agreements for a Job Scheduler by Visualizing Simulations

by

Dina M. Betser

Submitted to the Department of Electrical Engineering and Computer Science
on May 1, 2012, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Currently, job owners at Google do not have a good way to generate suitable Service Level Agreements (SLAs), which means that they cannot accurately communicate their intentions to the job scheduler. This means that the owner's job might not finish on time or at all. The solution described in this thesis helps users visualize design changes to SLAs and use simulation to explore the behavior resulting from the SLAs. I have designed and begun development of a visualization and simulation framework that allows users to see how the job scheduler's behavior might vary under different SLA parameters. This thesis describes the steps made towards designing and implementing a system that both helps users visualize SLAs and their reward functions, and allows users to create an SLA and gain an idea of the behavior of a job scheduler with the SLA as input.

Thesis Supervisor: Dr. John Wilkes, Principal Software Engineer, Google, Inc.
Title: VI-A Company Thesis Supervisor

Thesis Supervisor: Asuman Ozdaglar, Associate Professor
Title: MIT Thesis Supervisor

0.1 Acknowledgments

I am so grateful for the many people who made this thesis possible. My M.Eng. project has been the greatest academic challenge I faced up to this point in my life, and as such, it proved to be an extremely educational growing experience. I really appreciate everybody who has touched my life up to this point and has enabled me to achieve what I have, but there are a few people whose contributions deserve additional recognition.

There are a lot of people who helped me during my time at Google. Dr. John Wilkes, my Google intern host on the Omega team, was incredibly giving in both his time and creative ideas. I was very fortunate to have a thesis supervisor who was so knowledgeable and patient in helping me with so many aspects of this project. The other person whose assistance with this project has been invaluable was Malte Schwarzkopf, who was my go-to grad student for technical guidance. He has done everything from explaining the details of the simulator used in my project to helping me debug arcane source control issues. I am very grateful to Evan Adams, another engineer at Google, who taught me a lot about software engineering and helped prepare me for a career in software engineering by emphasising software skills. Others who made a significant impact during my time at Google include Surabhi Gupta, Gloria Guo, and Dr. Dimitris Nakos. I was privileged to work with the Roma team, where I gained valuable skills in data visualization and learned a lot about the infrastructure that manages Google's compute clusters. There have been many more amazing, brilliant people at Google who have been invaluable over the past three years. Although I have not listed them here, I greatly appreciate their time and am looking forward to working with them again in the future.

I acknowledge my MIT thesis advisor, Professor Asu Ozdaglar, for providing guidance on the MIT-end of the thesis process. Professor Paul Gray was and continues to be an outstanding Course 6 advisor, and I feel so lucky to have been able to benefit from his advice and tutelage over the past four years. Thank you to those involved with the 6A program, including Professor John Guttag and Anne Hunter.

Thank you to my friends, whose encouragement and advice kept me going throughout the process. Special thanks go out to Jenny Liu, who acquainted me with the MIT thesis process and assisted me in fine-tuning my final project presentation, and to Jesse Dunietz, who helped me in the initial phases of designing and executing my project.

A very special thank you goes out to my parents and grandparents for always being supportive and believing in me. Without my family, I'd hardly be where I am today. They always know just what I need and make sure I'm headed on the right track.

Contents

Chapter 1

Introduction and Background

1.1 Problem

Google has one of the most complex distributed computing infrastructures in the world. Within Google's cluster management system, job owners run computing jobs—essentially, programs within a cluster environment—that must be scheduled and managed by Google's job scheduler.

It would be advantageous for job owners to inform the job scheduler how important it is for their jobs to be run soon or at all. Google's current job scheduler is unaware of this information – currently, the scheduler uses a job's details about its priority, job shape (i.e., resource requirements), and constraints (e.g., machine type, external IP) to determine which jobs to schedule first. Schedulers could use more information from jobs to exert finer-grained control over jobs' timing constraints. Moreover, job owners could use help in expressing these kinds of needs in a formal way, such as in the form of SLAs (Service Level Agreements). An SLA is a machine-readable document that describes a contract between a service provider and consumer. An SLA defines contractual obligations that produce revenue or result in cost penalties; it specifies a service consumer's needs along with rewards provided to a service provider upon meeting those needs. WS-Agreement [7], a Web Services protocol to establish agreements between two services, details one method of creating SLAs, and Google has expanded upon this model to support SLAs for some of its services.

Currently, job owners do not have a good way to generate suitable SLAs. They are not directly aware of the effects that setting certain SLA parameters will have upon the services in question, so they lack all the necessary information to generate an SLA that will support their desired levels of service. This causes delayed and dropped jobs because job owners cannot communicate their intentions.

1.1.1 Motivating example

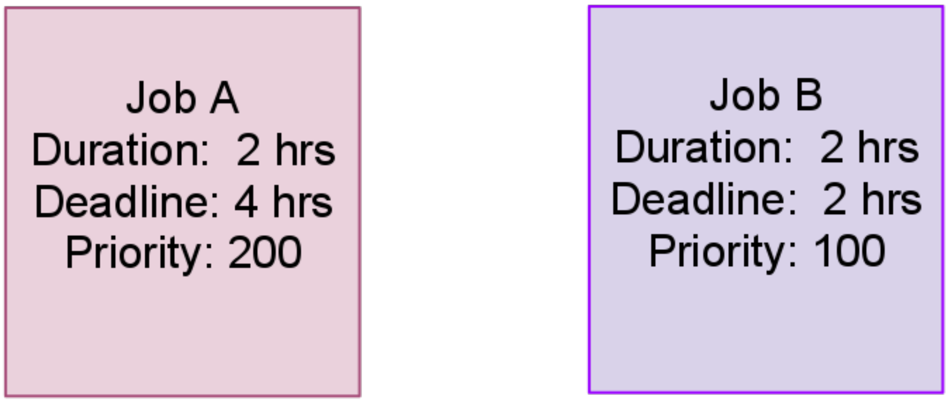


Figure 1-1: An example of the type of scheduling problem the current job scheduler cannot successfully manage. Out of the three job attributes of duration, deadline, and priority, the job scheduler currently only uses priority in making scheduling decisions.

Figure 1-1 provides an example of how the current cluster management system is ill-equipped to address certain job scheduling scenarios. Job owners may have different scheduling desires, but out of the three job attributes of duration, deadline, and priority, the job scheduler currently only uses priority in making scheduling decisions. In this example, there are two batch jobs to be run on the cluster. Both jobs have a duration of 2 hours; Job A needs to be completed within 4 hours, while the other is worthwhile to its owner if and only if it is done within 2 hours. Currently, jobs have no way to specify these types of temporal constraints (e.g., “completed within X hours,” and similarly, “started within Y hours”). Instead, the current job scheduler simply uses the priority value of the jobs in determining which job to run first. In this case, Job A would be run first because of its higher priority value, so Job B would miss its deadline. However, if the jobs were run in the opposite order, both could be completed while satisfying their respective constraints. If the scheduler knew about these temporal constraints, it could make a better decision when scheduling the jobs and start Job B before Job A, allowing both jobs to be run successfully. In other words, the ordinal ranking of job priority is not everything. As this example suggests, more should be done to take into account other attributes of a job such as duration or deadline when making scheduling decisions.

1.2 What is Omega?

This work focuses on Omega, Google's next-generation cluster management system. Omega will be responsible for scheduling and running computing jobs on Google's extensive computing infrastructure. Omega's underlying motivation is to support multiple, independent, application-specific schedulers in a single cell, with each scheduler's scheduling policy adjusted based on the type of workload the scheduler is designed to face. In the Google context, a "cell" refers to a logical unit of some number of machines within a data center. These schedulers, or "scheduling verticals," could process different categories of jobs and use additional information about the jobs to make wiser scheduling decisions.

Plans are already in place to include at least two scheduling verticals in Omega: one for service jobs, and one for batch jobs. These two categories of jobs have different scheduling needs. Service jobs, such as continuously-running web servers, are typically user-facing and latency-sensitive, which means that a delay in job output can be very problematic. Batch jobs, such as MapReduce jobs [11], are often throughput-sensitive, which means that while they might be more tolerant of delays or bursts in processing, the rate at which the job is processed must remain high. A batch job tends to be concerned with how soon it is able to start and complete.

Omega has a central shared state, or "cell state." The Omega cell state tracks machines, collections (jobs), and the tasks within a job, and also maintains a calendar of allocation decisions. This calendaring means that the cell state has a notion of the future, so jobs can reserve resources in advance.

This work serves, in part, to explore the possibility of using SLAs to make scheduling decisions in Omega. Because Omega plans to support specialized scheduling verticals, it is possible that a scheduling vertical could be made to focus on jobs with timing constraints specified as Service Level Agreements. Eventually, jobs with SLAs could be processed by a specialized scheduling vertical in Omega.

1.3 Related Work

SLAs are being used by many computing services to improve resource utilization, price-performance, and user satisfaction. In the context of grid and web services, SLAs are essentially thought of as electronic contracts, which are expected to be negotiated fully automatically (i.e., without any human intervention) and, as such, must be machine-readable and understandable to the humans who generate the SLAs.

An SLA can be considered a legally binding contract that specifies the terms and levels of certain services. The parties of an SLA can be distinguished into providers and consumers of a service. The terms of the SLA are agreed upon between service providers and service consumers.

Forms of SLAs have been in operation since the 1960s, when they were used as a method for buying minutes of computer machine time [22]. More recently, SLAs became more widespread as a means of making agreements when providing network services. Most of these agreements were paper-based and were drawn up after some form of negotiation between appropriate parties. Sakellariou and Yarmolenko [22] have argued for providing more flexibility in the level of service offered by supercomputing resources, such as by making separate SLAs between the resource owner and the user who wants to submit and run a job on these resources. The idea of providing this kind of flexibility ties in with the concept of utility, which is closely intertwined with SLAs. Utility as it relates to SLAs is explored in [25], which also describes the negotiation process between clients and service providers to achieve the best possible utility.

Many of Google's peers, such as HP [23], are studying optimization of resource allocation, wherein utility plays a large role in how resources are allocated. Within Google, elastic resource scaling has been tested to use historical resource usage data to adjust application resource demands automatically [12]. This relates to SLAs in that SLAs are one way to determine how to allocate these resources by observing utility values. Issues specifically related to the usage of SLAs for resource management on the grid are addressed in more detail in a number of other papers in the literature, such as by Naik, *et al.* [19]

Another significant area of research relates to the economic aspects associated with the usage of SLAs for service provision (e.g., charges for successful service provision, penalties for failure, etc.). Menache, *et al.* [16] takes a theoretical approach to describe how to maximize the aggregate utility of individual users together with the service provider (minus load-dependent operating expenses), assuming that a central controller may regulate admission and resource allocation to each arriving job based on the job's type. The main assumption is that in a cloud computing environment, the completion time and user's utility may depend on the amount of computing resources applied to the job. Convexity arguments are used to establish existence and uniqueness of the "social optimum," where aggregate utility is maximized. Finally, it is suggested that the social optimum may be induced by a linear usage-based tariff, which charges a fixed amount per unit time and resource from all users. While [16] focuses on a theoretical abstraction of the scheduling problem rather than implementation details, it provides an additional lens from which to examine the work described in this thesis.

Recently, there has been a significant amount of research on various other topics related to SLAs. Issues related to the overall incorporation of SLAs into grid architectures were discussed by Mobach, *et al.* [18]. Additionally, NextGrid [3] proposed SLAs as well as a negotiation approach, which were modelled according to business objectives of both customers and service providers [26].

Work completed by the Open Grid Forum led to the development of WS-Agreement [7], a specification for a simple generic language and protocol to establish agreements between two parties. Each of the two parties can initiate or respond to the agreement. The agreement structure of WS-Agreement was composed of several distinct parts, specifically Name, Context and Terms of Agreement, the latter of which was also divided into service description terms and guarantee terms. Service descriptions terms mainly described the functionality to be delivered under the agreement. The guarantee terms defined the assurance on service quality for each item mentioned in the service description terms section of the WS-Agreement. In the specific context of job submission, such assurances were defined as a parameter (constant) or bounds (min/max) on the availability of part or the whole of the resource.

The ideas and specifications for the "cheap-and-simple" and "by-deadline" SLA reward functions described in this thesis were generated after looking at other examples. The shape of the "cheap-and-simple" SLA's reward function is similar to the "soft deadline" of [24], defined as a monotonically decreasing function determined by pairs of points. The "by-deadline" reward function shape was inspired by the utility function described in [20], which has a similar shape of a positive horizontal segment, followed by a downward-sloping segment, followed by a negative horizontal segment.

In addition to the work described in the literature, significant work has been done within Google to deploy SLAs as a way to further automate the interactions between providers and consumers of services. Within Google, the Census team defined SLIs, SLOs, and SLAs [6]. Ahmadi built a PID controller for SLAs for the service that serves as the lowest level of Google's storage stack. This controller tuned the service to maximize the reward defined in the SLA, thus demonstrating the use of machine-readable SLAs to Google. Ahmadi's code evaluating how well an SLA has been met was used as a basis for calculating rewards in this project.

Visualizing SLAs and SLA-induced system behavior required deciding along which dimensions to display the data. The work done by Kim [15] during Summer 2010 at Google in visualizing SLOs served as a tool for viewing the data dimensions of SLO performance. Kim's goal was to build a web-based visualization tool that took in machine-readable SLO documents, displaying the interesting/problematic parts of the service, and reporting causes of the problems. Kim's work with SLO visualization techniques demonstrated one way to display aspects of the multi-dimensional space of how a service is doing against its SLOs. Kim's project also provided some infrastructure for this thesis project's SLA visualization aspect. Kim's visualizations focused on SLOs, lacking the necessary component of the reward function to fully represent SLAs. Because the reward function of the SLA determines its performance and utility, the reward function for each SLA figures prominently in the final Omega SLA Simulator visualizations. The Google Web Toolkit (GWT) framework, as described in Section 3.1.2, was useful in implementing these visualizations.

Additionally, the concept of "job shape," which includes the amount of RAM, CPU, and disk required by the job, is detailed by Mishra, *et al.* [17] This concept was useful in encapsulating the dimensions used by batch jobs for job scheduling simulations. It provided a way for the users and the simulator itself to refer to jobs with consistent terminology.

1.4 Contributions

The first contribution of this project was to design a way for jobs to specify temporal constraints in the form of SLAs. Beyond that, the contributions of this work can be divided into two categories: improving understanding and improving predictability with respect to job scheduling.

Both of these tasks motivated the design of a simulation platform that takes into account the temporal constraints expressed in SLAs. As part of this work, I designed the platform to provide a way for job owners to visualize SLAs and the behavior induced by a given SLA. It is designed to help the user understand the design space of SLAs and guide the user towards an SLA that maximizes utility. This involves using the job owner's willingness-to-pay to inform job owners ahead of time how likely a job is to start successfully in the desired time frame. To help users visualize "what-if" scenarios, this project included the implementation of a user interface that supports manipulating the parameters of an SLA and observing the results of simulation.

At the time of this project's execution, Omega was still a prototype, not running in production. Therefore, exploration into Omega-style scheduling was begun through a discrete-event-based simulator. For our purposes, simulation proved useful in that simulators enable the capability of running “what-if” scenarios for a system, without having to use the real-world system [9]. In this case, using a discrete-event simulator allowed us to prototype SLA-based scheduling for our system without Omega yet being fully functional or fast.

This thesis builds upon the related work described in Section 1.3, by designing a system that includes a user interface to present the effects of different SLA settings on the start times of jobs within the Google cluster computing system. The system communicates with SLA users in a way that feeds back to the user the effect of proposed SLA parameters on job execution performance.

This thesis describes systems at Google that may or may not be deployed in production. It describes the state of affairs at the time I left Google, and current plans may no longer be the same. It also makes forward-looking statements about what could be possible; it does not mean to imply that these outcomes will actually occur.

Within this thesis, Chapter 1 has provided some background and an overview of relevant work on SLAs. Chapter 2 describes how SLAs are implemented at Google, including the contributions of this work that have extended support for SLAs at Google. Chapter 3 describes the design and implementation of the system built to address the problem described in 1.1. Chapter 4 provides ideas for future exploration and concludes this thesis.

Chapter 2

SLA Formulation

One of the main tasks within this project was producing Service Level Agreements that could be used to encapsulate a job's desired service levels and the user's willingness to pay for them. In order to communicate these to the service provider, which could be any of Google's shared services such as Bigtable [10] or Google's job scheduler, the SLAs must be translated into a machine-readable format.

SLAs address this problem by providing a way for people to indicate just how important it is to them to maintain their service levels in the face of situations that change resource availability. The job owner wants the job to perform as well as possible given whatever situation the scheduler faces. Using an SLA, the scheduler can know better how to make tradeoffs.

Thus, SLAs serve as a machine-readable negotiation mechanism between providers and consumers of services. Without an SLA between them, the service provider may fail to provide the service that the application requires. An SLA represents a contract that encourages increased utility for all parties where utility, here, means that both sides have their needs met: the service provider does not want to spend time on tasks that have no value, and the job owner wants to get as much done as possible.

Within this chapter, Section 2.1 details the existing support at Google for SLAs and my contributions. Section 2.2 describes SLAs introduced by this work, specialized for the purpose of job scheduling. Section 2.3 provides an example use case for the system.

2.1 SLAs at Google

Customers specify their service requirements in Service Level Objectives (SLOs), and an SLA augments SLOs with rewards and penalties for service levels provided. Previous work at Google provided design principles for the infrastructure of an SLA, including definitions for service levels, Service Level Indicators (SLIs)[1], Service Level Objectives[2], and Service Level Agreements (SLAs)[3] in the form of a set of machine-readable documents (protocol buffers [4]) for all the SLA's items.

SLAs are designed to support any reward scheme with arbitrary complexity [6]. In general, a reward is a function of all SLIs and targets defined in an SLO between the service provider and consumer. The reward value output by an SLA takes the following factors into account:

1. How much a customer cares about violating each SLO target individually.

2. How much a customer cares about violations of multiple targets when considered together.

SLAs are structured in a way such that they can assign reward values to the objectives defined in an SLO instance. SLAs support reusable and custom-made reward functions. Figure 2-1 demonstrates the structure of the components that make up an SLA at Google.

An SLA is composed of `SLALineItems` that describe the reward associated with `SLOLineItems`. An `SLALineItem` describes how the agreement assigns rewards for an `SLOLineItem` and penalizes its violation. Each `SLOLineItem` describes a target for a specific SLI. For example, an `SLOLineItem`'s SLI can specify that the average latency over a certain time interval should be less than 30 ms.

[1] Service Level Indicator (SLI): The measurements associated with a service level: a metric of the service's and service consumerr's actual behavior.

[2] Service Level Objective (SLO): The desired quantity of a given service level. This includes a clear specification of both what is to be measured and tools to take and collect the measurements.

[3] Service Level Agreement (SLA): The combination of an SLO and a specification of the results of meeting or failing to meet the SLO (a reward or penalty). With an SLA between a service consumer and a service provider, the consumer can specify the business consequences of not being able to meet its objectives, while the service provider can specify the consequences if a customer overloads it and can decide how to make tradeoffs if it cannot meet all its SLOs.

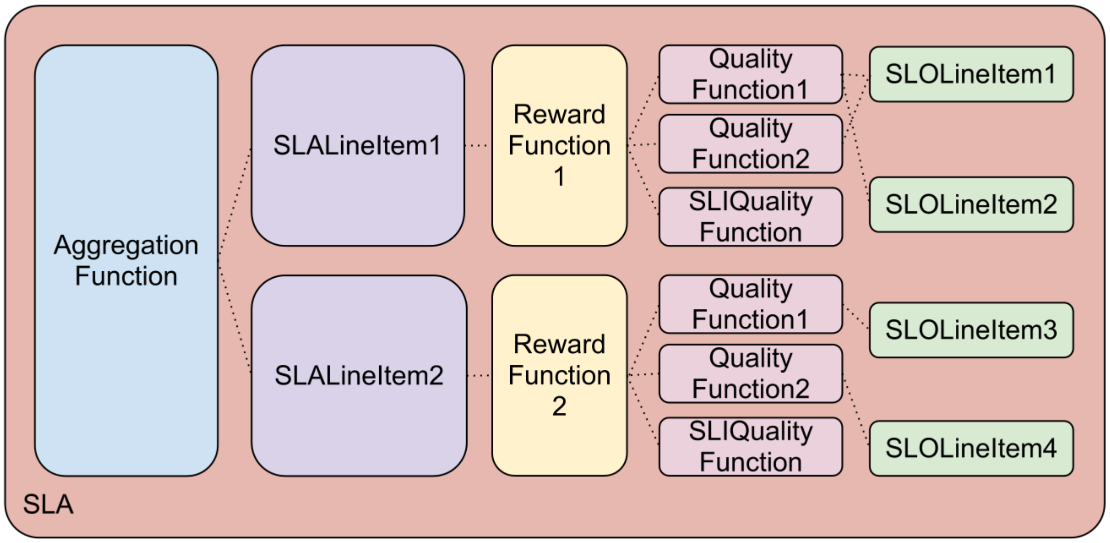


Figure 2-1: The components that make up an SLA at Google. An `AggregationFunction` consists of multiple `SLALineItem`s, each of which contains a `RewardFunction` that references `QualityFunction`s and `SLIQualityFunction`s. `QualityFunction`s can refer to any number of `SLOLineItem`s.

For the `SLOLineItem` described above, an `SLALineItem` in its simplest form can specify a reward of 100 if the average latency is less than 30ms and –50 otherwise. In the case where we are interested in particular combinations of latency and another factor specified by an SLI, multiple `SLOLineItem`s are mapped into a single `SLALineItem`.

In order to increase the reusability and readability of SLAs, the reward calculation process is split into two functions. Quality value only considers how the customer prioritizes violating a particular `SLOLineItem`. Reward value considers the relative importance of different performance targets for a single customer (e.g., latency versus throughput).

Based on this decoupling, `SLALineItem`s consist of a `QualityFunction` and a `RewardFunction`. A quality function takes exactly one `SLOLineItem` and produces a positive scalar value that indicates how well the `SLOLineItem` has been met. A value of 0 means that the objective has not been met at all and 1 means that it is fully satisfied. Values larger than 1 can be used to indicate that the performance is better than the objective. A reward function assigns reward to the value produced by the quality function. Penalties are merely negative rewards. Unlike quality functions, reward functions are simply a general mathematical function.

Within Google, reward is specified in units of SWE-hours, which can be thought of as "funny money," as these units are used to obfuscate the actual dollar value of the resources. When looking at reward, the relative values of the units that are used are what really matter.

2.1.1 Contributions to SLA Infrastructure at Google

As part of my work, I made modifications to Google's existing set of SLO protocol buffers to support SLAs for Google's job scheduler. These modifications included allowing an SLA's reward function to change based on how the service did in absolute terms, rather than how it did in relation to a target SLO. Within this model, `QualityFunction`s generate a quality value, a measure of how well an SLO is being met by an SLI. The quality value is then used directly within the reward function. I added support for `SLIQualityFunction`s. Whereas a regular `QualityFunction` requires an SLO for evaluation, `SLIQualityFunction`s generate a quality value for an SLI without reference to an SLO.

Additionally, because part of my work emphasized allowing job owners to tweak the values for various SLA parameters from the parameters' default values, I incorporated support for `max` and `min` values for the reward function parameters so that service providers could specify the extent to which parameter values could vary from the default values.

2.2 SLAs for Job Scheduling

With SLAs, different desired behaviors are signalled by different shapes in the SLA's reward function. Thus, the challenge is to devise a set of default SLAs for a service that encapsulates the tradeoffs that the service is willing to make.

In Omega, specialized workloads, such as MapReduce jobs, may be handled by their own specialized scheduling verticals. The goal is to enable jobs in Omega to specify SLAs. Then, an SLA-aware scheduling vertical might be able to look at SLAs when making scheduling decisions.

This work focused on Omega's batch job scheduler for a number of reasons. Batch jobs are concerned with their start and finish times, unlike service jobs, which may run continuously, so the metrics of start and completion time could be analyzed to generate simple, useful SLAs. Additionally, there are a lot more batch jobs than service jobs, so there is greater potential for impact. Finally, with Google's current batch job scheduler, a pricing scheme already exists; as part of this work, I investigated the preexisting pricing scheme and used it as a basis for formulating new SLAs and their accompanying reward functions.

In this work, the focus was on both the current batch job scheduler and the next generation batch job scheduler within Omega, which may consider looking at start time- or completion time-based SLAs in making scheduling decisions. The time constraints of this project mandated that the focus be on SLAs for job start times rather than job completion times; however, generating and using SLAs for job completion times is an extension of this project that should work well for Omega in the longer term.

2.2.1 SLAs for Google's Batch Job Scheduler

In order to get a sense of the type of Service Level Agreement that might work for Google's future batch job scheduler, I began by formalizing a set of SLAs for the existing batch scheduler. SLAs were generated by examining the current pricing scheme used at Google for resources that are requested for batch jobs.

At Google, there are two prices, peak and off-peak. These prices represent the cost of using resources at that time of day. There are a variety of options for scheduling batch jobs that take this into account. One option runs the job as soon as the resources are available, paying the price for the resources consumed, which ends up being a mix of peak and off-peak pricing depending on when the job was requested.

Two factors enter into the reward function for this option's SLA. The first factor is the cost of the resources actually used, which is a function of how long the job was running and using resources in the cluster. This is modeled by an `SLIQualityFunction` for each of the number of hours during off-peak and peak times. Thus there are quality values for both the amounts of runtime during peak hours and runtime during off-peak hours. The other factor that enters into the reward function is the amount of time it took for the job to start. This factor is represented as a `QualityFunction` with an associated desired start time. This `QualityFunction` returns a quality value less than 1 if the job does not start by this desired start time, and a value greater than 1 otherwise. The expression for this SLA's reward function can be found in Example 1.

Example 1 Reward function for the existing batch scheduler. There are two components to this reward function; if the SLO is unmet, the reward is 0. If the SLO is met, the reward is determined by how long the job actually spends running during peak times and non-peak times according to the batch scheduler resource pricing model, where peak times are M-F 6am-6pm PST and non-peak times are all others.

```
if (delay_quality < 1)
    return 0
else
    return peak_runtime_quality + nonpeak_runtime_quality
```

With this reward function, jobs that start sooner are assigned a higher quality value, which increases the overall reward for the scheduler.

One major difference between the SLAs for the current job scheduler and the future job scheduler is that the current job scheduler does not support a penalty if a job could not be scheduled by a desired time.

2.2.2 SLAs for Google's Future Batch Job Scheduler

In discussion with those developing the Omega batch scheduler, there were two use cases, each of which seemed to require its own SLA and accompanying reward function.

The first category includes jobs that desire a "cheap-and-simple" SLA – one that is not expensive but is of reasonable quality. The reward function, in this case a function of start time, is initially constant, and then asymptotically decreases towards a minimum reward value as the start time increases. The reward is never negative (see Figure 2-2). With this SLA, a job will always garner a positive reward for the scheduler if it is scheduled, but the scheduler has an incentive to schedule it sooner rather than later. The "cheap-and-simple" SLA includes parameters that control the maximum and asymptotic minimum reward value and rate of decline.

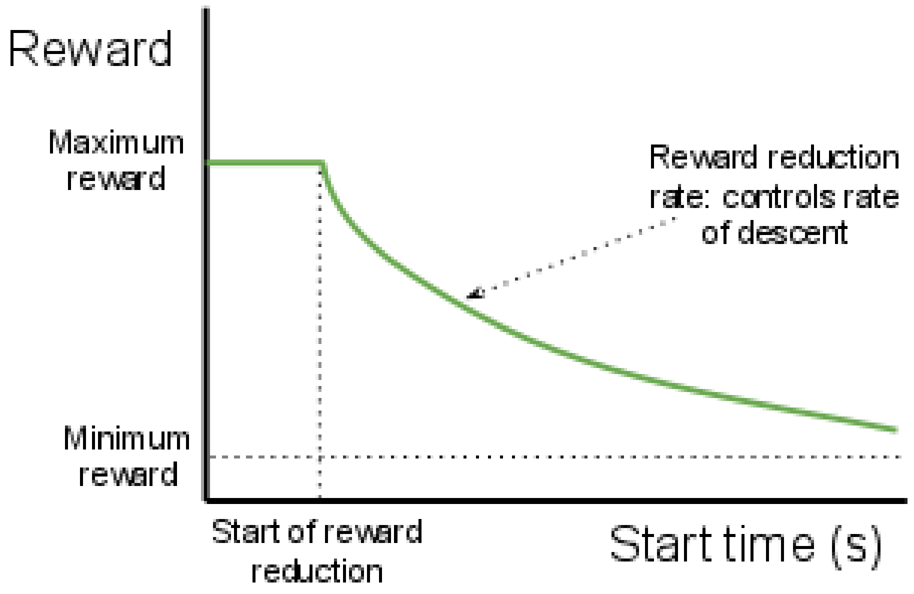

Reward
Maximum reward
Reward reduction rate: controls rate of descent
Minimum reward
Start of reward reduction
Start time (s)

Figure 2-2: The reward function of the "cheap-and-simple" SLA and its parameters.

The other SLA that batch job owners may wish to use is the "by-deadline" SLA. This reward function is represented as a level positive reward, followed by a downward-sloping reward line segment, followed by a level negative reward, also known as a penalty (see Figure 2-3). This reward function shape was inspired by similar utility functions derived by Popovici and Wilkes [20], and Irwin, *et al.* [14] The parameters are the coordinates of the point where the reward starts to drop and the point where the reward levels off after dropping. Users control the "max reward" and "max penalty" values as well as at what start times the reward function has "knee points." With this SLA, the job scheduler can use the reward function to determine the best time to run the job based on the rest of the scheduler's load.

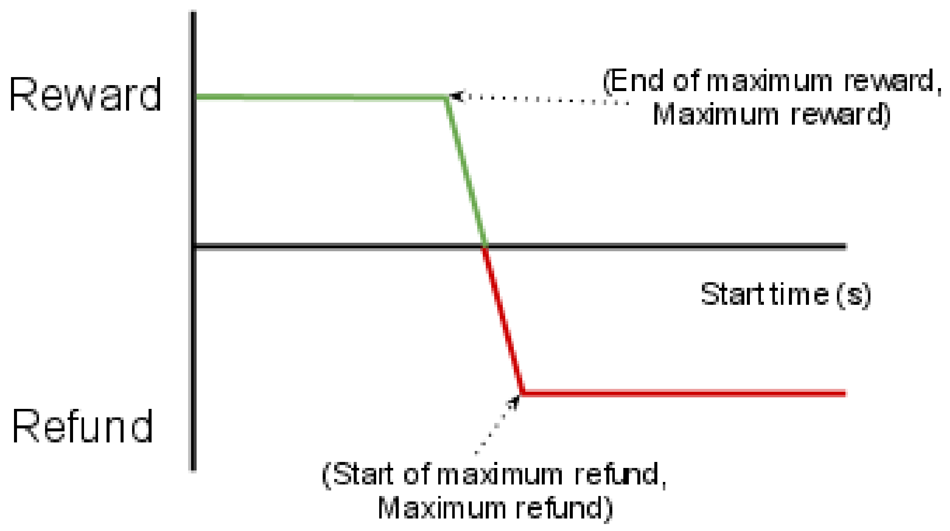


Figure 2-3: The reward function of the "by-deadline" SLA and its accompanying parameters.

2.3 Use Case

As previously described, the Omega batch scheduler hopes to offer two default SLAs, a "cheap-and-simple" SLA and a "by-deadline" SLA. Users would like to find the SLA that will meet their needs, so they will want to tweak parameters on these default SLAs to obtain the agreement they want.

To look at an example of what users might wish to do, let us examine "Bob," an imaginary user who represents a typical batch job owner at Google.

Bob has a batch job with two tasks that each have a demand shape of `CPU: 1 core, Memory: 30 GB, Disk: 4.5 GB`.

Bob wants his job to start in the next hour. Will Bob's SLA give him what he wants? The reward function of Bob's SLA can be seen in Figure 2-4.

Bob's goal is for his job to start by a certain time. Thus, the system needs to allow Bob to communicate how soon he would like his job to start, for which he could use a "by-deadline" reward function. Bob could then play "what-if" experiments to explore and understand the scheduling behavior that his proposed SLA induces. In this manner, job owners could use this tool to generate a proposed reward function and observe its effects on a job for themselves. Visualization of the simulation results is

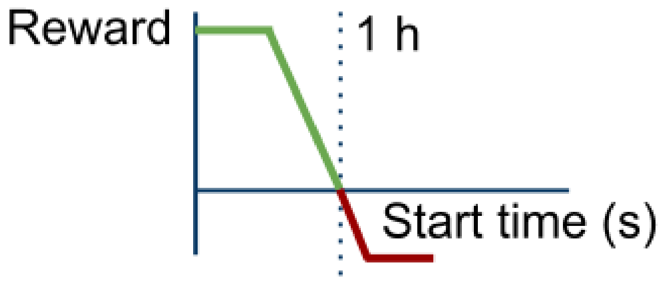


Figure 2-4: The reward function for Bob's SLA.

key in reporting to the user how well the scheduler can satisfy the SLA on the desired cell. By seeing the visualization of the simulation results, the user can ascertain if the SLA encourages the desired behavior, and the user can tweak the SLA until the visualized simulation results are closer to those that the user intended. This process of tweaking the SLA helps the user understand how much he values the job being started or completed by a particular time.

Chapter 3

System Design and Implementation

This project is divided into a frontend and a backend. The frontend supports visualization; the backend's goal is to make predictions. These two functions correspond to the contributions described in Section 1.4. The design of the system can be seen in Figure 3-1.

The frontend, described in Section 3.1, consists of the Omega SLA Simulator Dashboard, while the backend, detailed in Section 3.2, consists of a simulator, which is divided into several components.

3.1 Frontend – User Interface

3.1.1 Overview

The user interface component of the SLA simulation system illustrates to users what their options are in defining SLAs. It also informs the user of the results of various SLA selections. This UI component allows those who wish to run jobs using the batch scheduler to specify the job's resources requirements ("shape"), the cell in which the job should run, and an SLA for the job specifying its desired start time via a reward function.

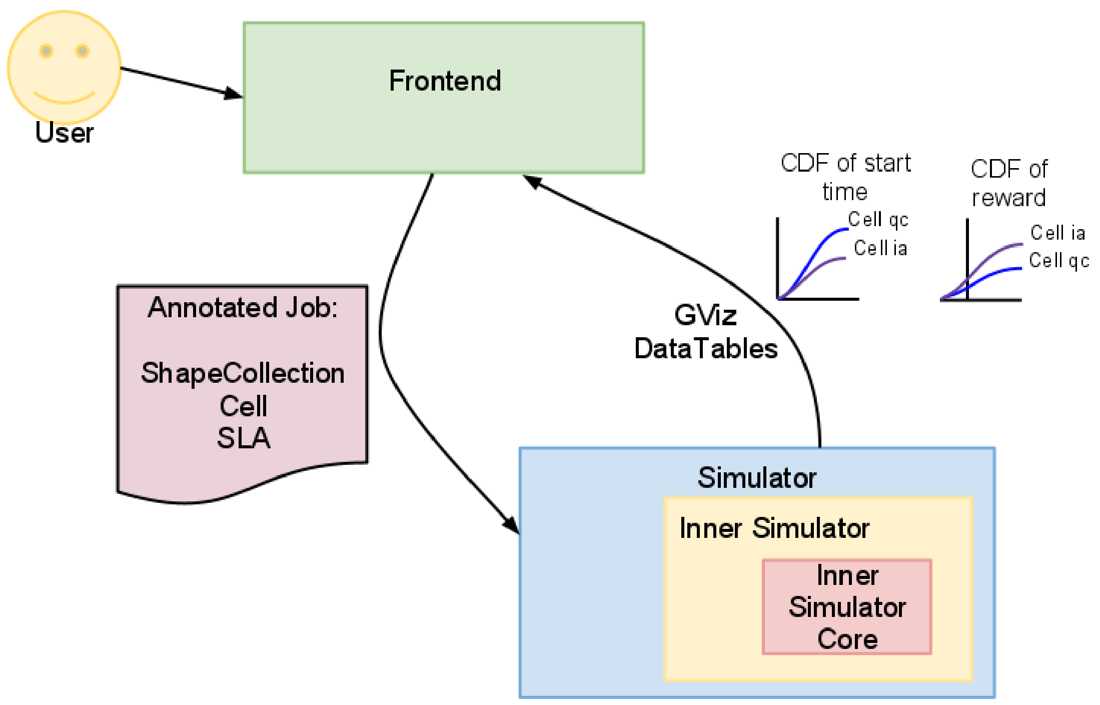


Figure 3-1: The design of the entire system.

This user interface is built using Google Web Toolkit (GWT) [2], a framework for writing web applications in Java. It is based on existing dashboard software at Google used elsewhere within cluster management.

3.1.2 Why GWT?

GWT is cross-browser compatible: the Java code is compiled into JavaScript/HTML for all the major platforms. The JavaScript code needed for every browser is generated, but using deferred binding, each client can optimize which version is loaded at runtime. On the programmer's side, all the server and client code is written in Java, which makes the development environment more uniform and consistent. The GWT Protobuf RPC mechanism exposes the client-side API for GWT applications, allowing for serialization and deserialization between the GWT client and server. Additionally, pure JavaScript is hard to debug and lay out, whereas GWT has a complement of cross-browser widgets that can be combined to build the interface. The

Model-View-Presenter (MVP) framework separates the concerns of the presentation logic. Furthermore, GIN (GWT INjection) brings automatic dependency injection to Google Web Toolkit client-side code. GIN is built on top of Guice, the generic Java dependency injection framework at Google. These features of GWT make it a useful framework with which to implement the user interface of the Omega SLA Simulator.

3.1.3 Dashboard infrastructure

The SLA Simulator Dashboard is based on existing dashboard software. Written using GWT, the structure of this existing dashboard was similar to that needed by the Omega SLA Simulator dashboard. Additionally, the dashboard relied upon constructs relevant to the SLA simulator as well; for instance, the dashboard used a `ShapeCollection` as the descriptor for a job, and executed its service requests for a given set of cells or regions, just as the SLA simulator needed to do. The server logic in the dashboard was replaced by code that communicated with the Omega simulator on the backend.

The dashboard followed the MVP (Model-View-Presenter) design pattern, which separates logically distinct layers in an application [21]. In the MVP paradigm, the **model** stores data that can be displayed and modified by the user. The **view**, in addition to displaying the data, responds to user input and UI events. The **presenter** is the link between the model and view; it fetches data from the model and passes it along to the view for display.

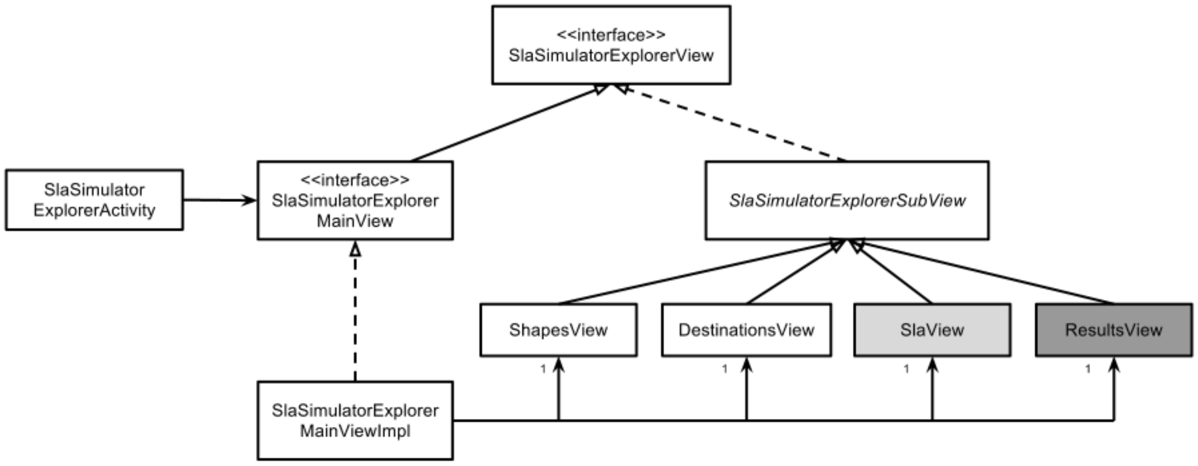
The MVP framework ties in with GWT's Activities and Places paradigm. `Activity`s are used to represent what the users are doing, and tend to perform actions such as restoring state, loading a UI, or performing setup operations. `Place`s are Java objects that correspond to URL history tokens [5]; in this way, a `Place` represents a given UI state. The dashboard only used one `Place`, the `SlaSimulatorExplorerPlace`, and this structure was retained in the Omega SLA Simulator Dashboard.

3.1.4 Modifications

All changes to the dashboard focused on the dual functions of 1) allowing SLAs and their parameters to be used as input to the simulator and 2) obtaining and displaying results from running simulations.

New view and associated component

Figure 3-2 demonstrates the relationship of the modified view (`ResultsView`) and added view (`SlaView`) with the existing infrastructure.



<<interface>>
SlaSimulatorExplorerView
SlaSimulator ExplorerActivity
<<interface>>
SlaSimulatorExplorer MainView
SlaSimulatorExplorerSubView
ShapesView
DestinationsView
SlaView
ResultsView
1
1
1
1
SlaSimulatorExplorer MainViewImpl

Figure 3-2: UML diagram of the Java classes for the view hierarchy. The component added (light gray) enabled SLA configuration. The existing results component (dark gray) was modified to query the Omega SLA Simulator on the backend. Other dashboard components were renamed to start with `SlaSimulator`, but were otherwise unchanged.

SLA view

Because the user needs to be able to select and customize a default SLA, there needs to be a uniform method of selecting an SLA template from a list of options, and specifying parameters for the chosen template.

Additionally, because SLA customization was an entirely new process that needed to occur before the RPC calls were made to the simulation backend, a new pane on the main page was built for this purpose. In addition to SLA selection, this pane needed to support customization by providing input fields specifying parameters specific to the selected SLA (see Figure 3-5 on page 31, which depicts the finished interface including this SLA Pane). SLAs are built by the SLA component to be sent along to the SLA simulator by RPC.

Results view

The existing dashboard already supported integration with GViz, Google's Visualization API, so modifications to the set of `Result` classes were aimed at displaying the desired graphs for the SLA simulator. To do this, the data returned by the SLA simulator needed to be translated into a GViz-ready format.

3.1.5 Simulation Service

The dashboard's web server consisted of a collection of services (implemented by GWT servlets) specialized for different purposes needed by the dashboard. The service that provided a list of cell names (`SupplierInfoService`) was usable. However, to fetch simulation results from the SLA simulator, a new service needed to be created (see Figure 3-3).

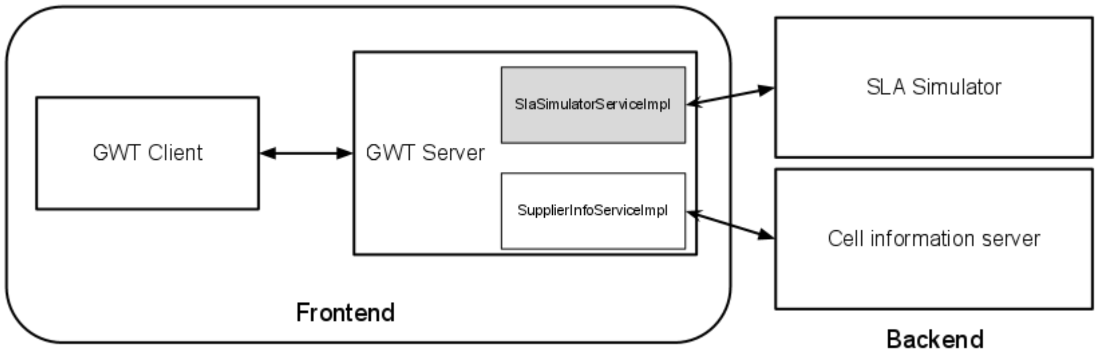


Figure 3-3: The relationship between the "frontend" and "backend." The frontend is a GWT application that consists of a GWT client and GWT server that communicate using GWT ProtoBuf RPC. The GWT server communicates with the backend using Stubby (Google's RPC layer). The gray component was built to communicate with the SLA Simulator.

The class diagram in Figure 3-4 demonstrates how the GWT ProtoBuf RPC pattern was used for this service. `SlaSimulatorClientService` has the same method signatures as `SlaSimulatorService`, except that an asynchronous callback function was added as a parameter for the return value. The `SlaSimulatorExplorerActivity` instance, which serves as the Presenter in the MVP framework, is responsible for passing request and response objects to and from the GWT server.

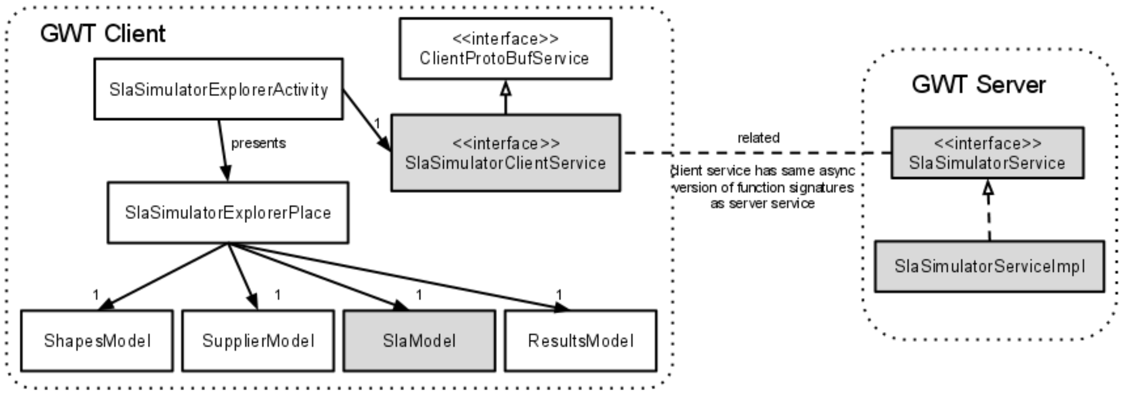


Figure 3-4: This figure explores the inner structure of the GWT client and server depicted in Figure 3-3. Each box pictured is a Java class. `SlaSimulatorExplorerPlace` has an instance of each model, and represents the state of the client. `SlaSimulatorExplorerActivity` acts as a presenter in the MVP pattern by presenting `SlaSimulatorExplorerPlace`, and is also responsible for making calls to the GWT server to receive simulation results. New components have a gray background.

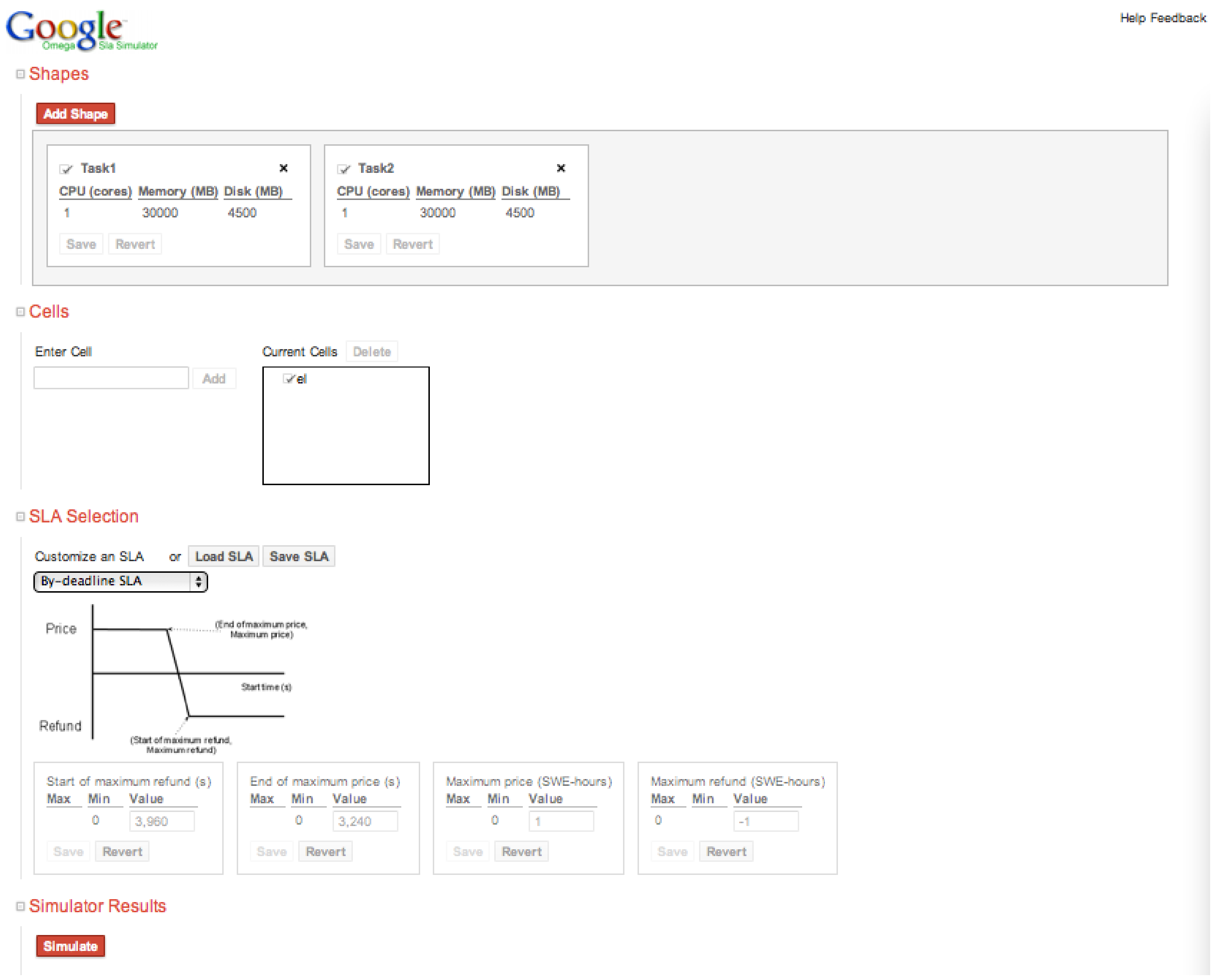
The class `SlaSimulatorService` was implemented as a client stub that issues RPCs to the Omega SLA simulator, thus delegating all user requests to the simulator.

3.1.6 Screenshots

The final version of the Omega SLA Simulator user interface has four panes, each separated by a header in red text, (see Figure 3-5). The first two panes, the Shapes pane and the Cells pane, are largely reused from the existing dashboard. The Shapes pane uses a set of shapes to represent a job; each shape corresponds to a task. The third pane, the SLA pane, allows for selection of a default SLA and customization of the fields of the selected SLA. The fourth pane, the Results pane, displays two graphs using GViz.

In the third pane, SLAs can be selected from the drop-down menu, and their parameters can be updated via the widgets that appear after selection of an SLA. As an extension, logic could be implemented for the currently stubbed-out “Load SLA” and “Save SLA” buttons to load an SLA from an ASCII file or to save the current SLA to an ASCII file

Two GViz charts of cumulative density functions (CDFs) for job start times and rewards are displayed in the Results pane. In these CDFs, the X axis represents the metric of interest, in this case, start time, and the Y axis stores percentiles. These percentiles correspond to the results of running the job specified by the user multiple times in a Monte Carlo simulation.



Task1

CPU (cores)	Memory (MB)	Disk (MB)
1	30000	4500

Task2

CPU (cores)	Memory (MB)	Disk (MB)
1	30000	4500

Current Cells: ✓ el

By-deadline SLA

Start of maximum refund (s)

Max	Min	Value
	0	3,960

End of maximum price (s)

Max	Min	Value
	0	3,240

Maximum price (SWE-hours)

Max	Min	Value
	0	1

Maximum refund (SWE-hours)

Max	Min	Value
0		-1

Figure 3-5: The Omega SLA Simulator user interface.

To demonstrate the type of visualization this tool is able to show the user, consider the following scenario. Bob, from the use case described in Section 2.3, wishes to develop and test an SLA that will induce the job scheduler to start his job in the next hour. Figure 3-5 shows how Bob would go about this: he enters his job's task information to the Shapes pane, selects his desired cell, and then chooses the "by-deadline" SLA from the drop-down menu. Using the default SLA that is loaded to the UI, with the value for the maximum refund parameter set to "-1," he requests simulation by clicking the appropriate button, at which point the visualizations in the Results pane are displayed (top half of Figure 3-6). Bob can see from the left chart that at 3600 seconds (1 hour), the job would be started with about a 40% likelihood. Seeing this, Bob might want to alter the default SLA to encourage the scheduler to start his job sooner. For instance, he could change the value for the maximum refund parameter to -3, thus increasing the penalty to the scheduler for starting the job later. Bob would want to see that the likelihood of his job starting within one hour increased. An improved result panel can be seen in the bottom half of Figure 3-6, where the likelihood of the job starting within the hour has jumped to 80%.

3.2 Backend

The main components of the Omega SLA simulator backend are the outer simulator layer, the Monte Carlo simulator layer, and the inner simulator layer. Components will be described from outermost layer to innermost.

3.2.1 Outer Simulator

The outer simulator layer of the Omega SLA simulator is responsible for communicating with the GWT frontend and for performing data processing on the simulation results of the Omega multischeduler simulator on the backend. It consists of a Stubby (Google's RPC layer) server to interact with the Stubby client contained within the GWT server and the modules that process raw probability density functions (PDFs) output by the Monte Carlo simulator. The data flow diagram for the outer simulator can be seen in Figure 3-7.

The outer simulator layer of the Omega SLA simulator is responsible for communication between the GWT frontend and the Monte Carlo simulator. This is a simple

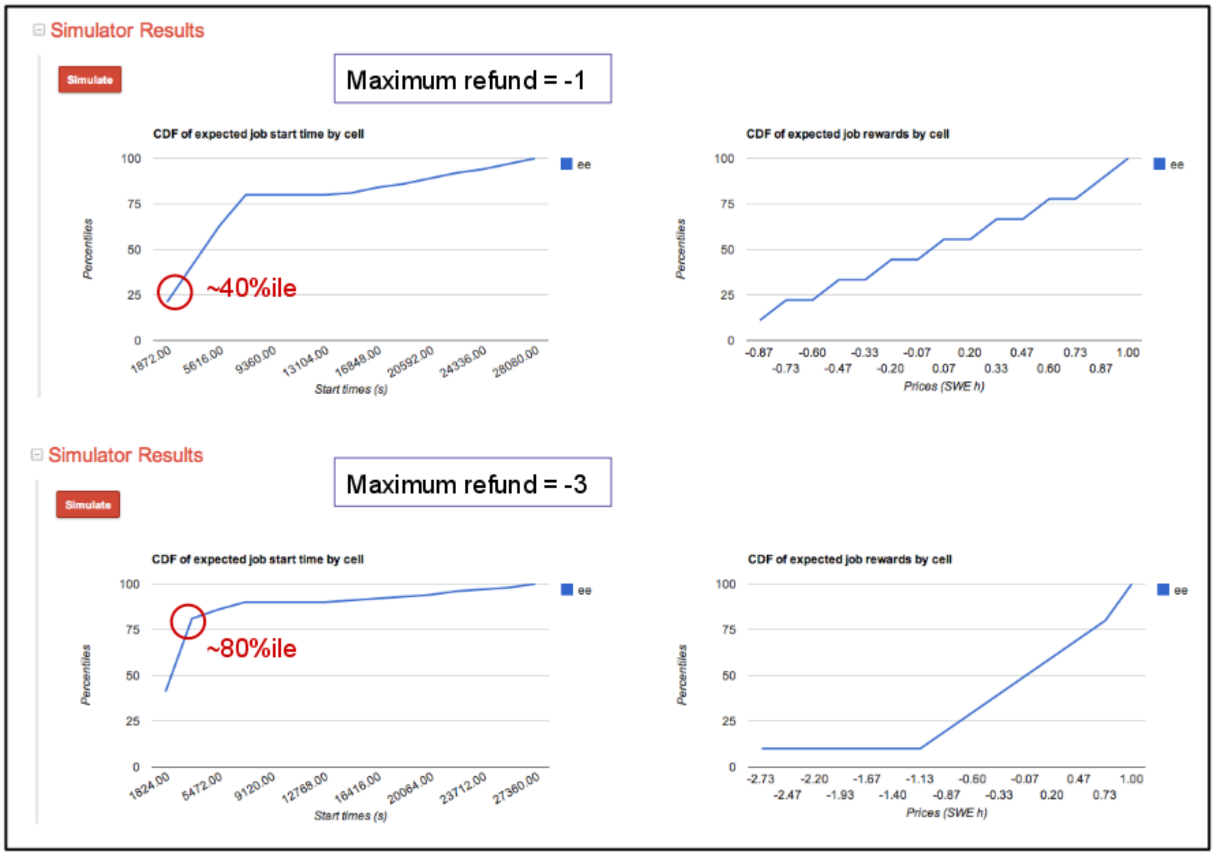

Simulator Results
Simulate
Maximum refund = -1
CDF of expected job start time by cell
Percentiles
Start times (s)
~40%ile
ee
CDF of expected job rewards by cell
Prices (SWE h)
Simulator Results
Simulate
Maximum refund = -3
CDF of expected job start time by cell
~80%ile
CDF of expected job rewards by cell

Figure 3-6: An example of how the Results pane might look when Bob runs the simulation with different values for the "Maximum refund" parameter of the "by-deadline SLA." In the top half of this figure, the parameter's value is set to −1, while in the bottom half, the value is −3.

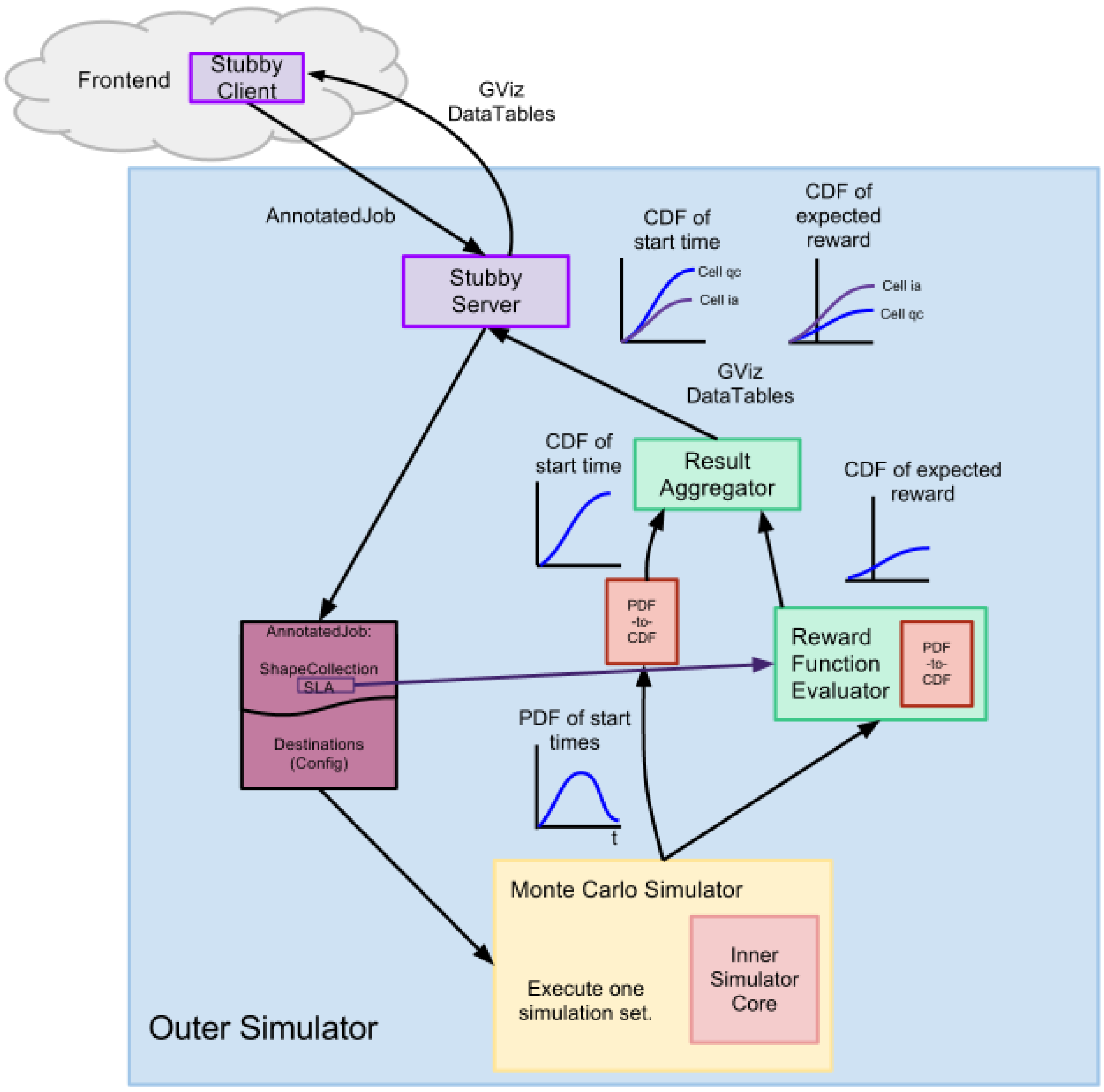


Figure 3-7: The data flow diagram for the outer simulator component of the Omega SLA Simulator.

Stubby server module written in Python that is designed to accept RPC calls from the frontend and return the data needed to visualize the CDFs obtained from the Result Aggregator module.

The PDF-to-CDF Converter module converts PDFs of one format into CDFs of another format. The Monte Carlo Simulator will output a PDF of results, which is translated to a CDF in the form of a GViz DataTable [1]. These CDFs, when produced for multiple cells, can be placed onto a single set of axes by the Result Aggregator. This way, a single graph can contain the CDFs of a given type for all the desired cells.

The Reward Function Evaluator accepts a distribution of job start times. This module produces a distribution of expected reward values, which is then fed into the Result Aggregator. It does so by substituting each start time value in the distribution of job start times into the reward function, and evaluating the resulting expression to obtain a reward value. The reward values generated in this way are combined to form a distribution of reward values. The Reward Function Evaluator makes a call to the PDF-to-CDF Converter within its logic.

The purpose of the Result Aggregator module is to collect the results from simulations for one or more cells and to reformat the results as a set of data for display in two GViz DataTables, one for start time values, and one for reward values. Therefore, when a set of cells are selected by the user for analysis, results can be displayed succinctly. The GViz DataTables returned by the aggregator are sent by the Stubby server to the frontend for display.

3.2.2 Monte Carlo Simulator

The purpose of the Monte Carlo simulator is to generate a distribution of start times for a given job. It does so by submitting a job that has been annotated with an SLA to the inner Omega SLA simulator. On a given submission, the question answered is, “if the scheduler tried to run the job now, how long would it take to start the job (or would this job fail to start)?”

Monte Carlo simulation builds models of possible results by substituting a range of values – a probability distribution – for any factor that has inherent uncertainty. It then calculates results over and over, each time using a different set of random values from the probability distributions. In this way, Monte Carlo simulation produces distributions of possible outcome values. A single value cannot sufficiently describe the uncertainty inherent in a variable with an unknown outcome. Probability distributions are a more realistic way to describe such a variable.

Monte Carlo methods vary, but tend to follow a common pattern:

1. Define a domain of possible inputs.

2. Generate inputs randomly from a probability distribution over the domain.

3. Perform a deterministic computation on the inputs.

4. Aggregate the results from the previous step into a final computation to generate a probability distribution of outcomes.

The result is an approximation to some unknown quantity.

The Monte Carlo simulator layer is responsible for making calls to the Omega SLA simulator. Because every run of the simulator produces only one prediction of the job's start time, the inner simulator must be run a number of times to generate a distribution of start times for the job.

Therefore, Monte Carlo methods are particularly applicable, as the Omega SLA simulator is designed to arrive at the same deterministic result based on a given set of inputs. The simulator is initialized with an initial cell state, and proceeds by running a trace file that has been processed for simulation. These trace files contain the event timing information necessary for replaying the execution history of the job scheduler for a specified time period. This information includes details such as when a job's tasks started within the cluster and when they finished.

The normal use case for the simulation tool would be as a means of predicting behavior for a live cell; users aiming to run their jobs now or in the near future would like to know what would happen if they ran their jobs on live cell with current resource constraints. However, to achieve a particular goal, it is often necessary to use historical cell state data.

Currently, the historical traces and cell states that are used for simulation may not represent a time when the cell was particularly busy. Because of this, a job that is injected into the trace may not face much contention, and would be likely to schedule quickly or immediately. In order to simulate a future, busier world, the cell size may be shrunk or the offered load increased before running the simulation to increase the competition for available resources. More contention leads to more interesting behavior where SLA-based scheduling might make a difference.

To apply Monte Carlo methods, the main decision to be made is how to use randomness to arrive at a reasonable simulation of the cell state. The Omega simulator's deterministic computation needs to be run multiple times, including an element of randomness each time, to satisfy steps 2. and 3. above. This requires a source of randomness, which, in this case, arise from two random number generators. The random generator within the Monte Carlo simulator controller uses a deterministic seed to produce a random number. This random number is provided as a seed to the second random number generator that is within the inner Omega SLA simulator. These random numbers will be used by the Omega SLA simulator to draw input values from a probability distribution in order to achieve variance in the load that the job of interest faces, and thus allow us to predict the future.

As described in Section 2.2, a distribution of job start times is desired as an intermediate goal toward obtaining a distribution of job completion times. Users want to know how long their job will take to complete if submitted now, and the corresponding expected reward for the job. The job's start time is used as a crude means of approximating the job's completion time. Adding running-time information to the simulation to support completion time prediction is an extension that is discussed in Section 4.1.2.

There are a number of possible methods to achieve variance in the job start time distribution. Because the time constraints of this project did not allow me to implement these methods, the ideas for achieving variance are discussed in the future work section (see Section 4.1.1).

3.2.3 Inner Simulator

The Omega multischeduler simulator is an event-driven simulator that utilizes real Google job scheduler scheduling logic and real workload traces. The Omega simulator includes multiple scheduling verticals with a single underlying cell state, just as Omega plans to do. Omega is described in more detail in Section 1.2.

The workload used by the simulator consists of historical trace data from Google's current job scheduler running in real cells. The simulator has been used for experimental purposes within Google, and as such, there is comfort within the company that its simulation results are a reasonable reflection of the scheduler's anticipated behavior.

The multischeduler simulator is a discrete event simulator written in C++. It maintains a queue of events for arrivals and completions of jobs ("collections") and their tasks. The types of events processed by the simulator are as follows:

1. Collection Submitted
2. Collection Scheduled
3. Task Restarted
4. Task Ended

Within the simulation, the initial cell state is loaded from historical cell data, and an initial trace-processing pipeline prepares historical trace data for the simulator.

Because the Omega multischeduler simulator supports specialized scheduling verticals, a key step in this project was to create a specialized SLA-aware vertical. In this vertical, SLAs serve as the basis for decision making; in other words, for jobs that have an associated SLA, the SLA's reward function determines the order in which jobs are scheduled. There are a number of algorithms that could be used to determine this order:

Greedy with respect to maximum reward

This algorithm uses the instantaneous reward provided by the jobs' reward functions to decide which job is scheduled next whenever a pending job needs to be popped off the queue. For instance, in Figure 3-8, we see that Job P has been waiting to start for 100 seconds and has a current reward of 1.2 SWE hours, while Job Q has been pending for 50 seconds, with a current reward of 1.5 SWE hours. This algorithm would pick Job Y to schedule first because it would provide a larger reward to the job scheduler.

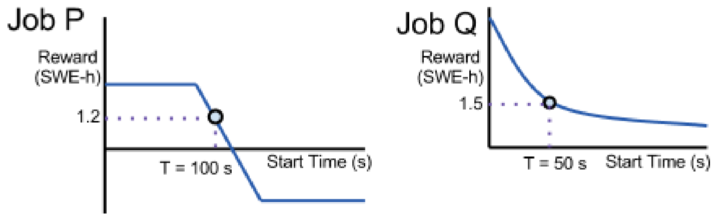


Figure 3-8: Graphs of the current location on the reward functions for two jobs with different reward functions that have been pending for different durations. Job P has an SLA of type "by-deadline," while Job Q has an SLA of type "cheap-and-simple."

Greedy with respect to soonest deadline

For jobs with SLAs of type "by-deadline," meeting the job's deadline is a huge incentive for the scheduler, as jobs provide a much larger reward before their reward function's deadline. Thus, if two jobs are of type "by-deadline," and both are still in the positive portion of their reward function, the one that is closer to the downward-sloping portion of the reward function (Job M in the case of Figure 3-9) should be started first.

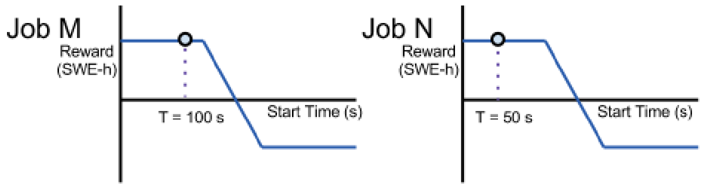


Figure 3-9: Graphs of the current location on the reward functions for two jobs with "by-deadline" SLAs.

Greedy with respect to negative rate of change

Another technique is to use the rate of change of the reward function – with multiple jobs that are about to lose large amounts of reward, the rate of reward loss can be taken into account. The job that is losing reward at a faster rate should be started earlier. In Figure 3-10, both Job R and Job S are at the "knee point" of the curve, where the reward begins to diminish sharply. Because the absolute value of the rate of change is larger after the "knee point" in Job S, Job S should be run first. A modified version of this algorithm could additionally take into account the values of the maximum penalties to decide which job to run.

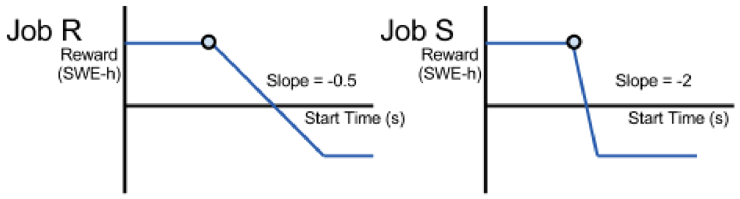


Figure 3-10: Graphs of the current location on the reward functions for two "by-deadline" jobs with differing rates of change. Job S has a larger negative rate of change than Job R.

Inner simulator modifications

While the work done here was not able to provide quantitative evidence for the effectiveness of SLA-aware scheduling, a number of steps were taken to alter the simulator to support SLA-aware scheduling, which could be built upon in the future.

1. An SLA-aware vertical was added that schedules jobs with SLAs. A sorting function was written for the queue of pending collections that depends on SLAs. It includes the logic to compare jobs with SLAs using an algorithm specified in Section 3.2.3.

2. Support for shrinking the cell size was implemented to increase contention within the cell.

3. Support was added for annotating jobs from traces with SLAs. This required adding an SLA field to the protocol buffer containing the information for a collection.

4. A shell script was created for the Monte Carlo simulator layer to interface with the Omega simulator and allow for input and output.

5. The simulator was altered to stop running upon the start of the job of interest, outputting that job's start time at that point. This was because after the simulation has output the job's simulated start time, there is no additional information to be extracted from the simulation.

6. As the job to simulate does not come from the preprocessed trace file, logic was added to inject the job with an SLA into the simulator.

As a result of these modifications, the inner simulator reached a point where based on a given injected job and starting state, a single start time value was output for visualization. However, because a single simulated value is not enough to give a job owner a thorough understanding of what might happen in a variety of situations, a discussion of how to improve the variation in start times obtained from the simulator is included in Section 4.1.1.

Chapter 4

Future Work and Conclusion

This document has described the work done to design and implement an SLA-based simulation and visualization system for Google's Omega job scheduler. Steps have been taken to build this system, though there are more features and improvements that could be used to extend the functionality of the existing system and enhance it in the future.

4.1 Future Work

There are a number of ideas for extensions that could be made to the current Omega SLA Simulator that merit further investigation.

4.1.1 Monte Carlo Simulator – Achieving variance in start times

There are a few ways to introduce variation in order to generate a meaningful distribution of job start times (see Section 3.2.2). However, the time constraints of this project prevented detailed exploration, so they are left as future work. The following are some ideas for ways to allow for simulation runs to produce variable results.

Method I: Start trace at different points

Given a single job trace, we could select a random value at any time during the time covered by the trace and replay the trace until the desired point was reached, injecting the job at that point. The fraction that is the output of the random number generator (see Section 3.2.2) from the Monte Carlo simulator layer can be thought of as a fraction of the total trace duration. The state of the cell would be different based on where in the trace the simulation started, which would lead to different loads on the cell and thus different start times for the job. A simulation of this type makes a weaker statement than desired because the simulation could only present what would happen if a job showed up at a random time within a trace, as opposed to what would happen if a job showed up "right now." The latter case is more useful to a job owner who wishes to run a job on a cell. Figure 4-1 displays the process of injecting the job into different points of the trace.

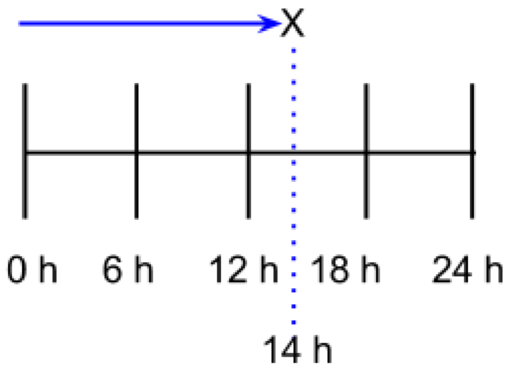


Figure 4-1: Within a single day's trace, a time (represented by $X = 14$ h above) can be chosen at random between 0 and 24 h. The trace can then be replayed until that point, at which time the job can be injected.

Method II: Use different initial cell states/traces

Randomness can be used to select which day's trace is used to replay the state. Thus, instead of choosing the time within a given trace to inject a job, as in Method I, the job is always injected at the beginning of the selected trace. The trace chosen is the trace from X days ago, where X is a number chosen from a distribution of positive numbers. The cell state and trace for the chosen day can then be replayed. Figure 4-2 shows how the day to simulate could be chosen.

This method is feasible, provided much of the setup work is done in a preprocessing

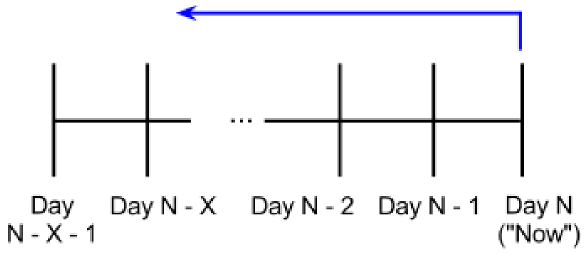


Figure 4-2: The day chosen can be a random number of days (X in the figure) behind the current day (N).

stage to collect the historical job data for many different days. The random seed would be used to select which day out of the set of possible days to try to run the job. While this method is less complex than Method I, it similarly provides a weaker result than desired, as it does not provide an estimate of the cell's current ability to start the job based on the current cell state; rather, it provides an estimate of the cell's ability to start the job were the cell usage to look like it did in the past.

Method III: Forward-looking predictions through synthesizing end times for already-started jobs

The idea in Method III is to do forward-looking predictions from "now" for an injected job. We assume that no new jobs arrive; therefore, it is only the already-started jobs that contend with the injected job. We also assume that resources are released when existing jobs end, but because no new jobs appear, only the injected job has access to the resources as they are released. These assumptions are made to simplify the simulation, as they mean that only the already-started jobs must be considered in addition to the injected job. The random element is that the expected end time of each of the already-started jobs is calculated by selecting a job duration from a distribution of expected job durations.

In order to introduce variation from run to run, we pick running times at random from a distribution for already-started batch jobs. Service jobs effectively run continuously, and as such, have no end time. A curve can be made to fit the historical run time data of previously-completed batch jobs using regression analysis. A random value can be selected from this curve to assign job timing information to each already-started job in the trace.

Ideally, we could produce a graph that would allow us to calculate the future remaining run time given the job's current running time, as in the graph in Figure 4-3.

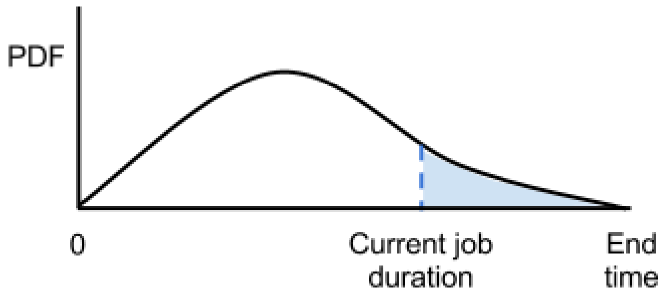


Figure 4-3: The distribution of job durations. The future run time can be predicted by looking at the portion of the distribution following the current job duration.

Implementing this would require measuring the distribution of running times for jobs. Factors such as job type (batch or service), job size, and cell history for similar jobs may be used to generate this distribution.

One issue is that the simulator does not know how long jobs that do not have a "job arriving" event (i.e., already-started jobs) have already been running. The simulator assumes that they were started at time $t = 0$ in the simulator. Therefore, it would still be possible to produce an expected total duration distribution for the jobs that can be applied as if the "already-started" jobs in the trace all started at the time of the trace's beginning. Alternatively, another way to introduce more noise to the system is to generate both the start time and end time of the job from the distribution. Then, the total time between start time and end time should be drawn from the distribution (the "duration"), and this value can be apportioned between the "elapsed time" (X) and "future time" (Y) (as in Figure 4-4). To apportion the total duration into X and Y, a random fraction between 0 and 1, p, can be used as follows:

$$
\begin{aligned}
X &= duration \times p \qquad (4.1) \\
Y &= duration \times (1 - p) \qquad (4.2)
\end{aligned}
$$

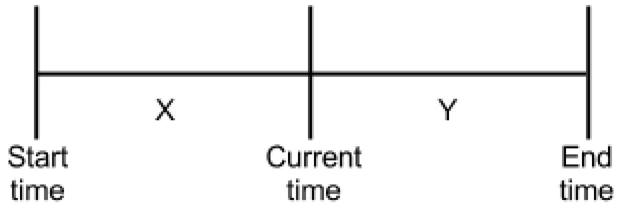


Figure 4-4: The value of the job's duration could be randomly apportioned between X and Y. The total duration is the sum of X and Y.

Updating the end times of the "already-started" jobs would be fairly straightforward because when the cell state is initialized by the simulator, the jobs of type "already-started" receive special treatment and could be assigned a synthetic end time by using a random number to select a value from the distribution of end times.

Method IV: Forward-looking prediction through synthesizing jobs to introduce into the simulation

An extension of Method III would be to synthesize additional jobs to introduce into the simulation. Doing so would populate the future cell state with competitors in the form of synthesized jobs. Factors such as the job's size (resource requirements), arrival time, and reward/penalty provided would need to be synthesized for every job using a distribution of these factors along with the random number generator.

There are a number of tradeoffs involved in implementing Method IV instead of Method III. Method IV is advantageous when trying to make a prediction of the future occupancy of the cell. For instance, if a job owner submits a batch job at noon, but the cell will be busy with other (short- and long-lived) work until 8pm, Method III would fail to take that into account, but Method IV might be able to do so because synthesized jobs could be incorporated at any point in the simulation. Thus, Method IV removes one of the limiting assumptions of Method III, that no further jobs are introduced past the initial already-started jobs. However, Method IV would also add complexity because it involves generating a new workload consisting of completely new jobs based on a multitude of inputs.

Recommendation for future investigation
Method II most likely requires the least change to the logic of the simulator code itself, and as such should be the quickest method to implement.

Method I might be the next method attempted, because unlike method II, it requires only one preprocessed trace, though it requires changing the simulator logic to inject the job at a specific location in the trace.

Finally, Method IV could be implemented to improve the quality of results; however, because it involves generating a model for the distribution of start times and changing the end times of jobs in the trace accordingly, it is more difficult and should be implemented after one of the previous two methods has produced results.

One way to validate the results of the repeated simulation would be to compare the synthesized conditions with conditions that happen later in the trace.

4.1.2 Job completion times

As suggested in Section 2.2, in addition to knowing when their jobs will start, batch job owners would like to know how long it will take for their jobs to be completed. To use simulation to gain a measurement for job completion times, a model and accompanying "what-if engine" for batch job completion times would need to be developed. Such a model would be analogous to a model previously developed by Herodotou and Babu [13] for elapsed times of MapReduce jobs on Hadoop.

4.1.3 Job admission control using SLAs

Once the job scheduler is able to use SLAs in deciding which job to schedule first, SLAs can be taken a step further: the scheduler can use the information provided by a job's SLA to decide which jobs to accept. If a given job's expected outcome yields a negative reward for the scheduler, then the scheduler should not agree to run the job, thus forcing the job owner to change the SLA to a more profitable one for the scheduler. In this way, SLAs could enable a profit-aware job admission control algorithm, similar to the work done by Auyoung, *et al.* [8]

4.2 Conclusion

This work takes a step in using SLAs as a means of encapsulating additional information about job owners' service level requirements. The Omega SLA Simulator, as designed, aims to help job owners understand what service levels they want and gain better predictability with respect to job timing. The user interface helps users visualize SLAs and reward functions and the scheduling behavior these SLAs induce. Machine-readable SLAs have the potential to improve communication between job owners and the scheduler, and it is my hope that their use will become more widespread, thus narrowing the gap between services and their users.

Bibliography

[1] Google Visualization Data Tables. http://code.google.com/apis/chart/interactive/docs/reference.html#DataTable. Accessed on October 20, 2011.

[2] Google Web Toolkit. http://code.google.com/webtoolkit/. Accessed on August 30, 2011.

[3] NextGRID: Architecture for Next Generation Grids. http://www.nextgrid.org/. Accessed on March 1, 2012.

[4] Protocol buffers. http://code.google.com/p/protobuf/. Accessed on June 17, 2011.

[5] Tokens in GWT. http://code.google.com/webtoolkit/doc/latest/DevGuideCodingBasicsHistory.html#tokens. Accessed on January 29, 2012.

[6] Hossein Ahmadi, John Wilkes, et al. Census: The SLA Stack at Google. Project Documentation (Internal to Google), June 2011.

[7] Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke, and Ming Xu. Web Services Agreement Specification (WS-Agreement). *Global Grid Forum*, 31:1–47, 2007.

[8] Alvin Auyoung, Laura Grit, Janet Wiener, and John Wilkes. Service contracts and aggregate utility functions. In *In Proceedings of the IEEE Symposium on High Performance Distributed Computing*, pages 119–131, 2006.

[9] Jerry Banks, John Carson, Barry L Nelson, and David Nicol. *Discrete-Event System Simulation, Third Edition.* Prentice Hall, 2000.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'06, pages 205–218, Seattle, WA, 2006. USENIX Association.

[11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, OSDI'04, pages 137–150, San Francisco, CA, 2004.

[12] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *Proceedings of the 6th International Conference on Network and Service Management*, CNSM'10, pages 9–16, Niagara Falls, Canada, 2010.

[13] Herodotos Herodotou and Shivnath Babu. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. *PVLDB*, 4(11):1111–1122, 2011.

[14] David E. Irwin, Laura E. Grit, and Jeffrey S. Chase. Balancing risk and reward in a market-based task service. *International Symposium on High-Performance Distributed Computing*, pages 160–169, 2004.

[15] Han Suk Kim, Sunita Verma, and John Wilkes. Interactive visual exploration of service level objectives. Technical Report CS2011-0966, University of California San Diego, May 2011.

[16] Ishai Menache, Asuman Ozdaglar, and Nahum Shimkin. Socially Optimal Pricing of Cloud Computing Resources. In *ICST International Conference on Performance Evaluation Methodologies and Tools*, ValueTools 2011, Paris, France, 2011.

[17] Asit K. Mishra, Joseph L. Hellerstein, Walfredo Cirne, and Chita R. Das. Towards characterizing cloud backend workloads: insights from Google compute clusters. *SIGMETRICS Perform. Eval. Rev.*, 37:34–41, March 2010.

[18] David G. A. Mobach, Benno J. Overeinder, and Frances M. T. Brazier. A Resource Negotiation Infrastructure for Self-Managing Applications. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC 2005)*, pages 381–382, Seatle, WA, June 2005. IEEE.

[19] Vijay K. Naik, Swaminathan Sivasubramanian, and Sriram Krishnan. Adaptive resource sharing in a web services environment. In *Middleware*, pages 311–330, Toronto, Ontario, Canada, 2004.

[20] Florentina I. Popovici and John Wilkes. Profitable services in an uncertain world. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC'05, page 36, Seattle, WA, USA, 2005.

[21] Mike Potel. MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java. Technical report, Taligent, Inc., 1996.

[22] Rizos Sakellariou and Viktor Yarmolenko. Job Scheduling on the Grid: Towards SLA-Based Scheduling. *Computer*, 16:207–222, 2008.

[23] Thomas Sandholm and Kevin Lai. MapReduce optimization using regulated dynamic prioritization. In *Proceedings of the international joint conference on measurement and modeling of computer systems*, SIGMETRICS'09, pages 299–310, Seattle, WA, USA, 2009. ACM.

[24] Xiaodan Wang, Christopher Olston, Anish Das Sarma, and Randal Burns. CoScan: cooperative scan sharing in the cloud. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SoCC '11, pages 1–12, Cascais, Portugal, 2011.

[25] John Wilkes. Utility Functions, Prices, and Negotiation. In *Market-Oriented Grid and Utility Computing*, pages 67–88. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2009.

[26] Wolfgang Ziegler, Philipp Wieder, and Dominic Battre. Extending WS-Agreement for dynamic negotiation of Service Level Agreements. *Research Report CoreGRID Project*, TR-0172, 2008.