Ninety-nine percent of computers work tolerably satisfactorily.
- John Buxton

University of Alberta

Supporting Quality of Service, Configuration, and Autonomic Reconfiguration using Services-Aware Simulation

by

Michael Smit

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

For my mother.

Abstract

Service-oriented architectures (SOA) enable interaction among multiple entities, loose composition of components, service substitution, dynamic run-time binding, and a network-driven infrastructure. These potential benefits, regardless of the technology providing them, introduce challenges: multiple organizations interacting, behavior that changes at run-time, and “black box” functionality. Service interactions can be governed by service level agreements (SLAs) specifying quality of service standards; meeting these standards is an ongoing challenge.

This dissertation advances the state of the art for configuring, deploying, and managing service systems. First, it demonstrates that authoring a simulation of a service-oriented system need not be prohibitively difficult, and that such simulations can produce a narrative that offers useful and realistic information about the predicted performance of a software system. This is in contrast to the results of a systematic survey of current frameworks. A framework is described and implemented that improves on the state-of-the-art in key areas.

Second, this approach is used to simulate two real-world service systems. They are validated to accurately predict performance, and serve as testbeds for demonstrating simulation-driven methodologies.

Third, a novel view on how service level agreements are negotiated, deployed, and evaluated is described. A simulation-driven methodology and tool allows consumers to explore trade-offs among configuration goals, based on a desire to produce an SLA that maximizes perceived value for the consumer and the provider. Another simulation-driven tool answers questions posed by administrators seeking configurations that will adhere to an SLA. A third tool enables run-time testing and monitoring of a service system. The tools are implemented and tested in both simulated and real scenarios.

Finally, an autonomic manager capable of re-configuring an application at run-time is presented. A decision model is created before the service is deployed by running the simulation (either manually or automatically), collecting traces of performance, and constructing a state-transition model that identifies the (abstract) states of an application and the transitions among them. This is implemented and tested both in simulation and in a real-world cloud computing environment. Questions about the granularity of the abstract states and the size of the state space are asked and answered using empirical results.

Acknowledgements

I could not have completed this document or this degree without the help and support of many people. The credit for this is theirs; the blame is mine.

My wife Ann provided unfailing support and patience, for which I am very grateful. My family was very supportive, despite not really being 100% sure what I do exactly. I lack the space to say all the wonderful things that need to be said about my mom and dad.

My supervisor Eleni Stroulia was everything you could want in a supervisor and more. Her insight and wisdom, her intuitive understanding on when to encourage and when to excoriate, her genuine care and concern for her students, her ability to identify opportunities for others, and in general her approach to mentorship are qualities I will strive to emulate.

IBM CAS provided funding and introduced me to Marin Litoiu and Gabriel Iszlai who were very helpful in starting this project. NSERC and the University of Alberta also provided funding at various points.

I am grateful to Nelson Amaral for all I learned from him in my CMPUT 603 TA experience, lessons that continued with Russ Greiner.

Andrew Nisbet and Hyeon Rok Lee devoted many hours to helping with the implementation of some of the ideas in this document. Thank you!

Of course, this dissertation is not my sole achievement at this university, and I'd have no hope of naming the people who have worked with me over the years or the experiences I value. My friends and colleagues helped make my time here more enjoyable; I don't name you all here, but you know who you are. The staff in the department are invaluable and it saddens me to see how many are not available to provide students the same attention and experience they offered me - Sheryl, Karen, and Edith were especially important to my experience.

Interested readers should also consult the acknowledgements page of my Masters dissertation. In helping me achieve that milestone, the people acknowledged there contributed to this milestone as well, particularly Jacob Slonim, Mike McAllister, and Kelly Lyons who have continued to be valuable mentors.

Contents

List of Figures

List of Tables

List of Acronyms

ACUS	Autonomic Configuration using Simulation
ANOVA	Analysis of Variance
API	Application Programming Interface
AWS	Amazon Web Services
BPEL	Business Process Execution Language
BPMN	Business Process Modelling Notation
CDS	Content Delivery Service
CDS-M	Content Delivery Service Manager
CDS-D	Content Delivery Service Depot
CGI	Common Gateway Interface
DAML	DARPA agent markup language
DARPA	Defense Advanced Research Projects Agency
DDSOS	Dynamic Distributed Service-Oriented Simulation Framework
DEVS	Discrete Event System Specification
DEVSJAVA	Java implementation of Discrete Event System Specification
DFS	Depth First Search
DMS	Device Manager Service
FIFO	First-in, First-out [queue]
GB	Gigabyte
GHz	Gigahertz
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IDDFS	Iterative-deepening Depth First Search

IO	Input/Output
IT	Information Technology
JIML	Jimmie Markup Language
JIMMIE	Just Imagine Many Many Interesting Experiments
MAPE	Model, Analyze, Plan, Execute
MB	Megabyte
OASIS	Organization for the Advancement of Structured Information Standards
OS	Quality of Service
OWL	Web Ontology Language
PRISM	A probabilistic model checker
PSML-S	Process Specification and Modelling Language for Services
QoS	Quality of Service
RATER	Reliability, Assurance, Tangibles, Empathy, and Responsiveness
REST	Representational State Transfer (*e.g.* HTTP)
SASF	Services-aware Simulation Framework
SDI	Software Distribution Infrastructure
SERVPERF	A service quality assessment method
SERVQUAL	A service quality assessment method
SLA	Service Level Agreement
SLM	Service Level Management
SLOC	Source Lines of Code
SOA	Service Oriented Architecture
SOAD	SOA-compliant DEVS
SOAP	Simple Object Access Protocol (no longer an acronym)
SOPM	Service-Oriented Performance Modelling
SQL	Standard Query Language
SSL	Secure Sockets Layer
STM	Service Testing Module
SUB	Satisfactory, Unsatisfactory, Boundary state descriptors

TAPoR	Text Analysis Portal for Research
TCA	Tivoli Common Agent
TPM	Tivoli Provisioning Manager
UI	User Interface
UML	Unified Modeling Language
URI	Uniform Resource Identifier
WS	Web Services
WSDL	Web Services Description Language
XML	eXtensible Markup Language
XSD	XML Style Document
XSLT	Extensible Stylesheet Language Transformations

Chapter 1

Introduction

Service-oriented architectures and flexible, scalable computing infrastructures promise useful functionality such as interoperability across organizations, software service systems that can be deployed and run quickly, managed infrastructure, and alignment of business processes with technical implementations of those processes. Service-oriented architectures are loosely coupled, componentized, and standards-based. This composability, interoperability, and standardization enables re-use and the seamless replacement of one service with another offering the same interface. SOA is a potentially flexible architecture that enables interaction among multiple service providers and service consumers, creation of complex functionality by composing several simpler components, and a network-driven infrastructure.

The current state of the art for providing this functionality is services-oriented architectures and cloud computing. Regardless of the ability of these specific technologies to meet their promises, this functionality will continue to be sought.

This functionality introduces challenges inherent to complicated systems: there are multiple organizations involved, the behavior of the system is difficult to predict based solely on the behavior of the individual components, and technical decisions may be dictated by business decisions made without consideration of technical challenges. Generally speaking, flexibility leads to complexity in decision-making. Complexity cannot be removed from a closed system; it can only be moved from one component or stage to another. By allowing organizations to integrate software systems, the complexity is transferred from information sharing to creating and implementing the agreements that govern such interactions. The missing step is hiding that complexity from non-technical decision makers by offering tools and methodologies that simplify this process. A flexible computing infrastructure poses the challenge of actually configuring and managing that infrastructure in the face of changing business requirements and technical abilities. Distributed, loosely-coupled software systems solve interaction problems and can reduce development time, at the cost of making configuring, deploying and testing the software system more complex.

1.1 Research Problem and Objectives

Research in configuration management has attempted to address the challenge of configuring software of increasing complexity. The typical approach to configuring a deployed software service system involves a trial-and-error reactive approach loosely based on a set of application-specific best practices. There is some expertise in the area of configuring at the systems level - servers and networks - but application-level configuration expertise is domain- or application-specific when it exists at all. The result is an expensive and time-consuming process of configuring a deployed service, or maintaining a sophisticated replica testing environment, or investing in an emulation environment to test possible configurations. In the case of a multiple-entity service composition, such testing may be impossible or prohibitively expensive.

Another way to meet the configuration challenge is self-managing (autonomic) systems. A key challenge for such systems is producing a decision model - somehow translating raw metrics monitored from a service system into actions that re-configure the system (or, inaction). An approach more involved than simply making changes when a threshold is exceeded is desired; in particular, an approach that is able to make pro-active configuration changes to avoid thresholds completely. There is general distrust of many decision models because they are difficult to understand - the model is seen as a black-box oracle that produces re-configuration actions that are difficult to understand.

Given that service providers have difficulty understanding and configuring their own services, it may be surprising that service consumers are expected to participate in establishing agreements specifying minimum performance standards for the services they use. Research shows that consumers who are able to experience a product are better able to assess its quality and make decisions about cost versus quality trade-offs. Typically this can't happen until a service is deployed - after which changes are more expensive and more disruptive.

A solution that addresses each of these problems is to produce a simulated version of the target service system, then use simulation-generated data to assist in configuring services, to construct decision models for self-managing systems, and to inform and educate service providers and consumers. Unlike a pure analytic model, simulation can predict a narrative for a service system in a given scenario over time, making the results explainable and understandable. The current state of the art in simulating service-oriented systems is a) more focused on composition bottlenecks instead of on the QoS attributes of each component of a service system, and b) less focused on tools to help communicate a plausible narrative (visualizations, metrics generation, probabilistic models). Existing solutions also require substantial modeling and/or development effort and expertise specific to the simulation tool.

The aim of this dissertation is to absorb some of the complexity of configuring, deploying, governing, and managing service systems. The four research objectives are as follows. First, to demonstrate that authoring a simulation of a service-oriented system need not be prohibitively difficult, and that such simulations can produce a narrative that offers useful and realistic information about the predicted performance of a software system. Second, to use this approach to produce simulated versions of real-world service systems using real performance data. Third, to show that simulation-driven tools can be used to help manage the governance of software systems throughout the cycle of negotiating standards for service performance, configuring the service to meet those standards, and evaluating and monitoring ongoing compliance with those standards. Fourth, to demonstrate that a decision model generated in simulation can be used to reason about a real-world software system, to the point that such reasoning can be trusted to re-configure the service at run-time.

1.2 Achieving the Research Objectives

In support of these objectives, a series of contributions advancing the state of the art in understanding and managing services are described.

To achieve the first objective, a *standardized set of dimensions and characteristics* is used to describe and compare existing service simulation frameworks in a *systematic, comprehensive literature review.* No such review or set of characteristics existed previously. To address the identified shortcomings, a *simulation framework and prototype implementation* that improves on state of the art simulation frameworks is described. The framework excels at producing a narrative of the predicted performance of a service system based on past exemplars. It achieves this by offering a powerful and extensible engine for collecting and visualizing metrics and the ability to generate a simulation from an existing standardized description of the service, reducing development effort. Integration with real-world components enables simplified validation in real situations. The unique ability to systematically run large numbers of varied experiments simplifies generating volumes of performance data.

The second objective is achieved by using this framework to produce *simulations of two real-world service-oriented systems*, which have been demonstrated to accurately predict performance-related metrics. The resulting simulations produce narratives of predicted system performance over time for a given configuration. One of the simulations is statistically validated and used as a test-bed for later contributions; in cases where simulated implementations are tested in the real world, the simulation is shown to be an accurate predictor of their behavior.

In support of the third objective, a *novel view on the lifecycle of a Service Level Agreement (SLA)* is described. It improves over traditional perspectives by learning from known problems with SLAs and incorporating research from other disciplines on perceived quality and value. This view of the lifecycle is supported by a series of novel methodologies and tools. First is a *novel simulation-driven approach to sharing information, exploring trade-offs, and increasing understanding of a service while negotiating desired service levels.* This is motivated by an understanding of perceived value and the problems with SLA negotiation and the need to translate low-level metrics to higher-order constructs that assess value. A tool implementing the approach is also presented. The second contribution to SLA lifecycle support is an *innovative simulation-driven question-answering methodology and tool* to assist administrators responsible for configuring and deploying a service system to meet a service level agreement. Finally, a state-of-the-art tool allows *run-time testing and monitoring of a deployed service system* to ensure its compliance with a service level agreement by using the emulation capabilities of the simulation framework. The focus of the SLA management is on measurable (or approximable) qualities with defined thresholds, which are sometimes called Service Level Objectives (SLOs), rather than the more vague textual descriptions offered by SLAs. The term SLA is used generically to refer to any workable expression of the goals of the SLA, even if they take the specific form of SLOs. From a consumer standpoint, the exact form the SLA is expressed in is less important. Certain qualities described within the SLA - *e.g.*, privacy, security, usability - are more difficult to quantify and the current tools and methodology will not support them. Other qualities - *e.g.*, availability, reliability - can be managed using the approach described here, though the motivating example used throughout this work is capacity planning.

The fourth objective is achieved through an *abstract decision model in the form of a state-transition model, constructed from simulation-generated data*, useful for identifying the state of an application and predicting future states. No existing decision model generates a state-transition model from simulated data. A *novel method for creating a self-managing software system* is described, using the decision model to make configuration and re-configuration decisions based on predicted futures. This is implemented and tested both in simulation and by using the self-manager to make configuration changes in a real-world cloud computing environment. Finally, a novel approach (and implementation) *systematically explores the state space of an application to construct a decision model automatically.* This leads to an assessment of what level of granularity offers the best abstraction of continuous data to produce the best translation between simulated environments and the real world.

1.3 Organization of this Dissertation

The existing state of the art is described in Chapter 2, including a background on services; a comprehensive literature review of simulation support for service oriented software; an introduction to value-related concepts like perceived value, perceived quality, and perceived cost; and current research in capacity planning and configuration management and self-managing software. A novel contribution is provided in the form of a set of characteristics useful for characterizing and comparing simulation frameworks for service-oriented systems (§2.2).

Chapter 3 introduces the Services-Aware Simulation Framework (SASF), a novel contribution designed to create a virtual model of a service oriented system. SASF improves on existing solutions by offering automatic generation of simulations from existing data about the service, a powerful approach to recording & visualizing simulation-generated metrics, support for integrating with real-world components, and an extensible library of common service tasks. SASF uniquely offers accurate replication of performance characteristics of services, an API and a user interface for interacting with a running simulation, and an innovative language and tool to systematically modify simulation configuration files and execute simulations.

In Chapter 4, SASF is used to simulate two service systems. The first, a proprietary enterprise-level system that includes SOA-based interfaces, is used to demonstrate simulations at a higher level of abstraction with less library support. The simulation models a distributed architecture used to publish files to thousands of remote endpoints. It is shown that the simulated version produces results comparable to the real-world system (§4.1 and §4.2). The second is a text-analysis tool that provides a public web services interfaces. It is used to demonstrate automatically generating a simulation with a one-to-one mapping between simulated components and real-world components (§4.3 and §4.4). Its CPU-bound operations are complex and interesting operations, but the relatively small number of these operations make it more manageable as a case study. This simulation, called TAPoRsim, is used as a platform for demonstrating simulation-driven methodologies in the remaining chapters.

Chapter 5 is organized around the first contribution, a novel formulation of the life cycle of a Service Level Agreement (SLA). These documents govern the interactions between a service provider and a service consumer; they are cyclic because they should be re-negotiated and re-evaluated periodically as business and technical needs change. The second contribution draws on an understanding of perceived value to describe the problems with SLA negotiation and identify a simulation-driven approach to sharing information and increasing understanding of the service (§5.1). The third describes a simulation-driven approach to answering specific questions posed by those responsible for configuring and deploying a service system that complies with the SLA (§5.2). The fourth contribution describes a SASF-supported approach to testing and monitoring a deployed service system (§5.3). Each contribution is implemented and demonstrated using TAPoRsim.

A novel method for creating a self-managing software service system is described in Chapter 6. Configuration management is a complex task, even for experienced system administrators, which makes self-managing systems a desirable contribution. The first contribution is an abstract decision model in the form of a state-transition model, useful for identifying the state of an application and predicting future states (§6.1). A second contribution uses this model as the basis for self-management features that use predicted futures to make decisions on configuration changes (§6.2). This is validated both in simulation and by using the self-manager to make configuration changes in a real-world cloud computing environment. Third, an approach and an implementation to systematically explore the state space of an application to support constructing a decision model (§6.3). Finally, the granularity required for such a decision model to be effective is described and tested empirically.

The contributions are summarized and future work is presented in Chapter 7.

Chapter 2

Background and Related Work

Chapter Contents

This chapter introduces background, terminology, and the current state of the art. Given the breadth in the areas touched by the contributions described in this work, the various subsections cover a diverse range of topics, ranging from industry standards to simulation to autonomic computing to marketing literature.

Basic services terminology is introduced in §2.1, including how service provision can be governed by Service Level Agreements (SLA) and how typical implementations exhibit shortcomings (§2.1.1). Services are a recurring theme throughout this work, but this section is most relevant to Chapter 3 where a simulation framework for services is described, and to Chapter 5 where some of the shortcomings of SLAs are addressed.

§2.2 describes a set of characteristics for characterizing, describing, and comparing simulation frameworks, with a focus on simulation frameworks used to simulate service-oriented or componentized software systems. This set is a novel contribution, described here as it is important to §2.3.

In §2.3, the results of a comprehensive survey of simulation frameworks for service-oriented and componentized software systems are presented. The systematic survey identified 6 specific frameworks; each is described and characterized using the characteristics defined in §2.2. Finally, a discussion guided by the characteristics shows while each framework has strengths in several dimensions, the framework presented in this work meets several previous unmet requirements and broadly covers the desirable aspects of a simulation. This survey is most relevant to the description of the services-aware simulation framework (Chapter 3).

An introduction to concepts like perceived value, perceived cost, and perceived quality follows in §2.4. Included is a discussion on whether an implementation of a service-oriented architecture should rightly be considered a product or a service (§2.4.1). A survey of marketing literature reveals how individuals evaluate value and service quality when making purchasing decisions. This background is important to the discussion of an SLA creation methodology in §5.1 and §5.1.1.

Selected state-of-the-art work on configuration management and capacity planning is presented in §2.5. Provisioning sufficient resources to meet expected demand is most relevant to the question-answering methodology contribution that is intended to help system administrators translate customer non-functional requirements to a configured software system (§5.2.1).

Autonomic computing related work is reviewed in §2.6. The goal of an autonomic system is to be self-managing; the more specific focus is on self-configuring software systems. The archetypical architecture is presented, and the areas advanced by this work are identified. This section is most relevant to the simulation-based autonomic computing methodology presented in Chapter 6.

2.1 Services Overview

Service-oriented Architecture (SOA) is a relatively new software architecture formalization. It has been embraced by academic researchers and companies. It's primary defining characteristic is meeting needs with capabilities: broadly speaking, units of functionality are discovered and loosely-coupled to meet a need. The building blocks of a Service-oriented Architecture are services. In the context of service-oriented computing, a service is a mechanism for accessing a capability via a well-defined interface [10].

While this definition is implementation-agnostic, a common software implementation specification is Web Services (WS), which offers a well-defined interface to server-side software and a suite of standards[1]. This narrows the definition to a network accessible endpoint that communications using XML standards. A Web service is "a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP[2]-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards" [20].

A *binding* describes how to move from a specification to implementation; it is an "association between an interface, a concrete protocol and a data format. A binding specifies the protocol and data format to be used in transmitting messages defined by the associated interface" [20].

An *operation* is an action supported by the service. A service is capable of providing at least one action and potentially more. For example, a car rental service might have two operations: searching for a car to rent, and booking a car to rent.

Another specific form of service is a RESTful service. Such services use HTTP

[1]These standards, sometimes called the WS-* standards, are agreed to by major corporations, freely available online, and will not be discussed at length here.

[2]SOAP is officially no longer an acronym, it is a term unto itself; the original meaning was Simple Object Access Protocol.

without a SOAP message wrapper to transmit messages. The responses are usually still in a standardized form: XML or JSON are common. One school of thought considers RESTful services to be architecturally different, to be resource-oriented rather than service-oriented. The terms and language used throughout this document are those used for Web Services; this is done without loss of generality and without endorsing one type of service over the other. The implemented tools do have dependencies based on WS technologies; including RESTful services is not a methodological question, but rather an implementation issue.

Services can be *composed* to provide more complex functionality. Composed services can be managed by choreography (where the services are told how to work together, and possibly intermediaries are created to help the two communicate, but there is no central control) or orchestration (where a central controller guides the sequence of services and transforms data as needed). The specification of how services are composed can use the *Business Process Execution Language (BPEL)*, which "defines a model and a grammar for describing the behavior of a business process based on interactions between the process and its partners" [29]. It is a script multiple partners can follow to ensure execution of their shared business process.

Web services allows for multiple-entity compositions. While not part of the WS-* standards, *Service Level Agreements (SLAs)* can be used to govern these interactions. A service level agreement (SLA) is a contract between two parties promising a certain level of service. The level of service expected can be technical (response time, CPU load) or non-technical (time to helpdesk problem resolution), but is typically measurable. There may be penalties for dropping below this level of service.

2.1.1 Service Level Agreements

Service Level Agreements (SLAs) can be used to govern interactions between service providers and service consumers. A service level agreement (SLA) is a contract between two parties promising a certain level of service. The level of service expected can be technical (response time, CPU load) or non-technical (time to helpdesk problem resolution), but is typically measurable. There may be penalties for dropping below this level of service. The items in a service level agreement are called Service Level Objectives (SLOs).

One type of SLA is static and used for all customers, like those dictated by service providers (*e.g.* Amazon EC2). One can be a customer or not; this is the extent of "customization" available. Metrics are recorded by the consumer and must be reported to and validated by the provider in order for the penalty clause of a 10% refund to take effect. The other type are individual and specific based on the needs of the consumer and capabilities of the provider, in which case they are negotiated and include organization-specific guarantees. Typically these service providers have an SLA template which can be tailored to the needs of the customer. This tailoring can happen in a series of meetings prior to beginning the service experience; there are also methods for negotiating SLAs automatically at run-time (*e.g.* [65, 44, 85, 35]).

Services covered by SLAs typically have reporting requirements so both the provider and consumer can compare actual performance to target performance. These reports are typically high-level; for example, "green" means a service level objective was met, “red” means it was missed, and “yellow” means that there was trouble meeting the objective.

Although SLAs are the current standard practice, they do not always ensure customer satisfaction. Blomberg [6] conducted a study of interactions between consumers and providers. She identified five problems with how SLAs were used in those interactions, three of which are particularly relevant. A 2008 Forrester Research study [19] and a paper identifying SLA principles and best practices by Fitsilis [18] support her conclusions and offer additional problems, as follows:

- Information is difficult to understand. A Forrester study [19] concluded most SLAs are defined in technical terms not accessible to business users. Fitsilis [18] emphasizes the difficulty in mapping service levels from low-level measurable information to higher-level meaningful information, reporting that only technical specialists understand SLAs. He also emphasizes the importance of changing this: the first two best practices identified are “service level definitions should be business-based, meaningful to the users...” and “service level definitions should be easily defined and measurable”. The author also recognizes that “the ultimate measure of service-level performance is customer perception and satisfaction”, but that this end-user experience is difficult to measure.

- Satisfied SLA metrics don’t mean satisfied customers. The SLA reports are useful in the first few months as the provider adjusts service delivery to ensure the metrics are “green”. However, once the service delivery is satisfied, they appear to become less important, and may not indicate customer satisfaction. A 2008 Forrester study [19] refers to “misalignment”, the distance between consumer expectations and what the provider is trying to achieve. In pointing out that SLAs often aren’t actually agreements, the report claims “... at the end of a budget period, business managers can say ‘it was not good enough for me to do my business’, even if the IT service levels were met on a mathematical basis from the IT point of view.” They advocate for improved understanding of higher-level requirements. Fitsilis [18] reports that requirements are often poorly specified and difficult to enforce.

- Information is hoarded. Providers are reluctant to provide unfettered, unnuanced access to performance data. This sometimes extends from an honest desire to provide meaningful interpreted performance data that has been analyzed and summarized for the client. A reasonable theory, though one as yet unsupported by evidence, is since SLAs are legally binding contracts, the provider may hesitate to give clients access to information that could be used to enforce penalty clauses if not properly “interpreted”.

- SLAs are not proactive enough. Clients remarked that they wanted more from their service providers: a proactive approach to help the client identify their needs and proactively meet them. As Blomberg points out, this requires a broad sharing of information that may not always be possible or popular. It also requires technical staff to have the ability to elicit requirements from business users, and these two groups may not be able to communicate on the same level. Fitsili [18] remarks that by only penalizing performance below

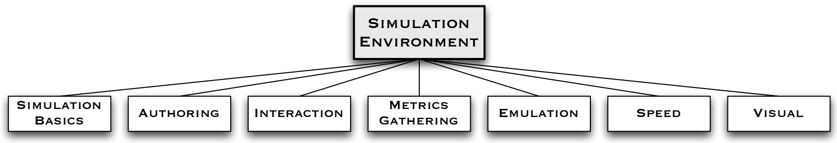


Figure 2.1: The overall Simulation Environment properties.

a minimum standard, providers are incentivized to strive for minimum levels and no better.

- SLA obligations are not met. Forrester [19] reports that SLAs are unmet 75% of the time, and suggests the problem is that IT has moved from managing servers and networks to managing the applications (or at least the middleware) that runs on them.

Chapter 5 describes a set of contributions that improve on the challenges with SLAs above; respectively:

- Simulation-driven metrics visualization relates low-level metrics to higher-level constructs. The tools give consumers a chance to perceive quality in simulation before deploying to a real environment.

- The lifecycle proposed involves periodic re-evaluation of SLAs. Simulation allows customers to view a narrative of the expected performance of the service before it is deployed. The configuration question answering tool helps service providers align their configurations to service level obligations.

- By making information easier to understand, the incentive to hoard information is minimized. Substantial simulation-generated data can help satisfy the desire for more information or prepare customers to receive larger volumes of information from providers.

- The SLA negotiation tool enables an approach where the SLA is co-created to maximize value to the client and help resolve trade-offs to the satisfaction of the customer.

- The question-answering tool helps translate SLA requirements to configuration values. The evaluation and monitoring tool simplifies periodic testing of the service to ensure SLA obligations are being met. Additionally, Chapter 6 describes an approach to automatically re-configure a service system to meet SLAs in rapidly changing environments.

2.2 Characteristics of Simulation Frameworks

To better classify simulation frameworks and compare them to the framework proposed in Chapter 3, the following set of simulation framework characteristics is defined:

1. The first interesting characteristic feature of a simulation framework is its **purpose**. Generally, a framework has one of two general objectives: (a) the

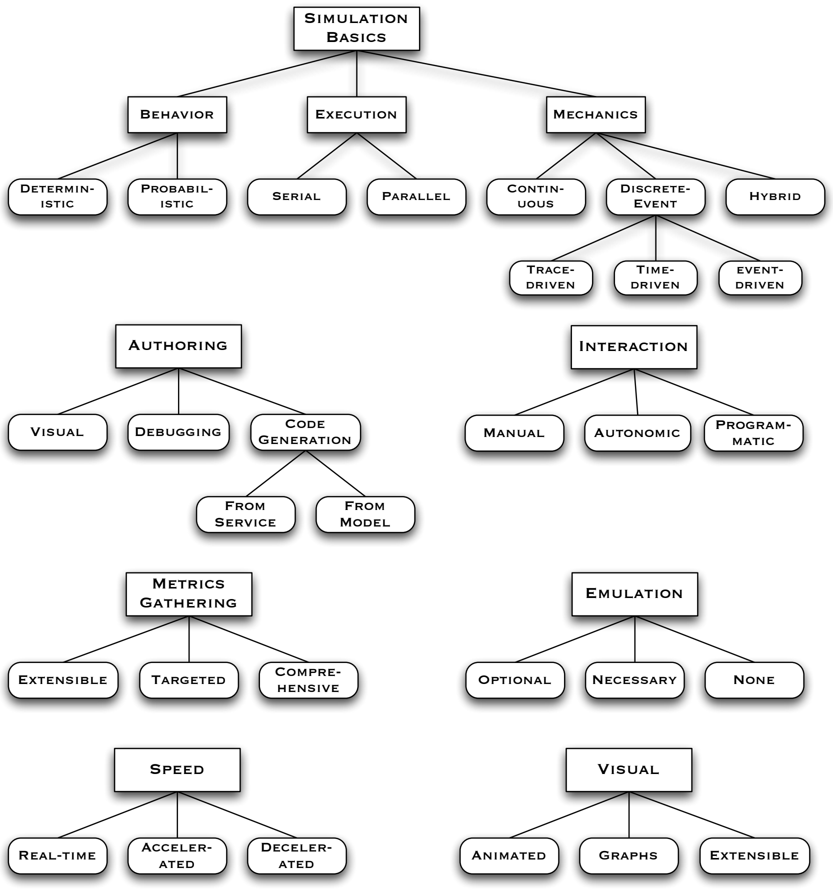


Figure 2.2: The details of each category in the Simulation Environment. The leaf nodes are properties.

examination of the system behavior in order to identify possibly undesirable interactions in the composition of the system constituent services, and (b) the analysis of the system performance.

2. The software design and functionalities of the **simulation environment**. An overview is in Figure 2.1 with the details of each branch in Figure 2.2.

   - Basics: Based loosely on Sulistio *et al.* [73], this covers the fundamentals of the simulation. Continuous simulations are capable of tracking a system through any point of time, usually based on differential calculus. Discrete-event simulations track a system based on specific moments in time (time-based), as events occur (event-driven), or based on traces from real-world applications. A deterministic simulation will produce the same output for a given set of inputs every time it is run; stochastic or probabilistic simulations use probability distributions and will produce output that varies based on the given distributions. Simulations can run in parallel, serially, or distributed (perhaps in a service-oriented architecture).
   - Authoring: Typically a simulation is implemented using a language or by programming using an API or library. Some simulation frameworks aim to ease the process of simulating a system by offering authoring assistance such as visual simulation authoring, generating code automatically based on the model or on the service itself (e.g. from WSDLs). Some simulation development environments offer tools intended to debug simulations.
   - Interaction: Once a simulation is running, can the configuration, simulated entities, or other components be modified by a user (manual), by the system itself (autonomic), or via an API (programmatic)?
   - Metrics gathering: Real-world systems can be instrumented to produce a variety of data. While simulations can be created to generate many types of data, often the focus is on a particular type of data (targeted). Others collect every metric generated by the simulation (comprehensive). Frameworks may be easily extensible to collect other metrics (for example, via an API).
   - Visual Interface: Simulation results are usually reported in some textual representation during the simulation run, or upon its completion. In addition, some frameworks include a visual component to visualize the simulation or the metrics with either static images (default) or animated images. Also of interest is the ease with which the framework can be extended to improve, augment, or annotate the visualizing functions.
   - Emulation: Some simulations offer the ability to interact with real-world services or entities. Others could require it as they cannot function on their own (for example, a simulation of a service broker might rely on real-world services).
   - Speed: Simulations can run faster than real time, in real time, or theoretically slower than real time (perhaps due to computation required).

3. Different simulation frameworks adopt different **modelling** languages, abstract representations, or formalisms for specifying the systems under examination. Each of these languages may make explicit (or alternatively, ignore)

different aspects of the system's constituent services, their composition, their underlying infrastructure, their behavior and performance, *etc.*.

4. Finally, each of the examined simulation frameworks has been **evaluated** or validated, by simulating real-world systems, by simulating toy problems, or by proofs.

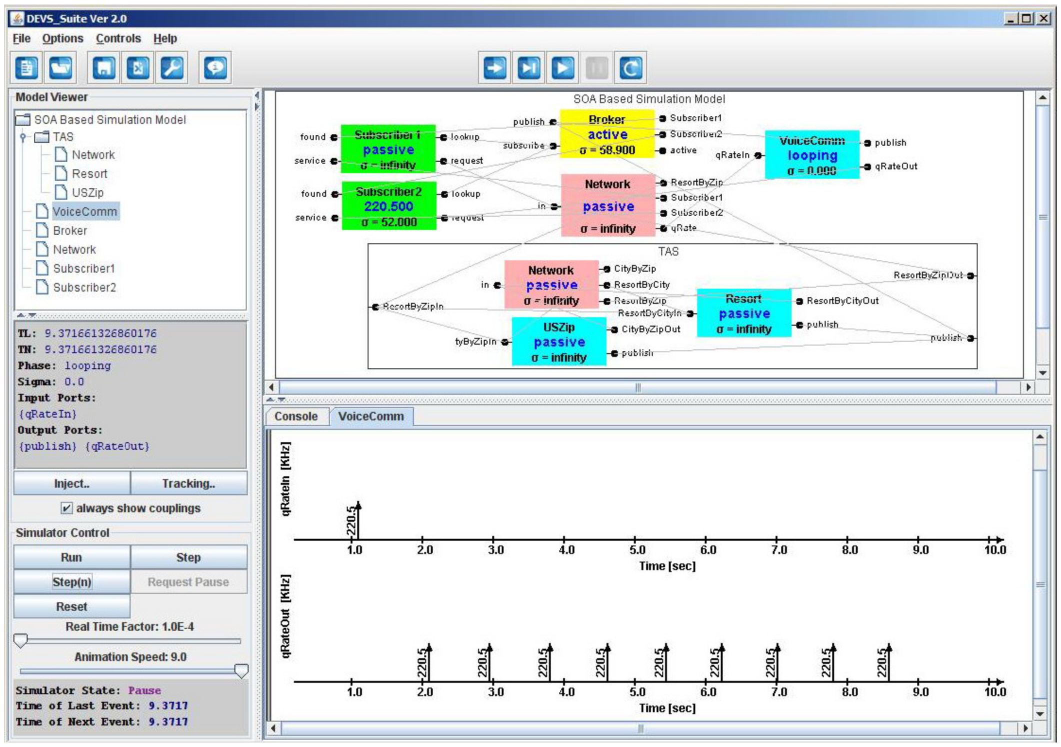
2.3 Simulation Frameworks for SOAs

Simulation involves building a model of an entity or phenomenon, implementing that model, and running experiments by way of executing the implementation. The purpose of a simulation is to gather information from the experiments to understand the entity, to reason about it, or to evaluate it. In support of computer-based simulations, there exist many simulation languages, tools, methodologies, applications, standards, and frameworks at varying levels of generality. Specific frameworks can reduce implementation time and effort by providing functionality specific to a domain in libraries, but have less flexibility. On the other hand, general approaches allow the user to gain proficiency in a methodology and with a suite of tools that can be used for multiple purposes. Here, a *simulation framework* is a set of software components and tools intended to specify, execute, visualize, and analyze simulation implementations reflecting a range of systems within a domain.

Services, as described in §2.1, are challenging to understand and reason about given the possibility for multiple participants, complex deployments, and composition. Similarly, deploying and testing service-oriented systems is time-consuming and expensive. A contribution described in Chapter 3 includes a simulation framework for service-oriented systems that addresses the limitations of existing frameworks. This section discusses other frameworks in detail, and characterizes each of them using the dimensions described in §2.2; see Tables 2.1, 2.2, and 2.3. A discussion of the frameworks is provided in Section 2.3.7.

The focus here is specific frameworks that support the simulation of service-oriented or, more generally, componentized software systems. More general frameworks and tools can also be used to simulate such systems, but more general approaches are covered in other surveys (no single survey could cover the depth and volume of all simulation support software; see for example [49, 61, 74, 42]). As the term "service-oriented simulation frameworks" is already used to refer to simulation frameworks that are implemented based on service-oriented principles, the term "framework" will be used to refer to those frameworks that are within scope of the survey, unless specified otherwise.

A systematic search for frameworks was conducted using the ACM Portal, IEEE Xplore, Google Scholar and Springer Link. The search terms were "services simulat(e|ion|ing|or)", "software component simulation", and "service-oriented simulat(e|ion|ing|or)". The results were pre-filtered using the scope defined here, by reading titles and abstracts. A total of 16 candidate papers were identified. These papers were read in sufficient detail to either exclude them from the survey or complete a detailed reading and incorporate them. For each the list of references was examined for other papers to be included in the survey; this process resulted in three additional candidate papers. Ultimately, the list was narrowed to six frameworks, described here.



```
TL: 9.37166132686017б
TN: 9.37166132686017б
Phase: looping
Sigma: 0.0
Input Ports:
{qRateIn}
Output Ports:
{publish} {qRateOut}
```

```
Simulator State: Pause
Time of Last Event: 9.3717
Time of Next Event: 9.3717
```

Figure 2.3: The DEVS-suite visual representation of a simulation (from [33]).

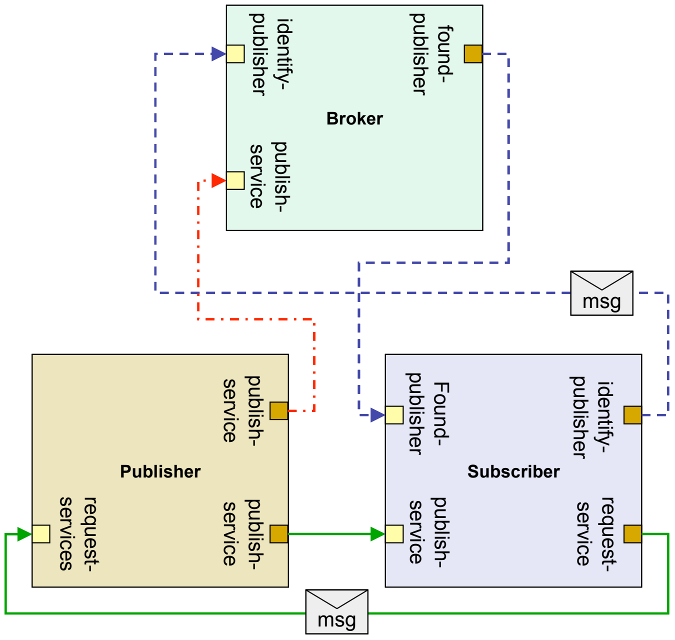


Figure 2.4: The atomic models of Sarjoughian's DEVS-based model of SOA (from [67]).

2.3.1 SOAD and DEVS

The Discrete Event System Specification (DEVS) [80] is a formal language for modeling system structure and behavior. Its behavioral model defines states, events (inputs), and outputs, where each state has a lifespan, and transitions occur in response to inputs or the end of a states' lifetime, and optionally generate output. A component modeled in this fashion is considered atomic, but can be hierarchically coupled together with other atomic components via similar formalism. These hierarchical DEVS models can be simulated [81]. Since it was introduced, various extensions, simulation engine implementations, and variations have been introduced. Implementations include DEVSJAVA [66] and DEVS-suite [33]. The latter offers visual construction of simulations, as well as observation of and interaction with the simulation using a GUI (Figure 2.3).

Sarjoughian *et al.* propose an extended version of DEVS called SOAD [67] which maps service-oriented concepts to the existing component-based model, and describe a general simulator based on that model. Their goal is to verify logical correctness of the service composition in terms of its throughput, timeliness, accuracy and quality. The majority of the modeling and simulation are based on the existing DEVS formalism and tool sets, with some extensions for SOA concepts for which there is no direct translation; for example, composition is modeled as a series of nested hierarchies. They build their simulator on DEVS-suite.

They consider a "simple" Web Services scenario, with composed, loosely-coupled services that communicate using SOAP over a "simple" network hardware model (which includes delay and transmission volume). They use three atomic components: producers, subscribers, and brokers. Figure 2.4 (from [67]) shows the communication links between the atomic elements. Metrics are primarily QoS metrics, and are recorded and visualized by DEVS-suite in an animated time series. To validate, hard-code deterministic values are used [67]: the simulator is implemented and used to model a fictional travel-planning service composition with deterministic timings. The validation is declared successful because the output metrics whose expected value is obvious based on the fixed input metrics have the correct value; the fact that the composition produces output indicates the validity of the composition. There is no integration with real-world implementations, and no metrics are generated.

Related work called DEVSI4WS from Seo *et al.* [70] uses a similar translation to a DEVS model and implements a simulation in DEVSJAVA (which includes a visual simulation builder). The focus is on service composition in the context of BPEL: sequentially calling services based on identifying common primitive data types in the WSDLs of two or more services to identify valid BPEL compositions. Functions of BPEL other than message sequences are not implemented. Their model uses WSDLs and SOAP messages. Matching is based on the name of the parameter in the WSDL message specification or the specification of complex data types. The simulation is based on the flow of data through the composition and not on performance or QoS metrics. A proof-of-concept composition of services related to purchasing and registering a car is presented. This example is also not validated using real-world systems.

2.3.2 DDSOS

The Dynamic Distributed Service-Oriented Simulation Framework (DDSOS) is the work of Tsai *et al.* [75, 76]. Their goal is to support the development of service systems by testing compositions in simulation. The approach begins at design-time, when a prospective service-oriented system is modeled using PSML-S (Process Specification and Modeling Language for Services) [77]. They intend for this language to provide the same features for SOA design that UML provides for object-oriented design (there is no translation support between the two). At the composition-level, the relationships can be specified visually; the service details and specifications are modeled manually. The PSML model allows for model-checking of an SOA design, as well as the generation of test suites. From the PSML-S model and a simulation configuration file, code is automatically generated and deployed to a distributed, multi-agent simulation engine, where it can be validated using the test suites. [76] is a thorough explanation how how SOA concepts map to PSML concepts, and how the result is simulated.

The simulation engine is a discrete-event, event-driven, distributed, extensible, engine. It is itself a service, which processes incoming queues of events. The simulation engine and all other framework components all run as services themselves, and are deployed at run-time for execution. A simulation federation client looks up deployed engines to which a generated simulation can be uploaded and run. The results are recorded at each agent involved in the simulation and retrieved once execution completes. The simulation runs as quickly as the engines are capable of processing the event queues. No emulation is supported; the service-oriented architecture would allow it theoretically, just as it would theoretically support programmatic interaction with a running simulation, but neither are available "out of the box". Similarly, the potential for visualization is there but is limited in support.

They describe [76] the process and results of simulating a service-oriented solution to an escape problem: a runner and a bait entity collaborate at run-time to get past a guard. There is no real-world service for comparison, however they do identify several issues with their collaboration strategy from iteratively running simulations, analyzing results, and changing their general collaboration strategy.

2.3.3 MaramaMTE

Grundy *et al.* used an existing performance test-bed generator (MaramaMTE) [21] to generate stress tests for static service compositions. Their goal was to support the meeting of non-functional requirements (particularly QoS) in service compositions by providing architects with information about potential interactions. They use BPMN or their own custom visual WS composition modeling language (ViTABaL-WS [41]) to model the high-level service composition. A custom software architecture notation can be used within the tool to model lower-level service interface details. Performance requests can be sent to the actual services or to generated stub services (in the paper, the service stubs are SQL queries executed on a database; there does not appear to be an attempt to replicate actual service behavior).

The composed services are stress-tested with load intended to emulate a remote client. Loads are created manually (using time delays between requests of services in a composition) and by a method based on the Form Charts formalism from [17].

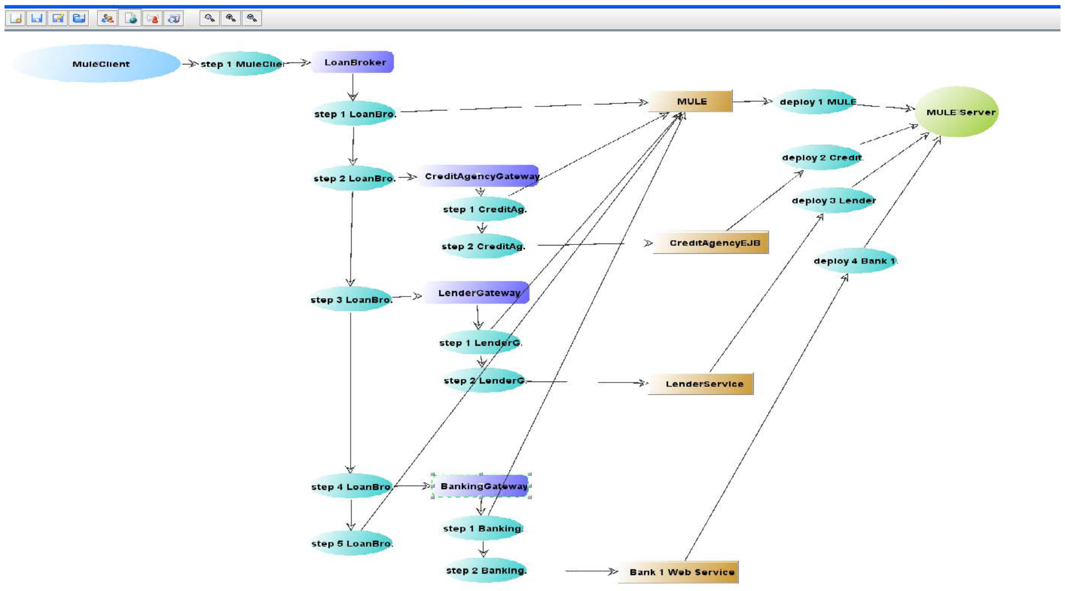


Figure 2.5: Screenshot of the SOPM visual model building tool (from [7]).

The intent is to emulate the actions of a user issuing a series of service requests by navigating a web site front-end.

They modeled and implemented a service composition based on searching for flights, choosing a flight, and choosing a seat. They identify this implementation as “simplistic”, but were able to identify potential resource contention issues in concurrent requests. The results are accurate when assessing composition but do not predict or emulate real-world service performance. The strengths of their approach are suited to design-time performance considerations.

2.3.4 SOPM

Service-Oriented Performance Modeling (SOPM) is a framework developed Brebner *et al.* at NICTA[3] [7, 9, 8]. It models a system of services using a visual tool to define existence of services and structure of compositions. A performance model is constructed for each component by sending a series of single requests to identify single-request response time, or by using instrumentation of the code to report performance metrics. This model is deterministic and cannot be modified stochastically. The model is made of building blocks like services, servers, workloads (workflows annotated with timing details), and metrics (Figure 2.5). The building blocks are not intended to be extensible; only simulations with a one-to-one mapping from real to simulated components are possible. From this manually-created model, a simulation is automatically generated to be run on their custom simulation engine (discrete-event, serial, local). The simulation allows for interaction at simulation-time to modify parameters dynamically. The simulation runs produce metrics for each simulated component.

Their system has been used to model substantial services and many-service work-

[3]National ICT Australia.

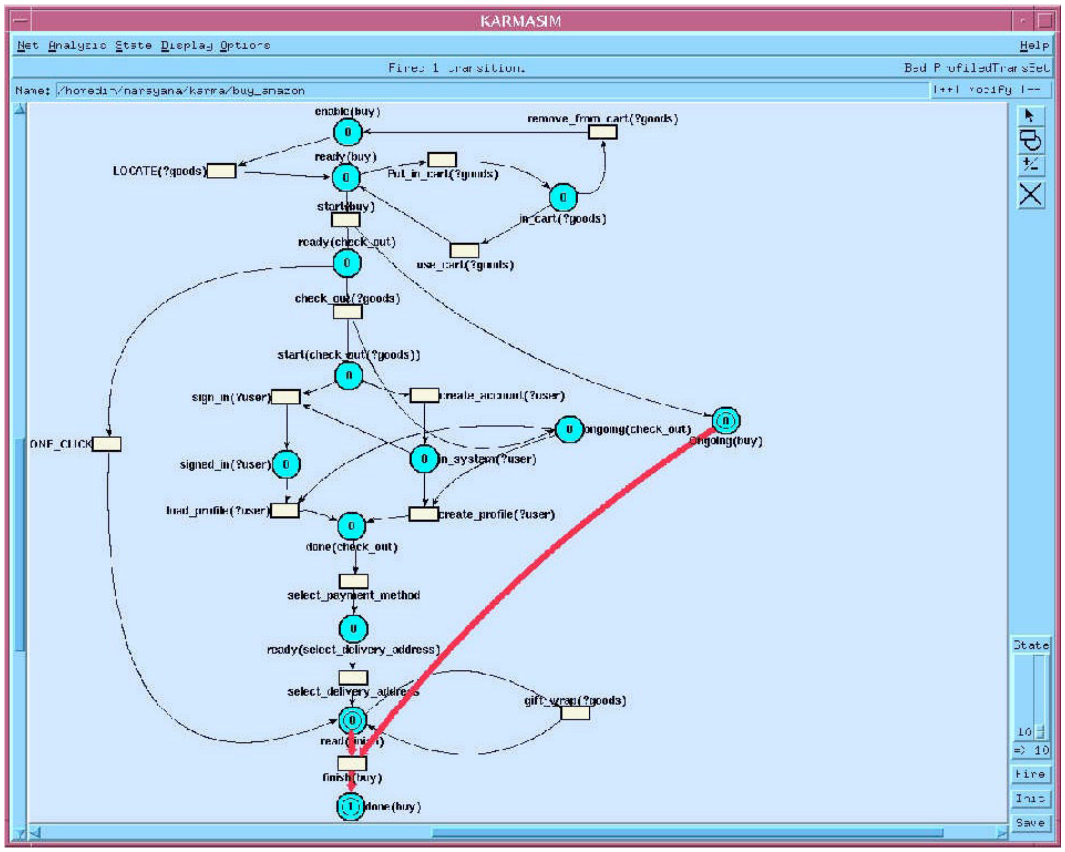


Figure 2.6: The KarmaSIM environment, showing the Congo example in execution (from [50]).

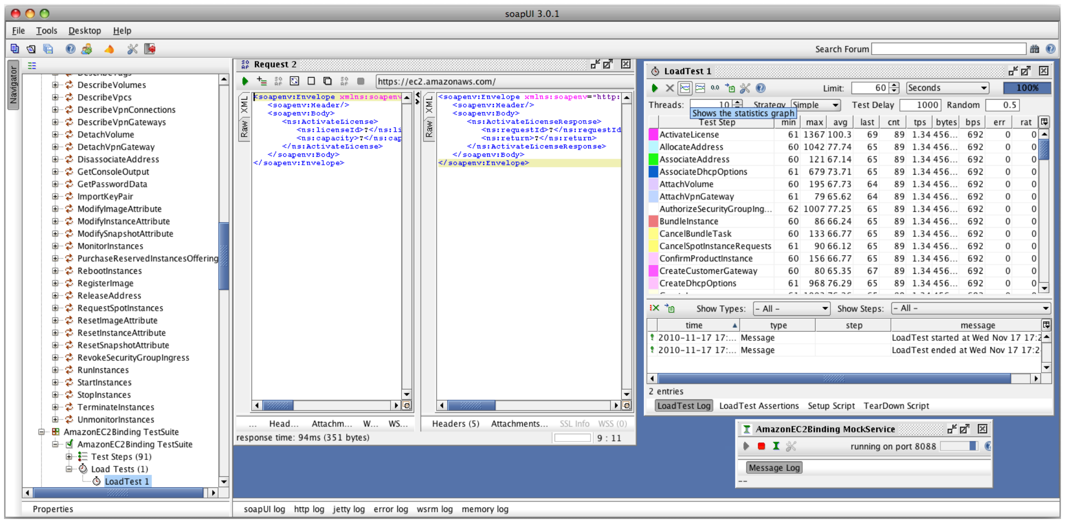
flows for service systems currently being developed. In [7], they describe using it to model an Enterprise Service Bus called Mule and calculate the total throughput. Their validation compared simulation-generated metrics to real-world metrics in identical configuration. They found a margin of error of up to 15%, which is explained by elements in the real world not included in the simulation model. They were able to make performance predictions based on the scenarios they simulated.

2.3.5 Narayanan (DAML)

Narayanan *et al.* [51, 50] described a knowledge representation that can be used for simulation (among other things) for Web service compositions. They do not mention SOA or most of the WS standards, but their composition-oriented approach is in line with SOA principles. They model services using a markup language intended for the semantic web, the DARPA agent markup language for services (DAML-S), which has since been superseded by OWL-S[4]. DAML-S is a process modeling language. This model is mapped to a first-order logic language to allow for situation calculus-based reasoning.

To simulate the modeled system, they translate the DAML-S representation to Petri nets. Petri nets have visual representations, yet allow mathematical analysis

[4]`http://www.w3.org/Submission/OWL-S/`



https://ec2.amazonaws.com/

```
<soapenv:Envelope xmlns:soapenv
   <soapenv:Header/>
   <soapenv:Body>
      <ns:ActivateLicense>
         <ns:licenseId>?</ns:li
         <ns:capacity>?</ns:cap
      </ns:ActivateLicense>
   </soapenv:Body>
</soapenv:Envelope>
```

```
<soapenv:Envelope xmlns:soapenv="http:
   <soapenv:Header/>
   <soapenv:Body>
      <ns:ActivateLicenseResponse>
         <ns:requestId>?</ns:requestId
         <ns:return>?</ns:return>
      </ns:ActivateLicenseResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

response time: 94ms (351 bytes)

Limit: 60 Seconds
Threads: 10 Strategy Simple Test Delay 1000 Random 0.5

	Test Step	min	max	avg	last	cnt	tps	bytes	bps	err	rat
🟪	ActivateLicense	61	1367	100.3	69	89	1.34	456...	692	0	0
🟦	AllocateAddress	60	1042	77.74	65	89	1.34	456...	692	0	0
🟩	AssociateAddress	60	121	67.14	65	89	1.34	456...	692	0	0
🟦	AssociateDhcpOptions	61	679	73.71	65	89	1.34	456...	692	0	0
🟪	AttachVolume	60	195	67.73	64	89	1.34	456...	692	0	0
🟦	AttachVpnGateway	61	79	65.62	64	89	1.34	456...	692	0	0
	AuthorizeSecurityGroupIng...	62	1007	77.25	65	89	1.34	456...	692	0	0
🟥	BundleInstance	60	86	66.24	65	89	1.34	456...	692	0	0
🟨	CancelBundleTask	60	133	66.77	65	89	1.34	456...	692	0	0
🟨	CancelSpotInstanceRequests	61	90	66.12	65	89	1.34	456...	692	0	0
🟪	ConfirmProductInstance	60	156	66.77	65	89	1.34	456...	692	0	0
🟪	CreateCustomerGateway	60	80	65.35	67	89	1.34	456...	692	0	0
🟪	CreateDhcpOptions	61	968	76.29	65	89	1.34	456...	692	0	0

time	type	step	message
2010-11-17 17:...	Message		LoadTest started at Wed Nov 17 17:2
2010-11-17 17:...	Message		LoadTest ended at Wed Nov 17 17:2

2 entries

running on port 8088

--

Figure 2.7: SoapUI, shown exploring the WSDL for Amazon EC2: a generated request and an Amazon response, a running mock service, and a load testing applet.

of processes; they support stochastic modeling; and they are commonly used to model distributed systems. They include places, transitions, and arcs: transitions are essentially events, places are roughly states, and a set of input arcs to a transition are roughly pre-conditions on the occurrence of events. Though Petri nets support continuous values, they choose to model a discrete-event system in their approach.

Their implementation translates DAML-S representations to Petri nets that can be executed in a simulator called KarmaSIM [52]. This visual tool shows the path of "execution" through the network (Figure 2.6). Since each transition has a probability of actually being taken even when pre-conditions are met, a variety of analyses can be conducted - detecting deadlocks, for example. Interactive simulations can be used to test compositions. A related benefit is the ability to automatically compose services to achieve a provider's goal. [50] describes modeling a book-buying web service for Congo.com. Their analysis identified a potential deadlock in user account creation workflow.

2.3.6 Commercial Solutions

This set of services testing and development tools available commercially (which includes soapUI[5], GH Tester[6], and CloudPort[7]) offers the ability to generate a service stub from a WSDL (emulation, which they refer to as simulation), to generate and send SOAP messages based on a WSDL (collecting performance metrics from the responses), and to perform similar functions for other communications protocols. The stated purpose is typically testing, including composition testing (using service stubs in place of outside services), automated testing (with a variety of messages), and load testing (with a large number of messages sent in parallel). Figure 2.7 shows the soapUI tool, including a load testing panel with the results of a load test on all

[5]`http://www.soapui.org/`
[6]`http://www.greenhat.com/ghtester/`
[7]`http://www.crosschecknet.com/products/cloudport.php`

	Goals
SOAD	verify logical correctness of the service composition pre-development
DDSOS	support the development of service systems by testing compositions pre-development
MaramaMTE	help meet non-functional requirements in service composition before and during development
SOPM	understand performance and scalability pre-deployment
Narayanan	"enable ... automated reasoning technology to describe, simulate, automatically compose, test and verify Web service compositions" [50]
Commercial	composition and load testing automatically from WSDLs
SASF	create a virtual model of a componentized software application focused on capacity planning

Table 2.1: Goals of various simulation frameworks.

of the Amazon EC2 operations. The raw numbers shown there can be displayed in a line graph, which is typical of the entire set. The metrics are focused on response time and bytes transferred.

These tools meet the inclusion criteria as they offer simulated services and metrics gathering, but the simulated services are based on a model of the interface (*i.e.*, the WSDL) and not on the behavior of performance of the service being simulated. Though performance metrics can be gathered from the simulated services, their use would be limited to identifying bottle-necks or incompatible compositions. The purpose of the performance testing tools is to gather metrics from real services, which is useful but outside the scope of this survey.

The focus on individual services means that support for composition requires the manual piecing together of a workflow, often in the form of scripting or perhaps in a GUI. Other tools, like Oracle BPEL Process Manager[8] offer superior support for load testing compositions, but offer fewer options at the individual service level.

2.3.7 Discussion and Comparison

The simulation frameworks described in this survey each have strengths best-suited to certain scenarios. This section describes the differences and similarities among the frameworks, based on the characteristics defined in §2.2. The properties of each framework are summarized in Tables 2.1, 2.2, and 2.3.

There are varied reasons for simulating service-oriented systems, and each simulation framework has strengths best-suited to certain scenarios.

1. **Goals and tasks** of the simulation. The most common task supported by the frameworks is to test compositions before full-scale deployment. SOPM and SASF were the exceptions, focusing instead on capacity planning and performance testing. Both SOPM and SASF offer library support for composing services; rather than the main goal, it is a means to enable capacity planning based on simulated performance. Commercial solutions are also focused on the performance testing of individual services, though composition can be scripted into the load testing manually.

[8]`http://www.oracle.com/technetwork/middleware/bpel/overview/index.html`

	Basics	**Authoring**	**Interaction**	**Metrics**	**Emulation**	**Speed**	**Visual**
SOAD	Deterministic, Serial, Discrete-event, Event/Time-driven	Visual, code generation from model	–	Targeted	–	Real-time	Animated, Graphs
DDSOS	Deterministic, Parallel/Serial, Discrete-event, Event-driven	Partial visual, code generation from model	–	Targeted	??	Accelerated	–
MaramaMTE	Deterministic, Serial, Discrete-event, Time-driven	Partial visual, code generation from model	–	Targeted	Optional	Real-time	–
SOPM	Limited stochastic, Serial, Discrete-event, ??	Partial visual, code generation from model	Manual	Comprehensive, extensible	–	Accelerated	Animated, Graphs
DAML	Stochastic, Serial, Discrete-event	Code generation from model	Manual	–	–	Accelerated	Animated
Commercial	Emulation	Automatic from WSDL (editable)	–	Targeted	Complete	Real-time	Graphs
SASF	Stochastic, Serial, Discrete-event, Time-driven	Automatic from WSDL and performance profile	Manual, Autonomic, API	Comprehensive, extensible	–	Accelerated	Animated, Graphs

Table 2.2: Simulation environment properties.

	Modeling	**Coupling**	**Evaluation**
SOAD	DEVS (SOAD)	broker, static composition (choreography)	Simple proof-of-concept
DDSOS	PSML-S	static composition (choreography)	Simple composed problem, identified composition problems
MaramaMTE	BPMN, ViTABaL-WS	static composition (orchestration)	"Simplisti" composed service, identified resource contention issues
SOPM	Visual, JMeter for performance	stochastic composition (orchestration, choreography	Numerous uses; sample app within 15%
Narayanan	DAML-S	stochastic composition	Modeled example network, found deadlock
Commercial	WSDL	via script authoring	–
SASF	WSDL + performance profiles	stochastic composition (orchestration)	Modeled two applications; statistically validated metrics

Table 2.3: The modeling, coupling, and evaluation properties.

2. Properties and features of the **simulation environment**[9].

    - Basics: A surprising number of frameworks offer only deterministic simulations: given the same set of input data, they will return the same result every time. Only DAML and SASF offer completely stochastic simulators; of these, SASF best supports the option to deterministically fix otherwise stochastic elements for systematic and repeatable testing. All of the frameworks examined offered or employed discrete-event simulation engines (this is the more common simulation method in other domains, as well). DAML offers the most potential for a continuous simulator, as the underlying model is petri nets, which have been extended in a number of ways to enable continuous simulations (*e.g.*, [16, 2]). The majority are time-driven simulations; only SOAD and DDSOS offer event-driven simulations, which make them a better solution for simulating systems where events are sparse. The engines all run individual simulations serially, except DDSOS which can optionally be run in a parallel mode. Only DDSOS offers the ability to run a distributed simulation.
    - Authoring: Only two frameworks offer automatic generation of a simulation, each with its own caveat. The Commercial tools generate a running real-world service from the WSDL, but this version makes no effort to replicate the performance or behavior of the model. SASF generates simulated services from the WSDL, but uses performance testing results obtained semi-automatically: the SOAP requests need human input to be parameterized before being sent to the real-world system to generate a performance profile. Both approaches work for SOAP-based services, but not REST. The other tools offer visual environments to help build models of the system; simulations can be generated automatically from

[9]The various Commercial solutions are disregarded here; their environment is determined by the real-world and not a simulation engine.

these models. If a WSDL does not exist, a visual tool for modeling is a better option. Some frameworks (like DAML) use the same approach to model the service at design time and guide implementation, which gives dual-purpose to the modeling stage.

- Interaction: Here SASF is quite distinct, as compared to the other approaches. Only SOPM, DEVS, and DAML offer even the basic ability to interact with a running simulation; the others are launch-and-forget. SASF goes further, offering both an API and internal functions for modifying the parameters of a running simulation (including properties of the simulation environment and the simulated application).
- Metrics gathering: Again SASF is a standout; the features and APIs enabling the collection and dissemination of metrics require minimum effort by the simulation author. Only SOPM offers such customizable metrics gathering.
- Visual: The standard here is the generation of visual depictions of metrics or of the system architecture, often animated. Only DDSOS and MarmaMTE fall short in this category. SASF requires less developer effort when introducing additional metrics to the visualizations.
- Emulation: Commercial solutions offer total integration with real-world services, where simulated services can even participate in compositions with already implemented services (with syntactically correct but semantically meaningless data). Only SASF and MarmaMTE offer any ability to integrate real-world services with simulated services that accurately reflect the behavior of the service.
- Speed: As a consequence of their emulation abilities, the Commercial and MarmaMTE frameworks cannot run faster than a real-world service. SOAD also has this constraint. SASF runs at an accelerated pace, except when in optional emulation mode.

3. **Modeling** language and methods, abstract representations, or formalisms. SASF and commercial solutions rely on the existing model of a service as expressed in WSDL. This is advantageous if the service already exists and is described using a WSDL; if not, the increases expressiveness of other models may be superior. The others use typically more rigorous formalisms that allow the use of formal methods and proofs. SOPM offers a purely visual model.

4. Support for **service coupling**. All of the frameworks support composition to some extent. MarmaMTE is notable for its use of BPMN as a modelling language.

5. **Evaluation** method. SASF is noteworthy, as it has been used to model full-size real-world componentized software systems and statistically validated to accurately predict service performance, then used to reason about the system. SOPM has also been used to address research problems.

2.4 Value, Quality, and Cost

To make configuration decisions by reasoning with simulation-generated data, a target must be defined. This target is often defined using technical metrics such as response time or throughput. Though these metrics are easily definable and measurable, they are not sufficient when asking non-technical users to understand or compare configurations. As discussed in Section 2.1.1, SLAs are often described in technical terms not accessible to business users. This section turns to marketing and business literature to understand higher-order attributes like value, quality, satisfaction, and cost. In particular, how *perceived* value, quality, and cost inform the decision-making process is described.

Value is not a constant definition with all consumers, and it varies from person to person and from product to product. Zeithaml [82] offers an overall definition: "*perceived value* is the consumer's overall assessment of the utility of a product based on perceptions of what is received and what is given." To us, that means "they do a cost-benefit analysis". There is a trade-off inherent to value: customers are willing to sacrifice some attributes in exchange for gains in other attributes. For example, they may give up convenient twist-top lids in exchange for 100% pure juice, or they may pay more for a product they consider environmentally friendly. Decisions can be guided by single attributes (*e.g.*, monetary cost) or some combination. In simple terms, one can think of perceived value as "what I get for what I give", where each individual may consider various attributes to 'give' in exchange for various attributes they 'get'. The total of what is given up (or *sacrificed*) is the *perceived cost.*

Researchers agree that value is a conclusion reached based on a variety of factors and attributes. Essentially, low level attributes imply quality, quality and cost imply value, value affects value to me personally. Sawyer and Dickson [68] defined value as "a ratio of attributes weighted by their evaluations divided by price weighted by its evaluation". They separate price from the other attributes. Though this is a formula that implies numeric analysis, it can be difficult to quantify the variables.

Quality is, at an abstract level, a measure of excellence or superiority [82]. It is not an attribute; rather, it is a high-level construct formed in the minds of individuals based on their assessment of a product or service. Research distinguishes between *actual quality* and *perceived quality. Actual quality* is some quantifiable and verifiable measure of excellence, compared to some norm or ideal. We measure water quality in parts-per-million of substances that should not be in pure (ideal) water. We measure manufacturing quality in terms of the number of defects per item produced, with an ideal of "0". *Perceived quality* is a more complex measure that factors in customer expectations, customer experience, comparator sets, and other factors to form a global assessment of quality. It is a customer's judgment about the superiority or excellence of a product. It is this definition that most influences perceived value.

Perceived quality, in its reliance on experience and knowledge, is assessed based on what the consumer knows about the product and all competing products. Which products are 'competing' is up to the consumer (Ford might think their competition is Toyota, but to some consumers it may be Vespa).

There are varying schools of thought on how to assess quality. Perhaps the best known is SERVQUAL [59], which assesses quality based on the gap between what the consumer expects and what they actually experience. Over the 25 years since was proposed, it has been refined several times (*e.g.*, [57, 83]). It measures the gap between expectation and perception in five primary categories: Reliability, Assurance, Tangibles, Empathy, and Responsiveness (RATER). It is driven fundamentally by satisfaction, where the smaller the gaps, the greater the satisfaction. Over the same time period, it has also attracted criticisms and alternatives. One of the best known contenders is SERVPERF [13], which is performance-based or attitude-based: Cronin and Taylor assert that perceived service quality can be best measured by a customer's "perceived" attitude about the service. The debate (*e.g.*, [14, 60]) has continued since and will not be re-produced here[10]. This background section is influenced by the gap/satisfaction model.

Lutz [43] proposes that we consider two other forms of quality: "affective quality" and "cognitive quality". A cognitive judgment is one based primarily on attributes that can be determined without actually consuming a product (brand, reputation, packaging), while affective quality is determined by actual experience. He suggests that service quality is an affective judgment: that is, global assessment of service quality is based on personal experience.

§2.4.2 presents a set of propositions that help understand perceived value, perceived quality, and perceived cost. These propositions motivate an SLA negotiation approach described in Chapter 5. The propositions relate to products; to assess their applicability, we must first establish that a web service shares properties with traditional definitions of products (§2.4.1).

2.4.1 Web Services as Products

In business and marketing literature, organizations can traditionally market products and/or services. The intuition of these concepts is easily determined; however, the distinction between them is more subtle. The academic community does not entirely agree on the distinction. The idea of a service offered online without ever talking to a person was still a foreign idea when much of the foundational literature in service quality was written. The expectation given the name "web service" is that it would align with the notion of a service in marketing literature. In fact, a web service has qualities in common with both.

Zeithaml, Parasuraman, & Berry [58] conducted a review of services literature and concluded that the prevailing characterizations of services were "intangibility, inseparability of production and consumption, heterogeneity, and perishability". That is, services are difficult to measure, they are consumed at the moment they are produced, they are not always consistent, and they cannot be stored. The difficulty in measuring the outcome of a service led to the consideration of the process of a service rather than the outcome (*e.g.* [47, 53]). Table 2.4 lists how electronic web services compare with this traditional definition.

Nilsson explored intangibility in more depth and concluded that there are few objective reference points for consumers to use in perceiving value in services. Any cues are ambiguous at best; therefore, the personnel offering the services can influence how the consumer perceives quality. The literature in marketing frequently refers to "service encounters": quality assessment, decision-making, and so forth are

[10]The notion that a one paragraph summary can adequately represent almost 3 decades of ongoing debate, refinement, and extension is, of course, ridiculous. Google Scholar reports 6776 citations for the original SERVQUAL paper and 3557 for SERVPERF, and in the thousands for the key debate papers.

Similarities between services and Web Services
Production and consumption cannot be separated.
Services are "perishable": they cannot be stored.
The quality of a service is difficult to measure (outside of a few non-deterministic cues, and necessary-but-not-sufficient attributes).
The assessment of quality *can* be guided by the personnel helping offer the service, though they may be distant from actual service encounters.
Distinctions between services and Web Services
They are "intangible" because they are electronic, but some aspects of a web service can be measured: transactions per second, response time, up time, and so forth.
Generally a service is NOT heterogeneous: for a given input, it will produce the same output.
"Service encounters" are two pieces of software "encountering" each other for milliseconds at a time.

Table 2.4: Comparing traditional definitions of services to Web Services.

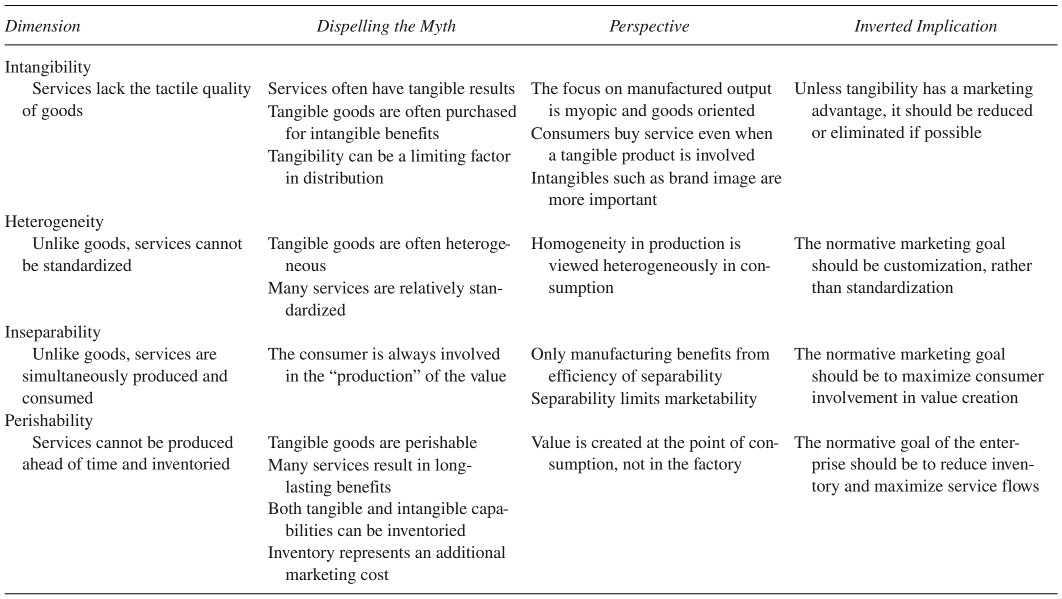
based on individual encounters, with people involved. To Levitt [38], everything is a service to a greater or lesser degree: sometimes the service was "back stage" and out of view, other times it was at the forefront.

Vargo and Lusch [78] argue that distinguishing a good from a service is meaningless, and is tied to a manufacturing-driven model of the world. They address each of the classic characterizations of services and explain why each is not applicable to services as we see them today (Figure 2.8). Two of their implications are particularly interesting - that customization is preferable to standardization, and that consumer involvement in value creation should be maximized. The SLA negotiation approach in §5.1 describes how simulation can be used to customize SLAs by involving the consumer and improving their ability to contribute to the creation of an SLA that meets their unique needs.

2.4.2 Propositions on Value

Zeithaml [82] presented propositions about value and quality; in particular, how low-level attributes map to higher-order constructs like quality. Her work focuses on regular consumers making personal decisions about products (groceries, travel plans, and so forth), and not individuals making decisions about electronic services on behalf of an enterprise. The assumption in this work is that the concepts are largely transferable. The propositions described here are limited to those that impact the contributions of this dissertation. Unless otherwise noted, they are paraphrased directly from [82].

- **Consumers use lower level attribute cues to infer quality.** The perception of "high quality" is based on cues from low-level attributes. Empirical studies have shown that "large size" is a cue indicating a high-quality speaker, that "100% pure" is a signal of good fruit juice, and that the amount of suds indicates high-quality detergents. Though more complex metrics are available for each of those products, one or two easily-compared attributes are used to



Dimension	Dispelling the Myth	Perspective	Inverted Implication
Intangibility			
Services lack the tactile quality of goods	Services often have tangible results; Tangible goods are often purchased for intangible benefits; Tangibility can be a limiting factor in distribution	The focus on manufactured output is myopic and goods oriented; Consumers buy service even when a tangible product is involved; Intangibles such as brand image are more important	Unless tangibility has a marketing advantage, it should be reduced or eliminated if possible
Heterogeneity			
Unlike goods, services cannot be standardized	Tangible goods are often heterogeneous; Many services are relatively standardized	Homogeneity in production is viewed heterogeneously in consumption	The normative marketing goal should be customization, rather than standardization
Inseparability			
Unlike goods, services are simultaneously produced and consumed	The consumer is always involved in the "production" of the value	Only manufacturing benefits from efficiency of separability; Separability limits marketability	The normative marketing goal should be to maximize consumer involvement in value creation
Perishability			
Services cannot be produced ahead of time and inventoried	Tangible goods are perishable; Many services result in long-lasting benefits; Both tangible and intangible capabilities can be inventoried; Inventory represents an additional marketing cost	Value is created at the point of consumption, not in the factory	The normative goal of the enterprise should be to reduce inventory and maximize service flows

Figure 2.8: Limitations of the distinguishing characteristics of services, from [78].

serve as reliable signals of quality. Consumers instinctively look for low-level attributes that consistently signal high-quality services.

For complex decisions, relying on one or several low-level attributes can be misleading. To avoid this, a high-level or aggregate attribute should be used to help decision-makers determine quality more explicitly, rather than relying on each individual to choose their own set of low-level attributes.

- **The intrinsic product attributes that signal quality are product-specific, but dimensions of quality can be generalized to product classes or categories.** Researchers divide attributes into intrinsic (physical composition of the product: color, shape, size, taste) and extrinsic (product-related but not part of the product: brand, price, level of advertising). Intrinsic properties are consumed as the product is consumed. Of course, there is ambiguity here.

  The few intrinsic attributes a consumer considers when determining quality vary by product. The cues that signal quality for juice differ from the cues for lamps. Even similar products show variation: thick tomato juice is good, thick grape juice is bad. However, some aspects of quality apply generally to whole classes of products. The more abstract an attribute, the more likely it is to apply to a large set of products. Comparing two different things, like stereos and vacations, requires consumers to use very abstract concepts like "entertainment value".

  When asked to consider service industries, Parasuraman [58] found that consumers consistently judged quality based on reliability, empathy, assurance, responsiveness, and tangibles.

- **Extrinsic cues serve as generalized quality indicators across brands, products, and categories.** Unlike intrinsic cues, most extrinsic cues are already abstract and generalizable across product classes. Brand, price, warranty, seals of approval, and endorsements are all extrinsic cues. Research in this area has shown that in the absence of all other information, high price means high quality. Brand name offers a bundle of information that can intellectually be the same as experiencing the product: if you've used a service provider to book a flight and preferred it, you may be more likely to rent a car from the same provider.

- **When customer's rely on intrinsic or extrinsic attributes.** Consumers depend on intrinsic attributes more than extrinsic attributes:

  1. at the point of consumption,
  2. in pre-purchase situations when intrinsic attributes are search attributes (rather than experience attributes), and
  3. when the intrinsic attributes have high predictive value.

  Consumers depend on extrinsic attributes more than intrinsic attributes:

  1. in initial purchase situations when intrinsic cues are not available (e.g., for services),
  2. when evaluation of intrinsic cues requires more effort and time than the consumer perceives is worthwhile, and
  3. when quality is difficult to evaluate (experience and credence goods).

  Intrinsic attributes are more useful while consuming a product: when drinking a beverage, you measure your satisfaction by the taste, not by the brand name. Search attributes are those that can be assessed prior to purchase (e.g. "100% pure" for juice, "no blemishes" for fruit), whereas experience attributes are assessed only during consumption (e.g., "smooth" for juice, "not under- or over-ripe" for fruit). In pre-purchase situations, intrinsic cues are used when they can be assessed prior to purchase. When they cannot be, extrinsic cues are used. Services have few intrinsic cues, and those available can typically not be assessed until the service encounter has begun. Thus, services are frequently initially judged using only extrinsic cues.

  Intrinsic attributes are used when they can be readily assessed and are known to have high predictive value. If there are barriers to evaluating intrinsic attributes, extrinsic ones will be used instead. Similarly, if easily evaluated intrinsic cues are not sufficient to differentiate between two products (e.g., identical memory and processor configurations), the consumer will resort to extrinsic cues.

  Intrinsic attributes are useful and generally preferred, but are limited by being requiring experience to assess them. This is a strong motivator for the use of simulation: experience can be simulated, allowing the use of intrinsic attributes in situations where it would otherwise be impossible or cost-prohibitive.

- **Monetary price is not the only sacrifice perceived by consumers.** This point is perhaps obvious; when you make a decision, you weigh many trade-offs, only one is price. The literature divides the elements of a purchase decision into two sides: what you give versus what you get. Those things you “give” are considered sacrifices, and this includes price. However, consumers are also aware that there are other sacrifices. If you pay a lower price but have to assemble your own furniture, the additional work required is part of the sacrifice. Travel time, search time, evaluation time, maintenance time, and more may all be factored in to the total “sacrifice”. The result is a distinction between the actual price and the perceived price. People have different ways of encoding the price or sacrifice, for example: “expensive”, “a bargain”, “you get what you pay for”. The overall sacrifice is part of the equation for determining perceived value.

- **Extrinsic attributes serve as “value signals” and can substitute for active weighing of benefits and costs.** Consumers are generally “mindless” [36]; there is not careful consideration so much as there is overall impression. “Trusted brand”, “it’s cheaper than it was last week”, and similar cues are used in place of hard facts. There are some consumers who spend time and effort to give purchases careful consideration; Zeithaml theorizes that more rational evaluation is used when there is lots of information available, lots of time, lots of processing ability, and high involvement in the purchase. One theory is that as the cost of the product goes up (or, as the cost of choosing the wrong product goes up), more careful consideration is more common.

  Ongoing service delivery requires more than an initial purchase decision; a decision based on facts is likely to prove more satisfying than one based on instinct. Thus, when establishing an SLA steps should be taken to encourage rational evaluation: high involvement with the customer in establishing the SLA and furnishing the customer with as much information as they need are two key ways to do this.

- **The perception of value depends on the frame of reference in which the consumer is making an evaluation.** Value determination is situational and depends on context. Zeithaml reports that value changes between purchase time, preparation time, and consumption. The use of simulation allows consumers to incorporate their (predicted) value at consumption time at purchase time, limiting “buyer remorse”.

- **Perceived value affects the relationship between quality and purchase.** Customers do not always purchase the highest quality option; they make a decision that balances their perceived quality and perceived sacrifice: their perceived value. The trade-off is the key factor and should be incorporated into decision support tools.

2.5 Capacity Planning and Configuration Management

The problem of capacity planning is very general and highly relevant to manufacturing and service-delivery business strategy. According to Balachandran *et al.* [5], the cost of capacity resources typically account for 30% to 60% of the total costs in manufacturing and 70% to 80% of the total costs in service organizations. This high cost percentage gives rise to the need for capacity planning, so that organizations can obtain all necessary resources at once, ahead of time, as opposed on as-needed basis, which would potentially further increase the cost ratio. In their 1997 work, they considered the general problem where product production requires resource consumption (under some stochastic model) and product price is a function of product cost (as implied by the cost of the consumed resources) and a random profit. They distinguished between 'soft' and 'hard' constraints on resources that can (and cannot) be acquired on an as-needed basis in case of shortfall. Note that this formulation clearly relates the problems of capacity planning at deployment configuration time and autonomic capacity adjustment at run time. Through simulation experiments they found that simplifying capacity planning to focus on an expected bottleneck resource dominates product- or resource-level capacity planning, when all resources impose hard constraints and the firm's product mix is significant.

More specifically, in the context of computer-based service delivery systems, research has been conducted on capacity planning for different types of architectures. Kant *et al.* [30] examined, both with analytical and simulation models, the performance behavior of a traditional symmetric multiprocessor web server, under different types of benchmark loads. They found that using the virtual interface architecture improves performance, especially in the case of workloads with increased memory bandwidth. Furthermore, they predicted that more I/O capacity is required when the network processing is offloaded.

Almeida and Menasce [46] developed a general method for capacity planning in client-server systems, making explicit the need for models of workloads that the architecture is expected to support and the way its performance may change when aspects of the workload change. This approach is not service-focused and does not use simulation to produce data.

Zhang *et al.* created R-Capriccio, a capacity-planning tool that uses available performance information to make predictions and decisions about future capacity requirements [84]. The tool identifies the most popular client transactions, estimates the CPU requirements, and uses queueing networks to project capacity requirements for changing workloads. The tool is limited to CPU-bound services, and requires logs of both server access and CPU utilization under existing load.

Finally, Rossi and Tari developed a method for web-service self-adaptation to mitigate the impact of changes in the environment on performance and resource utilization [64]. They use middleware to move services among nodes in a cluster to balance load, based on real-time performance metrics. Services are distributed among nodes in order to optimize an "efficiency value" derived from various performance metrics.

2.6 Autonomic Computing: Self Management

The demand for dynamic, distributed, component-driven software has led to increased adoption of service-oriented software systems. The nature of these systems limits the effectiveness of design-time configuration decisions; additional copies of service instances may be deployed at run time; new services may be removed and new services may be introduced in an existing composition; the provisioned network bandwidth can be changed. This is why deploy-time and even run-time configu-

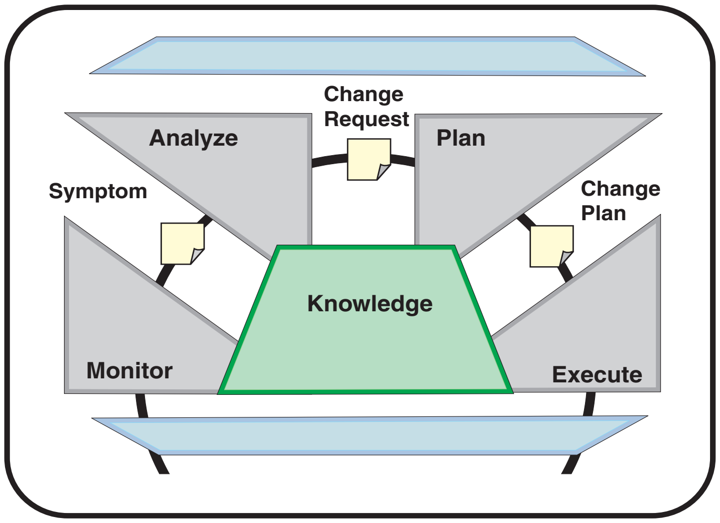


Figure 2.9: The Model, Analyze, Plan and Execute model of autonomic computing, from [26].

ration decisions are required to manage the system in the face of varying business requirements, service providers and consumers, and available resources [54]. The frequency of changes and decisions required for just-in-time resource provisioning precludes manual intervention; an autonomic, self-managed system is desired instead [32, 48, 56].

Like its namesake in biology, the goal of an autonomic system is to be self-managing. According to the IBM blueprint for autonomic computing [26], self-management involves four main activities: *Monitoring*, *Analysis*, *Planning*, and *Execution* (MAPE). The *monitoring* stage uses sensors to measure key load and performance attributes of the system. The *analysis* stage identifies any metrics that are outside the expected/desired ranges or violate rules stated in the system's SLA (Service-Level Agreement). Furthermore, it attempts to identify the cause of the problem: perhaps a server is under-performing, or perhaps the load suddenly increased. Proactive-analysis methods might also attempt to predict the future by identifying trends or matching current behavior to past behavior patterns. In the *planning* stage, the system decides how to react to the fault by identifying a set of actions that may remedy the situation. These actions are implemented in the *execution* stage via actuators.

The autonomic adaptation of software systems is studied extensively. Approaches to run-time adaptation vary in their approach, for example explicit models of the system's performance [11, 40, 79] like ACUS, linear programming formulations [3, 12], and hybrid approaches [39].

The contributions described in Chapter 6, referred to here as Autonomic Configuration using Simulation (ACUS), focus on the *analysis* and *planning* stages. Monitoring and execution are out of scope. The underlying assumption is that monitoring data can be acquired accurately and with enough frequency; similarly, that execution (i.e., reconfiguration) can occur without prohibitive overhead or delay. Allowances are made for monitoring overhead and for reaction time to autonomic changes. The evaluation throughout this work used environments where these assumptions hold true.

ACUS distinguishes itself from **state of the art autonomic approaches** in several ways. A popular web services autonomic adaptation is to substitute one service for another equivalent service. ACUS is focused on enabling a service provider to meet service targets; substitution is not a viable solution. Service providers are not incentivized to participate in automatic discovery and dynamic composition if their service will simply be substituted. Service consumers can derive more value from a service interaction if the provider is instead incentivized to meet service level targets without substitution.

Much work in autonomic adaptation is at the server level, monitoring load averages and memory. In contrast, ACUS is generally applicable to many applications, but the simulation creation and the knowledge base once created are done at the application level, monitoring application-specific metrics and capable of making changes to the application rather than to system infrastructure. The ACUS knowledge base is created entirely in simulation, then augmented in reality. ACUS deliberately uses a model that can be visualized and used to explain why the autonomic manager made a certain decision, in contrast to black-box approaches.

Mancini *et al.* [45] use simulation to autonomically configure applications. A simulation is created based on a high-level modelling language, a parameter to optimize is identified, and a target for the parameter is defined - all manually. At run-time, a decision module generates a list of simulations to be executed based on a set of simple rules, passes the list to a simulation engine, and retrieves the results. The configuration that produces results closest to the target is chosen and the configuration changes made. The claim is that this process is predictive, but there is no indication of how and when to trigger this adaptation process. As a run-time system, only a few simulations can be performed before it will be too late to make configuration changes proactively. No evaluation is presented. In contrast, ACUS does not need to be told what its goal state is. It does not generate a decision model at run-time, but instead simulates each scenario once and caches the results in a decision-model, adding to the model at run-time if needed. It predicts future behavior based on past performance in similar situations, and proactively makes changes based on these expected futures. The simulation that informs ACUS can constructed automatically. ACUS is evaluated in a real-world cloud computing scenario.

Calinescu and Kwiatkowska [11] use a Markov model of a software application to generate an autonomic manager for that application, using the PRISM probabilistic model checker as an engine. The Markov model is created during a formal-verification process of the application (or later following the same process). Decisions are chosen by the engine based on the maximizing of a utility function. They describe empirical results for Markov chains with a small number of states, for two scenarios: energy saving by putting hard drives to sleep and cluster availability in a data center. They call their method model-driven architecture for autonomic computing. It is not clear how accurately the deterministic model reflects reality, or how well this method translates to a real system at run time. Constructing the decision model is labor-intensive, particularly if the application was not previously formally verified. This approach is system-level management, in contrast to the application-level management enabled by ACUS.

Bahati *et al.* [4] use a state-transition model to encode past decisions and their results. A state consists of metrics (based on the metrics described in the policies being enforced) and the various transitions already tried when in this state. The dividing line between states is the threshold of any metric. For example, if there is only one policy, which says response time must be less than 2 seconds, there would be two states: one for response time over 2 seconds, and one for response time under 2 seconds. A state-transition graph is built over time as the deployed system is manually transitioned from state-to-state. The autonomic manager identifies the state in which the system is and the transitions that could potentially move the system to an acceptable state. In comparison, ACUS builds a behavioral model offline (skipping the expensive learning stage) and offers variable granularity for each metric which allows more fine-tuned control, rather than a simple binary compliant/non-compliant discretization. Finally, their approach does not address the problem of over-provisioned systems; ACUS provides a more complete solution to the problem of self-configuration.

Nou *et al.* [55] describe a framework for autonomic control of Grid computing environments that used simulation. This is system-level autonomic management, not application-level, and is not specific to services. At run-time, they build a model of the load of the system based on the jobs running and their respective SLAs and predict the expected QoS and throughput using Queueing Petri Nets. Resources can be added if needed to achieve desired QoS levels. They evaluated this approach in a real-world computing environment (though with an artificial workload), and found that their approach was capable of meeting almost all of the SLAs. In contrast, ACUS offers application-level management with a focus on the predicted user experience (and not on provider throughput). They use online simulation (at run-time), which is repetitive and requires resources at run-time. A limited number of configurations with a limited amount of flexibility can be tested in the narrow time frame afforded by the necessity of real-time changes.

2.7 Summary

This chapter introduced services and service-oriented architectures, particularly as implemented using W3C standards like SOAP. Services are governed by Service Level Agreements, but these agreements are often poorly understood, measure the wrong things, are not observed, and in general are not perceived by service consumers to add value to services-based interactions.

A set of proposed canonical characteristics for describing, comparing, and measuring simulation frameworks was offered in §2.2. This set draws from existing characteristics in simulation literature but is tailored for services-specific simulation frameworks. A comprehensive and systematic survey of simulation frameworks used these characteristics to guide the description and comparison of seven simulation frameworks, including the one contributed in this dissertation. The existing frameworks provided only partial coverage of the desirable characteristics of a simulation framework.

§2.4 introduced service quality and perceived value, quality, and sacrifice, as understood in business literature. Propositions related to value were introduced and how they should influence the formation of service level agreements was described. Customization is preferable to standardization, consumer involvement in value creation should be maximized, the best way to measure and predict value and quality is based on experience with the product, that monetary sacrifice is not the only relevant trade-off when calculating value, and that the highest quality option isn't always the right choice (the trade-off is more important).

Existing approaches to configuration management (§2.5) do not make use of simulation to predict performance; instead, data from real-world deployed systems is used directly. This data is more expensive to produce, though is by definition accurate. When simulation is used, it is used to predict bottlenecks and resource contention issues; the work described in this dissertation takes a different approach, focusing on maximizing user-centric value and explicitly guiding users through managing trade-offs.

Related work in the area of autonomic computing was reviewed in §2.6; though state-transition models have been used before to construct an abstract representation of the behavior, their construction based on simulation-generated data is novel. There is little real-world validation of autonomic self-configuration approaches. Autonomic work in services focuses on substituting services with other services that perform the same function; the approach taken in this dissertation is to instead re-configure the original service to make its continued use possible. The approach described in this dissertation is at the application-level, instead of the more common approach of hardware-level or infrastructure-level management. Finally, a shortcoming of autonomic approaches is they are difficult to understand or explain; the simulation-generated narrative of predicted performance and the visualizable state-transition model presented here allow re-configuration decisions to be explained.

Chapter 3

Services-Aware Simulation Framework

Chapter Contents

The services-aware simulation framework (*SASF*, *sass-if*) is a suite of tools implementing a methodology designed to create a model of a service-oriented system and execute the model in a virtual environment. The intended use is the generation and capturing of data predicting the performance under different resource configurations and work loads. The advantage of time-based simulation over an analytic model is the ability to predict not just end results, but to predict an ongoing narrative of system performance at each moment of a simulation. The models built using this methodology, and the prototype tools, are focused on a set of metrics related to capacity planning (sometimes called performance metrics). An early version of the framework was first described in [71]; the complete description below reflects the current state of the framework, including several extensions.

The framework consists of four main components (Figure 3.1) and two extensions, each described further in the following sections.

1. the *Simulation Engine*, which provides functionality for running simulations (clock management, *etc.*) and a set of extensible libraries that implement common functionality (§3.1);

2. *wsdl2sim*, which takes the document describing the service interface and generates a simulation (§3.2);

3. *JIMMIE*, a tool to systematically re-configure the simulation and re-run it to test a variety of scenarios (§3.3);

4. the *Metrics Engine*, which facilitates collecting, storing, and visualizing metrics generated by the simulation or from recordings of simulations, and creating a dashboard for presenting the visualizations (§3.4);

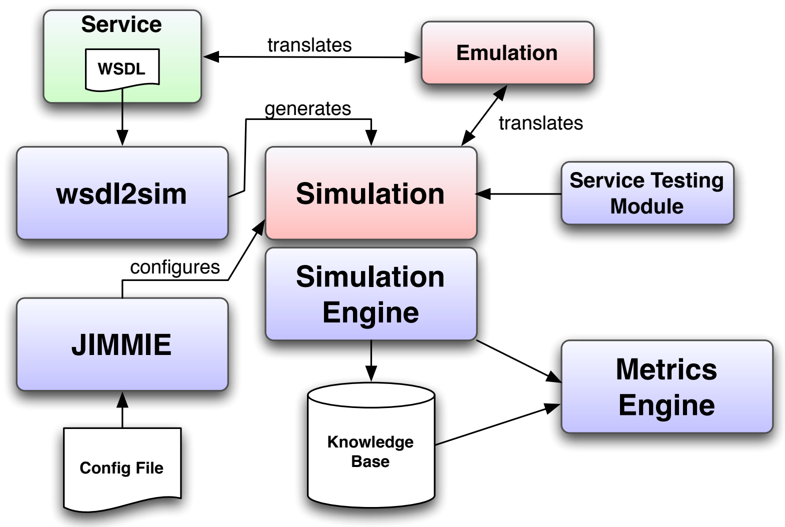


Figure 3.1: The primary components and extensions of *SASF*.

5. the *Emulation* extension, which enables integrating simulated components with real-world services (§3.5); and

6. the *Service Testing Modules* extension, which provides request generators and monitors requests/responses to produce metrics (§3.6).

3.1 Simulation Engine

The discrete-event time-driven simulation engine is implemented in Java, using objects to represent the real world entities. These simulation entities are designed to simulate the behavior of their real-world counterparts, in some cases at a higher level of abstraction[1]. Obvious mappings between simulated entities and real-world entities simplify the task of a simulation creator, one of the main goals of the engine. An extensible set of existing functionality reduces the "rote" code writing to create simulated entities, allowing the simulation creator to focus on implementing functionality.

The engine considers services and groups of services as entities on interconnected networks. These entities are linked by the exchange of messages over the network. By default, each entity is run in its own thread and only communicates with other entities over the network. The engine provides a set of Java classes implementing web services at different levels of granularity; simulation creators extend this base set of classes and override key methods with their own functionality (these implementations can be automatically generated, see §3.2). Several useful functionalities and sets of Java libraries are provided by the engine, including the following:

[1]For example, a simulated message is "transmitted" by the simulated network in the amount of time it would take the real-world network, but the process of creating packets and IP headers and so forth is abstracted away.

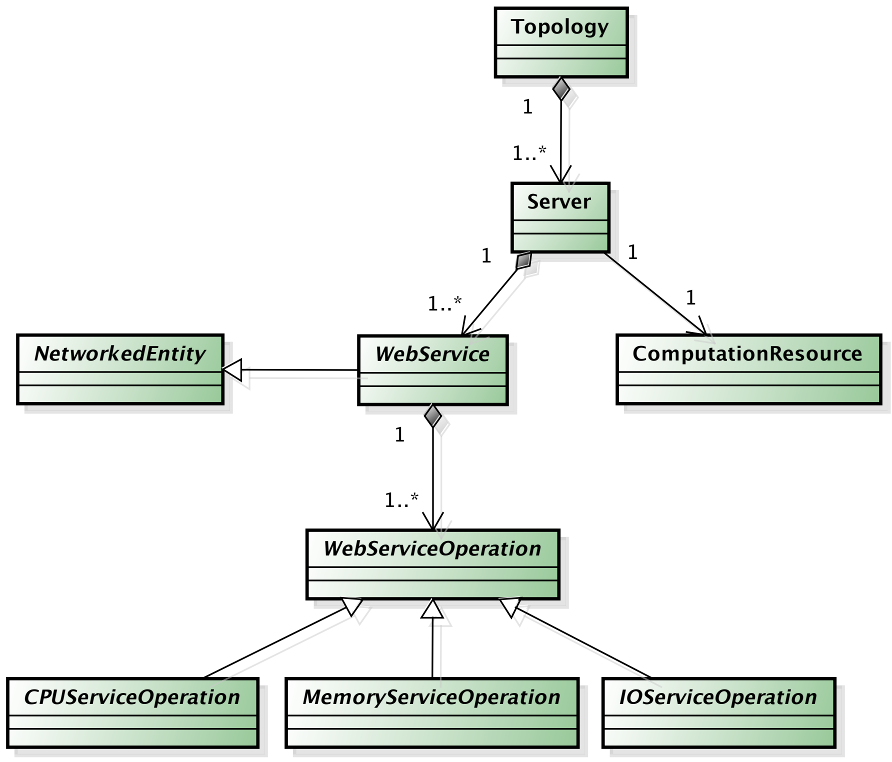


Figure 3.2: A class diagram of selected extensible service objects.

- Timing and clock control: All entities in the simulation deal with time in actual time units, and the engine handles conversion to simulation time. The number of real seconds that pass in a simulated environment second is configurable based on the complexity of the simulation and the computation power available. The clock also offers timed events (*e.g.*, the creator of the simulation can add requests to be notified in n minutes with a specific message or function call).

- Network transfers: Simulated services need only identify the intended recipient and the message to send. The required network bandwidth limitations, routing, and the actual transmission over time are handled by the engine. The engine enforces capacity limits based on configuration properties and records metrics.

- Basic service behavior: The structure of the simulated application is shown in Figure 3.2. This simplified UML diagram shows classes and their relationships; additional support such as interfaces and utility classes are not shown. Each application has a topology, describing its servers, networks, and what runs on those servers. Simulated entities are assumed to be resident on a particular server, which has a fixed amount of computation resources. Default implementations for topologies and servers with their computation resources are provided. These are integrated with the metrics engine, and are extensible. Each class is documented using `javadoc`, the standard Java API documentation.

Barring changes to the default structure, a simulation creator can start from any of the pre-defined starting points: `NetworkedEntity`, `WebService`, `WebServiceOperation`, or `*ServiceOperation`.

`NetworkedEntity` offers the most basic functionality: joining a network, packaging messages, sending messages and listening for messages. The simulation creator would implement the message creation and message processing entirely. It is also thread-safe, and is structured to be run in a `Thread`. This is the most flexible approach but also requires more development effort (though still less than starting from a blank page). It is useful to group similar services (or services that collectively perform a function) into a single simulated component.

A `WebService` has the functionality of a `NetworkedEntity`, plus it knows that incoming messages are destined for a `WebServiceOperation`. It provides some helper functions for its member operations (including metrics gathering/reporting functions that integrate with the metrics engine, and timing it obtains from the simulation engine). It manages queues of waiting requests and enforces service-level limits on member operations. This is particularly useful if the simulation creator does not want to simulate at the level of individual operations, and would prefer to have a single implementation simulate the behavior of all individual operations. Otherwise, the default implementation can simply be used in its current form, routing request to simulated service operations.

A `WebServiceOperation` receives requests routed by a `WebService` and sends a response. These offer a "blank slate" for simulation creators who wish to implement at the operation level but do not wish to make use of existing skeleton implementations.

Finally, the `*ServiceOperation`s offer stub implementations of `*`-bound operations; *i.e.*, CPU-bound, memory-bound, and IO-bound. A simulation creator need only extend these stubs and initialize a set of parameters for the equations used to calculate total processing time and resource consumption. The existing implementation will perform the calculations based on the limits expressed in the `ComputationResource`, wait the prescribed period of time, then send a response. The equations, and all other processing behavior, can be extended.

- Web Service messages: Simulated web services communicate using `Message`s. An interface is provided and can be extended by simulation creators. The `BaseMessage` can be used, but more commonly will be extended to include methods and fields required by a simulation (Figure 3.3). `Message`s are sent using the Network, which wraps all communication in `Data` packages that need to know the size of the message.

- Dynamic configuration: A simulation is configured and run based on XML configuration files, allowing simulation creators to easily move configurable parameters to external files outside the code base. The JIMMIE module can systematically modify these configuration files. In addition to a generic name-value style file, a topology file defines nested layers of servers, services, and

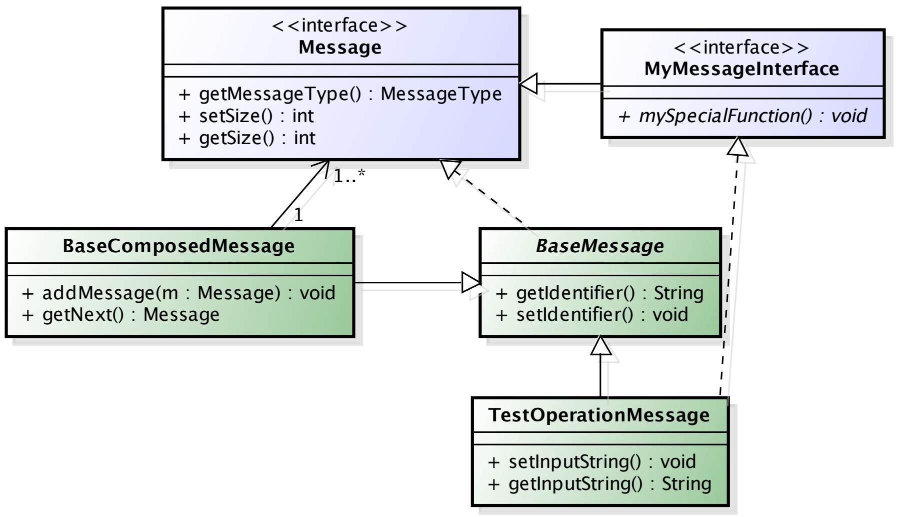


Figure 3.3: A class diagram of selected message objects.

```
<server id="1" memory="512" processor-count="2" processor-speed="2200">
   <service type="TaporWebService" class="org.ualberta.tapor.TaporWebService" id="1" parameters="2;1;">
      <operation id="2" name="ListWordsOperation" class="org.ualberta.tapor.ListWordsOperation" />
      <operation id="3" name="DateFinderOperation" class="org.ualberta.tapor.DatefinderPlainOperation"/>
      <operation id="4" name="AcronymFinderOperation" class="org.ualberta.tapor.AcronymFinderOperation"/>
   </service>
</server>
<server id="2" memory="512" processor-count="2" processor-speed="2200">
   <service type="TaporWebService" class="org.ualberta.tapor.TaporWebService" id="2" parameters="2;1;">
      <operation id="1" name="WordCloudOperation" class="org.ualberta.tapor.WordCloudOperation" />
   </service>
</server>
```

Figure 3.4: A sample two-server, four-operation topology descriptor. The simulation using this topology is described in §4.4.

operations that allow changes to the topology of the simulated web service at run-time. Each layer specifies the Java class file that implements that layer. A sample topology descriptor is shown in Table 3.4.

- Service composition: A `ComposedMessage` type is provided; the load generators can generate messages of this type, which consist of an ordered sequence of services to call. The orchestration of this sequence is handled by the engine.

3.2 wsdl2sim

This framework component generates a simulation from the WSDL specification of the service under consideration. The basic structure of a WSDL (shown in Figure 3.5) includes parts that describe the service with some abstraction, and other parts that describe implementation details (the URI endpoint of the service, for instance).

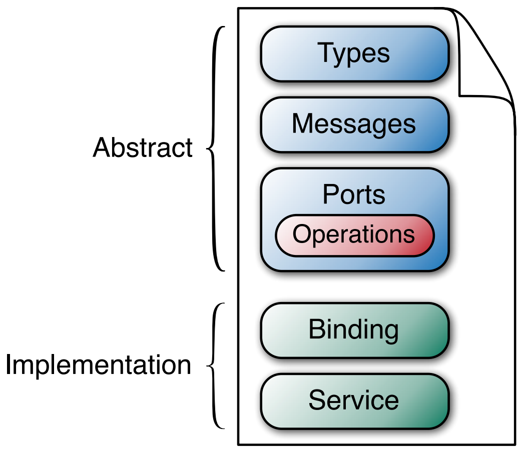


Figure 3.5: The high-level structure of a WSDL document. Each component can be characterized as a more abstract description or as being more about implementation.

XML Type	Java Type
xsd:string	String
xsd:decimal	double
xsd:float	double
xsd:integer	int
xsd:boolean	boolean
xsd:date	java.util.Date
xsd:time	java.util.Date

Table 3.1: Mapping from standard XML simple data types to Java data types.

The types section is for defining complex message types used by the service operations. The outer `<types>` tag wraps a series of tags based on the XML Schema Document syntax (XSD). Though not all tags are supported, wsdl2sim searches for `<xsd:element>` declarations inside `<xsd:complexType>` tags. (Nested complex types are not supported). A simulation data container object is created with fields for each simple data type listed there (based on the translation from standard XML types to Java data types given in Table 3.1). A constructor and getter and setter methods are also provided. This data container provides a mapping from the types being added to the XML schema to types in the simulation. Figure 3.6 shows one portion of the `<types>` section of an example included in the WSDL specification[2], and the corresponding automatically generated Java code.

The messages sent and received by the web service are generated from the `<messages>` section. Objects representing messages all extend the `BaseMessage` class. Each message stub class includes the necessary constructor, fields, getters, and setters to create and use message objects that represent actual messages. The simulation creator can then add or remove code to customize the representation. Code is also generated to instantiate a message object populated with syntacti-

[2]`http://www.w3.org/TR/wsdl`

```
<complexType name="Item">
      <all>
         <element name="quantity" type="int"/>
         <element name="product" type="string"/>
      </all>
</complexType>
```

(a) A `complexType` declaration from a WSDL file.

```
public class ItemComplexType {
   private int quantity;
   private String product;

   public ItemComplexType(int quantity, String product) {
      this.quantity = quantity;
      this.product = product;
   }

   public int getQuantity() {
      return quantity;
   }

   public void setQuantity(int quantity) {
      this.quantity = quantity;
   }

   public String getProduct() {
      return product;
   }

   public void setProduct(String product) {
      this.product = product;
   }

}
```

(b) The Java class produced to represent the `complexType`.

Figure 3.6: A portion of the `<types>` section of a WSDL document and the code produced.

cally correct but semantically meaningless data. Translation follows the mapping in Table 3.1, with the additional mappings from complex types to their Java class representations.

The operations of the web service are created in a series of classes that extend the base class `WebServiceOperation`. The approach and simulation methodology can use any number of models or implementations to adequately represent the behavior of a ServiceOperation. This proof-of-concept implementation focuses on CPU-bound operations that can be modelled using a linear equation with a single variable and margins of error. Extending this implementation to model using other mathematical functions is a straightforward matter of writing code.

The simulation does not rely on this performance profile-driven approach; the key element is that the simulated implementation of the operation adequately predicts the behavior of the operation in the dimensions important to the simulation. For instance, a modelling the operation using a queueing-theory model based on analysis of data or of code could produce accurate predictions for the behavior of interest.

CPU-bound operations, for which CPU time as a function of input size is the primary determinant of performance, make use of `CPUServiceOperation`. These generated classes include performance profiles. The simulation creator is asked if the operation is CPU-bound, and, if so, to provide a performance profile. This performance profile is a linear equation expressing CPU time used per byte of input, plus some constant initialization time, plus some error factor representing the distance between the linear performance profile and the actual service performance[3]. Thus, the CPU time used is $y = mx + b + e$, where m is CPU time per byte, b is initialization time, e is the error; for input of size x bytes. The CPU time is adjusted by a scale factor that represents how much faster/slower the test CPU is than the base processor. The performance profile also includes a similar equation for calculating the size of the returned response. The current implementation uses linear regression on metrics gathered automatically by sending requests of varying size to the operation under various service configurations and recording the results. This metrics generation is done using a tool that is functionally equivalent to WSUnit[4] or soapUI's stress testing features, with the addition of templates: an automated way to populate the content of a series of SOAP messages with semantically meaningful information. See §4.4.1 for more on the performance profile in practice.

In cases where no performance profile is available, the generated code extends the more abstract class, `WebServiceOperation`, instead. The generated code receives incoming requests and returns an empty response. When the performance profile is available, the generated code defaults to receiving a response, waiting an amount of time appropriate given the size of the input, processing queues based on service concurrency, and sending a response whose size based on the size of the input.

A web service is represented by generating a class that extends `WebService`. This foundation class is capable of receiving requests and sending responses over the network. The automatically generated code instantiates operation objects and handles routing of incoming requests to the appropriate operation. It then transmits the response received from the operation. It also includes a computation resource that calculates how long it takes to process requests based on the number of requests currently being processed, the size of the request, and the performance profiles for that type of request. A corresponding testing class is also generated, which simply sends a WSDL-compliant message to the simulated service for each operation and reports what response is received.

Finally, an overall simulation controller object is generated, extending from `BaseSimulator`. This controller loads configuration properties from an XML configuration file and instantiates the generated web services, as well as the network and other essentials.

The generated classes all use the dashboard component to track metrics such as CPU utilization, response time, and number of messages received over time. The final results of wsdl2sim compile and run, but may not be an accurate simulation un-

[4]`https://wsunit.dev.java.net/`

[4]This example describes the CPU-bound formula. The memory-bound and IO-bound formulae proceed analogously: for example, the linear regression expresses the expected memory use as a function of message size. If it depends on a factor other than message size, the base functionality must be extended.

```
<?xml version="1.0" encoding="UTF-8"?>
<p:experiment count="1000" saveOutput="true" xmlns:p="http://www.ualberta.ca"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="..">
 <configDocList>
  <use URI="configure.xml" name="config"/>
  <use URI="./topology[0].xml" name="topology"/>
 </configDocList>
 <do>
   <range from="1" to="1000">
      <document id="topology">
         <changeElement name="location" withId="main">
            <addAttribute name="width" to="3"/>
            <changeAttribute action="increment" by="100" name="height" />
         </changeElement>
         <addElement name="hello" parent="second" withId="hello_element_id">
         <removeElement id="some_id" name="useless" withId="123"/>
      </document>
   </range>
 </do>
</p:experiment>
```

Figure 3.7: A sample JIML document, which increments by 100 the `height` attribute of the `<location>` tag that has id "main" for each of the 1000 experiments, add an attribute `width` to the same element, add a new element `hello`, and remove the element `useless` with id 123.

til the automatically generated classes are further extended. Further extensions can add functionality not supported by the original application but relevant to simulation: for example, reporting metrics, allowing run-time modification of configuration parameters, or simulating failures / service timeouts.

3.3 JIMMIE: Systematic Simulation Configuration

The SASF framework is used for the simulation of a system under a set of systematically varied configurations and work loads. JIMMIE[5] is the framework component responsible for automatically and systematically modifying the simulation configurations, represented in an XML-based syntax. JIMMIE is essentially an XML transformation tool that makes potentially thousands of transformations based on a set of user-defined rules, saves each iteration, uses it to run an experiment, and manages the output produced by the dashboard/metrics module to allow for easy retrieval. Once configured it runs autonomously; the experimenter can view and compare the results as they appear in the database. JIMMIE is designed to work on any XML document with any schema and to handle any number of configuration files.

JIMMIE takes its directions from an XML document that conforms to the JIML (JIMmie Language) schema (*e.g.*, Figure 3.7). JIML defines a number of experiments and identifies to which experiments each instruction applies (start to end, 1 to 500, *etc.*). Within each range, it identifies the tags in the simulation configura-

[5]JIMMIE stands for Just Imagine Many Many Interesting Experiments.

tion XML document and describes the desired values and attribute values for that tag. The values can be specified as an enumerated list of items, as a range with step values, as a loop (like a range but repeated), as an equation, or "hard-coded". Hard-coding is based on the experiment ID: [on the k^{th} experiment, set value v to y]. A JIMMIE plugin based on an existing templating library[6] allows for the use of templates, which allows XML documents to be systematically modified with variable text content (replacing special template tags with defined blocks of text).

The modification of XML may remind the reader of XSLT. JIMMIE offers features akin to that of an XSLT engine specialized to systematically modify an XML file in a variety of ways producing potentially thousands of modified XML files. In addition, it manages invoking the simulation and tracking the location of the results; it is easier to learn and use (instead of being general and broad like XSLT); it can be integrated into an existing codebase, accessed via its library API, or run as a separate application.

3.4 Metrics Engine

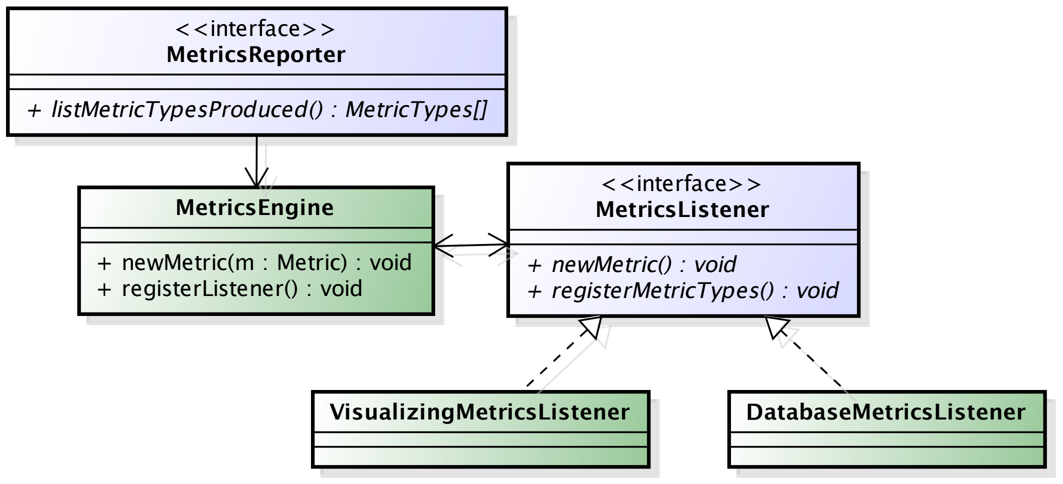
The metrics and dashboard module is a general library for recording, storing, and visualizing metrics. A simulation creator can have any component of the simulation register itself as a reporter, record metrics, and report them via one of several API calls. This module then stores and distributes the metrics to interested objects. Metrics are a three part tuple: a time stamp, a name for the metric, and a value (number or text). The simulation engine uses this module to report certain metrics for every simulation, such as network utliization, execution time, and the current state of services (if they implement the state reporting feature). Similarly, the service testing modules report request and response metrics.

The metrics module specifies *metrics listeners* which are notified each time a metric is added. Two listeners are provided by default. The database metrics listener caches metrics and periodically stores them in a database table (a new table is created for each simulation). The visualizing metrics listener generates graphs for each type of metric it receives and updates the graph as more metrics arrive. It also creates a 600 second rolling average line by default. In short, a simulation creator can store and visualize data with very little effort. Figure 3.9 shows the visualization metrics listener and automatically constructed graphs.

The overall structure of the metrics reporting / observing engine is shown in Figure 3.8. Simulation creators can easily add `MetricsReporter`s and `MetricsListener`s by implementing the appropriate interfaces. Base implementations suitable for extension are also provided.

Beyond the default functionality, a simulation creator can define their own metrics listener to handle metrics in other ways. For example, in one of the case studies (see Section 4.2), a visualization other than the standard graphs was needed, as was the ability to modify simulation parameters at run-time. Figure 3.10 shows a screenshot of the metrics listener created for the task. It uses a pie graph, a line graph, and a custom visualization of client state (green = served, yellow = waiting, blue = currently being served). The sliders on the right can be used to change the server capacity and the network capacity. Another example, extending the dashboard to

[6] `http://code.google.com/p/hapax/`



<<interface>>
MetricsReporter
+ listMetricTypesProduced() : MetricTypes[]
MetricsEngine
+ newMetric(m : Metric) : void
+ registerListener() : void
<<interface>>
MetricsListener
+ newMetric() : void
+ registerMetricTypes() : void
VisualizingMetricsListener
DatabaseMetricsListener

Figure 3.8: A class diagram of selected metrics objects.

allow for run-time modification of the simulation, is shown in Figures 4.11. For another application, a listener capable of modifying the simulation was implemented: an autonomic computing module that would monitor metrics, make decisions, and implement changes.

An additional feature of the metrics engine is the simulation player, which allows one to select an experiment and loads the metrics recorded for that experiment. Using the existing visualization metric listener, these metrics can be "played back" (rewound, fast-forwarded, at various speeds) to review the results of a past simulation.

The dashboard may also be used to modify simulation configuration parameters at run-time, though this requires more complex integration by the simulation creator.

3.5 Emulation Extension

The role of the Emulation extension is to integrate the simulated environment with a real-world environment. Simulated applications allow for inexpensive performance testing and enable the implementation of features that may be easier to develop and test in a simulated environment than in the real world. Validating these testing results or new features in simulation is a first step, and implementing them in the real world is the final step. In between lies testing a simulated feature in the real world.

The extension includes a translation layer that converts simulation `Message`s to corresponding SOAP messages. The simulation creator specifies a mapping between simulated messages and the SOAP messages described in a WSDL, and a mapping between the WSDL operations and the simulated operations. The emulation layer converts simulation-generated messages into SOAP messages, and vice versa. This allows the use of simulation request generators, a mix of real and simulated services, and the interaction of simulated components with real-world services. A second translation layer aggregates metrics from real-world entities and includes them in the

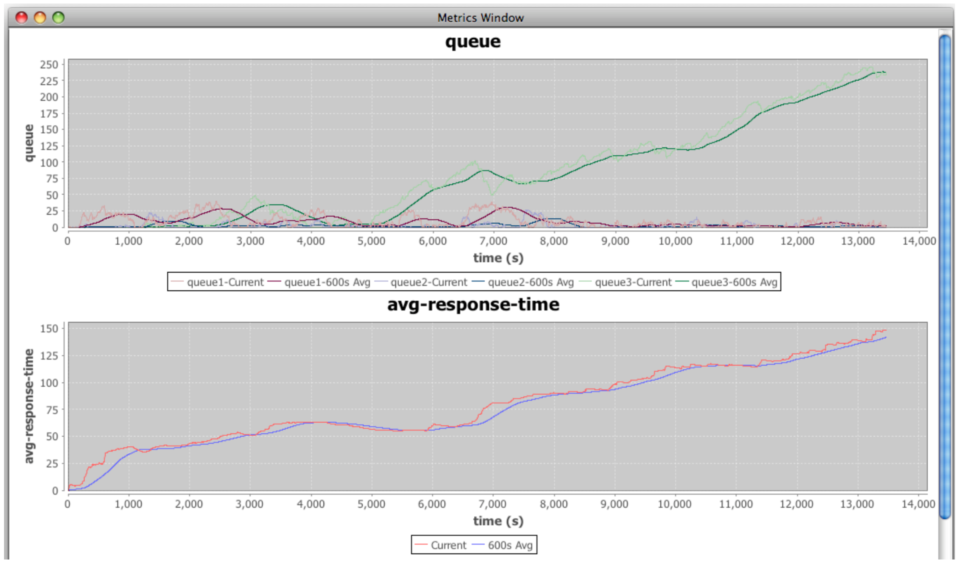


Figure 3.9: The visualization metrics listener displaying two of the metrics being captured by a simulation.

metrics engine; the existing metrics listeners, including the visualization dashboard, function as they do in simulation. The time-driven clock of the simulation engine is governed by actual time.

This extension was employed to test load balancing strategies and an autonomic management feature, both implemented in simulation, using simulation-generated requests.

The use of this extension requires a one-to-one mapping between simulated entities and real-world entities. Simulated `Servers` cannot host real services, and real servers cannot host simulated services.

3.6 Service Testing Modules Extension

The purpose of the STM extension is the generation, transmission, reception, and metrics reporting of simulated service requests to enable testing. Request generation for web servers itself is well understood; however, request generation for services, and in particular for services in simulation is under-developed. This extension provides an extensible `ServiceTester` class, which employs a `Generator` to create requests. The base testing class is notified on each "tick" of the simulation engine, and requests from the `Generator` a set of requests to send for that particular point in the simulation. A set of `Generator` implementations is provided with different strategies for producing this set. A simplified class diagram is shown in Figure 3.11.

Incoming requests vary based on several parameters: the arrival rate of requests, the type of each request (*i.e.*, the operation being invoked, as each has a different performance profile), and the size of each request. These values depend on the following generators, each of which records and reports metrics on how many requests were generated and of what size:

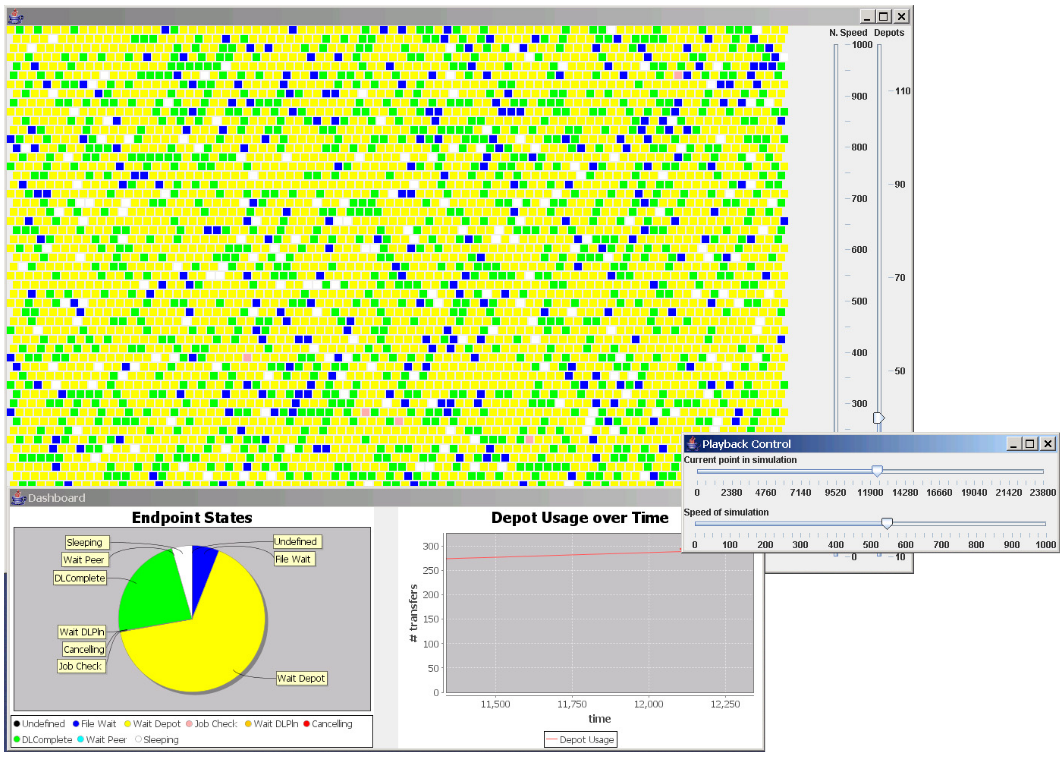


Figure 3.10: A custom metrics listener and the playback control.

- **Fixed**: The type of all requests, the size of each request, and the number of requests are described in a pre-defined configuration. These requests are generated as quickly as they can be processed. This is useful for performing tests that stress the application. This type of generation is used when validating the simulation.

- **Reading**: The generator reads from a log file that specifies a series of requests. For each request, the arrival time, type, and size are defined. This is useful when generating traffic based on logs of the existing service, or to generate identical traffic every time when conducting specific tests.

- **Stochastic**: Requests are generated based on a probabilistic distribution. The number of requests for any given second are determined by a Poisson distribution, where λ is a configurable job arrival rate. The type of each request is based on a weighted random distribution derived from logs of the real-world service. For the size of each type, the request sizes were extracted from the logs, then curve-fitting was used to identify and parameterize an appropriate distribution (usually exponential or Guassian). This is useful for generating roughly consistent but randomly varying traffic that can be used directly or recorded and used as input to the reading generator.

- **Modifiable Stochastic**: Requests are generated as in the stochastic generator, but some parameters can be modified at run-time: the job arrival rate and the weights for the type distribution (Figure 4.11). This is useful for creating

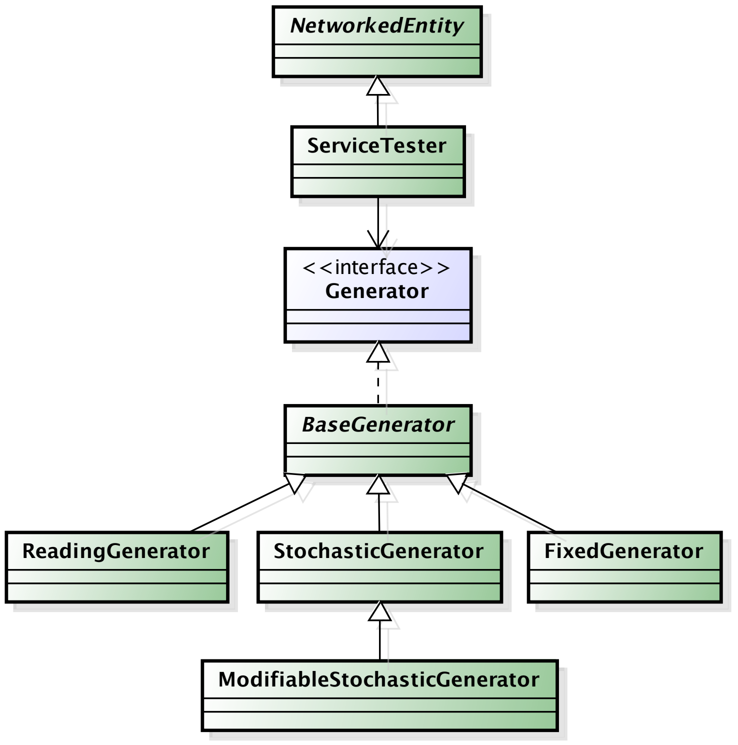


Figure 3.11: A class diagram of selected Service Testing Module (STM) components.

loads that are believably consistent for the same purposes as the stochastic generator.

3.7 Using Simulation-driven Methodologies

The goal of this simulation framework is to reduce the development time required to create a simulation capable of accurately predicting the behavior of a software system. This simulation is anticipated to be useful generally in better understanding the application, but can also be used for specific purposes and methodologies (in Chapters 5 and 6, a variety of simulation-driven methodologies are presented). For completeness, however, described here are the various scenarios in which such a methodology would be useful.

A simulation is useful in the **planning and design** stage. Alternate ways of structuring, architecting, composing, or generally designing the system can be simulated and tested to empirically evaluate design decisions. This framework was not originally intended to be used for this purpose, but is general enough that it could be. Some features of SASF (*e.g.*, the ability to generate requests from logs, and to generate a simulation from an implementation) are not applicable to simulating at this stage.

Simulation is useful in the **testing** stage, where implemented services can be used in combination with simulated services to create a complete service system for testing. This is the approach supported best by commercial simulation tools like soapUI, as described in §2.3.6. SASF is not intended to replace these commercial tools, but does offer integration between simulated and real-world services that could be used for this purpose.

One key area of applicability addressed here is the **pre-deployment** stage. Once all of the services are implemented, a simulation can be created and used to create an appropriate deployment for the service system. Supporting the process of better understanding, and systematically analyzing, an implemented service system is one of the primary uses of simulation contemplated in this work. This is addressed in §4.2 and Chapter 5.

The second primary use of simulation in this dissertation is **post-deployment and run-time**. After deployment, the simulation can be improved using the SASF tools by capturing more accurate performance profiles and logs of actual requests. Alternative configurations can be attempted in simulation to improve some aspect of the service system (here, primarily the performance). Off-line simulation can reason about the service system without interfering with it. This approach is used exclusively in Chapter 6, and in tandem with pre-deployment simulation in Chapter 5.

3.8 Known Issues and Threats to Validity

The prototype of SASF is a proof-of-concept and has a number of limitations, some of which have already been mentioned.

The currently implemented CPU-bound operations, on which the automatic generation depends, require a performance profile that is $O(n)$ in time complexity, where n is the size of the input. To support service operations with different performance profiles, the existing library would have to be extended. This is a matter of writing code; conceptually, the same approach applies. The methodology - curve-fitting to a variety of metrics produced by load-testing the service - still applies, but is not yet implemented.

Complex services, particularly more complex service compositions, will still require hand-coding, though less than would be required without a starting point. Support for composed messages is available for simple sequences, but more complex compositions - conditionals, loops, parallel processing - are not yet implemented in the framework. A simulation creator would have to provide a composition engine appropriate for their needs.

The wsdl2sim is mainly useful when considering a one-to-one simulation, where every operation in the web service has a corresponding object in the simulation. If a higher level of abstraction is required, its usefulness is limited. It is limited to only CPU-bound operations, though the automatically generated code can be modified to extend other service operation simulation templates. It also requires a request-response web service. Chapter 4 illustrates creating a simulation both with and without using automatic generation.

No one other than the author and those who helped implement the simulation framework have used it to simulate applications; no user study of its usability for producing simulated services has been conducted. Details of how to create a simulation and important metrics like lines of code manually written are presented in Chapter 4, where it is clear that less work is required when using SASF. The exact improvement is not quantified. It should be noted that while this is reported as a threat to validity, this level of evaluation actually exceeds current practice in the area.

3.9 Summary

This chapter introduced SASF, a novel simulation framework for services-oriented software systems. SASF is built around extensible libraries for common web service functionality that offer more features than the current state of the art. Development effort is reduced using automatic generation of basic simulations from existing data about the service: from the services WSDL and a performance profile, an executable simulation is generated automatically, ready for extension and enhancement. The basic generated simulation can accurately replicate the performance characteristics of a real service; existing solutions focus on composition issues and not on a minute-by-minute replication of performance metrics. A narrative of the predicted performance of a simulation can be produced using the extensible and flexible metrics gathering support system with its own API, which out-of-the-box is capable of visualizing, recording, and playing back metrics generated during a simulation.

SASF also offers the ability to interact with a running simulation, including not just the usual user interface, but also an API so both the simulation components and external applications can modify configuration parameters during simulation execution. It can integrate simulated components with real-world components by translating requests and messages from the simulated environment to a real-world environment. It can generate requests based on real request logs or stochastic distributions modelled on known request patterns. Finally, JIMMIE is an innovative language and tool to run systematically modified simulations in sequence or parallel to generate data.

These contributions are compared to the current state of the art in §2.3.7; SASF meets or exceeds the state of the art in all areas.

SASF is used to generate, test, and validate simulations in Chapter 4. The simulated versions are validated against their real-world counterparts and prove to produce the same performance results.

Chapter 4

Simulating Service Oriented Applications

Chapter Contents

This chapter demonstrates the utility of the services-aware simulation framework described in Chapter 3, by creating simulations for two service-oriented systems. Those systems, and the details of the simulation approach, are described in this chapter. One simulated system will be used as a test bed for additional simulation-related tools and methodologies in Chapters 5 and 6.

The first case study is a component of a proprietary enterprise-level system that includes SOA-based interfaces, Tivoli Provisioning Manager (TPM). Its focus is on the distribution of data to a large set of endpoints using a distributed architecture (potentially geographically distributed). It is simulated at a higher level of abstraction without using the automatic simulation generation feature; referring back to the simulation engine (§3.1), the least amount of library support (`NetworkedEntity`) is used. Instead of a one-to-one mapping of service operations to simulation objects, a set of composed or related services that implement a set of features is simulated as a single simulation object. This showcases the ability to simulate a complex application at an abstracted level, where a set of web services is grouped into a virtual component and a series of messages into a virtual communications group. This higher-level groups are then the simulated objects. TPM is introduced in §4.1; its simulation is described in §4.2.

The second is a text-analysis tool that provides a public web services interface, TAPoRware, the back-end support to the Text Analysis Portal for Research (TAPoR). Its primary focus is on data processing (CPU-bound), with the movement of data being a secondary but still important concern. It is simulated using the automated simulation generator wsdl2sim, using a one-to-one mapping of service operations to simulation objects, and the greatest amount of library support (`CPUServiceOperation`) is used. The application is described in more detail in §4.3, and its simulation is covered in §4.4. The simulated version of TAPoRware, TAPoR-sim, is used as a platform for demonstrating simulation-driven methodologies in the remainder of this work.

4.1 Tivoli Provisioning Manager

Tivoli Provisioning Manager (TPM) is designed to automate the management of an enterprise's software, servers, storage, and networks. It allows enterprises to manage and change the configuration of IT infrastructure including servers, operating systems, middleware, applications, storage and network devices[1]. Detailed descriptions of TPM and configuration best practices are outside the scope of this paper; consult the TPM information center[2], the deployment Redbook [22], the large-scale deployment white paper [28] and the scalability white paper [37] for more details. This description - and the simulation - is based on version 5.1.1, from 2008. Updates to TPM since (currently 7.2; there was no 6.x version) have changed some components and functionality. This description, and the accompanying simulation, are still suitable candidates for demonstrating the usefulness of SASF.

The specific motivating example is the scalable Software Distribution Infrastructure (SDI), an SOA-based distributed architecture used by TPM to distribute files (install files, patches, configuration files, *etc.*) throughout an organization, and to obtain inventory information in return. It consists of a collection of loosely-coupled services that can be federated to different locations to distribute load and reduce network load. Throughout the remainder of this section, the term TPM is used to mean the distributed component of TPM, the SDI.

One usage scenario is an organization managing several data centers around the world. Communication within the data center is fast, though bandwidth between the different data centers might be lower. The SDI allows the organization to deploy federated services at each site, enough to meet demand at that physical location. A central installation is chosen to host the federators and the non-federated services. A conceptual model of this scenario is shown in Figure 4.1.

The SDI consists of a number of logically or physically separate entities that communicate using web services, as follows:

1. **Endpoints**: Each entity managed by TPM is called an *endpoint.* Common examples of endpoints include the computers deployed to an organization's employees, or collections of servers in data centers. Endpoints all contain an installation of the Tivoli Common Agent (TCA), which handles all communications with the SDI. In general, references to "endpoints" can be interpreted as references to the TCA installation on the endpoint.

2. **Central Manager**: The central manager of TPM includes a database for persistent storage, the user interface, and most of the functionality of TPM not included in the SDI. Provisioning requests originate here, whether invoked by

[1] `http://www-306.ibm.com/software/tivoli/products/prov-mgr/`

[2] `http://publib.boulder.ibm.com/infocenter/tivihelp/v20r1/index.jsp`

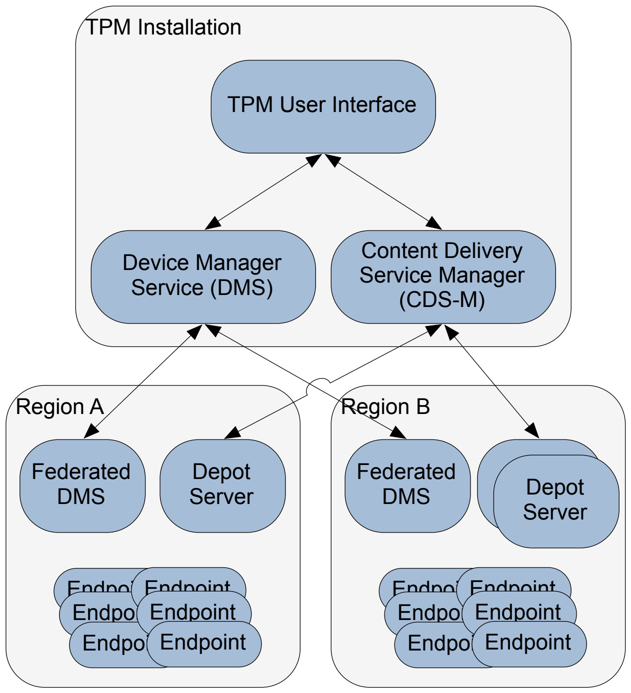


Figure 4.1: A conceptual model of the SDI in an example scenario.

an administrator through the user interface, called by an external command, or triggered autonomously. Provisioning requests are *jobs*, and include lists of files to download and actions to perform and are targeted to a defined set of endpoints. When a job is created, it is sent to the Device Manager Service, and the files needed are uploaded to the upload server.

3. **Device Manager Service (DMS)**: Endpoints regularly poll this service to obtain lists of outstanding jobs, and report the success or failure of these jobs. To share the workload, DMS services can be federated to servers in different regions. When DMS services are federated, the federator is co-located with the central manager, and is responsible for distributing job information to, and collecting job status information from, the federated agents.

4. **Content Delivery Service management center (CDS-M)**: This entity's primary role is to manage content uploaded by the central server, usually files needed to execute jobs. The CDS-M maintains a directory of these files, oversees the distribution to the appropriate CDS Depots (see below), receives requests for files from endpoints, returns download plans (listing appropriate download depots), and collects statistics from the endpoints and depots.

5. **Content Delivery Service Depot (CDS Depot)**: These are repositories of content for distribution to endpoints. The CDS-M distributes content on the depots, and endpoints request content from the depots when needed to complete their assigned jobs. Depots are placed strategically throughout an organization to meet the downloading needs of the endpoints, based on the expected jobs. File distribution is the one aspect of SDI that runs outside the SOA; file transfers are done using an efficient protocol.

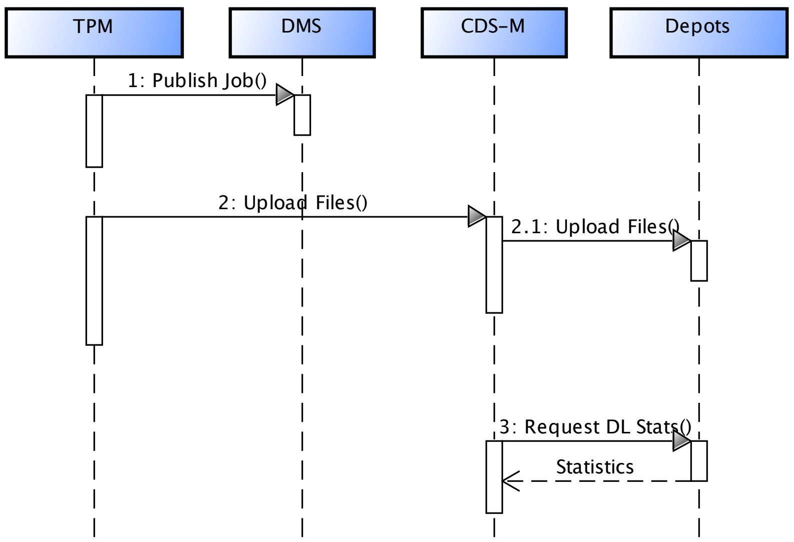


Figure 4.2: A high-level sequence diagram of job publishing.

6. **Regions**: Regions are not technically entities, but can be thought of as groupings that organize TPM into manageable areas. Each endpoint belongs to a region and communicates primarily with the federated DMS and the depots assigned to service that region.

These entities collectively execute provisioning requests (jobs), a process that involves first a top-down push of instructions and files, and then a bottom-up pull of the same information, as described in the following subsection. §4.1.2 describes the current state of the art for capacity planning for TPM and the motivation for a simulated implementation.

4.1.1 Executing Jobs using the SDI

Executing jobs on endpoints requires two separate but related processes. The first is *job publishing*, and the second is *job distribution*.

A high-level sequence diagram for job publishing is shown in Figure 4.2. Step 0, not shown, is for a job to be created using the user interface (or the API). The job includes a set of actions to be executed and a list of files required by the job. The administrator defines which jobs apply to which endpoints. For example, jobs can apply only to endpoints in a certain defined region, or only to endpoints with a certain operating system. Once the job is created, it must be published to the other services in the SDI. First, the job files are uploaded to depots, a process overseen by the CDS-M. Second, the job is sent to the DMS (in fact, the DMS polls TPM regularly to see if new jobs are available). The DMS stores the job.

Job distribution is shown in Figure 4.3. This process is driven primarily by the endpoint, which periodically contacts the DMS to request any pending jobs. When the endpoint receives a job, it first contacts the CDS-M to request a download plan. The CDS-M uses statistics from previous file downloads, the network topology, and the proximity of endpoints to depots to create a download plan listing a set of depots to contact such that a level of service quality is maintained. The endpoint

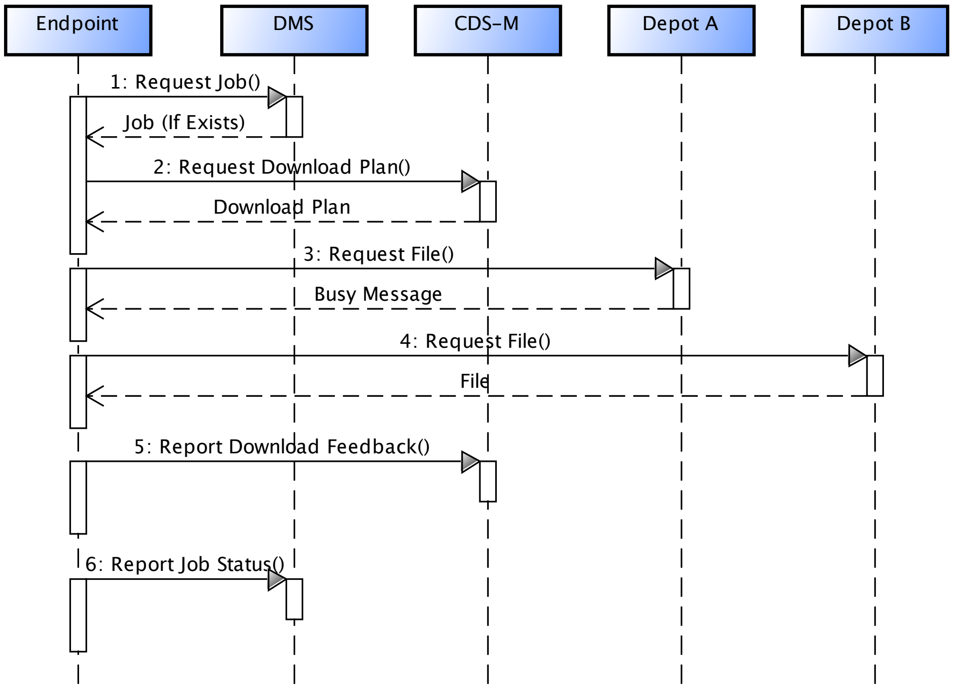


Figure 4.3: A high-level sequence diagram of job distribution.

then contacts the first depot to request the file. If the depot is busy, the endpoint retries later, or attempts to contact a secondary depot as listed on the download plan. Once it receives the file, the endpoint executes the job: applying a security patch, updating virus definitions, whatever the task may be. The results of the job (success or failure) are reported to the DMS. The DMS collects results and when enough results are received, notifies the central server that job results are available.

One variation on this sequence is that endpoints can download content from their peers, if peer-to-peer downloads are enabled. This shifts traffic from the wide-area network to the local-area network. When enabled, the CDS-M returns download plans that include other endpoints as sources. This is a useful feature for distributed sites not large enough to justify a dedicated depot server and with only a low-speed connection to other depot servers. Peering can also work in concert with depots; experiments have shown a 50% reduction in distribution time when both peering and depot servers are used on a topology with 55,000 endpoints [28].

4.1.2 Capacity Planning for TPM

The current state of the art for creating configurations for TPM, and in particular the SDI, is a manual process. As described in IBM white papers [28, 37], an emulation environment is manually set up and configured in a testing lab. Virtual machines running on 45 physical machines emulate up to several hundred thousand endpoints. Bandwidth throttling is used to emulate network speeds and closely model the target network topology (this part of the process is semi-automated). Using some rules-of-thumb known to the development team, regions are defined, and depots and federated DMS agents are deployed to these regions. A central server is installed alongside the DMS federator and the CDS-M. Based on the ex-

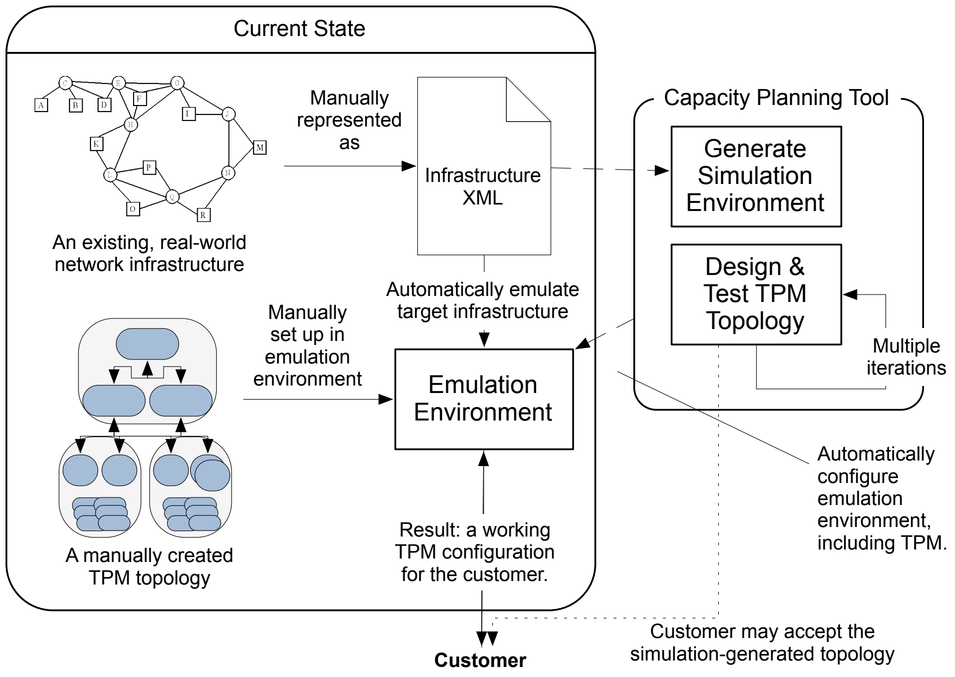

Current State
Manually represented as
Infrastructure XML
An existing, real-world network infrastructure
Automatically emulate target infrastructure
Manually set up in emulation environment
Emulation Environment
A manually created TPM topology
Result: a working TPM configuration for the customer.
Customer
Capacity Planning Tool
Generate Simulation Environment
Design & Test TPM Topology
Multiple iterations
Automatically configure emulation environment, including TPM.
Customer may accept the simulation-generated topology

Figure 4.4: Current process plus the capacity planning tool.

pected use of the infrastructure, jobs are configured and deployed using TPM. A variety of performance metrics are collected and use to refine the initial deployment.

Once the emulation is running, TPM is deployed in the actual target environment based on the working test deployment. The process of refining this deployment varies; however, changes are made to the initial configuration, and continue to be made as the topology changes or as the actual use of TPM varies from the expected use.

The expected role for simulation is in the form of a capacity planning tool. Configuration changes in the actual environment are very costly in terms of time and resources. Emulation changes are easier, but still occupy time and resources given the trial-and-error process of refining a configuration. These changes also at present occur at the development team's time and expense. Our simulation method offers negligible costs, but could have reduced accuracy. The expected role is to reduce the amount of real-world or emulation-level work required. Figure 4.4 shows how our capacity planning tool (once complete) would fit in with the TPM deployment process (and similarly, how it could fit with any deployment process).

The capacity planning tool, built as a simulation of TPM as described in §4.2 and called TPMsim, reads from the same topology configuration file as the emulation environment and builds a simulation environment. The infrastructure, topology, and a set of jobs are all determined by configuration documents. TPMsim can be used interactively, or automatically using JIMMIE-based systematic modifications to the configuration files, to determine an appropriate TPM configuration. The candidate configuration can be replicated in the emulation environment for validation, using configuration files generated by TPMsim, or provided to a customer to employ in their installation.

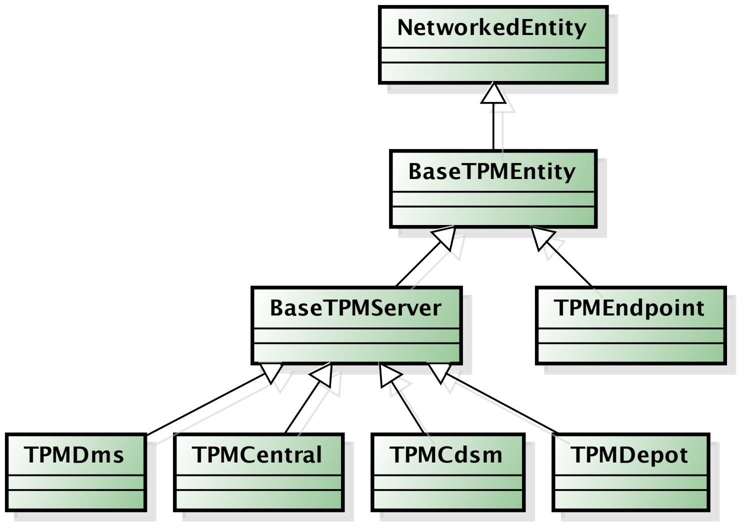


Figure 4.5: The class hierarchy implemented for the simulation.

A simulation TPM could also be useful in other ways. For instance, at runtime the download plans produced by the CDS-M could be modified in response to their simulation-predicted impact. At development itme, modifications to the architecture could be tested before the software is released.

4.2 Simulating TPM

The 6 entities described in §4.1 are treated as separate entities in the simulation; though each may comprise many different web services and operations, each is modelled as a single entity. Figure 4.5 shows the classes implemented to replicate the behavior of the TPM entities; each is extended from `NetworkedEntity`, part of the library support provided by the SASF simulation engine (§3.1). Each communicates with other entities using messages that implement the library `Message` interface.

In capacity planning for simple web requests, often the consumers of services are modelled only by a request arrival rate, with the performance metric being response time. In TPM, the endpoints are the entity most analogous to consumers; however, the behavior of endpoints is more complex. The types of messages an endpoint sends, and when it sends messages, is closely controlled based on timing and on its state. For this reason, the SASF service testing modules are not suitable; instead, an endpoint class is created, similar to but distinct from the TPM services. Thousands of endpoint objects are instantiated to simulate the behavior of endpoints. The implementation of endpoints is essentially a state machine, where the transmission or arrival of messages and certain timed events spur the moves from one state to another.

To illustrate this state transition, consider the sequence diagram in Figure 4.3. The endpoint starts out ‘sleeping’; it has no contact with the SDI during this time. At step 1, it sends a job request to the DMS and enters the “waiting for job” state. When the job response arrives, it either re-enters the sleep state, or sends a download plan request and enters the “waiting for download plan” state. When it receives the download plan, it will request a file and enter the “downloading file” state. If

```
<?xml version="1.0"?>
<topology>
  <location name="winward" count="1" endpointCount="2000">
  </location>
  <location name="mahwah" count="1" endpointCount="2000">
    <wan speed="10000000" targetLocation="winward"/>
  </location>
  <location name="winward-branch" count="240" endpointCount="100">
    <wan speed="1540000" targetLocation="winward"/>
  </location>
  <location name="winward-vpn" count="1" endpointCount="1500">
    <wan speed="10000000" targetLocation="winward"/>
  </location>
  <location name="mahwah-branch" count="240" endpointCount="100">
    <wan speed="1540000" targetLocation="mahwah"/>
  </location>
  <location name="mahwah-vpn" count="1" endpointCount="1500">
    <wan speed="10000000" targetLocation="mahwah"/>
  </location>
</topology>
```

Figure 4.6: The topology XML document used to configure the simulated networks.

```
<?xml version="1.0"?>
<infrastructure>
  <cam host="tca0103" registrationPassword="******"/>
  <cds serverURL="https://tca0104:9046"/>
  <dms-region name="central">
    <server name="tca0106.tca.tod.torolab.ibm.com" address="192.168.2.106"
       protocol="https" port="9046" uri="/dmserver/SyncMLDMServlet"/>
    <server name="tca0107.tca.tod.torolab.ibm.com" address="192.168.2.107"
       protocol="https" port="9046" uri="/dmserver/SyncMLDMServlet"/>
    <server name="tca0151.tca.tod.torolab.ibm.com" address="192.168.2.151"
       protocol="https" port="9046" uri="/dmserver/SyncMLDMServlet"/>
    <location name="central"/>
    <location name="backup"/>
    <location name="branch"/>
    <location name="ATM-1"/>
    <location name="ATM-2"/>
  </dms-region>
  <cds-region name="barclays">
    <depot name="tca0105.tca.tod.torolab.ibm.com" address="192.168.2.105"
       location="central"/>
    <depot name="tca0108.tca.tod.torolab.ibm.com" address="192.168.2.108"
       location="central"/>
    <zone location="central" peeringEnabled="false"/>
    <zone location="backup" peeringEnabled="false" peeringMaxSpeed="3000"/>
    <zone location="branch" peeringEnabled="true" peeringMaxSpeed="3000"/>
    <zone location="ATM-1" peeringEnabled="false"/>
    <zone location="ATM-2" peeringEnabled="false"/>
  </cds-region>
</infrastructure>
```

Figure 4.7: The document describing the TPM infrastructure.

a busy response is received, it will contact the other depots; if all busy responses are received, it will enter a "waiting for available depot" state where it sleeps for a period of time before waking to request files again. Once it finishes downloading the file, it runs the job, reports its status, and returns to sleep.

Since endpoints spend the majority of their time dormant, and since the engine must be capable of simulating up to hundreds of thousands of endpoints, each endpoint does not run in its own `Thread`. Endpoints wake up after timed intervals or whenever they receive a message, and request a thread in which to run. They are then allocated a thread from a pool of threads, which they use to complete their processing.

The other entities are considered service providers. Each runs in its own thread. For performance reasons, they do not constantly check for new messages; rather, the thread sleeps and wakes frequently to check for new messages. At this time, the load at the hardware level on most service providers is not simulated though it is recorded; they are capable of responding to requests almost immediately. The exception is the depot, which limits the number of files it will serve at any given time. This is a reasonable simplification because the depot is understood to be the primary bottleneck in the SDI.

The existing metrics engine is used to report some metrics (requests / hour for the services, network utilization, *etc.*), but the more interesting metrics component is a `MetricsListener` implementation that visualizes the current state of all the endpoints. Endpoints were augmented with `MetricsReporter` functionality, and periodically report their state. A dashboard was created that compiles live statistics on how many endpoints are in each possible state. Mentioned previously in §3.4 and shown in Figure 3.10, it provides a single-glance dashboard view of job completion status. This can be run at simulation time, or recorded and played back later. The dashboard also uses the configuration API to change the simulated application configuration at runtime, allowing the user to adjust network capacity and depot capacity to see the impact on the application.

TPMsim uses the existing emulation configuration documents as direct input, using the simulated entities to mimic the network topology (Figure 4.6) and the TPM infrastructure (Figure 4.7). These can be systematically modified using JIMMIE. Implementation-specific information (*e.g.*, IP addresses) is not included, but otherwise every component is simulated.

The final version of TPMsim is almost 3,000 lines of code developed manually; the total framework is almost 9,000 source lines of code (Figure 4.8). This case study offers a unique comparison point. TPMsim was first implemented entirely using hand-coding, without using the simulation framework[3]. This original pre-framework version of TPMsim was approximately 7,000 lines of code. The use of the framework, even without using automated features or the most detailed libraries, requires almost 60% fewer lines of source code.

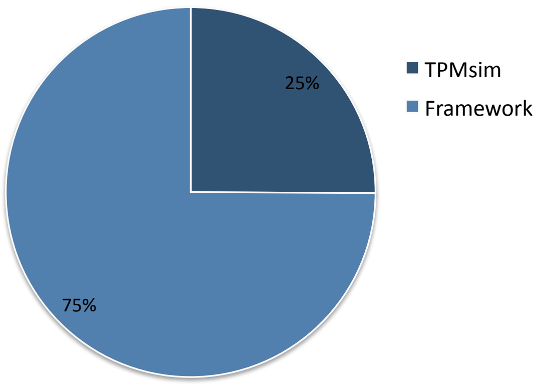


Figure 4.8: Source lines of code for the TPMsim-specific code and the framework.

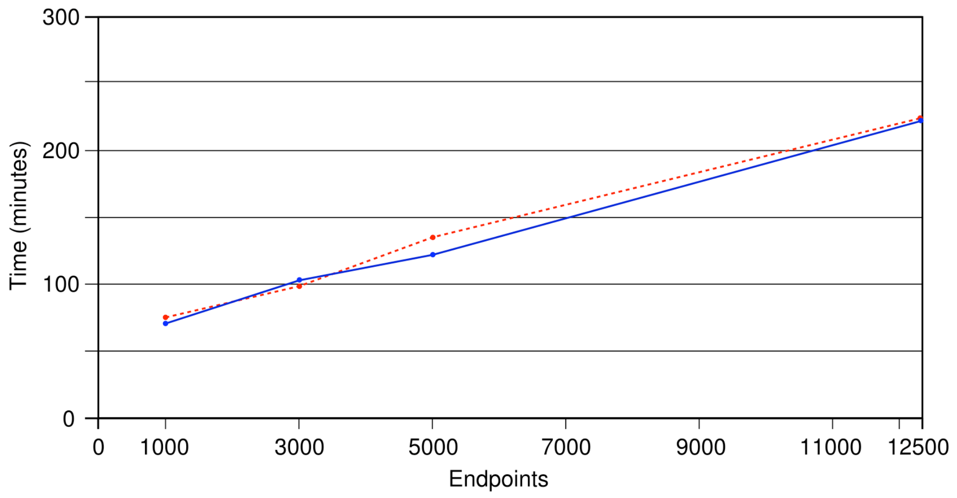


Figure 4.9: Time required to send a 10M file to endpoints, simulated vs. emulated.

4.2.1 Validation

The primary way in which the SDI differs from a 'typical' service composition is its role of file distribution. Network utilization is the expected bottle-neck, as opposed to individual server performance [37, 28]. Thus, accurate timing on message sending and network usage were priorities for this prototype. The primary metric collected in this evaluation is the amount of time required to distribute a file to all endpoints.

This metric is compared to the emulated performance tests described by Leitch *et al.* [37]. They describe a topology consisting of a single DMS and one depot server. The number of endpoints varies between 1,000 and 12,500. They test the time required to distribute a 10 megabyte file. These tests were done using TPM version 5.1. The white paper identifies some but not all of the values used for the configuration parameters. The important parameters, and the values used in this validation, are listed in Table 4.1. These values were based on the documentation and TPM best practices, and are reasonable assumptions.

[3]In fact, portions of the SASF implementation are generalized versions of the original TPMsim implementation; features general to service-oriented simulations were pulled out into a common set of libraries, tools, and features.

Parameter	**Value**
Speed of the network	100 Mbps
Max. speed per file transfer	2.5 Mbps
Depots max. number of simultaneous file transfers	40
Endpoint sleep time between checking for jobs	3600 s.
Endpoint sleep time waiting for available depot	600-2400 s.
DMS sleep time between checking for jobs	600 s.

Table 4.1: Configuration parameters.

To compare the simulation environment to the existing emulation environment, TPMsim was configured and executed with a job that included a 10 megabyte file. Simulation performance is an important consideration when simulating tens of thousands of entities over extended periods of time. The experiment described above with 12,500 endpoints and a 10M file was run at 25 times real-world speed (that is, 25 simulated seconds went by during each real second). The simulation ran on a Pentium 4 2.80 GHz machine. Depending on the state of the simulation, it required 10-20% of the CPU and less than 35MB of memory. The simulation can be either sped up or slowed down, depending on the available resources and performance.

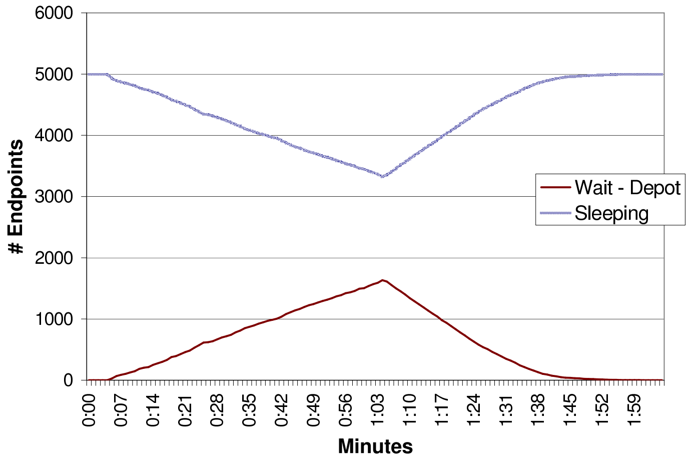
The total time to distribute this file to 1000, 3000, 5000, and 12500 endpoints was compared to the same metric from the Leitch results, and are shown in Figure 4.9. The solid line is the simulation; the dotted line is the emulation. Evidence indicates that for this limited evaluation, our simulation is producing similar results.

Using the state observation implemented in the metrics engine, the state of each endpoint was captured every minute during the 5,000 endpoint experiment. Figure 4.10a shows the number of endpoints sleeping and waiting for depot, the two most common states, over time. An endpoint "waiting" is a potential indicator of an overloaded depot. Figure 4.10b shows the number of endpoints actively downloading a file (or attempting to download file) and the number of endpoints doing any other task (checking for jobs, waiting for a download plan). Note the difference in the scale of the y-axis between the two graphs. There is no data with which to compare these figures, so they may not be an accurate reflection of what the endpoints are doing in reality.

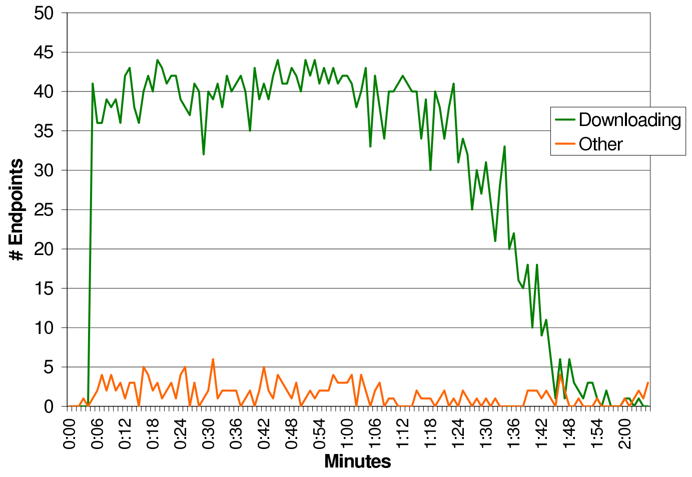
4.2.2 Known Issues and Threats to Validity

Some aspects of TPM are present in TPMsim but not validated here; most notably, configurations with a federated DMS and configurations using peer-to-peer transfers. Data from real-world or emulated installations of TPM is not available for these features. There was an emulation lab at the software development facility, but it is not available for use. Alternatively, this data could be produced by creating a local emulation environment, either on local hardware or on leased machines from a cloud computing provider such as Amazon. Approximately 5000 real or virtual endpoints would be required to produce meaningful results. Fine-tuned control over the network would also be required to emulate varying network topologies.

Similarly, though the simulation offers metrics tracking and recording, without



(a) The number of endpoints waiting for a depot and sleeping.



(b) The number of endpoints actively downloading a file and doing anything other than waiting, sleeping, or downloading.

Figure 4.10: Custom visualizations in the TPM simulation dashboard.

real world data with which to compare, validation of these metrics is not possible. Only the most crucial metric, total distribution time, is shown to match that of the emulated environment. For this reason, TPMsim is not used in further demonstrations of simulation-driven methodologies.

The data to which the simulation-generated data is compared appears in a white paper that does not describe the values used for a variety of configuration parameters. Statistical analysis of these results is not possible as detailed results of their testing is not available. However, the values assumed by the simulation are believed to be reasonable estimates, as indicated by colleagues of the authors of the source paper. They follow best-practice recommendations from the TPM deployment team. This threat could be further reduced by producing a validation data set where the configuration values are specified.

The load and performance at the hardware level (CPU, memory, or disk bottlenecks) for most TPM components is not simulated. The number of attempts to access each component per time unit is recorded, but the component is capable of

Operation	Description
List Words	Count, sort and list words of the source text in different orders. It can strip words user specified from the list, or common stop words.
Concordance	Find user specified word/pattern anywhere in the text with its context. Display the result in KWIC format.
Co-occurrence	Find user specified pattern and co-pattern in a given length of context. Highlight the patterns in the result.
Word Cloud	Using the top k frequent words reported by List Words, generate a visualization of those words and their relative frequency.

Table 4.2: TAPoR operations (from *http://taporware.mcmaster.ca/˜taporware/textTools/*)

responding to requests almost immediately (not including network transfer time). The exception is the depot, which limits the number of files it will serve at any given time. This is a reasonable simplification because the depot is understood to be the primary bottleneck in the SDI. A more detailed simulation could be authored which would use the computation resource tracking features of SASF to produce load and performance results for all TPM components.

4.3 Text Analysis Portal for Research

The Text Analysis Portal for Research (TAPoR) is a web-based application that provides a suite of text-analysis tools to scholars and researchers in the Digital-Humanities [63]. It includes a front-end portal and a back-end web service called TAPoRware.

TAPoRware is a single web service with 44 operations, implemented in Ruby using the SOAP4R libraries[4]. Each operation runs in time linear with respect to the size of the input. The subset of operations covered here includes listing the words with their counts, generating word clouds, finding the use of a word in context (concordance), and finding two words located near each other in text (co-occurrence). These operations are described in (Table 4.2). In addition to accessing the functions via web services, they may also be accessed via CGI.

Many of these operations appear "in triplicate", where the exact same functionality is applied to documents in different formats, namely plain-text, HTML, and XML. Some operations are implemented for only plain-text. When HTML and XML are explicitly supported, the functionality offered allows one to identify from which tag or tags to extract text. As the service matured, it created one operation for extracting text from HTML which could be composed with other operations, and operations implemented since this happened have only included plain-text versions.

A typical usage scenario begins with the end user identifying a piece of text to analyze with a given tool. Then via a web-services client or the web front-end, the user selects the relevant parameters for the analysis. Common options include word stemming, excluding stop words, and the number of words/sentences/paragraphs to display results in-context. In addition, each operation has its own configuration

[4]`http://rubyforge.org/projects/soap4r/`

options. The entire text to be analyzed and the configuration options are encoded in a SOAP request and transmitted to the web service.

The primary challenge facing TAPoRware is performance. The service performs CPU-intensive operations on potentially large text corpora. Though it is capable of handling a small number of users, performance challenges can become critical with an increased user population, which can critically throttle its adoption. The web service as implemented extends Ruby's Webrick server and can process only one request at a time; this cannot be changed in a configuration file. Though reasonable for single-processor machines, parallelism via an upgraded multi-core or multi-processor machine will not improve response time.

4.3.1 TAPoRware 2.0

To address the parallelism problems, the base system was modified in two important ways. First, the ability to process multiple simultaneous requests was "faked" to allow the software to take advantage of multiple processors or cores when available. This was implemented by running multiple instances of the single-request version on different ports on the same server. Incoming requests are services by one of the running processes. Any number of processes can be run, depending only on the resources available to the server: this parallelism will only improve performance when there is more than one processor.

To balance the requests among all running processes, Apache 2[5] was installed and configured to listen on a single port and load balance all incoming requests among the available single-request instances[6]. The `bybusyness` load balancing algorithm is used; it keeps track of the outstanding requests for each process and sends requests to the process that is least busy[7]. The underlying implementations are not modified in any way.

Second, instead of a single web service containing all of the different operations, the service is split into many web services, each containing a subset of the different operations. The division is a natural one, and is used to introduce the ability to partition the service over multiple servers (as opposed to replicating it). This partitioning makes the problem of composing services more interesting. Once again, the underlying implementations are not modified in any way.

This modified version, called TAPoRware 2.0, is the basis of the simulation described in the following section. Throughout the remainder of this work, "TAPoRware" and "TAPoRsim" will be used to refer to TAPoRware 2.0 and its simulation, respectively.

4.4 Simulating TAPoRware

This case study describes in more detail the process (following the SASF methodology and using SASF tools) for generating a one-to-one simulation of TAPoRware, TAPoRsim, where every real-world entity is mapped to a simulated entity. This allows more of the support provided by SASF - libraries, automatic generation,

[5]`http://httpd.apache.org/`

[6]The same approach can be used to load balance over multiple servers.

[7]`http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html`

metrics gathering, configuration APIs - to be showcased and shown to be effective. TAPoRsim is used throughout the remainder of this work as a motivating example or a platform for demonstrating simulation-driven methodologies and tools.

There are four steps in this process: 1) building a performance profile of the service, 2) automatically generating code and customizing the code to better mimic the service functionality and/or to offer simulation features (*e.g.*, additional visualizations, run-time configuration), 3) validating that the simulated service matches the real-world service, and 4) running the simulation systematically.

4.4.1 Building Performance Profiles

To determine the performance of a TAPoR operation, a performance tool sent repeated requests of varying size and with varying levels of concurrency. These requests were automatically generated using a combination of soapUI and a template engine. The empty SOAP messages generated by soapUI were augmented with template tags: notations where custom content is needed. Those tags were replaced by a template engine (Hapax[8]) with values from a configuration file. Most input options except the text to analyze were set to default values. The final SOAP messages included data to be processed, text of several fixed lengths (from 50 up to 175,000 words, or 250 to 1,000,000 bytes). The text was from English-language novels on Project Gutenberg (most frequently Mary Shelley's *Frankenstein* or *Last Man* or portions thereof).

These sample messages had to be both syntactically and semantically correct, as error messages produce a different performance profile than actual responses. Syntax was handled automatically, but semantics requires a user who understands the service to enter appropriate values. This could be automated by observing real traffic to the service if there is an existing deployment, or by using default values specified in documentation.

TAPoRware was installed on a web server running in a virtual machine (with a dedicated processor and reserved memory), which was configured with varying amounts of memory. The Apache Benchmark (ab) tool[9] was used to send the SOAP requests to the service with varying levels of concurrency. Where possible, only the plain-text tools were tested, as the performance of the plain-text, HTML and XML versions of the operation did not differ significantly.

To construct a profile, the amount of time required to process a single request of a given size was calculated. Requests of several varying sizes were sent repeatedly to acquire this information. Next, the time required when concurrent requests were sent was calculated; this determined the effect of concurrency on performance. Performance scaled linearly: when concurrency was doubled, so did response time. This indicates that a single request was sufficient to completely use the single processor of the test machine. From this data, a linear regression was computed, as well as the error in the regression (the distance of the line from the actual data). This linear regression represents the performance profile. It is limited to operations with performance of $O(n)$ or better. Finally, stress testing was used to identify the limit of the service: that is, the point at which requests start failing. The data for the List Words operation with concurrency of 1 is shown in Table 4.3.

[8]`http://code.google.com/p/hapax/`

[9]`http://httpd.apache.org/docs/2.0/programs/ab.html`

Bytes	Response Time (ms)	Time per Byte
250	62	0.2499
56469	3086	0.0547
217012	8010	0.0369
416818	13537	0.0325
1000732	29235	0.0292

Table 4.3: List Words operation response times for input of varying size submitted one request at a time.

4.4.2 Generating and Extending Code

Using a portion of the TAPoRware WSDL and the performance profiles from the previous section as input to wsdl2sim (§3.2), a simulation to be run on the simulation engine was generated automatically. This automatically generated portion was actually capable of replicating the performance of the service and its operations at a granular level. It is used in the validation process.

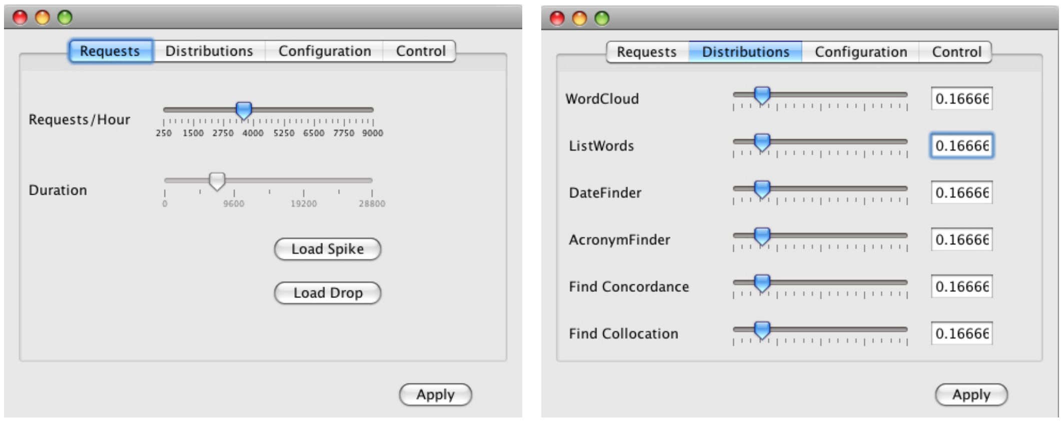
However, to make the simulation tool more useful as an experimental platform, features not included in the original application were added - generating a topology from an XML file, additional metrics capturing code to create a dashboard, a simulated load balancer based on load balancers used elsewhere but never with TAPoR, and extending the configuration files to allow for JIMMIE integration. Additionally, the ability to modify crucial elements of the configuration at run-time was added (Figure 4.11).

Future uses of TAPoRsim vary the inputs (systematically and repeatably) to predict behavior in different conditions. The behavior of the simulated application is determined by two main factors: (a) the configuration of TAPoRware in the simulation, and (b) input to the simulation (basically, incoming requests and their parameters). The latter are configured using the Service Testing Module extension (§3.6). For the former, TAPoRsim is configured using an XML topology configuration file supported by the simulation framework. This file identifies the servers, services, and operations that form the simulated web service(s). It includes unique identifiers and the names of the classes that implement the topology in the simulation. Modifying this file and another `Properties`-style file are sufficient to modify the configuration of the simulated application. These configurations can be modified systematically using JIMMIE. A sample topology descriptor was previously shown in Table 3.4.

This extended version was used to generate data later employed when reasoning about the application.

Using wsdl2sim, 1,145 source lines of code (SLOC) were created for a 6-operation sample simulation. (In comparison, the framework itself is 8,500, Figure 4.12.) The customizations collectively account for 1000 SLOC (primarily UI, run-time configuration, and metrics). An additional 1,000 SLOC were generated by a Java UI creator that was used to create portions of the new user interface. A breakdown showing simulation-specific code is shown in Figure 4.13.

The resulting simulation is the base TAPoRsim. Add-ons to this base are used in subsequent chapters to demonstrate approaches or tools, but do not form part of the core simulation.



WordCloud 0.16666
ListWords 0.16666
DateFinder 0.16666
AcronymFinder 0.16666
Find Concordance 0.16666
Find Collocation 0.16666

(a) Modifying request arrival rate (b) Modifying request type distributions

Figure 4.11: The modification tool for modifying parameters of request-generation in real time.

Request Block Configuration	**Mean Response Time Mean Absolute Error**
c=1, size=56000	11.8%
c=1, size=217000	11.7%
c=1, size=1000000	1.3%
Average	8.3%

Table 4.4: Mean Absolute Error when comparing the linear regression prediction for the List Words operation to the real-world operation.

4.4.3 Validation

To evaluate the accuracy of the simulation, the simulation-generated metrics are compared to the real-world metrics for each operation. Using Apache Benchmark, requests of various fixed sizes were sent to the real-world operation. The exact response time (to the millisecond) was recorded. This data represented reality.

The test requests were organized into blocks of requests. Each block included 50 requests with the same properties - same request size, same level of request-sending concurrency (with concurrency=1, only request is sent at a time). The sizes were 56469, 217012, or 1000732 bytes. The concurrencies were 1, 2, 5, and 10.

A single TAPoR service was deployed to a single-processor Pentium 4 machine with 2 GB of RAM and a 100Mbps network connection[10] between the client and the server (*i.e.*, network delay is a non-factor). It was configured to process up to two requests simultaneously, queuing any other requests (a single-processor machine will not benefit from additional parallelism). The metrics collected were response time and total time to process the blocks of request (*i.e.*, throughput).

This real-world environment was replicated in simulation - same hardware profile, same configurations, and same blocks of requests. The automatically generated code was used without modification.

[10]There is no motivation for hardware selection other than availability.

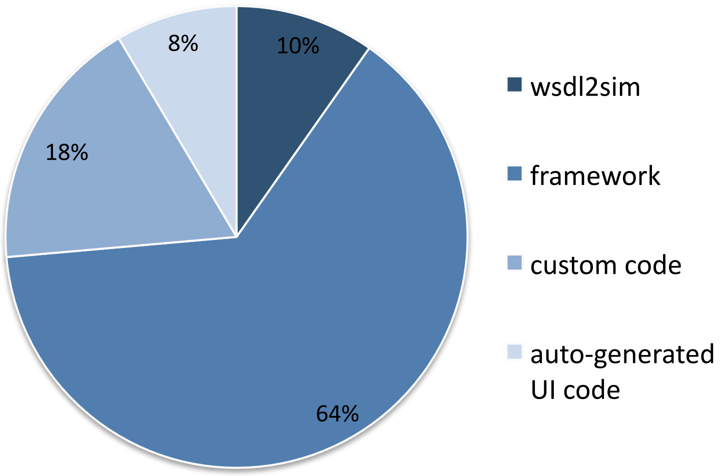


Figure 4.12: The proportion of hand-written code versus code provided by a library or automatically-generated.

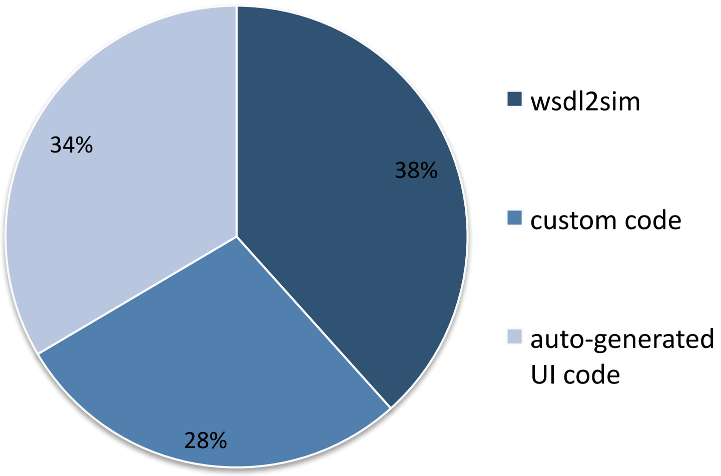
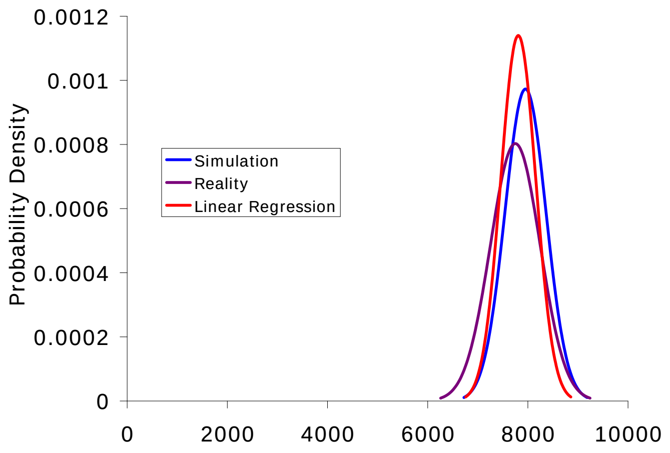
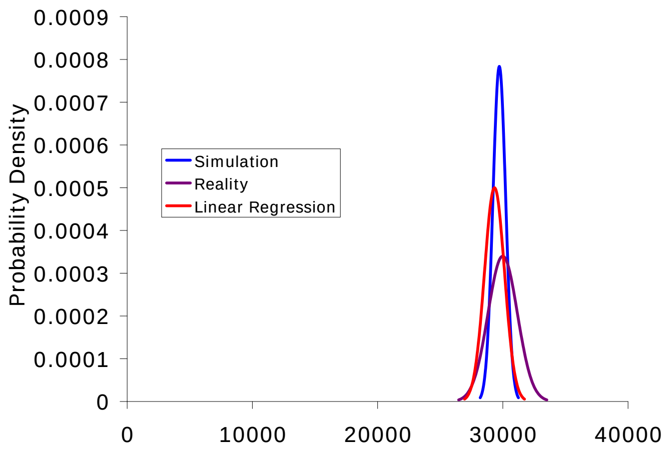


Figure 4.13: The proportion of hand-written code versus automatically-generated, excluding framework code.



(a) Input size 217 KB



(b) Input size 1,000 KB

Figure 4.14: Normalized distributions of service response times to single concurrent requests, comparing reality to the theoretic model (linear regression) and the implemented model (simulation).

For brevity, details are provided only for the results of evaluating the List Words operation. The results do not differ substantially across all operations.

Visually, Figure 4.14 shows the normalized distribution of response times for the List Words service under two input sizes for the real service, the simulated service, and for the response times predicted by the linear regression model of the service.

A two-way ANOVA (response time in milliseconds versus real/simulated) and found that the two sets of results were not statistically different (response time in milliseconds, $p < .05$).

Recall that the simulation is built on performance profiles generated from linear regressions. The predicted response time based on the linear regression (excluding the expected error information) was compared to the actual response time as measured empirically. The linear regression only predicts single-concurrency performance, so only three test request blocks are applicable. For each block, the Mean Absolute Error of the two values was calculated and expressed as a percentage. As

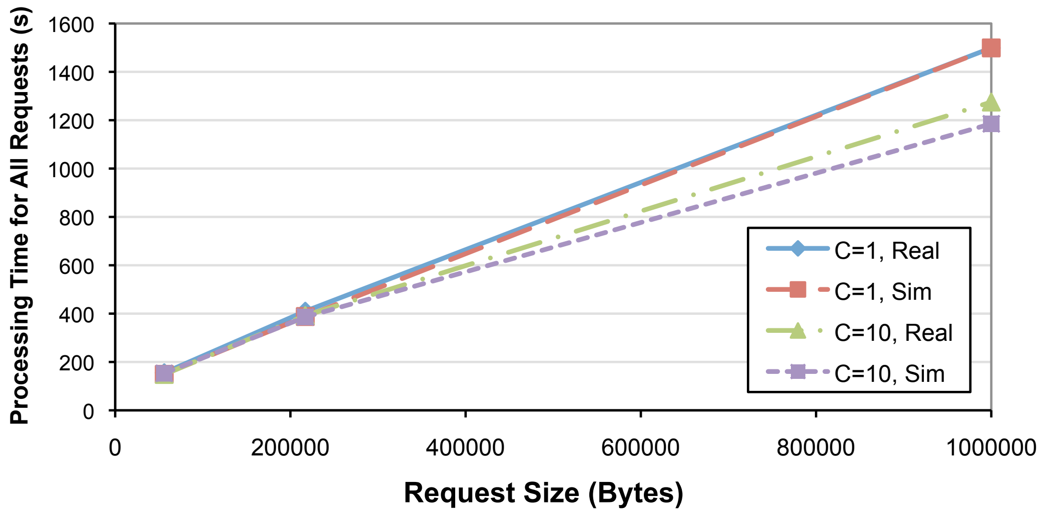


Figure 4.15: The total time required for the real service and the simulated service to respond to all 50 requests for varying input sizes.

shown in Table 4.4, the mean absolute error over all three applicable test blocks is 8.3%.

The performance model was based on real-world service performance for single concurrent requests, which are projected to scale to higher concurrency. Increasing concurrency beyond the approximate capacity of the server results in increased response times for clients, but may reduce overall time spent processing a fixed number of jobs. (Overall test time is of interest to the service owner; individual response time is of interest to the consumer). Comparing total processing time in the real-world service with the same metric in the simulation, for concurrency of 1 and concurrency of 10, the simulation accurately predicts total processing time (Figure 4.15). However, for concurrency of 10, the simulation is too optimistic about the improvement in overall processing time. Improvements were made to the model to decrease this margin of error: the original performance model failed to adequately account for request processing overhead when managing a queue of pending request.

4.4.4 Systematic Simulation

The simulation-generated data used for validation was created by varying three configuration parameters: the size of the requests sent, the concurrency with which the requests were sent, and the concurrency with which the service processes requests. In fact a broader set of data than that used for the validation was generated: 1, 2, 5, and 10 request concurrency; 1 or 2 service concurrency; and request size 250, 56469, 217012, 416818, and 1000732 bytes. There are 40 possible combinations of these values.

To demonstrate the systematic simulation of varying configurations, this data was generated using JIMMIE. The JIML file used is shown in Figure 4.16. A default configuration file is set with all of the configuration parameters required by TAPoR-sim; three of these are modified by JIMMIE: `serviceConcurrency`, `requestConcurrency`, and `size` (of requests generated). The default value for each variable is set in the default configuration file (to 2, 10, and 250, respectively). JIMMIE only modifies the attributes when the default must be modified. So for instance, one command changes `serviceConcurrency` to 1 instead of 2 for half of the simulations (experiments 1 through 20); for the remainder, the default value is used. JIMMIE systematically creates all combinations of the possible values, leaving untouched default parameters. All requests were sent to the List Words operation.

The simulation was run on a Pentium 4 machine with 1.75 GB of RAM, and simulated the TAPoRware service running on a Pentium 4 machine with 2GB of RAM. The results produced by JIMMIE are shown in Table 4.5. Sending more than one request at a time to a single-processor service increases the overall response time as requests contend for resources or spend time waiting in a queue; the total time taken to process the requests doesn't change.

Using this systematic simulation as well as stand-alone simulations, over 1,000,000,000 individual metrics have been collected during the experiments described in this dissertation.

4.4.5 Known Issues and Threats to Validity

The automatic generation (wsdl2sim) is limited to operations with performance of $O(n)$ or better. This could be extended by replacing the linear regression and linear models used; regression analysis and curve-fitting is generally applicable to higher-order polynomials (though limited by the abilities of curve fitting on higher-order polynomials), exponential growth, *etc.*.

The validation was performed on two key metrics (response time and total time) in a straightforward application configuration. The assertion is that the validity of (some) other metrics and the validity of metrics in more complex scenarios follows from this validation. Though these two key metrics are used predominantly throughout the remainder of this work, other metrics are occasionally employed and are not as thoroughly validated. In most cases, a real-world implementation is tested to further validate the simulation-based results.

Only the CPU use is tracked and used to govern response time; if a machine has substantially more CPU available than memory, memory may prove to be the bottleneck. Such scenarios are not anticipated by the simulation.

4.5 Summary

This chapter described the process of using SASF (Chapter 3) to simulate Tivoli Provisioning Manager and the back-end service for the Text Analysis Portal for Research. The first demonstrates use of basic library functionality, the simulation engine, and the metrics engine. The second adds the use of additional libraries, automated simulation generation, systematic simulation using JIMMIE, and the Service Testing Module. One remaining SASF extension, the emulation extension, will be used in later chapters and is not used in the base versions of TPMsim or TAPoRsim.

The use of the framework reduced development time and effort that would have been required otherwise. In the case of TPMsim, the framework is estimated to have offered a 45% reduction in development effort. In the case of TAPoRsim, its behavior was replicated without manually authoring a single line of code. A set of extensions was added, and still only 28% of the TAPoRsim-specific code was hand-coded.

TAPoRsim will be used to test, demonstrate, and validate the methodologies and tools in the following chapters.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<experiment xmlns="http://www.ualberta.ca"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" count="40" saveOutput="true"
  xsi:schemaLocation="jimmie.xsd">

  <configDocList>
    <use URI="config[0].xml" name="config"/>
  </configDocList>

  <do>

  <!-- table name changes in all documents -->
  <document name="config">
    <!-- [0] will be updated to a unique id for this sim run -->
    <changeElement name="variable" withId="tableName">
      <changeAttribute name="value" to="table[0]" action="increment"/>
    </changeElement>
  </document>

  <!-- serviceConcurrency = 1 -->
  <range start="1" end="20">
  <document name="config">
    <changeElement name="variable" withId="serviceConcurrency">
      <changeAttribute name="value" to="1"/>
    </changeElement>
  </document>
  </range>

  <!-- requestConcurrency = 1 -->
  <range start="1" end="5">
  <document name="config">
    <changeElement name="variable" withId="requestConcurrency">
      <changeAttribute name="value" to="1"/>
    </changeElement>
  </document>
  </range>
  <range start="21" end="25">
  <document name="config">
    <changeElement name="variable" withId="requestConcurrency">
      <changeAttribute name="value" to="1"/>
    </changeElement>
  </document>
  </range>

  ... omitted for 2 and 5 for brevity...

  <!-- requestSize = 56469 for 2, 7, 12 ... -->
  <range start="2" step="5" end="2">
  <document name="config">
    <changeElement name="variable" withId="requestSize">
      <changeAttribute name="value" to="56469"/>
    </changeElement>
  </document>
  </range>

  ... omitted for 217012, 416818, 1000732 for brevity ...

  </do>
</experiment>
```

Figure 4.16: The JIML file describing what parameters to modify for a series of experiments.

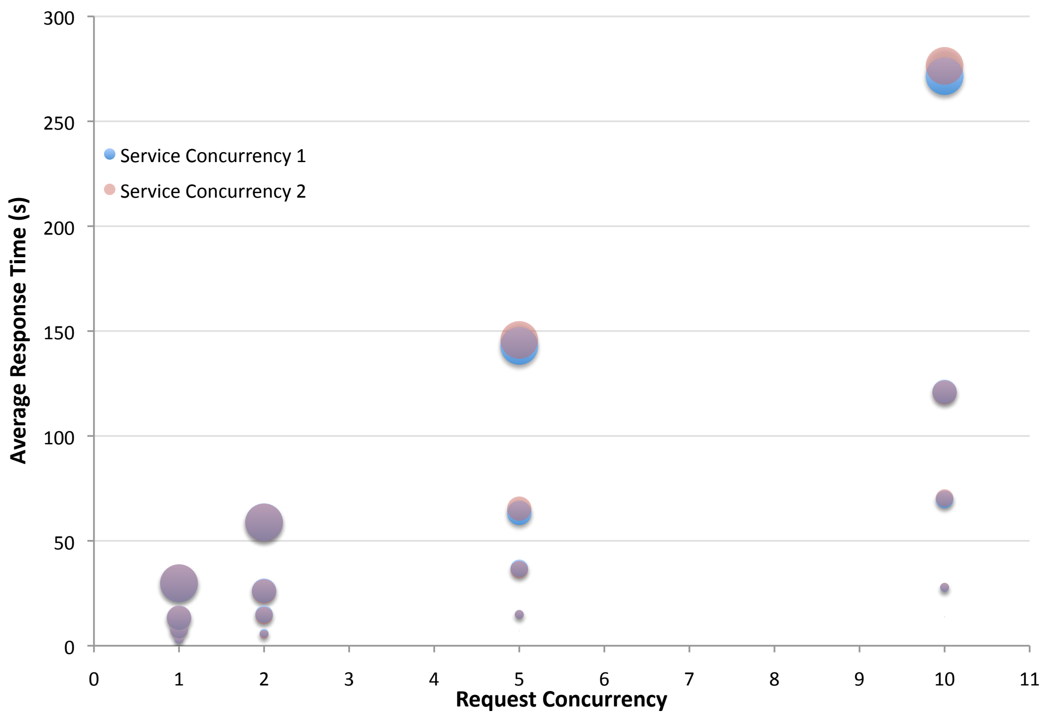


Figure 4.17: The data produced by JIMMIE; the relative size of the request is shown by the size of the point. The overlap between service concurrency of 1 and 2 shows that this factor doesn't affect the results. The average response time increases as the request load on the service increases.

Service Conc.	**Conc. Requests**	**Size**	**Total Time**	**Resp. Time**
1	1	250	74	1.47 ± 0.39
2	1	250	80.5	1.58 ± 0.5
2	2	250	46	1.79 ± 0.78
1	2	250	75	2.96 ± 0.76
1	5	250	71.5	6.67 ± 1.46
2	5	250	80	7.69 ± 1.43
1	10	250	75.5	13.62 ± 3.43
2	10	250	76	13.94 ± 3.58
1	1	56469	156.5	3.12 ± 0.51
2	1	56469	157.5	3.14 ± 0.55
2	2	56469	136	5.37 ± 0.97
1	2	56469	151.5	5.98 ± 0.69
2	5	56469	155	14.89 ± 2.39
1	5	56469	156.5	14.97 ± 2.35
1	10	56469	151.5	27.82 ± 6.63
2	10	56469	152.5	27.89 ± 6.2
2	1	217012	379	7.57 ± 0.57
1	1	217012	387.5	7.74 ± 0.56
2	2	217012	364	14.42 ± 1.26
1	2	217012	378.5	14.96 ± 1.27
2	5	217012	375	36.27 ± 6.52
1	5	217012	385.5	36.9 ± 5.75
1	10	217012	384	69.81 ± 16.9
2	10	217012	384	70.42 ± 15.66
2	1	416818	665	13.28 ± 0.52
1	1	416818	668	13.35 ± 0.5
2	2	416818	649	25.93 ± 1.13
1	2	416818	664	26.27 ± 1.85
1	5	416818	660.5	63.31 ± 9.84
2	5	416818	675	65.35 ± 12.93
2	10	416818	658	120.74 ± 27.18
1	10	416818	665.5	121.04 ± 29.48
1	1	1000732	1491.5	29.81 ± 0.48
2	1	1000732	1492	29.83 ± 0.53
2	2	1000732	1471	58.76 ± 0.97
1	2	1000732	1487	58.88 ± 4.12
1	5	1000732	1488	142.92 ± 22.34
2	5	1000732	1505	145.77 ± 32.28
1	10	1000732	1492	271.54 ± 65.77
2	10	1000732	1502.5	276.42 ± 60.93

Table 4.5: List Words operation response times and total processing time for configurations that vary in request concurrency, service processing concurrency, and size.

Chapter 5

Informing the SLA Lifecycle using Simulation

Chapter Contents

An SLA can be viewed as a three-way relationship among the producer, the consumer[1], and the service. The typical lifecycle of an SLA involves creation (negotiation), implementation and assessment, and eventually termination (Figure 5.1). This is the prevailing view on SLA lifecycles (*e.g.*, [23, 24]). Others recognize the need for the SLA to evolve and be re-negotiated over its lifetime (*e.g.*, [6, 25]). The latter view is adopted here, and extended: the SLA should evolve as business needs change, based on periodic review, and as the service implementation itself evolves (for example, to add new features).

This modified view of the SLA lifecycle is shown in Figure 5.2 by changing the linear process into a more realistic cycle. An initial SLA is negotiated to document the desired qualities of the service delivery. The service is configured and deployed to meet the SLA. Service interactions are monitored and evaluated to ensure that the terms of the SLA are met. The important part is the SLA is not static - it is periodically re-negotiated, or at least re-examined, to ensure it is still practical. Good points for re-examination are when the software is updated and when business needs change. The result of the evaluation or the re-negotiation may be to terminate the SLA.

Each phase of the lifecycle and a simulation-driven method addressing that phase is described in the following sections. §5.1 describes the challenges that face a consumer negotiating with a service provider who has more information and understanding about the service. It presents simulation-driven approaches to assisting in making choices and trade-offs explicit and well understood to strike a balance that provides actual and lasting quality to the consumer. §5.2 describes the process of translating an SLA into a configuration, and introduces a simulation-driven question

[1]The consumer is not an individual; it is any entity that consumes a service. A service is not technically "consumed", but the term "user" is even more overloaded.

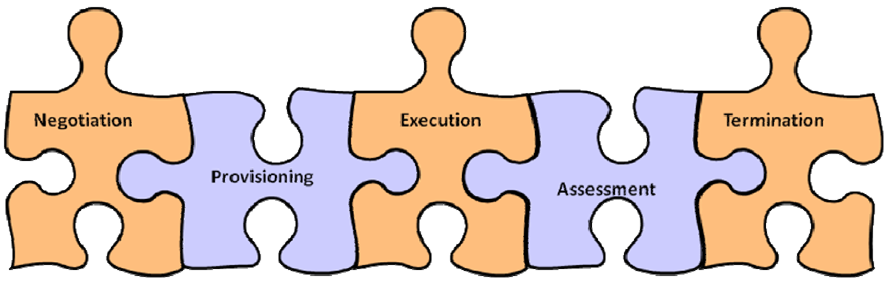


Figure 5.1: The traditional view of an SLA lifecycle (from [24]): a linear run from negotiation to termination.

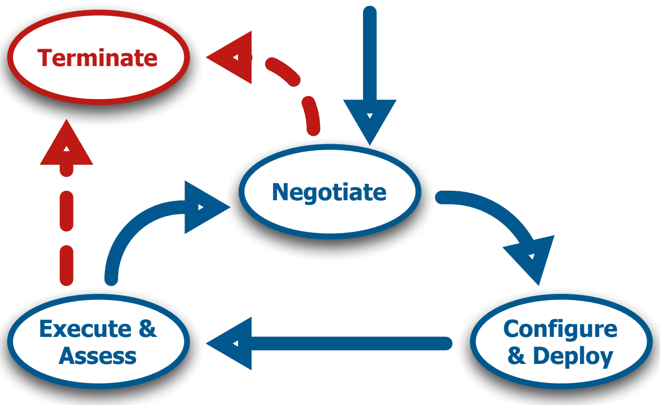


Figure 5.2: The lifecycle of an SLA: negotiated, implemented, assessed, then re-negotiated or terminated.

answering tool that a service provider can use to better understand the software application and create configurations more probable to meet a defined SLA. §5.3 briefly introduces service execution and assessment before describing a SASF-supported approach to testing and monitoring a deployed service system. Each simulation-driven approach is demonstrated using TAPoRsim.

5.1 Negotiation

An ideal SLA negotiation would involve two parties with equal knowledge and balanced power over the transaction (symmetric information and no duress), both with understanding of the technology and systems of the provider. The standards agreed upon would be based on a complete requirements elicitation from all of the stakeholders, trade-offs would be balanced optimally, and the proposed service levels would be tested vigorously before entering production. Penalties would provide exactly the right incentive to ensure the targets are met if at all practical. The terms of the SLA would revisited regularly, and when change events occurred in either the provider or consumer. The reporting metrics would be well-understood by all parties, and unvarnished understandable detailed data would be available to both the provider and consumer.

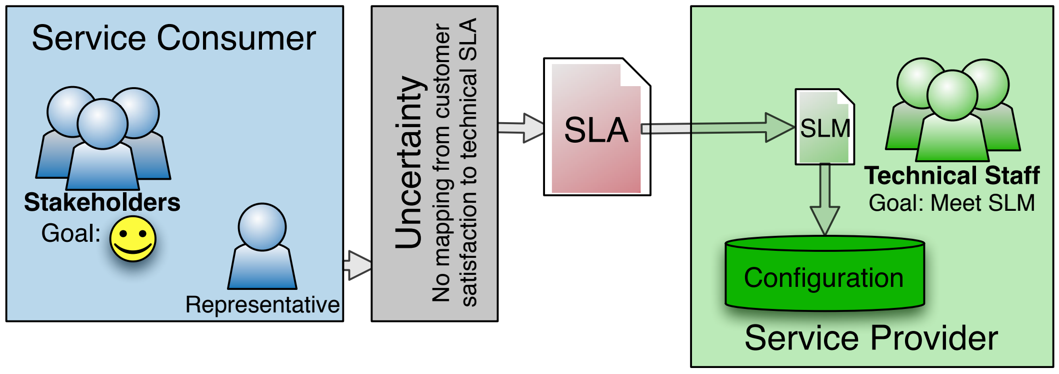


Figure 5.3: Diagram of the current state of SLAs: providers implement SLAs by converting them into SLMs and ultimately system configurations. Consumers interact with an SLA directly, but there is no information on how it maps to customer satisfaction.

This idyllic situation is not found in reality, as discussed in §2.1.1. The parties typically are not working with symmetric information, are not equal, and often the consumer has no complete understanding of the service system. A knowledge gap may be unavoidable, but the distance of this gap can be controlled by careful execution of the SLA lifecycle[2].

The assertion here is the root of this knowledge gap is a **fundamental disconnect** between the level at which non-technical service consumers evaluate services and the level at which SLAs are specified.

Figure 5.3 shows a broad view of a common SLA situation. The service provider uses the SLA to create *Service Level Management* (SLM) policies: internal objectives specifying what has to happen in order for the SLA to be met. Beyond simple service levels, SLM policies (and the entire SLM infrastructure) monitor and evaluate the provider's performance with regard to the objectives. Autonomic management, if applicable, is part of the SLM infrastructure. Based on the SLM (or on many SLA+SLM combinations), a system configuration is created (and updated over time) intended to meet the service levels. This translation can be semi-automated but mostly relies on technical staff who map SLA terms all the way to configuration-level decisions. One consequence of this process is the terms in an SLA tend to be dictated by the available configuration options and thus are inherently technical documents.

On the other side of the SLA, the service consumer has a negotiating team that attempts to identify the requirements of the stakeholders: end users, employees, and so forth. From these requirements, they specify required service levels. However, the mapping from what their stakeholders need to the measurable, enforceable objectives of an SLA is not clear. It is analogous to a requirements elicitation for a software project where the user is asked to translate "nice user interface" to what colour the shadow of the submit button should be. Without detailed knowledge and understanding of the provider's service system, the consumer is at a disadvantage:

[2]The largest gap, and a prevalent case, is when the SLA is unilaterally dictated by a provider and there is no negotiation. In this case, this section applies to the provider as they author and re-examine this mandated SLA.

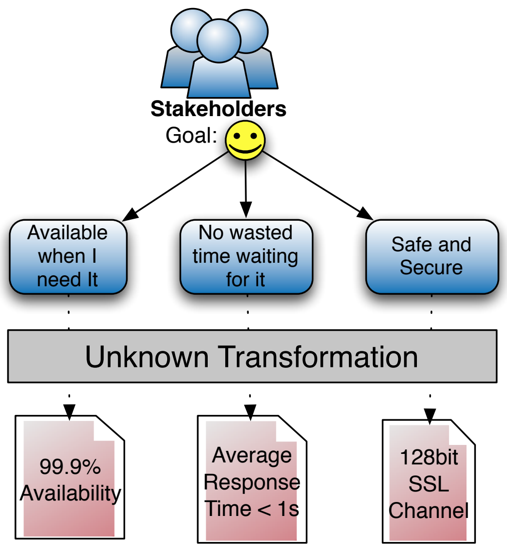


Figure 5.4: Expanding the top-level goal of "satisfaction" to one level is feasible; the direct link between SLA terms and satisfaction is less clear.

information is asymmetrical.

Figure 5.4 offers a more concrete example based loosely on a real-world SLA. "Customer satisfaction" is broken down into three high-level goals: service is up when I need it, service is as fast as I need it to be, the service is safe and secure. A fictional (and murky) translation to measurable metrics is also shown. "Available when I need it" is translated to three-nines (99.9%) availability. This level is likely over-provisioned and the consumer will end up paying for availability they don't need. Conversely, if the .1% of allowable downtime each year occurs as 8 consecutive hours at a critical time, the customer is still dissatisfied. Regarding the average response time of 1 second, "1" is likely a magic number with no basis in the customer's needs. The SSL channel will secure the transport layer, but provides no guarantee of any other security.

In short, SLAs used for automated negotiation and enforcement rely on measurable data. This data exists at a relatively low level: at best, it consists of metrics calculated from a series of lower-level observations over a service period. Notions like customer satisfaction, perceived quality, and perceived value are higher-order constructs. The question is how to map the information available to the higher order, and vice versa.

From the discussion on perceived value §2.4, we know it is evaluated in the context of perceived quality versus perceived sacrifice. Sacrifice is everything exchanged to ensure the level of perceived quality, which is more than simply the cost of that level of quality. Perceived quality is evaluated based on the information available, which can include the perceived sacrifice. Extrinsic and intrinsic attributes[3], even at a low level, can be used as cues to infer perceived quality. Extrinsic cues - especially price - are more important when intrinsic cues are not available or are difficult to

[3]Recall that intrinsic attributes are measurable properties of the service - in this case, attributes like response time, latency, throughput. Extrinsic attributes are product-related like branding and pricing.

evaluate; they are used as "value signals". Intrinsic cues are more important when they are search attributes[4] and tend to be accurate predictors of value. Intrinsic attributes that are more abstract are more broadly application. §5.1.1 describes simulation support for negotiation that is an early attempt at addressing these issues.

Service providers are motivated to provide value to service consumers at evaluation time: in an ongoing service-relationship, customer satisfaction is necessary for the continuation of the relationship[5]. In service encounters, a trusted provider has the opportunity to influence how the consumer perceives quality (Nilsson, [53]). Though perceived value and perceived quality can be influenced at negotiation time by exploiting the understanding of value as described above, this influence will be diminished at evaluation time. As discussed in §2.4, assessment of perceived value is situational and can change between purchase time and deployment, as more information becomes available. An SLA negotiated with artificially skewed perceptions of quality and cost will likely prove to be unsatisfactory once implemented, and the trust relationship may be impacted. The service provider is incentivized to assist the consumer in reaching an SLA that satisfies them.

5.1.1 Simulation-supported SLA Negotiation

Creating an SLA involves reconciling a set of goals, which might be well-specified or only intuitively understood, and might be complementary or might conflict. This trade-off analysis methodology supports decisions involving the balancing of conflicting goals, whereby one might relax goals in one area in order to achieve other goals, perceived as more important. Without loss of generality, this discussion is focused on cost and performance metrics, as they are common and more readily quantifiable. One of the shortcomings of SLAs identified by Blomberg [6] was that the parties involved in creating SLAs lacked the information and intuition needed to make appropriate decisions about quality of service levels; this approach attempts to provide both information and intuition when planning.

Experts can ask and answer specific questions about a software service system. When the goal is a general exploration of the trade-offs, a large number of possible configurations can be simulated to generate a knowledge base that powers a decision support tool. Using this knowledge base, various conflicting or correlated metrics are visualized to understand the impact that one goal will have on another. The system manager can select potentially viable configurations using a visual tool. Based on their selections, a "fitness score" is calculated for each configuration.

Visual Selection Tool. The configuration exploration simulations are visualized as a series of weighted scatter plots in a two-dimensional space, where the two axes correspond to two metrics, offering a head-to-head comparison. For example, Figure 5.5 shows a sample plot of correlated values, the average response time (x) versus the total time required to complete the processing. The configurations are clustered based on their values for the two metrics. Each point on the plot corresponds to a cluster, and is sized based on the number of configurations that belong in this cluster. The user is shown a series of these plots and is asked

[4]Recall that search attributes can be assessed before experience with the product or service; experience attributes are assessed only during consumption.

[5]Excluding, of course, monopolies and oligopolies

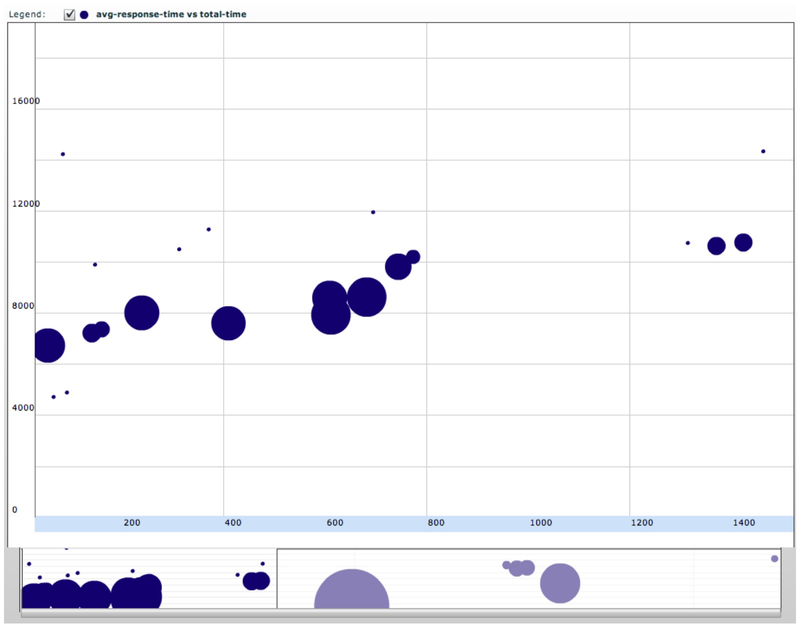


Figure 5.5: A visualization of configurations' Average Response Time versus their overall time to process.

to draw a rectangle around the configurations that meet the requirements of the system under configuration. Each successive plot would have clusters colored different shades based on how many configurations in that cluster have been previously selected. Once the user has made all of their selections, he is presented with a list of candidate configurations based on how often those configurations were selected on individual plots. To select a final configuration, the user can watch a recording of the simulation for each candidate to observe metrics in action, choose based on a simple metric (such as the cheapest, the most energy efficient...), or trust a more complex fitness score (as explained in the next section). Hovering over a point produces a "tool-tip" listing the configurations involved, important metrics, and key configuration parameters.

Fitness Score. Based on the user's selected simulations, the visual tool computes a fitness score. Each configuration, in each head-to-head comparison, is given a comparison score h_{ij} based on its values for the compared metrics i and j:

$$h_{ij} = weight_{ij}[i] * value_{ij}[i] + weight_{ij}[j] * value_{ij}[j]$$

The *weight* is determined by two factors. First, how many configurations were selected out of the total set: if the user accepted a metric's value for most configurations, that metric is relatively unimportant in distinguishing between good and bad configurations. Second, how much of the total range of that metric is selected. If the user was more selective, the metric is more important. For example, consider a comparison of average response time (x) and cost (y) that plots 100 configurations, where response time ranges from 1 to 101 seconds and cost from 200 to 500 dollars. The user draws a selection box from (1,200) to (10,400); there are 34 configurations with response time between 1 and 10, and 56 configurations with cost between 200 and 400. The first factor is $1 - \frac{34}{100} = .66$ for response time and $1 - \frac{56}{100} = .44$ for cost. The second factor for response time is $1 - \frac{(10-1)}{101-1} = .91$ and for cost is $1 - \frac{400-200}{500-200} = .33$. Thus $weight_{ij}[r.t.] = 1.57$) and $weight_{ij}[cost] = .77$. Note that response time and cost may have different weights when compared to other metrics.

The $value_{ij}[k]$ is normalized based on the relative position of the configuration's value for metric k: at the "good" end of the selection is 2, at the "bad" end is 1, everything in between on a linear scale. Values outside the selection have value 0. The intuition is that a configuration deserves value for even being in the selection, but even then should be valued differently depending on its relative "goodness". Returning to the example, a configuration with response time of 1 (best in the selection) and cost of 500 (outside the selection) would have values 2 and 0, respectively. The fitness score for that configuration for this pair of metrics would be $h_{ij} = 1.57 \times 2 + .77 \times 0 = 3.14$ out of a possible 8 points.

This fitness score is open to revision; the key contribution is producing configurations and the metrics associated with them. From there, any selection algorithm can be implemented. Though little work has been done on negotiating an SLA such that the service delivery meets consumer expectations, more research exists on selecting services from a pool of functionally equivalent services based on their non-functional attributes (like quality of service). Similar methods are appropriate for choosing a configuration from among a set of functionally equivalent configurations (for instance, mid-level splitting [72] or the web services relevance function [1]). Some even explicitly use hypothetical services to establish weights (*e.g.*, [72]); this simulation-driven methodology allows one to instead use the same service with different hypothetical configurations to establish weights.

Demonstration. JIMMIE was used to generate all possible distributions of 6 services on 2 and 3 servers, where each service corresponded to a single operation. Traffic was generated directly from the logs of the existing service. This provides data representing a performance versus cost trade-off for approximately 800 candidate configurations. Using the visual selection tool, ranges were selected in head-to-head comparisons based on the perceived desires of the user community (fast response, consistently, at low cost). Comparisons deemed unimportant were ignored (value = 0). There were 6 metrics, and $\binom{6}{2} = 15$ potential head-to-head comparisons; only 6 comparisons were considered meaningful. Using the visual selection tool, 153 candidate configurations were identified, each having been selected 4 times.

This reduced the candidate configurations by 81%. There remained variation within the remaining configurations; the fitness score allows us to rank the configurations using preferences expressed in the visual tool. The fitness score was computed for each of the six comparisons. Of the 153 candidate configurations identified by the visual tool, 9 were missing from the top 153 of the new ranked list, in all cases due to being very close to the boundaries of the selection box (*i.e.* marginal candidates). This identified 49 top configurations (all within .1 of 41.2, with the remaining configurations under 40, on a theoretical maximum score of 48). This is a 94% reduction in the number of candidate configurations. $\square$

5.2 Configuration and Deployment

After an SLA is created, the service provider deploys, provisions, and makes available the services agreed upon. As described in §5.1, the service provider maps the terms of the SLA to service level management policies that are more technically specific. Based on these, the service is deployed and made available.

The difficulty lies in creating a configuration that is capable of meeting the negotiated SLA. The service provider is expected to have some expertise in this area, but may still wish to test candidate configurations, or to ask specific configuration-related questions. The following section describes using the simulation to answer questions.

5.2.1 Simulation-supported Configuration and Deployment

After creating an SLA using the methods and tools described in §5.1.1, the same tool set can be used to understand how an application reacts to changing configuration options (*e.g.*, by testing a variety of configurations and measuring their performance levels, or by watching traces of application behavior in simulation). This section addresses other ways to support configuration and deployment using simulation: question answering, for administrators with a solid understand of the application but in need of specific answers to specific questions. Answerable questions can be asked in the form *How does changing configuration variables* $x_1, x_2, \cdots, x_i$ *affect metrics* $m_1, m_2, \cdots, m_3$*?*

The question answering approach here systematically modifies a simulated application to test a series of configuration options, and the results are compared to answer the question of interest. The methodology is best described using examples. Consider two questions based on real-world TAPoRware issues, namely the impact of concurrent requests and what distribution of services on what number of servers is best suited to meet expected load.

First, *How does sending multiple concurrent requests to the List Words operation change the overall response time and throughput? What if more processors are added?* The question asks about outputs (response time and throughput) based on environments where two configuration variables are changed: request arrival concurrency, and number of processors. The performance profiling has established that one process processing one request can use one entire processor, so this serves as a demonstration of how question answering will proceed and does not offer any particular insight.

To answer this question, JIMMIE was used to generate configuration files specifying 50 requests of the List Words type, size 1000KB, but with requests being sent with various levels of concurrency: from 1 up to 6 requests sent simultaneously. The simulated application was configured with varying numbers of processors: 1, 2, or 4. Recall that TAPoRware (2.0) launches one process per processor and distributes requests to each process based on availability and queue length. A separate simulation was run for each of the 18 configurations; for each, the average response time for the 50 requests and the total throughput was recorded.

The results are shown in Figure 5.6. For the single processor configurations, as the number of concurrent requests increases[6], the average response time also

[6]Note that only concurrency up to 4 is shown; the line continues straight - 72.2 and 87.0 seconds

increases: a single request is using all of the resources available, so once more arrive they compete for the same resources. However, the throughput does not change. For two processors, moving from one concurrently arriving request to two doubles throughput without changing the average response time: there is spare capacity to handle the increases load. Greater concurrency has an insignificant negative impact on throughput (perhaps due to resources dedicated to queue management), and increases the average response time (though at half the rate of increase when compared to a single processor system). In general, k processors can handle from 1 to k concurrent requests without increasing response time. Maximum throughput for a server with k processors is achieved when the concurrency is k. This is the expected answer.

Validation: To verify the accuracy of the answers, a smaller set of tests was performed in the real world. The TAPoRware service was deployed to a high-CPU Medium Amazon Web Services instance (5 ECUs, 2 cores, 1.7 GB memory, moderate I/O performance). A 1-processor system was emulated by configuring TAPoRware to process only one request at a time, using only one CPU at a time. A Micro AWS instance (bursting to 2 ECUs, 613 MB memory, low I/O performance) was set up to imitate the clients sending the concurrent requests (using the Apache Benchmark tool). The set of requests was identical to those in simulation. Only 1 or 2 processors are tested, with request concurrency of up to 3.

The results are shown in Figure 5.7. Note that the results cannot be compared directly to the real world, as a) the hardware was similar but not identical, and b) the variability of cloud resources make direct comparisons to non-virtualized hardware difficult. However, the curves are correlated: the same factors of improvement and worsening are seen in the real world as were predicted by simulation. □

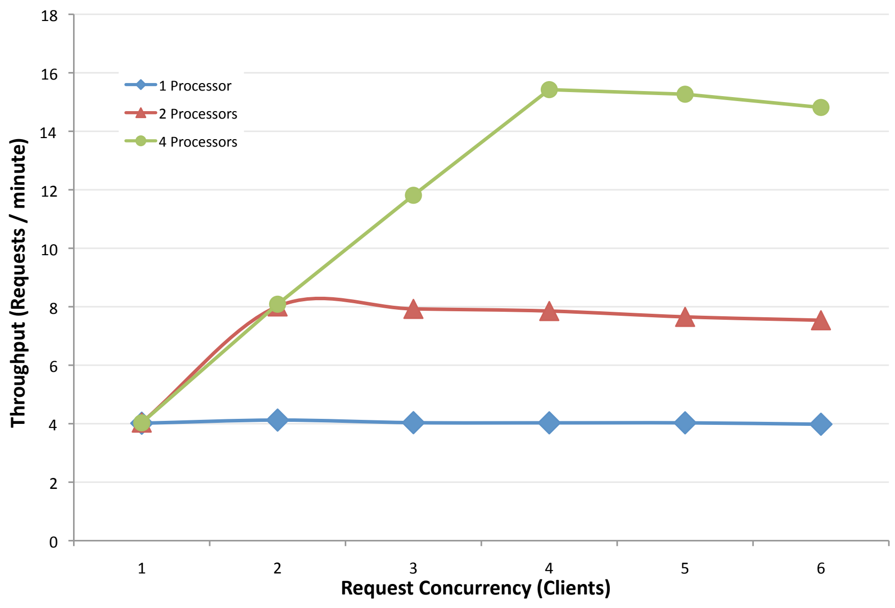
The second question considers the *preferable distribution of 6 operations over 2 or 3 servers in order to achieve the best throughput to cost ratio.* The goal is to partition the 6 operations used most often into separate services, distributed over several servers to balance the expected load. A normal approach might be to separate the more resource intensive operations, or to group them. In this example, the variable element is the number of servers, and which operations are assigned to which server. The measured outcome is throughput, in this case expressed as the total time taken to process a set of requests.

JIMMIE was used to generate all possible distributions of 6 services on 2 and 3 servers, where each service corresponded to a single operation[7]. Traffic was generated directly from the logs of the existing service. The simulation was run faster than real time, so each configuration was simulated in between 2 and 4 minutes.

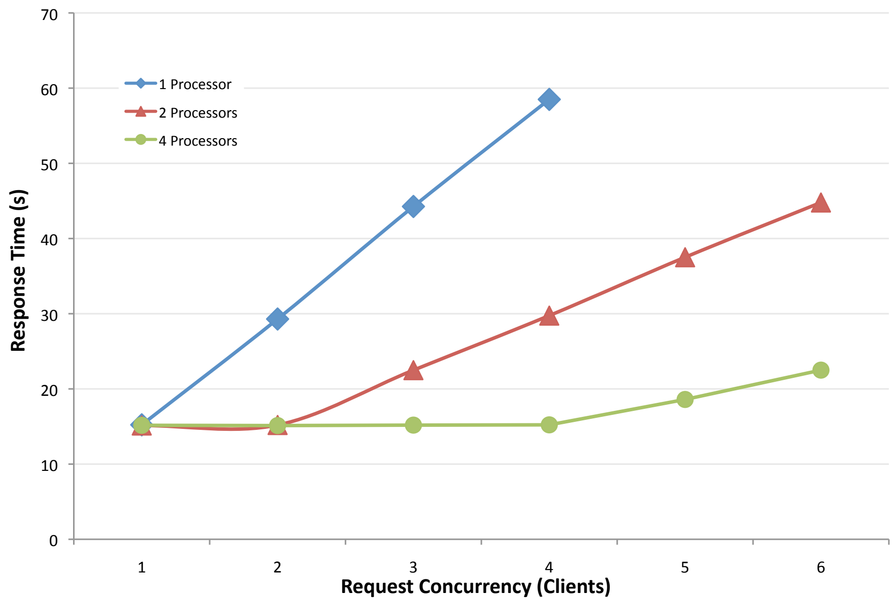
The best total times for 1, 2, and 3 servers are shown in Figure 5.8. The fastest two-server configurations took 140 minutes to process all of the incoming requests, *i.e.*, twice the cost, half the time. The fastest three-server configurations completed shortly after the traffic generator stopped generating requests (120 minutes), 57%

for concurrency of 5 and 6, respectively.

[7]The idea of using different distributions instead of a complete mirror of the service allows faster operations to be assigned to one server and slower operations to another; a fast operation stuck behind a slow operation in a queue will experience a more dramatic increase in its response time.



(a) Throughput (higher is better).

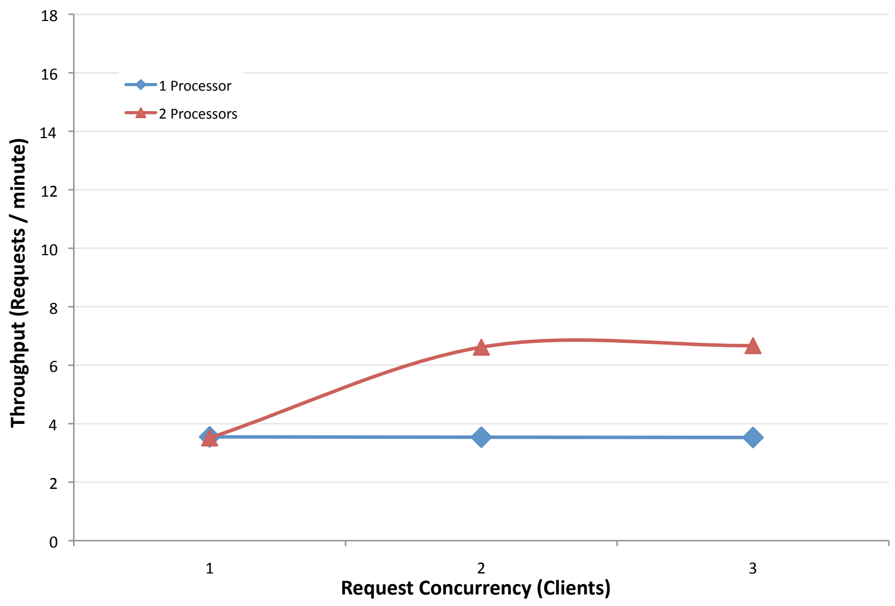


(b) Average response time (smaller is better).

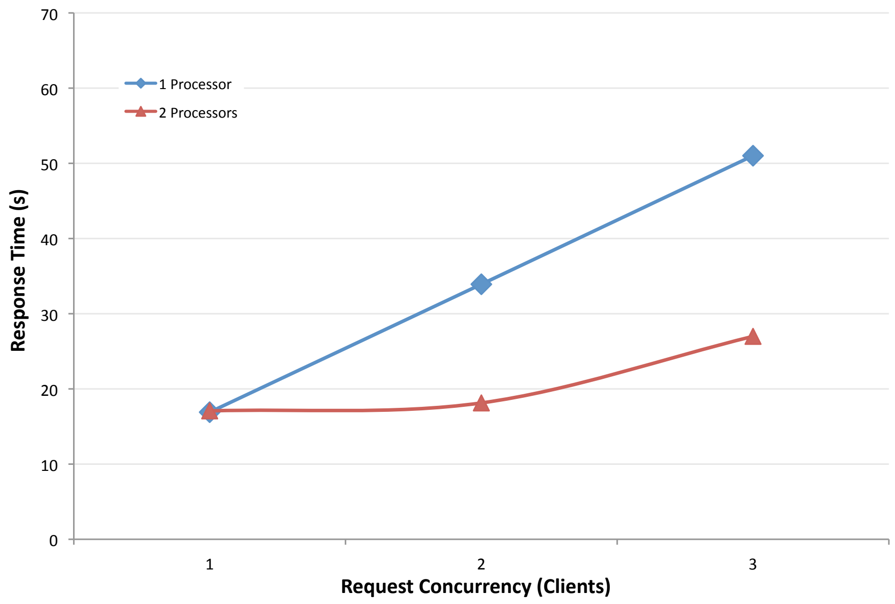
Figure 5.6: Simulation-generated answers for how the List Words operation handles varying levels of concurrency for varying numbers of processors/cores.

less time. This represents 3 times the cost for a 2.3 performance improvement factor: a more expensive improvement for the expected load. Service distribution is important: 75% of the three-server configurations performed worse than the best two-server configuration.

Validation: Before running the simulations, a configuration was created manually using the same information available to the simulation, as well as personal experi-



(a) Throughput (higher is better).



(b) Average response time (smaller is better).

Figure 5.7: Real-world answers for how the List Words operation handles varying levels of concurrency for varying numbers of processors/cores.

ence with the performance behavior of the application. This configuration made use of detailed information: expected arrival rate and frequency of each service, the total size of all requests to the given service, and so forth (Table 5.1). Based on the expected number and size of requests, as well as the total predicted processing time, the three most resource intensive operations were allocated to three different servers. The fourth busiest service was paired with the third busiest service, and the remaining two were put with second service. This manually-designed configuration took

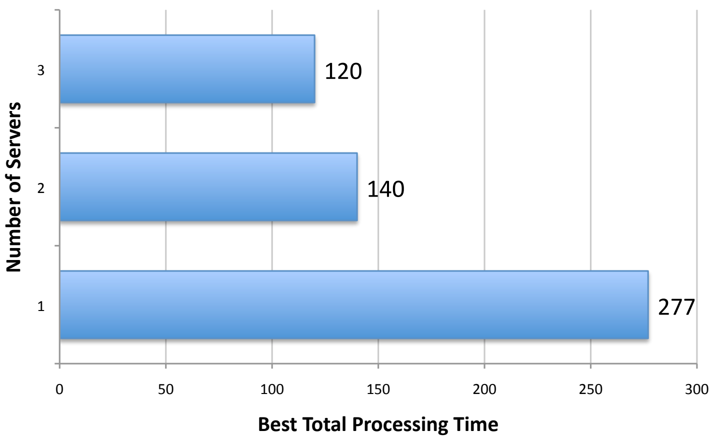


Figure 5.8: Distribution question answering results.

Operation	# Requests	Total size (MB)	Expected Total Processing Time (h)	Proportional Load
WordCloud	23421	1261	17.3	51.14%
ListWords	11283	1178	9.6	24.64%
Date Finder	265	61	0.0286	0.58%
Acronym Finder	95	7	0.0022	0.21%
Concordance	8376	1980	0.5767	18.29%
Collocation	2354	1774	10.3	5.14%

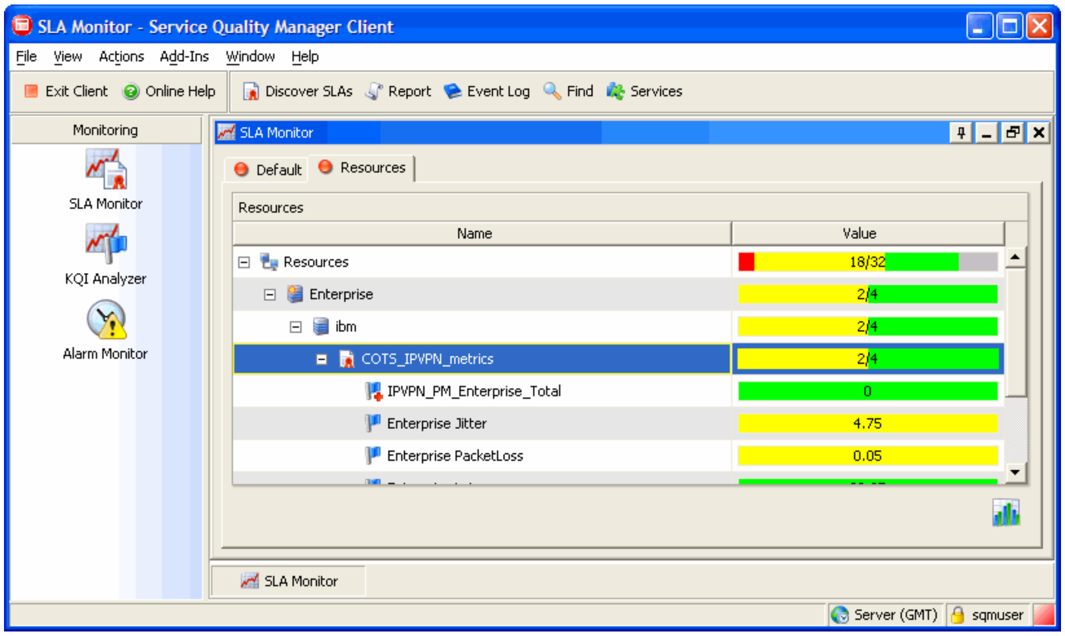
Table 5.1: Some of the meta data about the expected requests that informed the manual configuration.

137 minutes to run, 14% longer than the best three-server configurations and 2% faster than two servers. The question answering methodology was able to produce better results. This confirmed the intuition that configurations produced by experts based on their understanding of the software, expected load, and environment may not necessarily meet the properties desired of the system. □

5.3 Execution and Assessment

Once a configuration is chosen, the service is configured, deployed, and accessed. The mechanics of deployment and service integration are outside the scope of this work.

Once the service system is deployed in production, ongoing monitoring is required to ensure the service complies with SLAs. Service assessment involves monitoring the running service and the consumer experience to ensure compliance with the SLA and internal management policies. A balance must be struck so that reasonable monitoring is available, but does not impair the performance of the system. Performance data is sometimes made available to the customer, though typically abstracted. Ongoing testing can be used to test the system performance using requests that do not have SLA penalties for non-compliance associated with them (for



Name	Value
Resources	18/32
Enterprise	2/4
ibm	2/4
COTS_IPVPN_metrics	2/4
IPVPN_PM_Enterprise_Total	0
Enterprise Jitter	4.75
Enterprise PacketLoss	0.05

Figure 5.9: IBM Netcool Service Quality Manager (from [27]Distribution question answering results.

instance, during off-peak hours or scheduled maintenance windows). A dashboard-type view can be presented (for example, IBM Netcool Service Quality Manager; sample screenshot in Figure 5.9).

A complete discussion of monitoring is out of scope for this work; see *e.g.* [15, 62, 31, 27].

The following section describes a SASF-driven approach to ongoing testing and monitoring of a deployed service system, including a demonstration of integrating real-world and simulated components. Additional contributions to this portion of the SLA lifecycle are described in Chapter 6, which describes automatically adjusting a configuration after deployment.

5.3.1 SASF-supported Execution and Assessment

At this point, the service is deployed and assessed in reality; the utility of simulated results is diminished. However, one area where SASF can be employed to assist in executing and assessing services is post-deployment testing and monitoring, using the Emulation extension. Once a service is configured and deployed, it may be re-configured in response to a changing environment. Pure simulation is less useful for testing when a real-world system is already deployed. The methodology proposed here involves integrating simulation tools with real-world services and metrics. The advantage is the tools are familiar from the negotiating, training, configuring, and deploying stages. This is not a substitute for a comprehensive monitoring framework; rather, it is a straightforward way to test and monitor the non-functional behavior of a deployed software system[8].

[8]Real-world testing is also performed during the configuration and deployment stages, and the methods described here are useful for that as well.

The methodology starts with the Emulation extension (§3.5). The extension allows the Service Testing Modules (§3.6), the Metrics reporting and visualizing framework, and any features implemented in simulation to be combined with real-world services. The result is a hybrid emulation model where requests are generated in simulation and sent in the real-world to a real-world service, and responses are received back in simulation and reported and visualized using tools from SASF.

This extension is then adapted to also receive and visualize metrics directly from the hardware supporting the service: CPU use, memory use, disk IO, *etc.*, as well as the service itself. These metrics are combined in a single dashboard to provide an overall view of how the deployed service is responding to incoming requests. Using the STM, a service provider can vary the size, distribution, frequency, and type of requests arriving. Visual correlations can be made to diagnose system behavior (*e.g.* between high response time and high disk IO) using the unified view of both low-level hardware performance metrics and higher-level metrics like response time. Using the SASF metrics system enables visualization as well as recording directly to a database for replay later at any speed: the testing can be performed automatically at the slow pace of the real services, than observed at high speed.

Demonstration: To demonstrate the potential of this approach, the TAPoR simulation was unified with a deployed TAPoR service. The real deployment was to an Amazon Web Services instance (High-CPU Medium / `c1.medium` type, 1.7 GB of memory, 5 EC2 Compute Units on 2 virtual cores, 32 bit, Moderate I/O Performance). It was configured to run one TAPoR service per core, load balanced using Apache 2. The simulation component used a pre-recorded trace of requests to send a series of service requests to the real service, emulating tens of different clients. The simulation component was run on a Core 2 Duo 2.66 GHz, 4GB, 64-bit machine running Mac OS X on physical hardware.

The AWS feature *CloudWatch* was enabled for the deployed instance, with granularity of 1 minute. A translation component was written to query the CloudWatch API, obtain metrics, and pass them to the SASF metrics dashboard. There is a slight delay (60-120 seconds) after the measurements are taken before CloudWatch statistics become available to the API; the results shown in the dashboard are time-delayed. The API can report aggregate metrics for larger deployments, or can produce per-instance metrics. Though AWS was used in this demonstration, similar translators can be created to obtain metrics from virtualization frameworks, physical systems, or monitoring frameworks. Using Java and the Java library for AWS, the translator is under 100 lines of code.

A sampling of the graphs shown in the Metrics dashboard is provided in Figure 5.10, illustrating the blend of real and simulated data. The queued requests are created and managed by the simulation; the CPU utilization is a real-world metric that is recorded and reported entirely in the real world and brought in through the translator; average response time is a property of the real world system that is calculated by the STM. □

5.4 Summary

Service providers and service consumer mediate their relationship using service level agreements. This chapter introduced a different perspective on SLAs - that they are lasting, cyclic documents that last as long as the interaction between a service provider and service consumers. They must be periodically reviewed and renegotiated if necessary to match the changing business requirements or technical properties. Like existing SLA lifecycles, a configuration is created from the SLA and deployed; the running service is tested and monitored to ensure compliance. It will eventually be necessary to terminate the SLA.

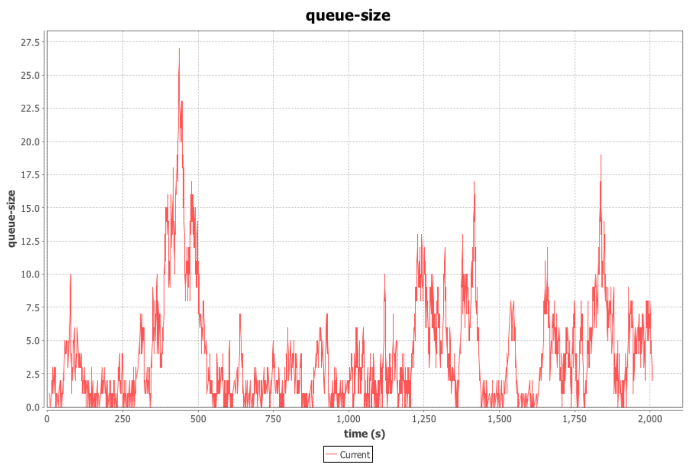
This chapter described simulation-driven contributions that assist each phase of the lifecycle. Each was implemented in simulation and validated to produce results that met the requirements; two were also tested on real-world deployments to verify their viability.

- A methodology, tool, and numeric representation of trade-off balance cost-benefit analysis to assist in understanding and choosing the balance between dependent metrics. It was demonstrated that hundreds of candidate configurations could be tested, visualized, and presented to a user who chose their preferred trade-offs. The tool narrowed down the candidate configurations by 94%.

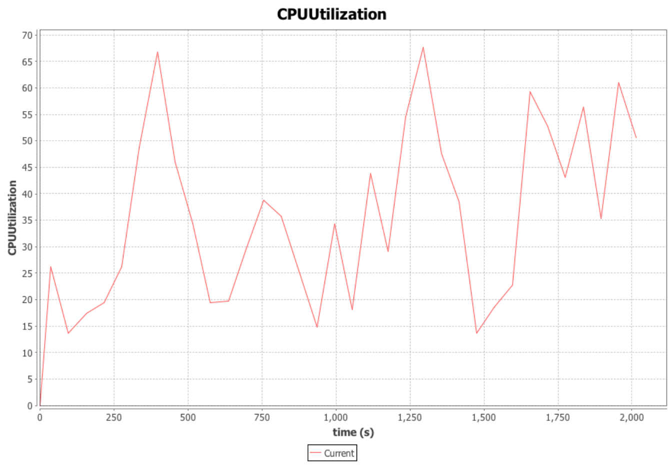
- A question-answering methodology and tool that assists a service provider in producing a configuration that meets the SLA (through his or her interaction with the tool), and in better understanding a service system generally. The approach was demonstrated on two questions, on both the simulation and on a real-world deployment. The question was answered with more detail than would be possible without a simulation; for the tests feasible in the real world environment, it was shown that the answers were the same as the simulation-produced answers.

- A monitoring and assessment tool that integrates simulated components with real-world services to use the flexible and powerful service testing modules and the metrics/dashboard framework of SASF to produce, visualize, and store real-world data. The tool was demonstrated exercising a real-world deployment of TAPoR and visualizing simulation-generated metrics alongside real-world metrics.

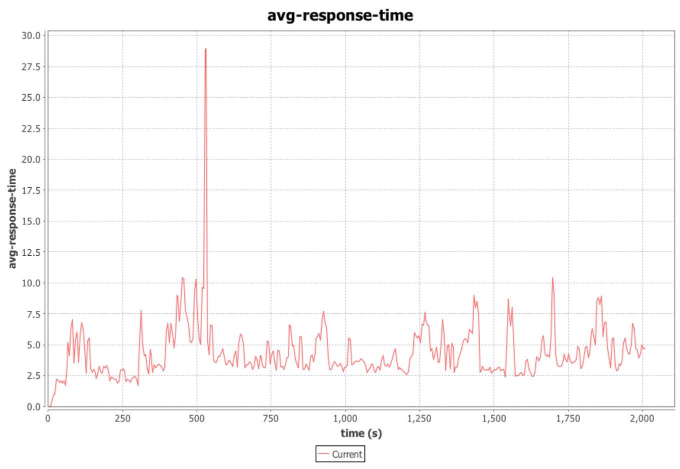
The primary threat to validity is these simulation-driven approaches have not been validated in user studies that demonstrate that SLAs, configurations, and assessments are actually producing results that satisfy users. Though the methodologies and tools meet the derived requirements (more co-created value, more information sharing, analysis of trade-offs), and are shown to perform as designed, the last step of a user study is not taken.



(a) Number of queued requests (created and managed in simulation).



(b) CPU utilization in percent (real world).



(c) Average response time in seconds (created in real world, measured by simulated component).

Figure 5.10: The actual graphs displayed in the Metrics Dashboard, unifying real-world metrics with metrics created or measured by the simulation.

Chapter 6

Autonomic Configuration using Simulation

Chapter Contents

Configuration management is a complex task, even for experienced system administrators, which makes self-managing systems a desirable contribution [32, 48, 56]. This chapter describes a novel approach to self-managing systems, implemented as an autonomic configuration tool.

The self-configuration decision-making methodology uses a simulation model of the subject system and systematically executes it in order to collect examples of its expected behavior under different conditions of load, configuration and SLA constraints. The motivating reconfiguration scenarios involve adapting the number of servers on which the system services are deployed.

From these examples of behavior, a behavioral state-transition model is created. The metrics recorded in each monitoring step are discretized to produce a more coarse-grained representation of the recorded data, which is clustered into classes, each of which represents a behavioral *state*. Two states are related with a *transition* between them if an instance of the source state is followed in time by an instance of the destination state. These transitions represent changes in demand (when the system load metrics increase) or changes in the system configuration (when the system configuration changes). Precise definitions of the states and transitions, and the details of how the complete model is created, are provided in §6.1.

The autonomic manager (§6.2) monitors the system at run-time, taking periodic snapshots of its load and configuration. This data enables it to identify in which of the behavioral states it is, tracking the progress of the system through the state-transition model. When the system is in a state known to violate (or to potentially lead to violations of) its SLA, a search for a configuration change transition (or a series thereof) that will lead to a satisfactory state is initiated. The required changes are executed; that is, the subject system is reconfigured and monitoring resumes.

To demonstrate and validate the methodology, a state model is created using TAPoRsim (§4.4). An autonomic manager is implemented and verified in simulation, then TAPoRware is deployed to a cloud computing environment and the autonomic manager is tested in a real-world environment. §6.2.1 and §6.2.2 describe the experiment and demonstrate a decision model generated entirely in simulation can be used to make accurate changes to a real-world application.

§6.3 examines the process of generating the state-transition model in more detail. The experiments used for verification use states gleaned from the simulation while it is being exercised by an expert user. The most expensive part of the process is the involvement of an expert. An alteration to the methodology involves exercising the simulation automatically and systematically, in an attempt to cover all feasible states. To test this approach, the performance of the autonomic manager is tested in two scenarios: one where the expert-generated traces are used to construct the state-transition model, and a second where traces from thousands of systematically configured simulations are used instead.

The nature of the state-transition model and the attempt to cover "all feasible states" raises questions about how much data is needed to produce an autonomic manager of sufficient quality. The volume and granularity of data is important to understand - the potential issue is that if too many states are required to accurately represent the system, problems of scale will arise. The usefulness of the autonomic manager with even a small number of states is demonstrated in §6.3.3.

6.1 State-Transition Model

The decision model used to inform the autonomic manager is a set of states and transitions between them. This section defines states and transitions, and discusses how the model is obtained by monitoring a simulation.

6.1.1 States

A *snapshot* is a tuple of metrics recorded within a period of time that show the behavior of the application at a moment in time. For each snapshot, a *snapshot descriptor* is calculated and recorded. A snapshot descriptor has three parts. The first part defines the state's compliance with the SLA: *Satisfactory*, *Unsatisfactory*, or *Boundary* (the *SUB metric*). Boundary states are states which meet the SLA, but which have transitions to unsatisfactory states[1]. The second element is a set of configuration-related parameters, *i.e.*, a set of metrics related to the current environment (number of incoming requests, *etc.*). Finally the third value of the snapshop triple is a set of performance metrics:

$$S = \langle (\mathrm{S} \mid \mathrm{U} \mid \mathrm{B}), \langle \text{Configuration} \rangle, \langle \text{Environment} \rangle, \langle \text{Metrics} \rangle \rangle$$

The internal structure of the snapshot descriptor, namely the actual metrics included in the configuration-parameter and performance-metrics set, depends on the

[1]Note that this value is not static: the same simulation data with a different SLA will produce different states as the SUB metric may change.

1. SUB metric
2. Configuration
   (a) Number of servers.
   (b) Number of processors.
   (c) Limit on concurrent requests
3. Environment
   (a) Requests / 5 seconds (sliding window)
   (b) Request size / 5 seconds (sliding window)
4. Performance
   (a) Total queued requests
   (b) CPU utilization (over all processors)
   (c) Average response time (rolling)
   (d) Response time standard deviation
   (e) Largest queue at single servers
   (f) Highest CPU utilization at single server
   (g) Smallest queue at single server
   (h) Lowest CPU utilization at single server
   (i) Throughput (responses / minute)
   (j) Memory footprint

Table 6.1: The elements of a snapshot descriptor for TAPoRsim.

subject system. The challenge is to identify a tuple that includes measurable information sufficient to characterize the performance of the system without including extraneous information. The snapshot descriptor used for TAPoRsim is shown in Table 6.1.

To control the size of the state space, snapshot descriptors are clustered into equivalence classes, called *states*. The snapshot descriptors within each state are "similar enough", in that there is no significant difference in their environment, performance, or configuration metrics[2]. Each state is annotated with a cost, the cost of the cheapest configuration of all its snapshot descriptors. The function used to determine the configuration costs could be the cost of hardware, the cost of maintenance, the "green-ness", *etc.*

Whether two values of a performance metric are "similar enough" depends on the level of precision/abstraction at which metrics are examined. At the lowest level of abstraction, any variation in any metric is considered a change. If the metrics involved are measured using real numbers (and not restricted to a range of integers), the state space is theoretically infinite. To restrict the size of the state space while still producing meaningful results, the ranges of values are discretized into "windows" where all values within a window are considered equal to each other. For enumerated performance metrics, each enumerated value defines a "window";

[2]Note that the SUB metric is ignored in the clustering process.

for example, the SUB metric has 3 possible values and has window size 3. For numeric metrics, the size of the window is chosen to give the desired number of windows and therefore the desired state space, while remaining appropriate for that metric. Three different discretization strategies are defined and implemented:

1. A fixed window size based on the theoretical range of values. For example, average response time can range from 0 seconds to 30 seconds (the timeout point). The window size is then $\frac{30 \text{ seconds}}{\text{desired \# of windows}}$.
2. A fixed window size based on the actual or expected range of values. Based on the collected simulation data, the actual range of observed values is divided by the desired number of windows, to produce the window size. For example, the average response time might actually range from .2 to 5.5 seconds, so $\frac{5.5-.2 \text{ seconds}}{\text{desired \# of windows}}$.
3. A window size chosen to ensure an equal number of values in each window. If $k = \frac{\text{\# of values}}{\text{desired \# of windows}}$, then the first k values make up the first window, the next k values make up the second, and so on.

The granularity and accuracy of the model determines the size of the state space and the eventual performance of the autonomic manager. If the window size is too large, it is not possible to understand what is actually happening in the application at each point in time, since one window many contain significantly different snapshot descriptors. If the window size is too small, the state space will be larger and as the number of transitions increases, the computation time involved in analyzing the possible future alternatives will also increase. To avoid both of these shortcomings when deciding on a window, one can empirically test a variety of window sizes and observe whether the resulting equivalence classes contain sufficiently similar simulation metrics. In parallel, one can calculate theoretical state size, based on the number of windows resulting from the chosen sizes, and ensure it is acceptable.

Once the number of windows for each metric w_m is decided, if d is the number of metrics included in the descriptor, then the state space size is the product of the window size of each descriptor as follows:

$$\text{State Space Size} = \prod_{n=1}^{d} w[n]$$

or

$$\text{State Space Size} = \prod_{n=1}^{d} \frac{r[n]}{ws[n]}$$

For example, consider a snapshot descriptor that includes only the SUB metric ($w_1 = 3$), a CPU utilization metric, and a memory usage metric. CPU utilization ranges from 0-100 (percent); with a window size of 5, there are 20 windows$w_2 = \frac{100}{5} = 20$. In the example, memory usage ranges from 0-4096 (MB); with a window size of 512, there are 8 windows. $w_3 = \frac{4096}{512} = 8$. The theoretical state space for the example is therefore $3 \times 20 \times 8 = 480$.

The actual number of states achievable in practice is less than this theoretical maximum, as some states are not possible in reality. For example, a snapshot that

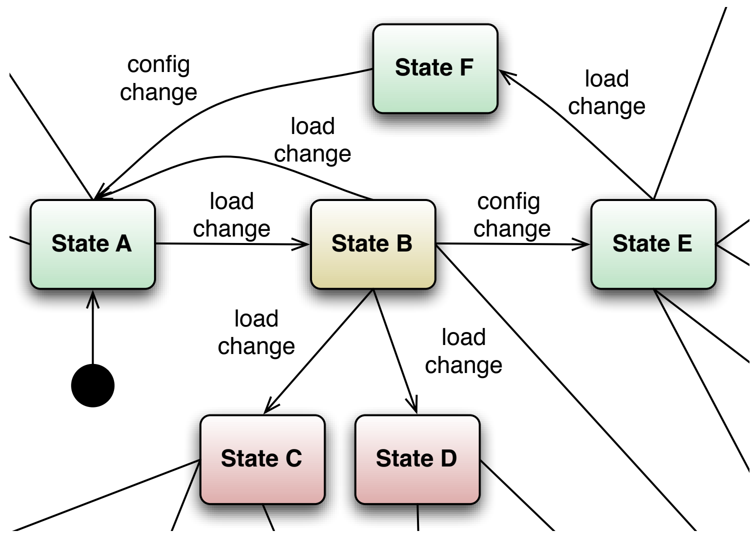


Figure 6.1: A hypothetical fragment of a state diagram; states can be satisfactory, unsatisfactory, or boundary; transitions occur when the load changes or the configuration changes.

shows the SLA being violated (U-state) but low CPU use and low memory use is not typically possible. If only the top window for CPU and the top 3 for memory are possible in a U state, then the theoretical state spaces is the number of possible U-states plus the number of possible B- and S-states. There are $5 \times 8 = 40$ possible B-states and S-states, and 1×3 U-states: 83 total. A simple constraint on the theoretical state space reduces it by almost 400 states.

6.1.2 Transitions

The model includes two types of transitions between states. First, *need-based transitions* (n-transitions) occur when the environment changes, and the software is asked to provide out-of-the-ordinary service (this new level may be the new "ordinary"). In the ongoing example of performance, this takes the form of a load change: increased requests, increased request size, network congestion, and so forth. Thus, a need-based transition implies that there is a pair of states, s_{src} and s_{dest}, such that (a) s_{src} is followed by s_{dest} in the simulation and (b) the load in s_{dest} has significantly different load metrics associated with it from the load metrics associated with s_{src} (*i.e.* the former load metrics fall in different windows from the latter load metrics). In addition to the metrics that change between s_{src} and s_{dest}, a $n - transition$ is characterized by its cost differential, namely $cost(s_{src}) - cost(s_{dest})$. Recall that the cost here need not be *explicitly* financial. For example, increased load will increase power consumption; decreased throughput may incur costs as specified in an SLA; increased response time may result in customer frustration or a decline in use. These costs are *implicitly* financial; that is, they have some quantifiable financial cost that can be determined. For instance, customer frustration can be mapped to lost revenue and failed conversions.

The second type of state transitions, *configuration-based transitions* (c-transitions), occur when the configuration changes (*i.e.*, s_{dest} has a different configuration from s_{src}). This transition may be caused by a system administrator or the autonomic manager. These transitions are also annotated by their cost differential. However, they also imply a secondary cost, namely the cost of actually making the change. For example, adding a second server involves a deployment cost in addition to the ongoing maintenance and energy costs (operational costs) implied by the additional server.

As each state may have any number of these transitions, the number of occurrences of a given transition t in the simulation traces is calculated. Then, the estimated probability of a given transition occurring is given by its occurrence count relative to the other transitions, namely $\frac{occ\_count(t)}{\sum_{i=0}^{n} occ\_count(t_x i)}$.

The process also identifies an additional type of transitions, *performance transitions*. While a configuration and the incoming load on the software will define the system's performance, it generally takes some time after a load or configuration change for the system to stabilize, during which the configuration and load will not change, but the performance will (*i.e.*, s_{dest} has the same configuration and load as s_{src} but different performance metrics). The process recognizes these transitions, but they do not alter the conceptual model: if there is a chain of performance transitions following a configuration or need transition, this "chain" can be followed to the eventual end state.

Figure 6.1 depicts a hypothetical fragment of a state diagram. The system begins in State A, a satisfactory state. There are three transitions out of this state. The transition to State B is a load-change transition (need-based). An implicit loopback transition represents the possibility that the system remains in this state. If the load changes, the system is in a boundary state, with direct transitions to unsatisfactory states, namely States C and D (with probability above a given threshold). At this point the load could change again back to where it started, and move the system to State A. Or, it could continue to increase and move the system into one of these two unsatisfactory states. The final option is for a manager to intervene and change the configuration, moving the system to the satisfactory State E. Note here that if the load decreases back to its original levels, the system moves to State F, where a configuration change will move back to the cheaper State A.

6.1.3 Expert-driven Model-Construction

The model-construction process used the stochastic but realistic request generator from the Service Testing Module (§6.1) to generate requests. The request arrival rate and the configuration (number of servers) were modified manually by an expert using the user interface. A total of 18 hours of simulation traces were created (in approximately 30 minutes of elapsed time). This is a small training set and was not intended to be comprehensive. The simulation records the system behavior at 5-second intervals. As each new snapshot descriptor is recorded, its abstracted representation is calculated using the second discretization method, substituting metric values for the corresponding intervals. If a state already exists for this abstract representation, the original descriptor is clustered in it, and the state-population counter is incremented. Otherwise, a new state entry is added to the model. A sample real state-transition model (from only 3 similar simulation traces) is shown

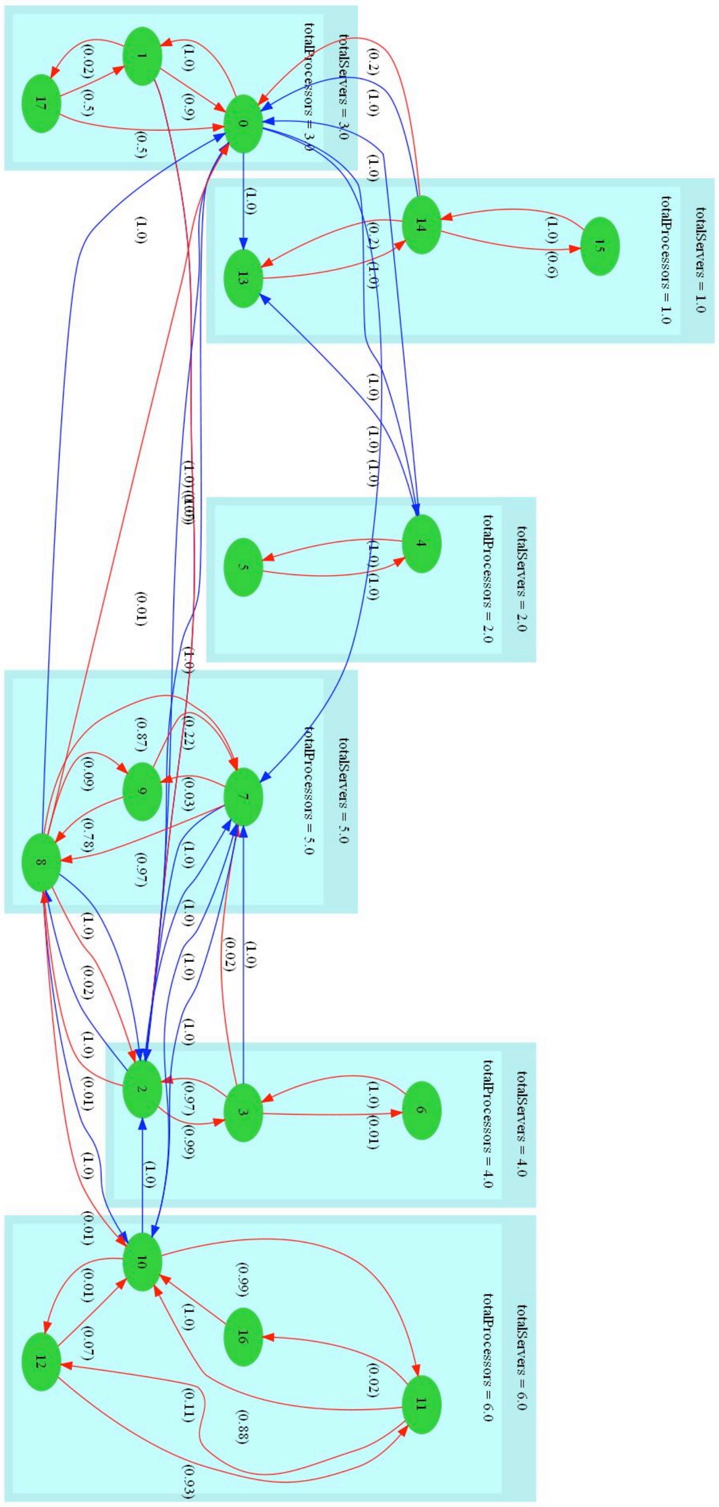


Figure 6.2: The states identified in three simulation traces; a permissive SLA ensures they are all S-states, each transition is annotated with its probability.

in Figure 6.2. The actual number of states identified was 83.

If two subsequent snapshot descriptors are not equivalent, the model-construction process analyzes what has changed: the configuration, the environment, or both. For configuration changes, the deployment cost is calculated and a c-transition is created. For load changes, a n-transition is created. When both types of metrics have changed, first a n-transition is created (where only load-related elements in the snapshot descriptor change), creating a new intermediate state if an equivalent state does not already exist. Next, a c-transition is also created (again creating a placeholder if need be). Two transitions are considered equivalent if they are both the same type (n or c) and they transition between the same two states; a counter is incremented when duplicates are encountered. The probability of each transition out of a given state is determined after state-transition detection completes. 228 unique transitions were identified (total 1,525).

The result of the model-construction process is a decision model containing the system states, their associated snapshot descriptors, and the transitions between them. Although the process does not ensure complete coverage of the system behavior, it ensures that any unsatisfactory states in the decision model have at least one c-transition from them to a satisfactory state. After construction of the state-transition model, and after any change to the SLA, all unsatisfactory states that do not have a c-transition are identified and new simulations are run to replicate each state. Once the target state is reproduced in the new simulation, configuration changes are made and the state-transition model is amended by adding the resulting states and transitions.

6.2 Autonomic Management

The autonomic management methodology specifies decision-making by monitoring the application, periodically recording a snapshot of its metrics, and classifying the snapshot as an instance of a corresponding state in the state-transition model of the system behavior. The decision on whether to make any changes to the system configuration, and which change exactly, rests on the SUB metric of the identified state.

If the current system state is classified as an unsatisfactory state, the state space is searched, starting from the current state and following only c-transitions until a satisfactory state is found. The search algorithm used for that purpose is iterative-deepening depth first search (IDDFS) [34]. Based on observations of the generated state spaces, a satisfactory state is likely to be within a few levels of the current state. IDDFS finds such states quickly with space-complexity similar to depth first search (DFS). It can be aborted and will return the best result found so far (*e.g.*, a state with improved performance but a boundary state) if it does not find a satisfactory state in the time allotted. The c-transition identified by the search algorithm is executed and the application is observed to ensure the target state is reached. A timestamp of the last change is recorded. If the search fails, the algorithm chooses from among a null action and any actions that offer an improved but not satisfactory state, based on the lowest total cost (= transition cost + (ongoing costs $\times$ expected time to remain in this state).

If the current state is classified as a boundary state, a search is initiated just

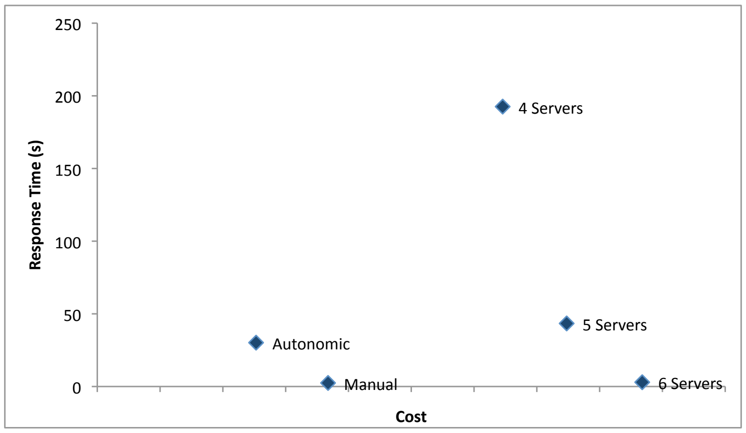


Figure 6.3: The performance / cost trade-offs for the linearly increasing data set in simulation.

as for the unsatisfactory state; however, the configuration changes implied by the transition sequence are only executed if the identified target state is superior to the current boundary state (an S-state, or a B-state with improved performance and/or lower cost).

If the current state is classified as a satisfactory state, a search for a path of c-transitions that leads to another satisfactory state is initiated. If a path is identified, it is executed if (a) the time elapsed since the last change is sufficient (for a defined threshold; 5 minutes in the implementation, but configurable), and (b) the new state is cheaper than the current state. This will remove over-provisioned resources while avoiding "churn" of repeated adds/removes for performance levels at or near SLA levels.

If the SLA changes, each of the recorded snapshot descriptors is revisited to recalculate its associated SUB metric. If after this process the system is found to be in an unsatisfactory state, the adaptation algorithm above is invoked in order to transition to a satisfactory state. Recall that every unsatisfactory state must have at least one c-transition to a satisfactory state; if new unsatisfactory states are added as a result of the SLA change, this requirement might be violated and additional simulations may be required to update the state-transition model.

Depending on the completeness of the simulation-generated state-transition model, the running application may encounter states not yet seen in the simulation traces or may transition to existing states via previously undetected transitions. In these cases, the model-construction algorithm adds the appropriate states and transitions to the model. If the new state is unsatisfactory, there is no strategy to adapt the configuration. The current implementation identifies this situation and notifies an administrator. The methodology specifies that ultimately an update to the state-transition model must be scheduled, but the immediate reaction is to use a modified state equivalence function to identify similar states that violate the SLA in the same way and use one of their c-transitions to attempt a move to a satisfactory state.

Config	Resp. Time	Cost	Performance
4	192.59	0.0%	0.0%
5	43.39	15.8%	77.5%
6	2.95	34.4%	98.5%
Manual	2.45	-43.1%	98.7%
Autonomic	30.17	-60.9%	84.3%

Table 6.2: Cost and performance metrics (and improvements) for 6 configurations and the linearly increasing data set in simulation.

6.2.1 Evaluation in Simulation

To demonstrate and test the methodology, an autonomic manager was implemented, comprising three components:

1. TAPoRsim and the SASF framework, which generate simulation traces;
2. Decision model constructor, which produces a state-transition model from simulation traces;
3. Autonomic decision maker, which uses the decision model to make re-configuration decisions.

The simulation traces and the decision model are produced offline in advance of the deployment of the autonomic decision maker; for this evaluation, the state model described in §6.1.3 was used. The autonomic decision maker is built on the SASF framework and runs in simulation. It uses a `MetricListener` implementation to monitor the running simulation, and modifies the configuration of the simulated application using the provided API.

The evaluation is based on monitoring and adapting the simulated version of the application by adding or removing servers in response to changing request loads. The service level objective was a very ambitious response time of 5 seconds, with a maximum queue length of 20 requests (recall this is not a simple web request, but CPU-intensive analysis of texts up to 7 MB in size).

Two pre-recorded but pseudo-randomly generated series of requests served as test input for the experiments. These sets of requests were not used or known when constructing the state model. For the first, an individual not familiar with this project generated 3 hours of request traffic using the modifiable stochastic request generator (variable data set). The second used a linearly increasing load, from 500 to 70,000 requests per hour, increasing by 600 requests a minute. The requests (size, type, and time) were recorded and used as input to the simulated application in 5 different conditions as follows:

- In the first three conditions, a fixed number of servers (4, 5, and 6, respectively) was used, reflecting static configurations, with no configuration changes made at run time.
- For the fourth condition, an expert user monitored performance metrics using the visualization dashboard and added or removed servers as he deemed appropriate (from 1 server to a maximum of 6 servers). The expert user had

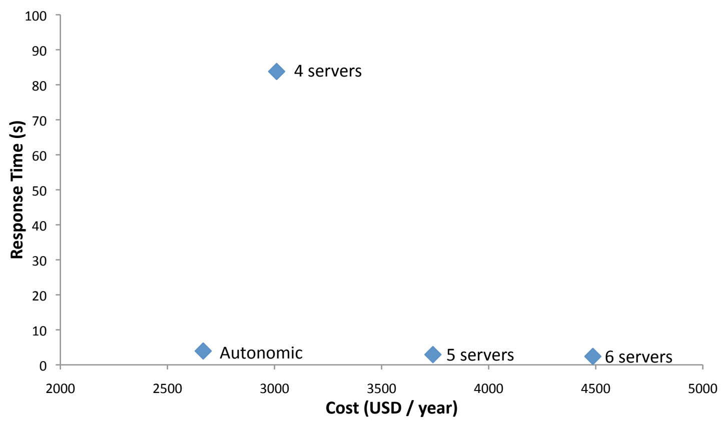


Figure 6.4: The performance / cost trade-offs for the variable data set in simulation.

Config	Resp. Time	Cost	Performance
4	83.792	0.0%	0.0%
5	2.927	24.2%	96.5%
6	2.412	49.1%	97.1%
Manual	2.445	-14.2%	97.1%
Autonomic	3.9521	-11.4%	95.3%

Table 6.3: Cost and performance metrics (and improvements) for 6 configurations and the variable data set in simulation.

no prior knowledge of the contents of the recorded requests; his decisions were based on watching the recorded performance metrics, primarily the load on each server, the queue at each server, and the overall response time. This scenario reflects configuration management by a system administrator whose only job is actively monitoring the system performance and reacting appropriately.

- For the fifth condition, the autonomic manager using 1-6 servers monitored and changed the system configuration as it deemed appropriate, based on management method described above.

For each of the above five conditions, the performance measure was recorded average response time. Configuration cost was based on total active CPU time (loosely, a private cloud computing scenario[3]). A server contributed to the cost if it was configured to process requests, whether it actually received requests or not. A slight delay was imposed to emulate start-up time for a new server, and another delay was imposed before server shutdown was initiated to give the autonomic manager time to reverse its decision and re-add the server.

A fixed four-server configuration was set as the baseline; comparisons were made relative to this baseline. Figure 6.3 and Table 6.2 show the cost/performance trade-off for each of the five conditions, for the linearly-increasing set of requests. The

[3]The estimated costs depend on the price model assumed; in this work, a hardware-leasing model is assumed. Given alternative cost-calculation models, the estimated costs would be different. This work is independent of any particular cost models.

Config	Resp. Time	Cost	Performance
4 Servers	91.59	0.0%	0.0%
5 Servers	6.36	18.8%	93.1%
6 Servers	3.36	42.1%	96.3%
Autonomic	6.61	-3.6%	92.8%

Table 6.4: Cost and performance metrics (and improvements) for 6 configurations and the variable data set as tested in a real cloud.

autonomic condition offered performance 33% better than the 5-server configuration, at one-third the cost, though it was an order of magnitude worse than the manual "expertly managed" condition. Given the monotonic increase in load, the cause of the inferior performance - and the lower cost - is that new servers were not added quickly enough; the rate at which the load increased was faster than autonomic manager was prepared to act. Nonetheless, an 84% improvement in performance with a 60% reduction in cost over the baseline is a substantial improvement.

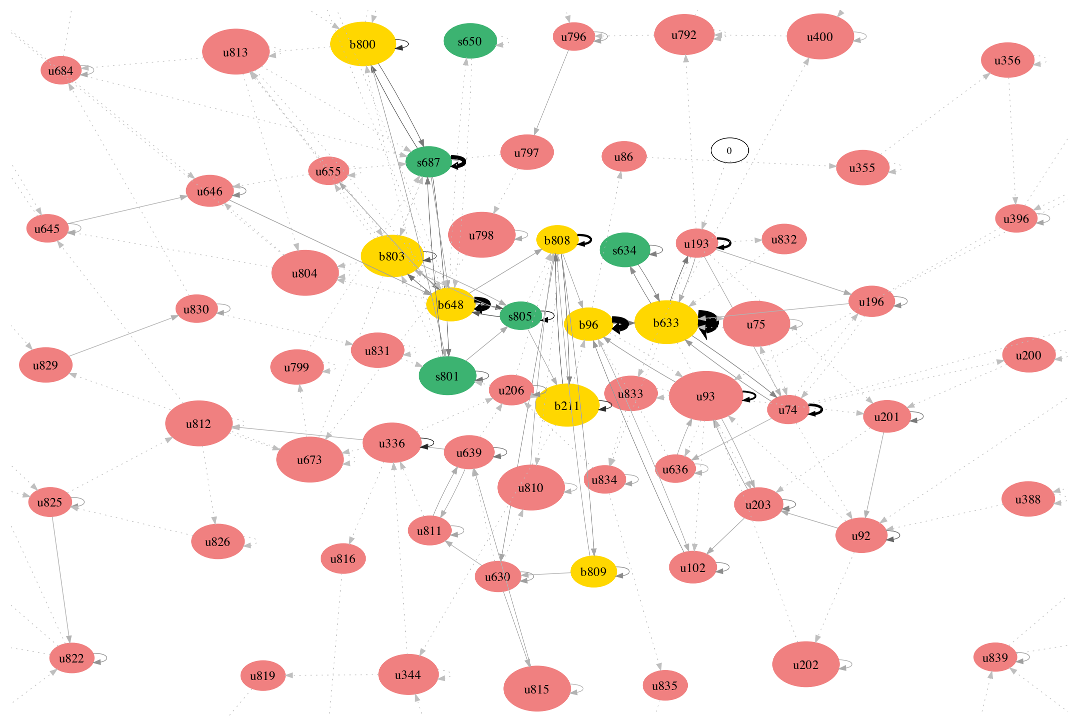
Turning to the more realistic data set (based on values from actual logs), Figure 6.4 and Table 6.3 show the same results for the variable data set. The manually and autonomically adapted conditions were cheaper than all three fixed-configuration conditions, and had similar performance: the manual adaptation had an average response time .03 worse than the best. The autonomic adaptation, though slightly worse then a manual process, offered a 95% performance improvement over the baseline and was well within the service level objectives. The autonomic approach was more expensive; this is likely due to the fact that the autonomic management process is conservative in removing resources from over-provisioned satisfactory states. Cost-improving reconfigurations only occur if some time has elapsed since the last change, where a system manager is not so constrained.

One advantage of a simulation-driven approach is the ability to see how an experiment unfolds. Figure 6.5a shows a directed graph that visualizes the sequence of states encountered in a three-hour trace of TAPoRsim being autonomically managed[4]. The set of variable requests was sent. The color of the state node indicates Satisfactory (green), Unsatisfactory (red), or Boundary (yellow). The directional arrows indicate that the simulation was in $state_{src}$ immediately before being in $state_{dst}$. Visually, the arrows indicate the number of times that sequence was encountered in simulation: the lightest dashed arrows (*e.g.*, 832 to 633) are the least frequent; solid arrows with progressively darker shades of grey are more frequent (*e.g.*, 633 to 193); black arrows with progressively greater thickness are the most frequent (*e.g.*, 633 to itself). Figure 6.5b uses the same visualization approach to the sequence of states encountered in a three-hour trace of TAPoRsim with a static 5-server configuration. This configuration was mostly compliant, but spent most of its time in boundary states.

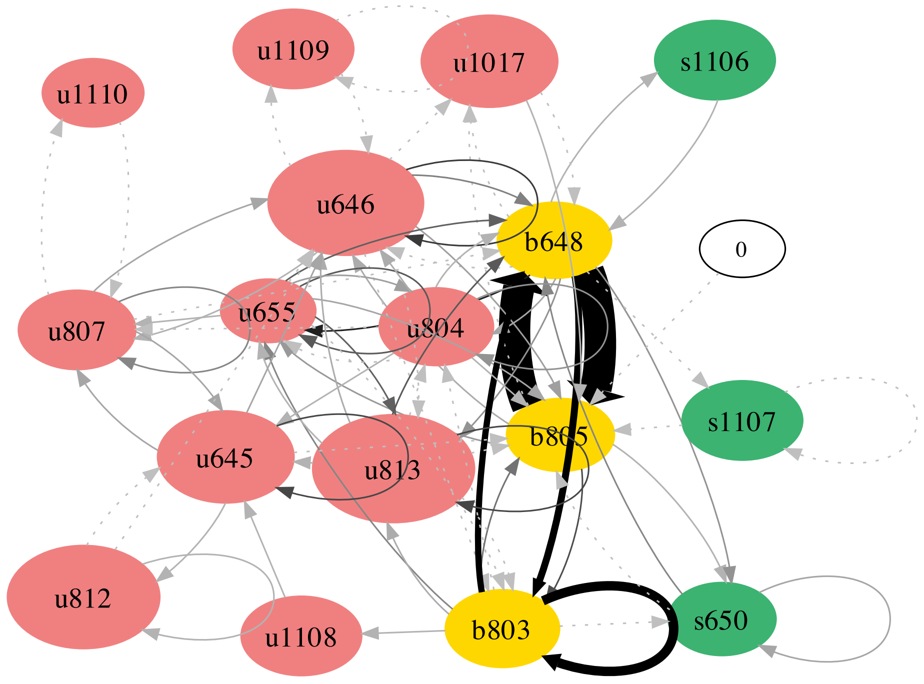
6.2.2 Evaluation in Reality

To validate this approach in a real world scenario, the experiment conducted in simulation was replicated in the real world. The components as described in §6.2.1

[4]Note: some outlying nodes were cropped from this image



(a) With autonomic manager.



(b) With fixed 5-server configuration.

Figure 6.5: Directed graph showing the states and transitions from a 3-hour execution of the simulation. Colors show the SUB metric; color and thickness of edges show frequency of transition.

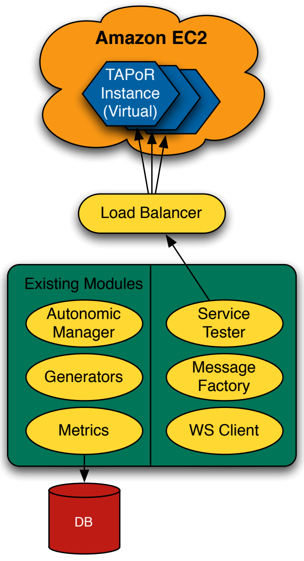


Figure 6.6: The components of the real-world testing architecture.

were used. Using the Emulation extension of SASF, simulated components were integrated with real-world components so actual service performance and performance metrics could be used. The actual manager still runs in a simulated environment, but with real data as input. The architecture of the testing infrastructure is shown in Figure 6.6.

The TAPoR web service was installed on an Amazon Elastic Compute Cloud[5] (EC2) instance, size `small` (32-bit system with 1.7 GB of memory and one EC2 compute unit, which is roughly a 1.0-1.2 GHz 2007 Xeon processor). TAPoRware was deployed as a single endpoint per instance, with requests balanced between two running TAPoRware processes per instance. An Amazon machine image was created to enable easy replication. Instances were deployed to match the number of servers used in the various simulated experiments (namely 4, 5, or 6 servers).

A load balancer acts as a single endpoint to which clients send requests, tracks the number of instances that are provisioned (either fixed or varied by the autonomic manager), and forwards incoming requests to the instance with the smallest number of outstanding requests.

On a local machine, SASF was run in emulation mode, emulating thousands of web service client requests to TAPoR services, in exactly the same sequence as in the variable data set described in the simulated evaluation, using the Service Testing Module. The metrics module of SASF was used to capture, store, and optionally visualize metrics generated by the testing application. The unmodified code used to autonomically manage the simulation framework was used to manage the real world system, via calls to the load balancer informing it that the number of servers has changed. The simulation-generated decision model as described in the previous section was used, unmodified.

The results are shown in Figure 6.7 and Table 6.4. The autonomic approach costs

[5]`http://aws.amazon.com/ec2/`

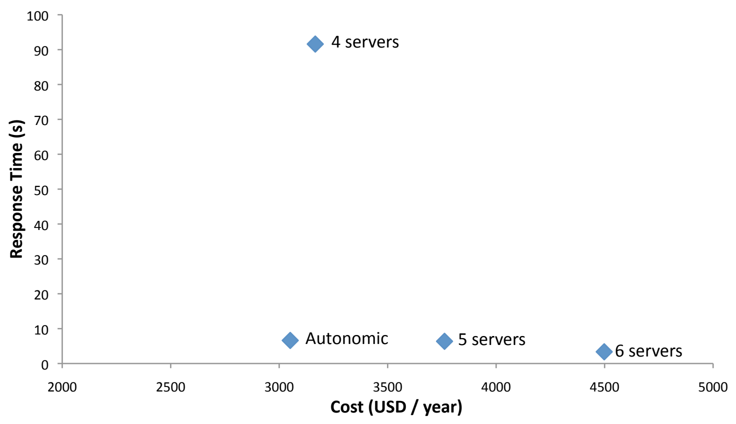


Figure 6.7: The performance / cost trade-offs for the variable data set as tested in a real cloud.

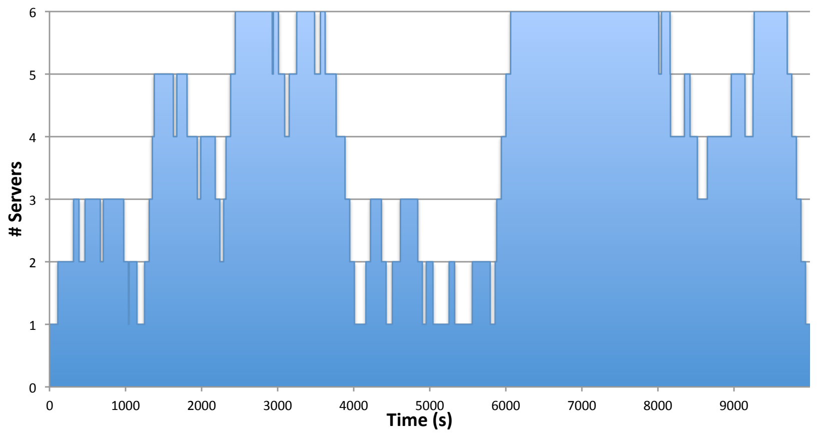


Figure 6.8: The number of servers over time as configured by the autonomic manager in the real-world testing.

less than a 4 server configuration, yet offers performance comparable to having 5 servers. Compared to the usual 4-server baseline, it reduces response times by 93% while reducing costs by 4%. Comparing these results to Figure 6.4 and Table 6.3 shows a strong correlation between the real world and the simulated testing.

Figure 6.8 shows the changing number of servers during the real-world testing of the autonomic manager. When a server is shown being added or removed only briefly, that change was typically not actually implemented in the testing infrastructure; a slight time delay is imposed on the changes requested by the autonomic manager to allow it to change its mind.

6.3 Systematic Exploration of the State Space

The autonomic approach, though effective, raises several questions. First, an expert user was required to "exercise" the simulation. This section explores replacing (ex-

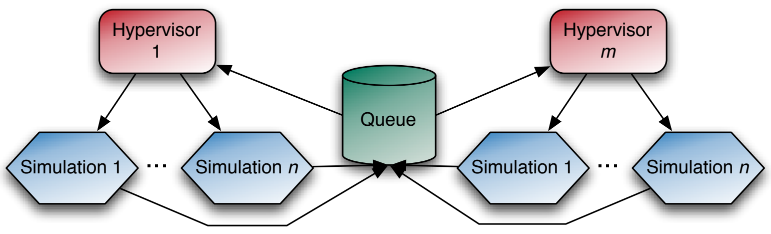


Figure 6.9: The high-level architecture of the systematic exploration.

pensive) expert knowledge with an automatic algorithm that tests the simulation in a variety of realistic simulations. Starting from a single simulation, each time a new state is encountered, a set of new simulations is launched to test mutations of that state - increasing load, adding servers, and so on. This approach does not require (expensive) expert domain knowledge, discovered more states and transitions, had fewer "missing" states at run-time, and produced better results in a new set of experiments. Statistics about the automatic generation of state-transition models are also examined.

Second, the challenge of defining a manageable search space with sufficient granularity to differentiate between states without impairing the decision-making ability of the algorithm was not addressed. This section uses various abstraction methods for calculating states from raw performance metrics, and examines how each impacts the state-transition model and the autonomic manager that uses it. A state space that is too big or too small loses effectiveness. Though intuitively a highly-granular model is desirable, the higher the better, this makes it more likely that the states encountered at run-time will be considered new and different and the quality will decrease. If there is too little granularity, predictive ability is lost.

6.3.1 Exploration-driven Model Construction

To systematically explore the configuration/performance state space of a service-oriented system, its behavior is examined in realistically possible states. There are four actions that influence the state of an application: adding a server, removing a server, increasing the rate at which requests arrive, and decreasing this rate. The simulated system must be observed to see what happens in each state when each of these actions occurs.

To achieve this, the model creation approach is changed. Rather than generating hours of simulation traces and from them extracting a state-transition model, the model is built iteratively, where every metrics snapshot from the simulation application is converted into a snapshot descriptor (using fixed window sizes, §6.1), which is evaluated to determine if it belongs to an existing state or if a new state should be added to the decision model. A simulation is started, and its metrics are monitored by the autonomic manager (via the metrics engine) at 5-second intervals. The state of the application at each interval is identified. If this state has already been seen, execution continues. If this is a new state, the steps taken to reach this state are stored: the application's topology, the configuration file, all requests sent, and any actions taken previously (and the time or state at which these actions occur; Figure 6.10). A new experiment is created for each action, where an experiment is a tuple $e =$ [id, topology, configuration, requests, actions]. The actions element includes all actions taken up to this point in the simulation, as well as the new action to be tested. A new experiment is not created if the action in this state would breach a threshold: a minimum / maximum number of servers, or a minimum / maximum request arrival rate. The new experiments are placed on a FIFO queue (implemented in a database), awaiting execution. The originating simulation continues execution.

A simulation hypervisor is responsible for retrieving experiments from this queue and executing them. The system architecture (Figure 6.9) allows for any number of hypervisors, each running any number of simulations (the limiting factor is system resources). The hypervisor is responsible for creating the environment described by the experiment, including the same topology, instructing the simulation to simulate the exact requests used in the originating simulation (then switching to stochastically generated requests), and informing the simulation of the times and states where actions would be performed. It also monitors and stores the output of the simulation, and waits for the simulation to exit before launching the next experiment.

As each simulation runs, new states are encountered and the queue grows. Each simulation will run until one of three possible normal termination points is reached: (a) the simulated application enters an implausible or failed state (e.g. the simulated server is so overloaded that it fails); (b) all actions have been taken but no new states have been seen for 20 minutes of simulated time; and (c) the state the simulation was launched to test is not reached within the time threshold. In the third case, the experiment is re-queued and attempted again. For each new state encountered, 2-4 new simulations are added to the queue.

The initial simulation is started and ended manually, and is called "experiment 0". This experiment forms *generation* 0. Each experiment it enqueues belongs to generation 1, which will enqueue experiments belonging to the 2nd generation, and so on. In this experiments, the mutations continue out to the 10th generation, after which no new experiments are enqueued. Figure 6.11 shows the number of experiments belonging to each generation. Figure 6.12 shows the number of experiments enqueued by each generation (except the 0th generation). The number of new experiments enqueued at each generation approaches 0 very quickly. Other than the start point, the only manual intervention occurred during the fifth generation when 2,914 experiments were manually pruned from the queue, as they were guaranteed to enter implausible scenarios immediately.

A total of 17,122 experiments were queued and executed over a 105 hour period on commodity hardware (Core 2 Duo processor, 4GB RAM). The majority of these experiments (82.7%) did not find any new states; 94.2% found only one new state (Figure 6.13). The total number of states identified was 5,123; 3,369 of them were unsatisfactory; there were 34,139 transitions in total. There were 720 abstracted states with 14,798 transitions.

6.3.2 Evaluation

To evaluate the automatic exploration model, another state-transition model is created using expert knowledge. This model uses the first discretization method, fixed window sizes, to match that used by the automatically generated model. As be-

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<configuration id="1815ADDSERVER">
 <atTime action="ADDSERVER" time="1637"/>
 <onState action="LOADINCREASE" stateId="367"/>
 <onState action="ADDSERVER" stateId="531"/>
 <onState action="ADDSERVER" stateId="1815"/>
</configuration>
```

Figure 6.10: A sample actions XML document, naming actions to be taken when the given states are encountered.

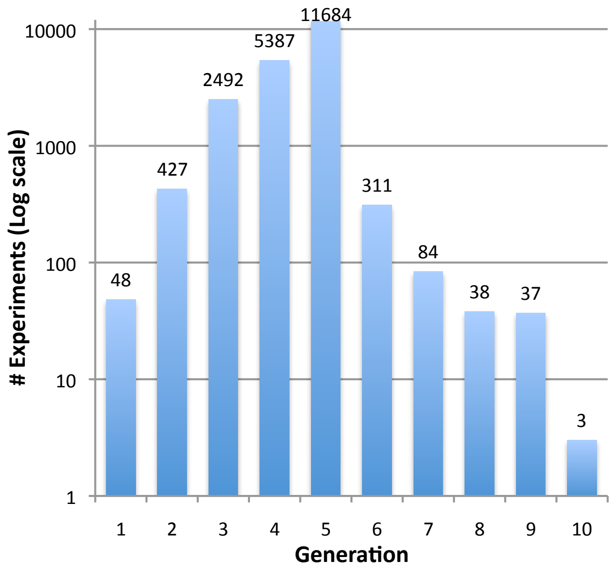


Figure 6.11: Number of simulations in each generation (log scale).

fore, using the interactive version of the simulation, an expert varied the request arrival rate of a stochastic but realistic request generator, changing the number of servers manually to adequately service the requests. The goal was to execute as many different scenarios as possible. The result was 18 hours of simulation traces with metrics captured every 5 seconds.

From this data set, 1,465 states were identified, with 2,698 total transitions. 1,304 of the states were unsatisfactory. The number of abstracted states used in the search process was 89, with 472 transitions. Given the values used in discretization, the theoretical state space is approximately 57,024,000. The majority of the theoretical state space is also impossible in reality.

The primary question for the evaluation is the relative quality of the models constructed (a) by the expert's monitoring of the simulated system and (b) by the automated state-space exploration manager, for the purpose of supporting autonomic run-time configuration management. To that end, in this experiment the simulated application is monitored and managed by adding or removing servers in

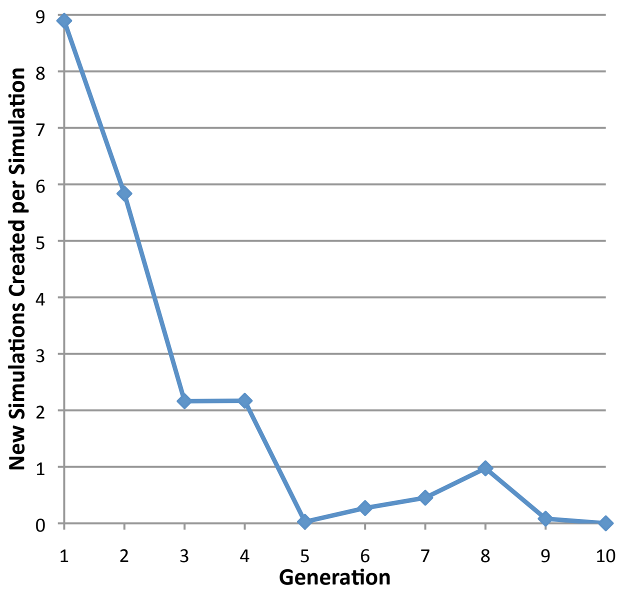


Figure 6.12: Average number of simulations enqueued per simulation for each generation.

response to changing request loads. The service level objective was a very ambitious response time of 5 seconds, with a maximum queue length of 20 requests (recall this is not a simple web request, but CPU-intensive analysis of texts up to 7 MB in size).

The input requests were the same as those used to evaluate the autonomic manager in §6.2.1 and §6.2.2 (3 hours of request traffic generated using the modifiable stochastic load generator, recorded and used for all configurations). The configuration conditions included a fixed number of servers [4-6], an autonomic manager deploying 1-6 servers based on the **expert** decision model, and the autonomic manager based on the automatic **exploration** decision model.

Figure 6.14 shows the results when comparing the expert and exploration conditions. The automatic exploration approach provided superior performance, servicing requests on average 3 times faster, with a cost increase of only 45%. This is when using the same discretization method; the expert data set does not perform as well using this discretization method as it does when using the other two. Another comparison metric is the number of unknown states encountered during the simulated tests. The expert data set encountered 730 new states (49.8% of the original total), 723 of them unsatisfactory states where remedial actions could have been taken. The exploration data set encountered 188 new states (3.6% of the original total, 185 unsatisfactory). (Most of the new states were beyond the threshold of what was considered plausible when generating the state-transition model; the thresholds could be adjusted).

6.3.3 Data Volume and Granularity

The second set of comparisons examines the relative merits of high versus low granularity when generating the decision model. If the window size is too large, it is not

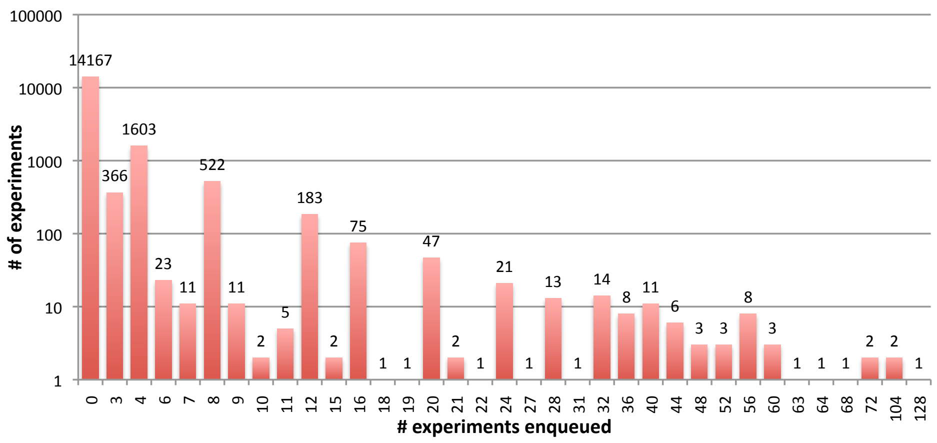


Figure 6.13: The number of experiments (y, log scale) that enqueued the given number of experiments (x).

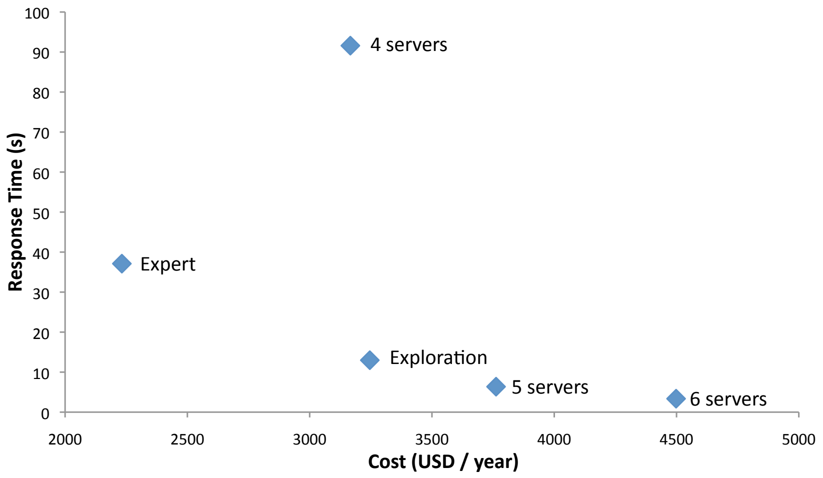


Figure 6.14: The performance / cost trade-offs for the test configurations.

possible to understand what is actually happening in the application at each point in time, since one window many contain significantly different snapshot descriptors. If the window size is too small, the state space will be larger and as the number of transitions increases, the computation time involved in analyzing the possible future alternatives will also increase. It is also not desirable to be too precise - this differentiates between states that are essentially the same.

To test for appropriate levels of granularity, the same simulation trace data is used, but with changed window sizes: the window size is both doubled and tripled (except for discrete values). The iterative construction of the state-transition model was *not* used; rather, the granularity of the existing model was reduced. The changes to state space and granularity are shown in Table 6.5. The theoretical state space is large, but many of the theoretical states are impossible in reality. The systematic exploration identified 3 to 10 times as many states as the expert-driven approach.

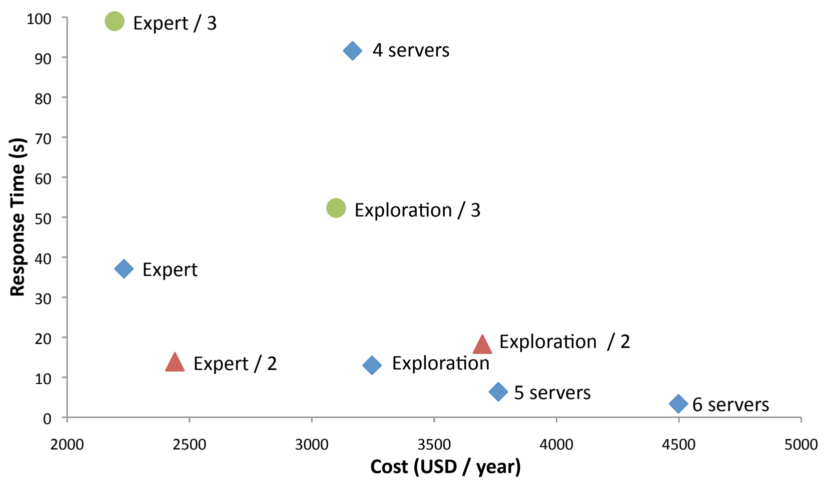


Figure 6.15: The performance / cost trade-offs for the test configurations with various granularity.

	WS 1	**WS 2**	**Drop**	**WS 3**	**Drop**
Total	57,000,000	150,000	99%	14,000	91%
Expert	1465 (.003%)	312 (.21%)	79%	231 (1.7%)	26%
Exploration	5123 (.009%)	3009 (2.0%)	41%	1306 (9.3%)	56%

Table 6.5: The changes to state space and coverage with changing granularity.

The exploration-generated model has better coverage and lost fewer states as the window size grew; from this, it appears that the automatically driven traces had a greater variety of states.

The impact on the autonomic modification of the system is shown in Figure 6.15 (overlaid on the data from Figure 6.14). Doubling the window size (decreasing granularity) actually improved the expert-driven modifications at first, though tripling the window size produced practically unusable results. It appears the tripled window size was not granular enough; the unmodified window size was too granular. The exploration model saw increased costs and worse response time for both window sizes. As the number of states decreases, the probability of encountering a state seen previously increases, improving recall at a cost to precision. This is more beneficial to the model that is sparse to begin with; the more complete model sees a smaller improvement to recall in exchange for the same cost to precision.

6.4 Threats to Validity and Shortcomings

Amazon EC2 provides one unit of computation and a certain amount of memory and IO performance, but one study showed the available resources and performance varied [69]). To minimize the impact of this variance, each experiment used three hours of requests, 12,000 in total. No direct numeric comparisons are made between local computing resources and cloud computing resources; the performance of the autonomic manager was assessed based on how a condition with a fixed number of servers performed in exactly the same environment.

Where an expert user was required, the author played this role.

The creation of the decision model used a request arrival rate and a configuration that were both set manually by the author over the 18 hours of simulated time. However, the request set used in the evaluation was entirely disparate and created by a third party.

A three hour sequence of requests is not insignificant, but repeated tests and statistical analysis are needed to ensure the correlation between the real and simulated evaluations and the improvements offered by the autonomic manager are statistically significant.

The evaluation of this approach relied on the relatively simple task of adding and removing servers based on load. Further, the service system used 6 web services per server, with the servers all hosting the exact same services. As the configurations grow more complex (distributing services across servers, for instance), the state space will grow and coverage will be more challenging.

6.5 Summary

This chapter introduced a simulation-based autonomic adaptation approach to configuring and re-configuring a service system at run-time. Simulation-generated data is used to develop a state-transition model of the behavior of the system, in terms of its configuration and performance. This model essentially captures how the system's performance is impacted by the changes in the request load that the environment imposes to the system and changes to its configuration. At run time, the system's actual behavior is tracked against this model and when the autonomic manager finds the system in a state that may lead to possible futures that violate service level agreements, its configuration is changed to move the system to a safer future state.

A case study implementation demonstrated the effectiveness of the process: the autonomic manager achieved results comparable to manual changes by an expert, though not quite at the same level. Given the potential impracticality of expert non-stop monitoring of an application, the autonomic approach has value.

In addition to the benefits provided by any self-adaptation system, the novel methodology and implementation described here offers several key benefits.

1. It is based on simulation. Assuming an accurate simulation (§4.4.3), it is known ahead of time what results any changes will produce without trial-and-error lag time which can be expensive if an SLA is being violated. Simulation data is less accurate than actual run-time data, but a greater volume of data can be acquired at a lower expense.

2. The majority of the "training" of this system occurs offline prior to deployment. There is no learning phase where the system is not useful while it monitors the behavior of a live application. At the same time, the system is still capable of learning at run-time: though it starts with a model built on simulation-generated data, the model grows and "learns" from manual changes or previously unknown states.

3. It is fairly explainable; a state model can be easily displayed showing what state the system was believed to be in, and what remedial action was chosen

as a result. Using the dashboard that is part of the simulation framework, a system manager can review simulated "what if" scenarios. This understanding is intended to help resolve issues that administrators may have with allowing an application to make its own decisions.

4. It supports dynamic SLAs, in that, changing the service level agreement does not require re-learning the entire model; new SLAs can be accommodated by a relatively inexpensive refresh of the state-transition model, where the classification of the equivalence behavioral classes are updated to reflect their conformance to the new SLA.

This set of contributions was further extended to replace the (expensive) expert user with an automated system that recursively simulated various mutations of each state of a simulated application, creating a more complete decision model. This automatic creation of a decision model offers a low-cost solution for autonomic self-configuration, which in this case was able to achieve average results closer to a defined service level objective.

Finally, the challenge of creating a state-transition model that balances accuracy, size of the state space, and the granularity of the discretization. Empirical results were presented on the growth of the state-transition model over the various generations of the automatic creation. Trade-offs in granularity when discretizing raw performance metrics into states that meaningfully differentiate between actual application states were studied, finding that it is possible to be too vague and to be too precise, and establishing empirically the appropriate balance for this application. Though the quality of the exploration-generated model deteriorated as granularity decreased, the low cost of CPU time versus expensive expert knowledge gives automatic exploration an advantage.

Chapter 7

Conclusion and Future Work

Service-oriented systems offer useful and desirable features like loose-coupling, encapsulation, network management, and enable interoperable multiple-entity service interactions. The complexity introduced by these features can result in unsatisfying interactions between service consumers and service providers: they are difficult for providers to manage and difficult for consumers to specify, particularly non-functional requirements like performance.

This dissertation explored some of these challenges: the problems with current approaches to service level agreements, the motivation to involve the customer in an understandable process to increase their perceived value, the difficulty in asking consumers to perceive value without experience and in understanding how consumers trade-off desired quality and trade-offs, and in general the benefits of understandable, explainable, available information.

Simulation was examined as a promising solution. A set of proposed canonical characteristics was defined and used to assess the existing state-of-the-art solutions for simulating service-oriented architectures. These solutions offered inadequate support for metrics gathering and visualizing, run-time interaction, emulation, extensibility, and automatic generation: generally, they had insufficient inability to accurately predict a performance-based predicted narrative for the behavior of a service, and required substantial effort to produce. One primary research objective was to develop a framework, an approach, and a set of tools to support simulating service-oriented systems with emphasis on metrics generation and minimizing development effort. A second objective was to employ this framework to produce working, accurate simulations for real-world service systems, to demonstrate not only the framework but also simulation-driven tools.

Existing approaches to configuration management do not make use of simulation to predict performance; instead, data from real-world deployed systems is used directly. This data is more expensive to produce, but is guaranteed to be accurate. When simulation is used, it is used to predict bottlenecks and resource contention issues. Consumer-driven trade-off management is rarely considered. The problems with SLAs and the potential of simulation to generate useful data to better inform the process of managing the lifecycle of an SLA led to the third objective, to use simulation to manage an SLA throughout each phase of its lifecycle: (re-)negotiation, conversion to a configuration, and deployment.

One option for managing complex systems is self-management, or autonomic computing. Existing approaches rely on real-world data, generated from a running system that is not self-managed. This training phase is expensive. Autonomic work in services focuses on substituting services with other services that perform the same function, ignoring managing at the application level. Decisions made autonomically may be difficult for administrators to understand; decisions that can be explained are rare in existing systems. This motivated the fourth research objective, constructing an autonomic manager based on simulation-generated data, using an explainable decision model, that re-configured the application and works in both simulation and in the real world.

The following section describes how each objective was achieved by describing a set of novel contributions in more detail.

7.1 Contributions Supporting Research Objectives

Objective: *Demonstrate that authoring a simulation of a service-oriented system need not be prohibitively difficult, and that such simulations can produce a narrative that offers useful and realistic information about the predicted performance of a software system.*

A services-aware simulation framework (SASF) was introduced in Chapter 3. A systematic, comprehensive review of simulation frameworks for service-oriented systems (using the proposed canonical characteristics) revealed shortcomings with existing approaches. This contribution excels at producing a narrative of the predicted performance of a service system based on past exemplars. It achieves this by offering a powerful and extensible engine for collecting and visualizing metrics and the ability to generate a simulation from an existing standardized description of the service, reducing development effort. Extensible libraries for common web service functionality offer support better than (or at least comparable to) the state of the art. The ability to replicate performance of service systems, and to test the services using requests generated based on probabilistic models derived from real data, improve the quality of the predicted narrative. Integration with real-world components allows for simplified validation in real situations. The unique ability to systematically run large numbers of varied experiments simplifies generating volumes of performance data.

These key contributions are compared to the current state of the art in §2.3.7; SASF meets or exceeds the state of the art in all areas.

Objective: *Use the simulation approach to produce simulated versions of real-world service systems using real performance data.*

SASF was used to simulate Tivoli Provisioning Manager and the back-end service for the Text Analysis Portal for Research. The first demonstrates use of basic library functionality, the simulation engine, and the metrics engine. The second adds the use of additional libraries, automated simulation generation, systematic simulation using JIMMIE, and the Service Testing Module.

The use of the framework reduced development time and effort that would have been required otherwise. In the case of TPMsim, the framework is estimated to have offered a 45% reduction in development effort. In the case of TAPoRsim, its behavior was replicated without manually authoring a single line of code. A set of extensions was added, and still only 28% of the TAPoRsim code was hand-coded.

TAPoRsim was statistically validated and shown to replicate response times with millisecond accuracy with 95% confidence. Both simulations contribute valuable test-beds to use for testing not only the specific application simulated, but also general simulation-driven methods and tools.

Objective: *Show that simulation-driven tools can be used to help manage the governance of software systems throughout the cycle of negotiating standards for service performance, configuring the service to meet those standards, and evaluating and monitoring ongoing compliance with those standards.*

The first step in achieving this objective was contributing a new SLA lifecycle, one with a focus on re-evaluation and re-negotiation of the SLA as technical and business needs change. Three additional contributions use simulation to assist each phase of the lifecycle.

First, a methodology, tool, and numeric representation of trade-off balance and cost-benefit analysis to assist in understanding and choosing the balance among dependent metrics. It was demonstrated that hundreds of candidate configurations could be tested, visualized, and presented to a user who chose their preferred trade-offs. The tool narrowed down the candidate configurations by 94%.

Second, a question-answering methodology and tool that assists a service provider in producing a configuration that meets the SLA (through his or her interaction with the tool), and in better understanding a service system generally. The approach was demonstrated on two questions, on both the simulation and on a real-world deployment. A positive correlation between simulation-generated answers and the real-world answers was observed. When asked to identify the best configuration from a set of hundreds of possible configurations, the answer produced by the question answering tool produced a configuration that performed better than the one proposed by an expert with access to perfect information about the test scenario.

Finally, a monitoring and assessment tool that integrates simulated components with real-world services to use the flexible and powerful service testing modules and the metrics/dashboard framework of SASF to produce, visualize, and store real-world data. The tool was demonstrated exercising a real-world deployment of TAPoR and visualizing simulation-generated metrics alongside real-world metrics.

Objective: *Demonstrate that a decision model generated in simulation can be used to reason about a real-world software system, to the point that such reasoning can be trusted to re-configure the service at run-time.*

A set of contributions centered on autonomically re-configuring a software system at runtime achieves this objective. Simulation-generated data was used to contribute a state-transition model of the behavior of the system, in terms of its configuration and performance. This model essentially captures how the system's performance is impacted by the changes in the request load that the environment imposes to the system and changes to its configuration. At run time, the system's actual behavior is tracked against this model and when the autonomic manager finds the system in a state that may lead to possible futures that violate service level agreements, its configuration is changed to move the system to a safer future state. Simulation allows for generating more data than might otherwise be available, all occurring offline. The state-transition model can also be updated at run-time, both by adding new states and by changing the SLA in response to business needs. This model can be visualized and is explainable, addressing a need not usually met by autonomic systems.

A case study implementation demonstrated the effectiveness of the process: the autonomic manager achieved results comparable to manual changes by an expert, though not quite at the same level. This was tested in both simulation and a real-world cloud computing environment. Given the potential impracticality of expert non-stop monitoring of an application, this contribution has value.

To further this contribution, the decision model construction process was extended to add an automated system that recursively simulated various mutations of each state of a simulated application, creating a more complete decision model. This automatic creation of a decision model offers a low-cost solution for autonomic self-configuration, which in this case was able to achieve average results closer to a defined service level objective.

For the final contribution, the challenge of creating a state-transition model that balances accuracy, size of the state space, and the granularity of the discretization was addressed. Empirical results were presented on the growth of the state-transition model over the various generations of the automatic creation. Trade-offs in granularity when discretizing raw performance metrics into states that meaningfully differentiate between actual application states were studied, finding that it is possible to be too vague and to be too precise, and establishing empirically the appropriate balance for this application.

7.2 Future Work

The SASF service testing module is useful for generating web service requests using statistical distributions based on real-world information. However, in §4.2, this module proved insufficient to model more complex client behavior. Adding a `StateMachineDrivenEndpoint` to the SASF Simulation Engine libraries would allow simulation authors to define a state machine that governs the behavior of an endpoint, and a set of messages that move it from one state to another. This would further reduce simulation development time when more complex behavior is required.

Currently, SASF's CPU-bound operation support is limited to performance profiles for operations that are $O(n)$ in time complexity, where n is the size of the input. Extending this support to more intensive operations, or modifying n to be another factor, is a planned extension. The `CPUServiceOperation` library class will be extended to offer a more flexible approach to instantiating a formula to calculate the processing time required for a request. This will better support simulation authors and will make wsdl2sim available to more types of services.

Though simple composition of services is provided, more complex composition support can be implemented by a simulated BPEL engine. This engine would read existing BPEL documents and orchestrate a series of simulated messages to replicate the behavior of a real-world BPEL engine. Though composition testing support is available in existing simulation frameworks, adding BPEL support to SASF would integrate composition testing support with a framework capable of predicting a narrative and generating accurate performance metrics. Such a model would be capable of supporting run-time composition.

No one other than the author and those who helped implement the simulation framework have used it to simulate applications; no user study of its usability for producing simulated services has been conducted. The framework will be released as an open-source project; this would allow further assessment of its usefulness. The framework could also be employed in a course project. This could identify additional features that would be useful to SASF.

Though TPMsim is shown to accurately replicate results from an emulation environment, no systematic approach using TPMsim to assist in configuring a real-world TPM deployment is described. Future work includes recommending deployments given an input network topology. The approach would use JIMMIE to systematically modify the number and location of content distribution depots, test how each performs when distributing files of varying sizes, and choose a configuration that best balances overall cost and total deployment time using the trade-off analysis tool.

An emerging area of research is migrating existing software infrastructure to cloud-based infrastructures. These systems have different hardware profiles and different cost-models, both of which are not always well understood. A future project involves using simulation to predict both performance and cost in a leased computing environment (*e.g.*, Amazon EC2). This migration question is believed to be application-specific, which makes an application-level simulation that can be automatically generated a useful tool for providing an answer. The current hardware profile of SASF is based on a non-virtualized machine. This profile would be modified to correspond to an Amazon compute unit, their defined unit of computation. The process of constructing a performance profile would also change: IO performance is the biggest differentiator between "real" and virtualized systems, so accurate representation of the predicted IO of a service is important. The first step in this work will be the migration of applications from physical computing environments to virtualized environments to better understand the differences.

The simulation-driven approaches to managing the SLA lifecycle have not been validated in user studies that demonstrate that SLAs, configurations, and assessments are actually producing results that satisfy users. Though the methodologies and tools meet the derived requirements (more co-created value, more information sharing, analysis of trade-offs), and are shown to perform as designed, the last step of a user study is a future project. The challenge of such a user study is asking individuals to make decisions to maximize their perceived value in a service-management scenario where they may have no intuitive sense of what is valuable. The study will translate this problem to a more common situation - instead of services processing requests, the scenario might be cashiers processing customers in a retail environment. Users will be asked to establish an SLA based on their intuition, then will be asked to use the trade-off analysis tool to choose SLA targets that maximize their perceived value. The SLA target they specify will be simulated and the results will be presented to them. Gap analysis will be used to measure their satisfaction with the performance of the resulting configuration; the hypothesis is there will be less of a gap between expectations and satisfaction when the trade-off analysis tool is used.

Bibliography

[1] E. Al-Masri and Q.H. Mahmoud. Qos-based discovery and ranking of web services. In *Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on*, pages 529 –534, 2007.

[2] H. Alla and R. David. Continuous and hybrid petri nets. *Journal of Circuits Systems Computers*, 8(1):159–88, February 1998.

[3] D. Ardagna and B. Pernici. Adaptive service composition in flexible processes. *Software Engineering, IEEE Transactions on*, 33(6):369 –384, June 2007.

[4] Raphael M. Bahati, Michael A. Bauer, and Elvis M. Vieira. Adaptation strategies in policy-driven autonomic management. In *ICAS '07: Proceedings of the Third International Conference on Autonomic and Autonomous Systems*, page 16, Washington, DC, USA, 2007. IEEE Computer Society.

[5] B.V. Balachandran, R. Balakrishnan, and K. Sivaramakrishnan. Capacity Planning With Demand Uncertainty. *The Engineering Economist*, 43(1):49–72, 1997.

[6] Jeanette Blomberg. Negotiating meaning of shared information in service system encounters. *European Management Journal*, 26(4):213 – 222, 2008.

[7] P. Brebner. Service-oriented performance modeling the mule enterprise service bus (esb) loan broker application. In *Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on*, pages 404–411, 2009.

[8] Paul Brebner, Liam O'Brien, and Jon Gray. Performance modeling for e-government service oriented architectures (SOAs). In Sonya Rosbotham Ashley Aitken, editor, *ASWEC*, pages 130–138, Australia, March 2008. ACS.

[9] Paul C. Brebner. Performance modeling for service oriented architectures. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 953–954, New York, NY, USA, 2008. ACM.

[10] P.F. Brown, R. Metz, and B.A. Hamilton. Reference model for service oriented architecture 1.0. *OASIS Standard*, 2006.

[11] Radu Calinescu and Marta Kwiatkowska. Using quantitative analysis to implement autonomic IT systems. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 100–110, Washington, DC, USA, 2009. IEEE Computer Society.

[12] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Francesco Lo Presti, and Raffaela Mirandola. Qos-driven runtime adaptation of service oriented architectures. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 131–140, New York, NY, USA, 2009. ACM.

[13] Jr. Cronin, J. Joseph and Steven A. Taylor. Measuring service quality: A reexamination and extension. *The Journal of Marketing*, 56(3):pp. 55–68, 1992.

[14] Jr. Cronin, J. Joseph and Steven A. Taylor. SERVPERF versus SERVQUAL: Reconciling performance-based and perceptions-minus-expectations measurement of service quality. *The Journal of Marketing*, 58(1):pp. 125–131, 1994.

[15] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web services on demand: Wsla-driven automated management. *IBM Syst. J.*, 43:136–158, January 2004.

[16] R. David and H. Alla. *Discrete, Continuous, and Hybrid Petri Nets.* Springer, 2010.

[17] D. Draheim, J. Grundy, J. Hosking, C. Lutteroth, and G. Weber. Realistic load testing of web applications. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering, 2006. CSMR 2006.*, pages 56–67, March 2006.

[18] Panos Fitsilis. Practices and problems in managing electronic services using SLAs. *Information Management & Computer Security*, 14(2):185–195, 2006.

[19] Forrester Research. Managing IT services from the outside in. `http://www.compuware.com/dl/ManagingITServicesFromTheOutsideIn.pdf`.

[20] W3C Working Group. Web services glossary. `http://www.w3.org/TR/ws-gloss/`.

[21] John Grundy, John Hosking, Lei Li, and Na Liu. Performance engineering of service compositions. In *SOSE '06: Proceedings of the 2006 international workshop on Service-oriented software engineering*, pages 26–32, New York, NY, USA, 2006. ACM.

[22] Vasfi Gucer, Alfredo Olivieri, Ghufran Shah, Scott Berens, David Campbell, Fabrizio Salustri, Johan Raeymaeckers, Marc Remes, Murtuza Choilawala, Philippe Hamelin, and Ignacio Fernandez Gonzales. IBM redbook: Deployment guide series: IBM Tivoli Provisioning Manager version 5.1, 2007.

[23] Nicole Hargrove, Ian Heritage, Prasad Imandi, Martin Keen, Wendy Neave, Laura Olson, Bhargav Perepa, and Andrew White. *Service Lifecycle Governance with IBM WebSphere Service Registry and Repository.* IBM, December 2009.

[24] P. Hasselmeyer, B. Koller, I. Kotsiopoulos, D. Kuo, and M. Parkin. Negotiating SLAs with Dynamic Pricing Policies. *Proceedings of the SOC@ Inside'07*, 2007.

[25] Qiang He, Jun Yan, Ryszard Kowalczyk, Hai Jin, and Yun Yang. Lifetime service level agreement management with autonomous agents for services provision. *Information Sciences*, 179(15):2591 – 2605, 2009. Including Special Issue on Computer-Supported Cooperative Work - Techniques and Applications, The 11th Edition of the International Conference on CSCW in Design.

[26] IBM. An architectural blueprint for autonomic computing. Technical report, June 2006.

[27] IBM. Education assistant: Service quality manager client user function. `http://publib.boulder.ibm.com/infocenter/ieduasst/tivv1r0/topic/com.ibm.iea.netcoolsqm/netcoolsqm/4.1/provisioning/sqm_client_user_function.pdf`, January 2011.

[28] Gabriel Iszlai, Igor Fedorenko, and Mark Leitch. White paper: Large scale deployment using Tivoli Provisioning Manager, 2008.

[29] Diane Jordan and John Evdemon. Web services business process execution language version 2.0. *OASIS Standard*, 2007.

[30] Krishna Kant and Youjip Won. Server capacity planning for web traffic workload. *IEEE Transactions on Knowledge and Data Engineering*, 11(5):731–747, 1999.

[31] Alexander Keller and Heiko Ludwig. The wsla framework: Specifying and monitoring service level agreements for web services. *J. Netw. Syst. Manage.*, 11:57–81, March 2003.

[32] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[33] Sungung Kim, Hessam S. Sarjoughian, and Vignesh Elamvazhuthi. Devs-suite: a simulator supporting visual experimentation design and behavior monitoring. In *SpringSim '09: Proceedings of the 2009 Spring Simulation Multiconference*, pages 1–7, San Diego, CA, USA, 2009. Society for Computer Simulation International.

[34] R.E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[35] Giannis Koumoutsos, Spyros Denazis, and Kleanthis Thramboulidis. Sla e-negotiations, enforcement and management in an autonomic environment. In *Proceedings of the 3rd IEEE international workshop on Modelling Autonomic Communications Environments*, MACE '08, pages 120–125, Berlin, Heidelberg, 2008. Springer-Verlag.

[36] E.J. Langer. Rethinking the role of thought in social interaction. *New directions in attribution research*, 2:35–58, 1978.

[37] Mark Leitch, Clement Au, Andrew Kaye-Cheveldayoff, and Cindy Lee. White paper: IBM Tivoli Provisioning Manager 5.1 performance and scalability results, 2007.

[38] T. Levitt. Production-line approach to service. *Harvard Business Review*, 50(5):41—52, 1972.

[39] Marin Litoiu, Mircea Mihaescu, Dan Ionescu, and Bogdan Solomon. Scalable adaptive web services. In *Proceedings of the 2nd international workshop on Systems development in SOA environments*, SDSOA '08, pages 47–52, New York, NY, USA, 2008. ACM.

[40] Marin Litoiu, Murray Woodside, and Tao Zheng. Hierarchical model-based autonomic control of software systems. *SIGSOFT Softw. Eng. Notes*, 30:1–7, May 2005.

[41] Na Liu, John Grundy, and John Hosking. A visual language and environment for composing web services. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 321–324, New York, NY, USA, 2005. ACM.

[42] Yoke-Hean Low, Chu-Cheow Lim, Wentong Cai, Shell-Ying Huang, Wen-Jing Hsu, Sanjay Jain, and Stephen J. Turner. Survey of languages and runtime libraries for parallel discrete-event simulation. *SIMULATION*, 72(3):170–186, 1999.

[43] Richard Lutz. Quality is as quality does: An attitudinal perspective on consumer quality judgments. Presentation to the Marketing Science Institute Trustees' Meeting, Cambridge, MA, 1986.

[44] Khaled Mahbub and George Spanoudakis. Proactive sla negotiation for service based systems. In *Proceedings of the 2010 6th World Congress on Services*, SERVICES '10, pages 519–526, Washington, DC, USA, 2010. IEEE Computer Society.

[45] E. Mancini, U. Villano, M. Rak, and R. Torella. A simulation-based framework for autonomic web services. In *Parallel and Distributed Systems, 2005. Proceedings. 11th International Conference on*, volume 2, pages 433 –437, 2005.

[46] D.A. Menasce and V. Almeida. *Capacity Planning for Web Services: metrics, models, and methods.* Prentice Hall PTR Upper Saddle River, NJ, USA, 2001.

[47] Peter K. Mills and Dennis J. Moberg. Perspectives on the technology of service operations. *The Academy of Management Review*, 7(3):467–478, 1982.

[48] Richard Murch. *Autonomic Computing.* IBM Press, 2004.

[49] Richard E. Nance. Simulation programming languages: an abridged history. In *Proceedings of the 27th conference on Winter simulation*, WSC '95, pages 1307–1313, Washington, DC, USA, 1995. IEEE Computer Society.

[50] Srini Narayanan and Sheila A. McIlraith. Simulation, verification and automated composition of web services. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 77–88, New York, NY, USA, 2002. ACM.

[51] Srini Narayanan and Sheila A. McIlraith. Simulation, verification and automated composition of web services. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 77–88, New York, NY, USA, 2002. ACM.

[52] Srinivas Narayanan. Reasoning about actions in narrative understanding. In *IJCAI'99: Proceedings of the 16th international joint conference on Artifical intelligence*, pages 350–355, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[53] Lars Nilsson, Michael D. Johnson, and Anders Gustafsson. The impact of quality practices on customer satisfaction and business results: product versus service organizations. *Journal of Quality Management*, 6(1):5 – 27, 2001.

[54] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 2008.

[55] Ramon Nou, Samuel Kounev, Ferran Julií, and Jordi Torres. Autonomic qos control in enterprise grid environments using online simulation. *J. Syst. Softw.*, 82:486–502, March 2009.

[56] Manish Parashar. *Autonomic Computing: Concepts, Infrastructure, and Applications / Editor(s): Manish Parashar and Salim Hariri.* Taylor & Francis, Inc., Bristol, PA, USA, 2007.

[57] A. Parasuraman, L.L. Berry, and V.A. Zeithaml. Refinement and reassessment of the SERVQUAL scale. *Journal of Retailing*, 1991.

[58] A. Parasuraman, V.A. Zeithaml, and L.L. Berry. A conceptual model of service quality and its implications for future research. *The Journal of Marketing*, 49(4):41–50, 1985.

[59] A. Parasuraman, V.A. Zeithaml, and L.L. Berry. SERVQUAL: A Multiple-Item Scale For Measuring Consumer Perception. *Journal of Retailing*, 64(1):12–40, 1988.

[60] A. Parasuraman, V.A. Zeithaml, and L.L. Berry. Reassessment of expectations as a comparison standard in measuring service quality: implications for further research. *The Journal of Marketing*, 58(1):111–124, 1994.

[61] Lissa F. Pollacia. A survey of discrete event simulation and state-of-the-art discrete event languages. *SIGSIM Simul. Dig.*, 20:8–25, September 1989.

[62] Franco Raimondi, James Skene, and Wolfgang Emmerich. Efficient online monitoring of web-service slas. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 170–180, New York, NY, USA, 2008. ACM.

[63] Geoffrey Rockwell. Tapor: Building a portal for text analysis. In Raymond Siemens and David Moorman, editors, *Mind Technologies; Humanities Computing and the Canadian Academic Community*, pages 285–299. University of Calgary Press, Calgary, AB, 2006.

[64] Pablo Rossi and Zahir Tari. Software adaptation for service-oriented systems. In *MW4SOC '06: Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*, pages 12–17, New York, NY, USA, 2006. ACM.

[65] Pawel Rubach and Michael Sobolewski. Dynamic sla negotiation in autonomic federated environments. In *Proceedings of the Confederated International Workshops and Posters on On the Move to Meaningful Internet Systems: ADI, CAMS, EI2N, ISDE, IWSSA, MONET, OnToContent, ODIS, ORM, OTM Academy, SWWS, SEMELS, Beyond SAWSDL, and COMBEK 2009*, OTM '09, pages 248–258, Berlin, Heidelberg, 2009. Springer-Verlag.

[66] H S Sarjoughian and B Zeigler. DEVSJAVA: Basis for a DEVS-based collaborative M&S environment. In *Proceedings of the International Conference on Web-Based Modeling and Simulation*, pages 29–36, 1998.

[67] Hessam Sarjoughian, Sungung Kim, Muthukumar Ramaswamy, and Stephen Yau. A simulation framework for service-oriented computing systems. In *WSC '08: Proceedings of the 40th Conference on Winter Simulation*, pages 845–853. Winter Simulation Conference, 2008.

[68] A.G. Sawyer and P. Dickson. Psychological perspectives on consumer response to sales promotion. *Research on Sales Promotion: Collected Papers, Marketing Science Institute, Cambridge, MA*, pages 1–21, 1984.

[69] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proc. VLDB Endow.*, 3:460–471, September 2010.

[70] Chungman Seo and Bernard P. Zeigler. Automating the devs modeling and simulation interface to web services. In *SpringSim '09: Proceedings of the 2009 Spring Simulation Multiconference*, pages 1–8, San Diego, CA, USA, 2009. Society for Computer Simulation International.

[71] Michael Smit, Andrew Nisbet, Eleni Stroulia, Andrew Edgar, Gabriel Iszlai, and Marin Litoiu. Capacity planning for service-oriented architectures. In *CASCON '08: Proceedings of the 2008 conference of the Center for Advanced Studies on collaborative research*, pages 144–156, New York, NY, USA, 2008. ACM.

[72] Abhishek Srivastava and Paul G. Sorenson. Service selection based on customer rating of quality of service attributes. *Web Services, IEEE International Conference on*, 0:1–8, 2010.

[73] A Sulistio, C Yeo, and R Buyya. A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools. *Software-Practice and Experience*, Jan 2004.

[74] James J. Swain. Simulation software survey: To boldly go. *OR/MS Today*, 36(5), October 2009.

[75] W Tsai, Chun Fan, Yinong Chen, and R Paul. DDSOS: a dynamic distributed service-oriented simulation framework. *Simulation Symposium, 2006. 39th Annual*, 2006.

[76] W.T. Tsai, Zhibin Cao, Xiao Wei, Ray Paul, Qian Huang, and Xin Sun. Modeling and simulation in service-oriented software development. *Simulation*, 83(1):7–32, 2007.

[77] WT Tsai, R.A. Paul, B. Xiao, Z. Cao, and Y. Chen. PSML-S: A process specification and modeling language for service oriented computing. In *The 9th IASTED international conference on software engineering and applications (SEA), Phoenix*, pages 160–167, 2005.

[78] Stephen L. Vargo and Robert F. Lusch. The four service marketing myths. *Journal of Service Research*, 6(4):324–335, 2004.

[79] K. Verma, P. Doshi, K. Gomadam, J. Miller, and A. Sheth. Optimal adaptation in web processes with coordination constraints. In *Web Services, 2006. ICWS '06. International Conference on*, pages 257 –264, September 2006.

[80] Bernard Zeigler. *Theory of Modeling and Simulation*. Wiley Interscience, New York, 1st edition, 1976.

[81] Bernard Zeigler. Hierarchical, modular discrete-event modelling in an object-oriented environment. *Simulation*, 49(5):219–230, Nov 1987. 10.1177/003754978704900506.

[82] V.A. Zeithaml. Consumer perceptions of price, quality, and value: a means-end model and synthesis of evidence. *The Journal of Marketing*, 52(3):2–22, 1988.

[83] V.A. Zeithaml, A. Parasuraman, and L.L. Berry. *Delivering quality service: Balancing customer perceptions and expectations*. Free Pr, 1990.

[84] Qi Zhang, Ludmila Cherkasova, Guy Mathews, Wayne Greene, and Evgenia Smirni. R-capriccio: a capacity planning and anomaly detection tool for enterprise services with live workloads. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 244–265, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[85] Farhana Zulkernine, Patrick Martin, Chris Craddock, and Kirk Wilson. A policy-based middleware for web services sla negotiation. In *Proceedings of the 2009 IEEE International Conference on Web Services*, ICWS '09, pages 1043–1050, Washington, DC, USA, 2009. IEEE Computer Society.