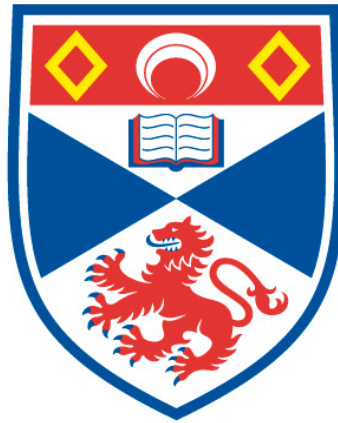


# A SCALABLE ARCHITECTURE FOR THE DEMAND-DRIVEN DEPLOYMENT OF LOCATION-NEUTRAL SOFTWARE SERVICES

Robert F. MacInnis

A Thesis Submitted for the Degree of PhD  
at the  
University of St Andrews



2010

Full metadata for this item is available in  
St Andrews Research Repository  
at:  
<http://research-repository.st-andrews.ac.uk/>

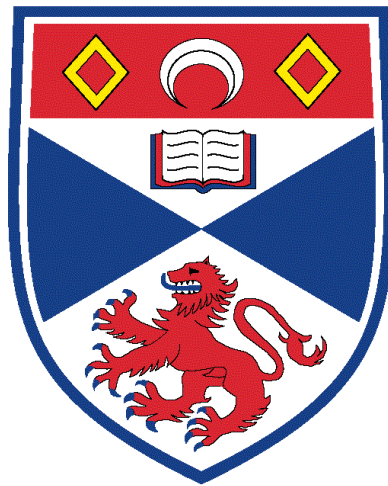
Identifiers to use to cite or link to this thesis:  
DOI: <https://doi.org/10.17630/10023-1844>  
<http://hdl.handle.net/10023/1844>

This item is protected by original copyright

A SCALABLE ARCHITECTURE FOR THE  
DEMAND-DRIVEN DEPLOYMENT OF  
LOCATION-NEUTRAL SOFTWARE SERVICES

ROBERT F. MACINNIS

PH.D. THESIS  
MAY 5<sup>TH</sup>, 2010



SCHOOL OF COMPUTER SCIENCE  
UNIVERSITY OF ST ANDREWS  
ST ANDREWS, FIFE KY16 9SX  
SCOTLAND

# Thesis Abstract

This thesis presents a scalable service-oriented architecture for the demand-driven deployment of location-neutral software services, using an end-to-end or 'holistic' approach to address identified shortcomings of the traditional Web Services model. The architecture presents a multi-endpoint Web Service environment which abstracts over Web Service location and technology and enables the dynamic provision of highly-available Web Services. The model describes mechanisms which provide a framework within which Web Services can be reliably addressed, bound to, and utilized, at any time and from any location. The presented model eases the task of providing a Web Service by consuming deployment and management tasks. It eases the development of consumer agent applications by letting developers program against what a service does, not where it is or whether it is currently deployed. It extends the platform-independent ethos of Web Services by providing deployment mechanisms which can be used independent of implementation and deployment technologies. Crucially, it maintains the Web Service goal of universal interoperability, preserving each actor's view upon the system so that existing Service Consumers and Service Providers can participate without any modifications to provider agent or consumer agent application code. Lastly, the model aims to enable the efficient consumption of hosting resources by providing mechanisms to dynamically apply and reclaim resources based upon measured consumer demand.

*To Mom & Dad,*  
*the best parents on Earth.*

# Acknowledgements

To those who have, by bearing my weight, enabled me to see farther than I ever could have alone:

Professor Alan Dearle, for putting up with me;

Dr. Graham Kirby, for noticing every, single, error, always;

and Professor Ron Morrison, for a healthier view of the world.

Thank you for teaching me what it means to learn something fully, and for letting me try on my own.

I, Robert F. MacInnis, hereby certify that this thesis, which is approximately 65,000 words in length, has been written by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree.

I was admitted as a research student in October, 2005 and as a candidate for the degree of Doctor of Philosophy in October, 2006; the higher study for which this is a record was carried out in the University of St Andrews between 2005 and 2010.

Date 9/9/2010                      signature of candidate

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date 9/9/2010                      signature of supervisor

In submitting this thesis to the University of St Andrews we understand that we are giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. We also understand that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that my thesis will be electronically accessible for personal or research use unless exempt by award of an embargo as requested below, and that the library has the right to migrate my thesis into new electronic forms as required to ensure continued access to the thesis. We have obtained any third-party copyright permissions that may be required in order to allow such access and migration, or have requested the appropriate embargo below.

The following is an agreed request by candidate and supervisor regarding the electronic publication of this thesis:

Embargo on both all or part of printed copy electronic copy for the same fixed period of five years (maximum five) on the following ground:

publication would be commercially damaging to the researcher, or to the supervisor, or the University;

publication would preclude future publication;

Pending analysis for submission of any patent application(s) we hereby request an embargo on chapters three (3), four (4), and five (5) of this thesis for a period of five years or until such time as necessary to evaluate the viability of such claims.

Date 9/9/2010                      signature of candidate                      signature of supervisor

# TABLE OF CONTENTS

1	Introduction .....	1
1.1	Software Services and Service-oriented Computing.....	1
1.2	The Web Services Model.....	2
1.3	Central Limitations of the Web Services Model.....	3
1.3.1	Rigid Addressing Model.....	3
1.3.2	Service Provider Role .....	4
1.4	Motivation.....	5
1.5	Hypothesis.....	6
1.6	Advancing the State of the Art .....	6
1.7	Thesis Contributions.....	8
1.8	Thesis Structure.....	8
2	Context Survey & Related Work.....	10
2.1	Introduction.....	10
2.2	Software Service Lifecycle .....	10
2.2.1	Terms.....	10
2.2.2	Lifecycle Stages.....	12
2.2.3	Design & Development .....	12
2.2.4	Deployment.....	14
2.2.5	Management .....	22
2.3	Web Services Model.....	27
2.3.1	Web Services Architecture .....	27
2.3.2	Web Service Actors: Roles & Responsibilities .....	32
2.3.3	Web Service Composition .....	34
2.4	Managing Change.....	36
2.4.1	By Service Consumers .....	36
2.4.2	By Discovery Services .....	40

2.4.3	By Service Providers .....	44
2.5	An Alternative Approach: Next-generation Architectures for Web Services on the Internet	46
2.5.1	DynaSoar .....	46
2.5.2	WSPeer .....	50
2.5.3	ServiceGlobe.....	55
3	A Next-Generation Architecture for Web Services on the Internet.....	60
3.1	Introduction.....	60
3.2	Requirements for a Next-Generation Web Services Architecture .....	60
3.2.1	Accommodating Technological Heterogeneity .....	60
3.2.2	Separating Entangled Concerns .....	61
3.2.3	Designing for Failure .....	63
3.3	A Next-generation Architecture .....	64
3.3.1	Introduction.....	64
3.3.2	Overview .....	64
3.3.3	Next-Generation Actors: Roles & Responsibilities .....	70
3.3.4	Architectural Entities.....	74
3.4	Intra-architecture Interactions.....	75
3.4.1	Service Consumer to Local Point of Presence (POP) .....	75
3.4.2	Publisher to Service Library.....	77
3.4.3	Service Host to Host Directory .....	78
3.4.4	POP to Active Service Directory .....	78
3.4.5	Active Service Directory to Active Service Directory.....	79
3.4.6	Active Service Directory to Manager.....	80
3.4.7	Manager to Service Library .....	82
3.4.8	Manager to Host Directory.....	82
3.4.9	Manager to Service Host .....	83
3.4.10	Service Host to Manager .....	84

3.4.11	Manager to Active Service Directory.....	85
3.5	Procedures .....	86
3.5.1	Endpoint Deployment .....	86
3.5.2	Service Invocation .....	89
3.5.3	Demand-driven Dynamic Deployment.....	90
4	Reference Implementation .....	92
4.1	Introduction.....	92
4.2	System Overview.....	92
4.3	Development Technologies.....	93
4.4	Developing Web Services .....	93
4.5	Local Point of Presence .....	96
4.5.1	Startup.....	98
4.5.2	Request Handling, Failure Detection & Recovery .....	101
4.6	Programming Consumer Agent Applications .....	114
4.7	ServiceHosts .....	116
4.7.1	Introduction.....	116
4.7.2	Role of the Administrator.....	117
4.7.3	The ServiceHost Web Service.....	118
4.7.4	Fulfilling Provisioning Requests.....	120
4.7.5	Collecting & Reporting Usage Data .....	124
4.8	Managers.....	126
4.8.1	Creating Custom Managers.....	127
4.8.2	Managing Service State.....	128
4.8.3	Deployment Planning.....	129
4.8.4	The Generic Manager.....	131
4.8.5	Bootstrapping Process.....	132
4.9	Security Considerations.....	133
5	Evaluation.....	132

5.1	Introduction.....	132
5.1.1	The Test Environment .....	132
5.1.2	Hardware.....	132
5.1.3	Software .....	132
5.1.4	Overview of Experiment Execution Process.....	133
5.1.5	The ‘TargetExperimentEndpoint’ Web Service .....	134
5.1.6	Describing Workloads with ‘WorkloadDetails’ .....	134
5.1.7	Reporting and Retrieving Experiment Results .....	137
5.1.8	Pre- and Post-Experiment Check-Lists .....	138
5.2	Efficacy of Mechanism to Reduce Average Invocation Response-time .....	139
5.2.1	Introduction.....	139
5.2.2	Hypothesis.....	139
5.2.3	Criteria for Success.....	139
5.2.4	Methodology.....	140
5.2.5	Background Work.....	140
5.2.6	Experiments.....	146
5.2.7	Custom Manager Behaviour.....	146
5.2.8	Analysis.....	152
5.3	Efficacy of Mechanisms to Increase Average Invocation Response-time and Re-distribute Load .....	154
5.3.1	Introduction.....	154
5.3.2	Hypothesis.....	154
5.3.3	Criteria for Success.....	154
5.3.4	Methodology.....	154
5.3.5	Results & Analysis.....	156
5.3.6	Experiment 2 Conclusions .....	158
5.4	Scalability and Dynamic Adaptation to Changes in Consumer Demand.....	159
5.4.1	Introduction.....	159

5.4.2	Hypothesis.....	159
5.4.3	Criteria for Success.....	159
5.4.4	Methodology.....	160
5.4.5	Experiments and Results.....	160
5.4.6	Experiment 3 Conclusions.....	187
5.5	Cost & Scalability of Infrastructure Services.....	190
5.5.1	Introduction.....	190
5.5.2	Baseline Infrastructure Costs.....	190
5.5.3	Cost of Deploying and Managing a Web Service with One Endpoint.....	192
5.5.4	Adding up the Worst-case Deployment Costs.....	197
5.5.5	Incremental Costs of Deploying and Managing Additional Web Service Endpoints	199
5.5.6	Incremental Burden Imposed on Infrastructure Services by Each Additional Service Consumer & POP.....	200
5.5.7	Incremental Burden Imposed on Infrastructure Services by Each Distinct Web Service Used by a Service Consumer.....	200
5.5.8	Burden of Increasing Web Service Demand Rate on Infrastructure Services....	201
5.5.9	Use-case: Total Cost of Providing Infrastructure for Experiment 3.3.....	202
5.5.10	Growth Rates & Scalability.....	205
5.5.11	Conclusions on the Cost & Scalability of the Infrastructure.....	208
6	Conclusions & Future Work.....	213
6.1	Introduction.....	213
6.2	Thesis Motivation.....	213
6.2.1	Central Limitations of the Web Services Model.....	213
6.2.2	Thesis Hypothesis.....	214
6.3	Thesis Summary.....	215
6.3.1	Conclusions from Experimental Evaluation.....	216
6.4	Thesis Contributions.....	219

6.5	Future Work .....	219
6.5.1	Optimizations to Existing Work.....	220
6.5.2	Integration with the ‘Cloud’ .....	221
6.5.3	Exploiting Locality in ActiveServiceDirectory Results.....	221
6.5.4	Managing the Geographical Proximity of Data & Computation .....	222
6.5.5	Universal StoreService for State-management.....	222
6.6	Finally .....	222
	References.....	224

# 1 INTRODUCTION

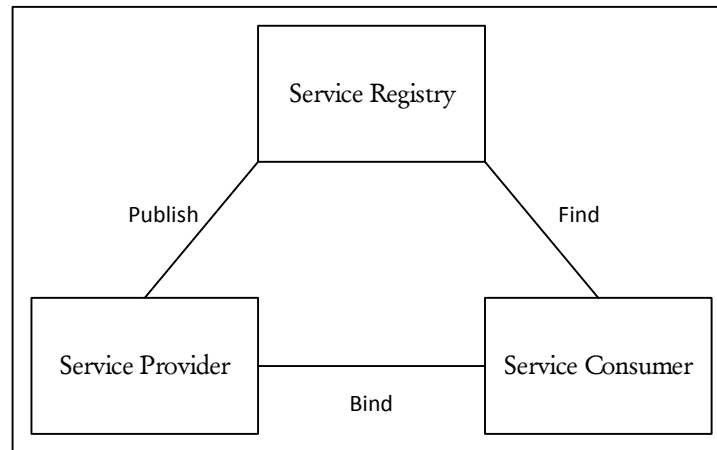
## 1.1 SOFTWARE SERVICES AND SERVICE-ORIENTED COMPUTING

Software services are understood to be autonomous, platform-independent computational elements that can be described, published, discovered and composed for the purpose of developing distributed applications. Software services are the primary unit of computation in service-oriented computing.

Service-oriented computing (SOC) promotes the assembly of application components into networks of services that can be loosely coupled to create flexible, dynamic business processes and agile applications that span organizations and computing platforms [1]. Service-oriented computing has emerged as the leading approach to evolving tightly-coupled, component-based distributed systems into wider networks of services which use uniform techniques to address, bind to and invoke service operations.

Service-oriented computing provides a way to create new architectures that reflect trends toward autonomy and heterogeneity [2]. Distributed architectures which enable the flexible and loosely-coupled processes of service-oriented computing are called service-oriented architectures (SOAs), of which software services are the primary component. The guiding characteristics of SOAs are the interoperation between loosely coupled autonomous services, the promotion of code reuse at a macro (service) level, and architectural composability.

SOAs are composed of three primary entities – a provider, a consumer, and a registry – as shown in Fig. 1. Providers publish service locations in a registry; consumers use a registry to locate services to program against and invoke. SOA implementations specify the language used to describe services, provide the means to publish and discover them, and dictate the protocols and communication mechanisms used to interact with them. The deployment of software services on the Internet is increasingly achieved using one such set of standards collectively known as ‘Web Services’ [3].



**Fig. 1 The three main components of a service-oriented architecture: a provider, a consumer, and a registry**

## 1.2 THE WEB SERVICES MODEL

Web Services are based on platform-independent standards developed through the efforts of the W3C working group [4]. These standards define the protocols, message formats, and service description language which enable interaction between clients and services on heterogeneous computing platforms across the Internet.

Web Services are widely used in industry and academia, exposing functionality ranging from basic mathematical operations to fully automated online booking systems. They have become the de-facto choice for providing Web-accessible business logic, consolidating and linking geographically dispersed operational entities and resources, and providing a common means of interaction between previously isolated business entities.

The standards introduced by Web Services provide the means to evolve distributed systems from tightly-coupled distributed applications into loosely-coupled systems of services. The standards enable interoperation between heterogeneous computing platforms through the exchange of messages (in SOAP [5] format) using well-defined interfaces (described with WSDL [6]). By abstracting over implementation and hosting technology, the platform-independent Web Service standards provide a means of homogenizing access to existing heterogeneous services while promoting the development of new services without forcing the adoption of any one particular implementation or hosting technology. The Web Services model is well suited to the emergence of Internet-wide service-oriented computing. While the Web Services model has been widely adopted, its promise is undermined by two central limitations.

### 1.3 CENTRAL LIMITATIONS OF THE WEB SERVICES MODEL

Two central limitations in the Web Services model negatively affect all stages of the Web Service lifecycle. Firstly, the Web Service addressing model is rigid and only suited to highly reliable networked environments with highly reliable hosts. It does not take into account the intrinsic dynamism and fallibility of hosts on the Internet, and applications which aim to be robust to the failure of hosts become littered with failure-recovery code. Secondly, an over-burdened and under-specified Service Provider role has led to the development of proprietary deployment systems and closed-world environments where the use of Web Services is only incidental. Saddled with these two drawbacks, the wide adoption of the Web Services model has resulted in a landscape of software services that is highly populated by applications which expose Web Service interfaces, but which are largely incompatible in terms of their required deployment systems and hosting environments.

#### 1.3.1 RIGID ADDRESSING MODEL

Web Service descriptor documents detail the data types and operations offered by a Web Service. These documents also include a fixed location for the Web Service, described using a URL – a static physical endpoint which, for any number of reasons, may be temporarily (or permanently) unavailable. The inclusion of a fixed URL in the Web Service descriptor document has far-reaching implications for all participants in a distributed Web Services framework, from developers of Web Services and administrators of Web Service hosting environments, to programmers and users of the applications which utilize them.

Programming against fixed endpoints results in tightly-coupled applications which are fragile with respect to changes in the location and availability of Web Services and are susceptible to the failure of a single host or service. If a Web Service fails or is migrated, users must either re-write code to reflect an updated endpoint, or include failure recovery mechanisms in each implementation of a client which uses a Web Service. The procedures for re-locating migrated Web Services (or locating alternative instances of a failed Web Service) are not specified by a Web Service standard, leading to the development of ad-hoc, application-specific failure-recovery techniques which are themselves susceptible to failure. This approach produces applications which are permeated by exception-handling code and tightly-coupled to both the location and the mechanisms used to locate Web Services. Due to the lack of standardized endpoint location procedures, existing work-arounds or ‘advancements’ are always proprietary, yielding non-portable, tightly-coupled, and platform-specific client applications which, in one fell swoop, mitigates all of the benefits of the Web Service model.

Providers of Web Services are also burdened by the rigid addressing model. To make Web Services locatable by clients, Web Service providers publish the location of active Web Service endpoints in a directory. Clients use this directory at design time to discover and write programs against Web Services. If service providers aim to provide a reliable and consistently available Web Service, the address of the endpoint must remain fixed, forcing an early decision on the location, amount of bandwidth, and amount of computational capacity required in the hosting infrastructure. In open-world systems, selecting an adequate provisioning level is difficult as it is difficult to predict demand levels before a service is deployed. Further, fixed-resource deployments are not scalable to demand and waste useful resources by statically provisioning them before deployment, either over- or under-shooting the necessary capacity. While dynamic deployment environments provide benefits of balanced resource consumption and high availability they typically present a closed-world view, assuming discrete capacity and highly reliable connections. Further, applications which utilize Web Services hosted in dynamic deployment environments must incorporate run-time dynamic binding techniques which are specific to the host environment. This solidifies the already tight coupling in client applications by tying them to not only the location of the Web Service, but also the location and implementation of an endpoint directory.

### *1.3.2 SERVICE PROVIDER ROLE*

The traditional Web Service model defines a ‘Service Provider’ actor which encapsulates all of the tasks of developing, deploying, hosting and managing a Web Service. Because of the implicit complexity of handling so many tasks, hosting environments are often built using proprietary ‘integrated technology’ packages: end-to-end platforms for developing, publishing, deploying, hosting, and managing services in homogeneous, fixed-resource infrastructures. The result of this broad set of responsibilities is the proliferation of proprietary deployment systems and hosting environments composed of services which cannot be deployed or managed by other systems and are only interoperable with other identical environments [7]. The resulting technology lock-in is contrary to the central ethos of platform-independence which is otherwise a hallmark of the Web Services model.

While Web Services themselves are platform-independent, the techniques used to deploy and manage them typically rely on particular platform-specific attributes or tools. In theory, developers of Web Services may use the most suitable platform for a new service, unencumbered by potentially restrictive technologies. In practice, however, this is not the case, as deployment systems are largely proprietary and hosting environments largely homogeneous. Platform-

dependence forces developers to develop for specific target environments – a closed-world approach which results in islands of incompatible services and collections of service implementations which are not portable.

Without clearly-defined sub-roles with concrete responsibilities, it is difficult for a Web Service provider to evolve their processes into more mature or automated systems. While tightly integrated systems can provide for efficiencies in design, the customized and often ad-hoc procedures devised in monolithic Service Provider implementations pose significant barriers to system evolution.

#### 1.4 MOTIVATION

The Service Provider role, as currently defined, involves high cognitive complexity and has high barriers to entry, discouraging participation by individuals due to the cost and complexity involved in fulfilling all the responsibilities of the role. It is argued that the Service Provider task list is too long, incorporates too broad a set of fields, and necessitates a significantly greater amount of domain-specific knowledge than should be required of any one actor. A modularization of the service provider role into multiple sub-roles, specifying their responsibilities, and defining concrete interfaces for interacting with them, will allow the sub-role tasks to be automated, outsourced or replaced within their environment – enabling service providers to progressively and transparently mature their systems and lowering the barriers to participation in the provision of Web Services.

This work rests on the belief that the traditional Web Service model has many entangled concerns which are in fact distinct, independent tasks. Implementing the business logic of a Web Service, for example, is very different from hosting a Web Service, requires a completely different skill-set, and a completely different set of software and hardware technologies. Similarly, while Web Services are developed for particular application servers (software applications which expose interfaces to Web Services' operations) the consideration of the target application server is very different from the tasks of preparing a host environment and deploying an instance of the Web Service – and even further removed from the decision of when and where a Web Service should be deployed.

From the point of view of application developers, it is argued that the task of implementing the business logic of an application is different than handling Web Service implementation- and distribution-related errors, and that a mechanism should be developed which abstracts over the location of a service while still preserving the current Service Consumer actor's view upon the

system – factoring out redundant failure detection and recovery code while maintaining compatibility with existing client applications.

It is argued that the rigid addressing model forces the entanglement of application business logic with remote-invocation failure recovery techniques, is detrimental to the usability and reliability of applications, adds significant time and complexity to the creation of client applications, and is a serious drawback to the adoption of the existing Web Services model. It is also argued that the over-burdened and under-specified Service Provider role has fostered the development of progressively insular, closed-world environments composed of Web Services and client applications for which the use of Web Services is only incidental. It is finally argued that the goals of universal interoperability, loose-coupling and platform-independence espoused by Web Services and service-oriented computing are worthwhile goals, that they are currently undermined by the traditional Web Services model, and that a new, next-generation model can be developed which better facilitates the realization of these goals.

## 1.5 HYPOTHESIS

The hypothesis of this thesis is that isolating the tasks and separating the concerns of participants in a distributed Web Services framework allows for abstractions over location and technology which enable the provision of pervasive Web Services which can be reliably addressed, bound to, and utilized, at any time and from any location.

Further, it is proposed that mechanisms can be developed to provide reliability and reduce complexity for users of Web Services, and that these mechanisms can be equally applicable to static, homogeneous computing environments as to dynamic environments composed of heterogeneous Web Service implementations and Web Service-hosting technologies.

## 1.6 ADVANCING THE STATE OF THE ART

This thesis presents a next-generation architectural model for Web Services on the Internet. The model belongs to a class of architectures defined by their use of abstraction layers between various stages of the traditional process of Web Service deployment, location, binding and use for the purpose of introducing further functionality or desirable properties, such as reliability and scalability. The presented model takes an end-to-end or ‘holistic’ approach to addressing the previously-identified shortcomings of the traditional Web Services model. It also reduces complexity for participants in the Web Service lifecycle by partitioning the responsibility of providing a Web Service into multiple independent roles, reducing the amount of overlapping domain-specific knowledge required by each actor and lowering the barriers to participation in a

distributed Web Services framework. Once decomposed, the tasks and responsibilities are re-structured and assigned to independent and autonomous functional entities, creating a loosely-coupled modular architecture which itself embraces the ideals of service-orientation.

The presented model also represents a break from the traditional approach to Web Service deployment by separating service substantiation from realization. Rather than being deployed explicitly, Web Service implementations are published into the infrastructure – with potentially many implementations of the same service. Hosts register their willingness to host Web Services, describing their available resources and joining a shared pool of latent hosting resources. The reference implementation utilizes this separation by applying the latent hosting resources to dynamically deploy Web Service endpoints as necessary to meet varying levels of demand. When usage levels drop, rarely-used Web Service instances are undeployed automatically in order to reclaim resources; aside from a passive bootstrapping process, an infrastructure with no demand will consume no hosting resources.

Service invocations are performed upon a client-side daemon - an indirection mechanism which transparently locates, binds to, and invokes Web Service operations on behalf of the user. The mechanism programmatically detects and recovers from endpoint failure, requesting a new endpoint from the infrastructure which returns an existing alternate or deploys a new one on demand. The model defines a failure model which, after exerting a ‘best effort’ to fulfill the request, returns users only non-recoverable errors which indicate that it is simply not possible to fulfill the request given the current state of the infrastructure.

The reference implementation eases the creation and publication of Web Services by consuming deployment and management tasks. It eases the use of Web Services by letting clients program against what a service does, not where it is or whether it is currently deployed. It extends the platform-independent ethos of Web Services by providing deployment mechanisms which can be used irrespective of implementation and deployment technology. Crucially, it maintains the Web Service goal of universal interoperability, preserving each actors’ view upon the system so that existing users and hosts can participate without *any* modifications to service implementation or client application code.

## 1.7 THESIS CONTRIBUTIONS

This thesis makes a 5-fold contribution:

- 1.) The current state of Web Service authoring, deployment, management and use is described and factors contributing to redundant human effort, high cognitive complexity, and inefficient resource utilization are identified in each of these areas. From these, requirements are specified for an architectural model which mitigates the identified limitations.
- 2.) The design of a next-generation architectural model for Web Services is presented which meets the defined requirements. The model lowers the barriers to participation in a distributed Web Services framework by reducing the cognitive complexity, effort, and overlapping domain-specific knowledge required of developers, hosts, application programmers and users of Web Services.
- 3.) A reference implementation of the next-generation architecture is described which achieves high levels of fault-tolerance and scalability and can dynamically adapt to changes in the levels of demand and available resources. Mechanisms are developed for dynamically provisioning Web Service endpoints, and are shown to be equally applicable to static, homogeneous computing environments as to dynamic environments composed of heterogeneous Web Service implementations and Web Service-hosting technologies.
- 4.) An extensible framework for the scalable and resource-efficient management of Web Services is described. This framework enables the application of both generic and customisable management techniques to the maintenance of high availability and predictable performance.
- 5.) A mechanism for transparently abstracting over the location of Web Service endpoints and dynamically detecting and recovering from endpoint failure is described. The mechanism disentangles the orthogonal tasks of implementing application business logic and writing failure-recovery routines, factoring out redundant endpoint-recovery code from client applications and thus lowering the likelihood of application failure. It provides a reverse-compatible solution to the rigid Web Service addressing model which, crucially, requires no changes to any of the Web Service standards and maintains the existing actors' views on the architecture.

## 1.8 THESIS STRUCTURE

This thesis continues in Chapter 2 with an introduction to the software service lifecycle and a detailed overview of the actors and standards of the Web Service model. A comprehensive survey of work addressing individual shortcomings of the traditional Web Services model is presented, followed by a body of work which aims to address multiple shortcomings through broader, higher-level modifications to the model. These approaches are compared with a set of requirements for a next-generation Web Service architecture at the beginning of Chapter 3.

Deemed inadequate, a new model is presented which achieves the newly defined requirements. The roles and responsibilities of each actor in the architecture are defined next, together with their interaction and failure models. A reference implementation is presented in Chapter 4 which leverages the new model in order to exemplify the beneficial properties it enables. Chapter 5 begins with an empirical evaluation of the reference implementation, exemplifying the ability to simultaneously provide simplicity, flexibility, and robustness in a distributed Web Services framework composed of heterogeneous service implementations and service-hosting technologies. The implementation is shown to be scalable – capable of effectively managing a dynamic pool of latent hosting resources in order to meet changing levels of demand; it is shown to be fault-tolerant – resistant to the failure of both the nodes which host Web Services and the architectural entities which users depend on to locate Web Services; the mechanisms developed to dynamically provision Web Services according to demand are shown to enable highly available Web Services which exhibit predictable performance across a broad range of implementation and hosting technologies. This thesis concludes in Chapter 6 with a summary of the contributions of this research and a discussion of future directions for the work.

## 2 CONTEXT SURVEY & RELATED WORK

### 2.1 INTRODUCTION

This chapter begins by establishing a concrete vocabulary for the discussion of Web Services. It defines the actors, artifacts, activities, and objectives of each stage in the software service lifecycle and uses these terms to introduce the concepts, roles and standards of Web Services. It continues with a discussion of each stage in the Web Service lifecycle and finishes with a comprehensive survey of work addressing shortcomings of the traditional Web Services model.

### 2.2 SOFTWARE SERVICE LIFECYCLE

The software service lifecycle may be simplified into three macro stages of design & development, deployment, and management, each of which has multiple sub-stages [8]. Authors ascribe various names to these stages and provide inconsistent groupings of their sub-stages and tasks. The following sections address this ambiguity by defining a common vocabulary and a concrete set of lifecycle stages which will guide the description and discussion of work presented in this thesis.

#### 2.2.1 TERMS

The terminology of Web Services is insufficient for discussing work which addresses the shortcoming of the traditional Web Services interaction model. This section begins by introducing the broader terminology of component-based distributed systems and then proceeds to the actors, actions and objectives of each macro lifecycle stage. Later, these definitions and descriptions will be used to describe the lifecycle activities of Web Services, followed by work which addresses shortcoming of the traditional Web Services interaction model.

This thesis adopts the widely-used terminology of the Object Management Group (OMG) [9] specifications. The following definitions represent terms commonly used to describe distributed software systems together with a sub-set of OMG deployment terminology described in [10]. More specific terms will be introduced later where necessary.

- **Artifact**
  - A physical piece of information that is used or produced by a deployment process. Examples of artifacts include models, source files, scripts, and binary executable files. An artifact may constitute the implementation of a deployable component.
- **Capability**
  - A feature offered by a component implementation.

- **Component**
  - A modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.
- **Domain**
  - A target environment composed of independent nodes and resources.
- **Installation**
  - The act of taking a published software package and bringing it into a repository.
- **Interface**
  - A named set of operations that characterize the behavior of an element.
- **Implementation Artifact**
  - An artifact used or produced as a result of an implementation (usually “executable code”).
- **Launch**
  - The process of instantiating components on nodes in the target environment according to a deployment plan.
- **Metadata**
  - Information that characterizes data.
- **Node**
  - A run-time computational resource which generally has at least memory and often processing capability.
- **Package**
  - An implementation, or set of interchangeable implementations, contained in a set of artifacts and compiled code modules.
- **Repository**
  - A facility for storing metadata, and implementations.
- **Requirement**
  - A feature requested by a component implementation. Monolithic implementation requirements must be satisfied by node resources.
- **Statefull Web Service**
  - A Web Service in which different service instances may store different information and may use this information to affect the results of invoked operations. Invoking the same operation on different instances of a service may produce different results.
- **Stateless Web Service**
  - A service implementation in which successfully invoking an operation of any service instance will produce the same result. This includes service instances which share ‘state’ in a commonly accessible data repository (e.g. a database).

### 2.2.2 LIFECYCLE STAGES

The following three sections introduce each stage and sub-stage of the software service lifecycle. These lifecycle stages consume portions of the generic lifecycle activities common amongst software services first characterized in [11] and later specified by the OMG in [10].

### 2.2.3 DESIGN & DEVELOPMENT

The design & development stage is composed of three sub-stages: *Specification*, *Implementation*, and *Packaging*. As shown in Fig. 2, the completion of these stages yields an installable package that includes the code and metadata describing the software component. The design process begins with a *Specifier* actor defining an interface which describes the behavior of the component. This interface is called a *ComponentInterfaceDescription* and is passed to the actor responsible for implementation.

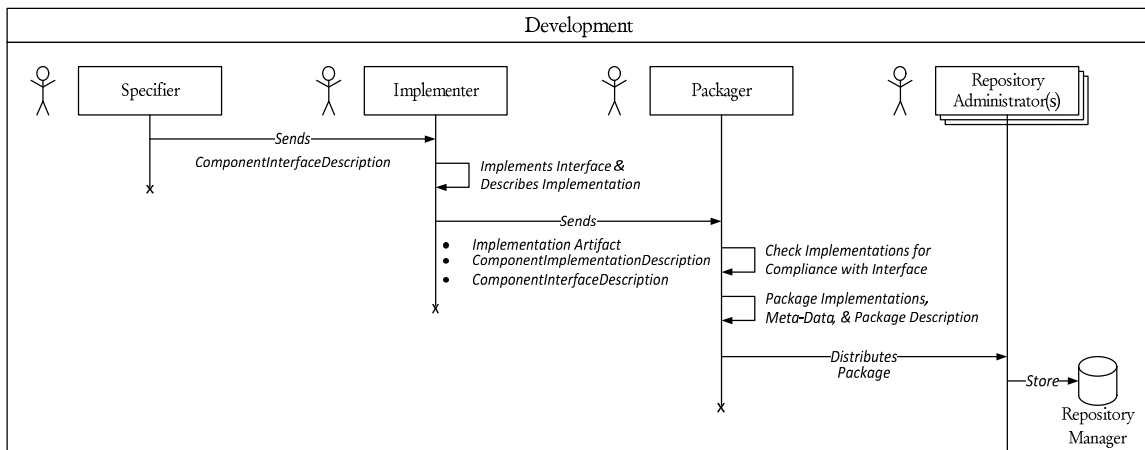


Fig. 2 The actors and actions of the Design & Development lifecycle stage.

The development process involves the *Implementer* actor writing the business logic of the component by creating an implementation artifact which implements the *ComponentInterfaceDescription*. This sub-stage can be completed by a *Developer*, who writes a monolithic implementation, or an *Assembler*, who uses existing components as building blocks. This sub-stage yields an implementation artifact which serves as the code module of the package.

The implementing actor must also describe the implementation with a *ComponentImplementationDescription*. An Assembler creates a *ComponentAssemblyDescription* which describes the component assembly in terms of its sub-components. A Developer creates both a *MonolithicImplementationDescription* describing the component implementation, as well as an *ImplementationArtifactDescription* describing the component's requirements of the target

environment. These descriptions are combined with the original *ComponentInterfaceDescription* and serve as metadata of the package.

Packaging is the final step in the development process and is carried out by the *Packager* actor. The Packager begins by combining the implementations with their descriptor documents, ensuring that the component implementations conform to the component interface, and describing the package as a whole using a *ComponentPackageDescription*. The combined artifacts and compiled code modules comprise the final package which is ready for installation.

#### 2.2.3.1 Installation & Configuration

The *Repository Administrator* actor is responsible for installing component packages – that is, storing them into a repository which is accessible to the entity responsible for deployment. The Repository Administrator is also responsible for setting and updating package configurations. Once a component package is created by the Packager it is distributed to Repository Administrators for installation and configuration.

Repository Administrators access repositories through a *RepositoryManager* interface which provides operations for installing, configuring, retrieving and removing component packages. While installation makes packages available for deployment, it does not involve moving them to the machines on which the components will be run. This task will be performed later during the ‘Preparation’ process of the Deployment lifecycle stage.

#### 2.2.4 DEPLOYMENT

Software deployment may be defined to be the process between the acquisition and execution of software [12]. Once a software component has been developed, packaged and installed, it is ready to be deployed. The deployment process consists of three steps: planning, preparation, and launch. Planning is carried out by the *Planner* actor and involves creating a *DeploymentPlan* dictating how and where software will be deployed. Preparation and launch are carried out by the *Executor* actor and involve executing the *DeploymentPlan*, preparing the target environment for launch, and orchestrating the ‘activation’ of each component in the deployment.

The following sections begin by defining the features and traits of target environments and the actors responsible for their administration. The Deployment sub-stages of planning, preparation, and launch are detailed next, followed by an overview of the techniques used to execute a *DeploymentPlan*.

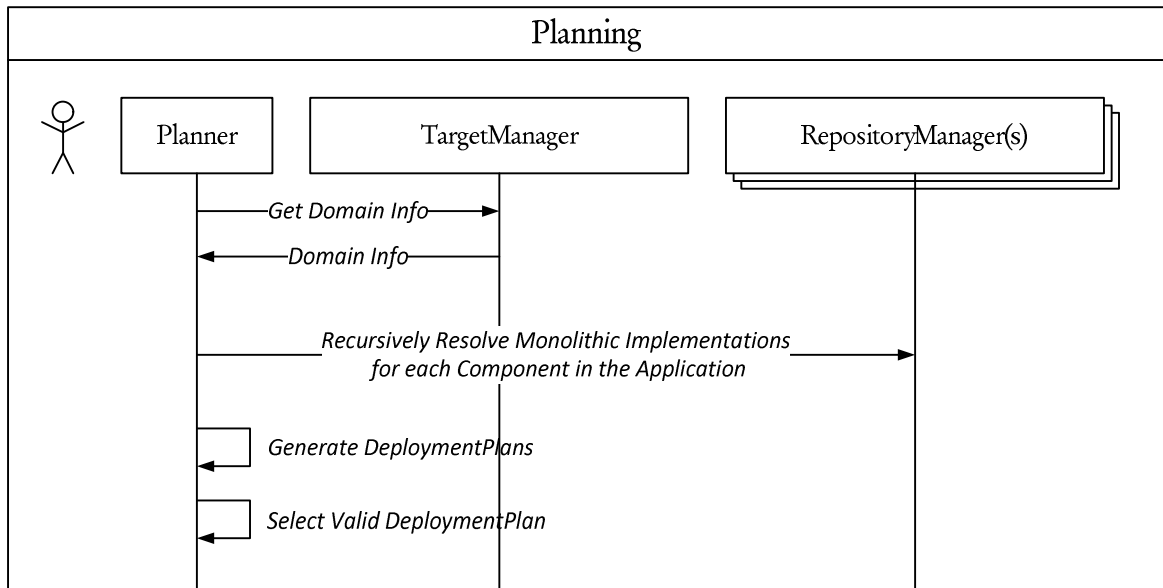
##### 2.2.4.1 Target Environments

Target environments – or ‘Domains’ – consist of a distributed system infrastructure comprised of nodes on which the software will ultimately run [10]. Target environments are administered by a *Domain Administrator* actor who describes the environment in terms of its nodes and shared resources. Each individual node in the domain is defined by its own resources – such as processing power, memory, and operating system – and domains may consist of nodes with varying resources. Nodes are managed by a *NodeManager* which is responsible for instantiating components on the node.

When the *DeploymentPlan* is executed in the following sub-stages, the deploying process does not control the target nodes directly. Instead, it interacts with them through their *NodeManager* interface, allowing for the software supporting component execution on the node to be functionally decoupled from the details of the deployment process as a whole. This separation allows a target environment to support heterogeneous nodes and component implementations without changing the implementation of the deployment system.

#### 2.2.4.2 Planning

The *Planner* actor is responsible for deciding how and where software will be deployed. The decision making process involves matching software requirements with target environment resources and creating a *DeploymentPlan* to guide the ‘Preparation’ sub-stage. A valid *DeploymentPlan* describes a deployment of an application using concrete implementations that match requested selection properties, and an assignment of these implementations to nodes so that node resources match or exceed the requirements of component instances that are deployed on them [10]. The activities of the planning process are shown in Fig. 3.



**Fig. 3** The processes of the Planning sub-stage, including information retrieval, generation of candidate DeploymentPlans, and selection of a valid DeploymentPlan to guide application Preparation and Launch.

The Planner begins the planning process by retrieving information about the target environment from a *TargetManager* controlled by the Domain Administrator. The Planner then contacts one of potentially many *RepositoryManagers* to find a package and *ComponentPackageDescription* for the application to be deployed. If the package contains an implementation which is an assembly then the Planner must retrieve packages for each sub-component in the assembly until it has monolithic implementation artifacts for all components in the application.

The next stage in the planning process involves generating candidate *DeploymentPlans* by matching node resources to implementation requirements, then selecting a plan for use. Deciding which plan to use, and when the choice is made, depends largely on the resource behavior of the target execution environment. In environments with static resource availability, *DeploymentPlans* may be pre-generated, reducing the time needed for deployment. Environments with dynamic resource availability will need to generate and select plans on demand in order to ensure that the *DeploymentPlan* is valid.

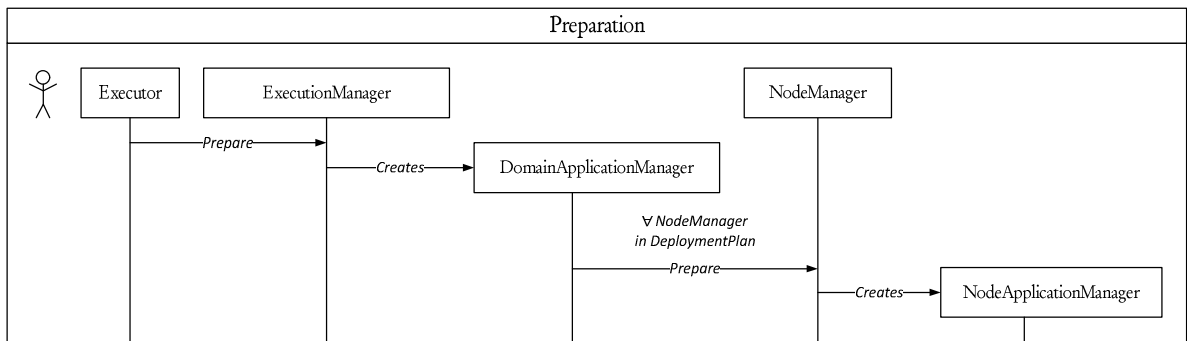
The process of generating and selecting *DeploymentPlans* is implementation-specific and can be one of the most complex lifecycle tasks. Generating all possible configurations for a low-complexity deployment on even a small set of nodes can yield an astronomical number of equally valid *DeploymentPlans*. Further, the process and criteria for comparing and ranking *DeploymentPlans* can be very complex and time-consuming. Choosing the right approach to

planning can have a large impact on the timeliness and effectiveness of the deployment process; the various approaches to planning and their implications for deployment are discussed further in section 2.2.4.5 ‘Deployment Techniques’.

#### 2.2.4.3 Preparation

A valid *DeploymentPlan* is used by the *Executor* actor during the Preparation stage in order to prepare the target environment for software launch. To prepare the deployment, the *Executor* sends a *DeploymentPlan* to the *ExecutionManager* who is responsible for managing the execution of the application into the domain.

The Preparation process, shown in Fig. 4, begins with the creation of *ApplicationManagers* at both the domain and individual node levels which are capable of enacting a particular *DeploymentPlan*, possibly multiple times. The *ExecutionManager* first creates a domain-level manager called a *DomainApplicationManager*. For each node in the *DeploymentPlan* the *DomainApplicationManager* creates a partial *DeploymentPlan* representing the single node’s deployment responsibilities. Each node’s *NodeManager* is sent their partial *DeploymentPlan* and creates a *NodeApplicationManager* capable of enacting it on the node. These domain- and node-level managers will be used later during the ‘Launch’ process to orchestrate and enact the execution of the application.



**Fig. 4 The Executor is responsible for preparing the target environment for application launch.**

As the semantics of the Preparation process are undefined, this phase can be a source of features which differentiate deployment system implementations. While typical Preparation tasks include retrieving component packages from a *RepositoryManager* and moving them to the nodes where they will be executed, the coordination and timing of these and various other tasks can be tailored to exhibit vastly different behavior. Synergies with other Deployment sub-tasks can also be realized during Preparation, affecting exhibited behavior even further. Pre-loading machines with artifacts, for example, may increase launch speed but will also reduce available

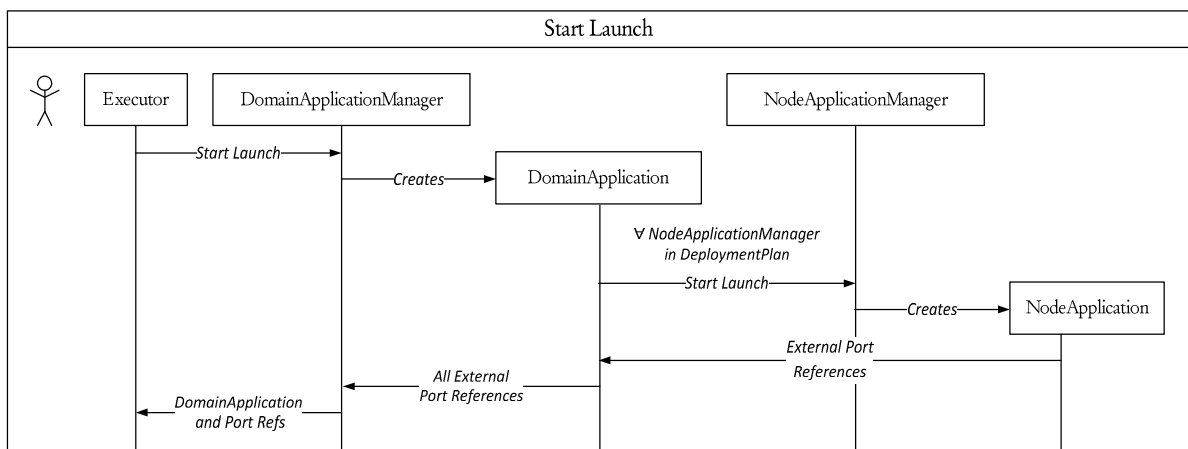
resources. The Preparation phrase could be intelligently coordinated with the Planning phase, however, by using pre-generated DeploymentPlans to *selectively* pre-load artifacts, reducing the original scheme's resource consumption while retaining the gained launch speed.

The Preparation process is finished when the target environment is prepared for launch according to the implementation-specific criteria of the deployment system. The stage is signaled as complete by the ExecutionManager returning the DomainApplicationManager reference to the Executor. Once returned, the Executor can use the launch operations of the DomainApplicationManager to instantiate the application and conclude deployment.

#### 2.2.4.4 Launch

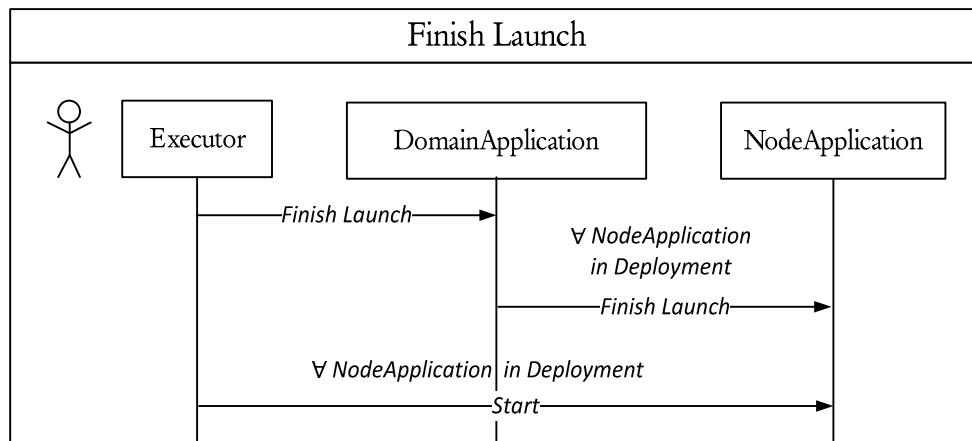
The Executor uses the DomainApplicationManager produced by the Preparation stage to finalize the software deployment process during the Launch lifecycle stage. A successful launch brings the application into an executing state and results in an object that satisfies the ComponentInterfaceDescription described by the Specifier in the Design & Development stage.

The Executor launches an application in two separate steps called 'start launch', shown in Fig. 5, and 'finish launch', shown in Fig. 6. The process begins with the Executor calling a 'start launch' operation on the DomainApplicationManager which creates a *DomainApplication* representing a single instance of the deployed application. This DomainApplication is responsible for enacting the DeploymentPlan by calling the 'start launch' operation of the NodeApplicationManager on each node of the deployment. The 'start launch' operation signals each NodeApplicationManager to instantiate a *NodeApplication* representing a single instance of the node-level partial deployment of the application.



**Fig. 5** The Launch stage is initiated through a series of 'start launch' calls resulting in the creation of domain- and node-level component deployments.

The NodeApplication can carry out component- and node-specific actions such as loading software into memory or opening ports in preparation for use – after which the application is deemed to have been ‘executed’ but not yet ‘started’. Once each NodeApplication is ready to start it returns the DomainApplication a list of the external ports provided by newly instantiated application components on the node. The DomainApplication waits until all nodes are ready to start before returning the NodeApplication port references to the DomainApplicationManager. The DomainApplication reference and the port list are both returned to the Executor who will use them during the ‘finish launch’ step.



**Fig. 6** The Deployment process is completed by calls to ‘finish launch’ on each deployed component; the timing and order of their ‘activation’ is controlled by the Executor issuing explicit commands to ‘start’.

The Executor finishes deployment by calling the DomainApplication’s ‘finish launch’ operation with the list of external ports. The DomainApplication coordinates the dissemination of the port references, calling ‘finish launch’ on each NodeApplication in the deployment and sending them the list of external ports so that they can communicate with other components in the deployment.

Deployment is concluded by the Executor sending each NodeApplication a command to ‘start’. At this point the entire deployed application is deemed to be ‘started’ and the launch stage is complete; the application has been deployed.

#### 2.2.4.5 Deployment Techniques

While a successful deployment always results in the instantiation of one or more objects that satisfies their ComponentInterfaceDescription, there are a great variety of techniques used to carry out the individual deployment tasks. All of the roles in Deployment can be carried out by different entities and through different means, such as manually by a system administrator or automatically by a computer-based deployment system. The ability of an application to adapt to

changes in the target environment – such as rising usage levels or node failure – is largely dictated by the techniques used to develop and deploy the application.

The techniques utilized for deployment can be categorized as manual-, script-, language-, or model-based, and are selected based on system scale, complexity, and the expectation of change. The effectiveness of the chosen approach is a function of its suitability to the deployment environment, the availability of management and monitoring facilities (if any), the skill of the developers, and a willingness to invest time and resources [13].

Using manual deployment, the Planner and Executor roles of the Deployment lifecycle stage are assumed by a person (such as a system administrator) who is responsible for performing all of the tasks by hand. The individual must acquire the software from a repository, resolve all component package references, gather and analyze target environment resources, then manually prepare and execute a DeploymentPlan. This approach is practical for testing individual components – monolithic implementations in particular – but is clearly unsuitable to any but the smallest and least complex deployments.

Executors may introduce automation into the process of executing a DeploymentPlan through the use of shell scripts. By formalizing the manual deployment tasks and chaining them together into a programmatic deployment script, individuals may repeatedly execute a single DeploymentPlan. Writing deployment scripts requires a larger initial investment of time during the development phase, but for low-complexity systems and highly homogeneous target environments it can provide a cheap means of performing large-scale deployments.

Manual- and script-based techniques are useful for the deployment of simple applications either singularly (by hand) or potentially multiple times (with shell scripts). When it is necessary to express highly complex and inter-dependent deployments, however, these approaches become limiting and other techniques must be considered.

In language-based deployment, constructs are provided in the programming languages which can be used to specify complex interdependencies between components in a distributed system. Frameworks such as SmartFrog [14] and eFlow [15] use language-based constructs to define the static and dynamic bindings between application components, which can then be combined with executable code in the same implementation (combining the Developer and Assembler roles).

Using a language-based deployment technique requires the Implementer to learn and understand the complexities of the language, but once implemented the remainder of the application deployment process can be fully automated. A deployment engine can first assume the role of Planner, generating and selecting valid DeploymentPlans for target environments with both static and dynamic resource availability. It can also perform the preparation and launch roles of the Executor, preparing the target environment for execution and coordinating launch according to the plan. Further, by analyzing the inter-component dependencies specified by the implementation artifact (before a specific DeploymentPlan is developed), change-management systems such as [15] can detect component failure and automatically reconfigure the deployed system – an activity detailed further in section 2.2.5 ‘Management’.

Model-based techniques take a higher-level approach by describing the business functionality and behavior of an application separately from the technology-specific code that implements it [16]. Applications are described as high-level models representing valid deployment configurations together with a set of transformation rules. By statically specifying all of the valid deployment configurations ahead of time, the use of models reduces the effort required during the Planning stage of Deployment. The generation of DeploymentPlans only involves matching requirements to resources, after which the Planner can decide which configuration to chose. Model-based approaches from IBM [17] and Radia[18] (recently absorbed as part of HP OpenView Application Manager [19]) allow developers to use desired-state modeling to define the ideal state of the system with regard to some set of constraints. These constraints guide the selection of an optimal DeploymentPlan for both the initial system deployment and any subsequent reconfigurations due to changes in the target environment.

Application deployment using models requires a comparatively large application of development effort before the actual business logic of the application can be written, and sophisticated model execution engines are necessary in order to leverage this effort. However, once the system is modeled and the required components implemented, all monitoring, management and change-enactment can be taken care of autonomically and intervention by the Executor or DomainAdministrator should be limited unless the system goals or requirements change.

#### *2.2.4.5.1 Comparing Deployment Approaches*

Each of the preceding deployment approaches has its strengths and weaknesses and no single approach is appropriate for all scenarios. The choice of approach depends primarily on the target deployment environment and the amount of time invested during Development in order to

save time adapting to change after Deployment. The burden of selecting an approach rests heavily on the Specifier of the application, while the repercussions for selecting the wrong approach are felt by the Executor, the DomainAdministrator, and the users of the application. Foreknowledge of the scale, complexity, and anticipated frequency of change in the system are required in order to make an informed decision. A quantitative evaluation of the preceding deployment approaches is presented in [20] and a qualitative comparison is presented in [11].

### 2.2.5 MANAGEMENT

The W3C defines management as “a set of capabilities for discovering the existence, availability, health, and usage — as well as the control and configuration — of manageable elements, where these elements are defined as services, descriptions, agents of the service architecture, and roles undertaken in the architecture” [21]. This definition identifies the two requirements of management: information about a system, and the tools to enact change in the system. A functioning management system can gather information about manageable elements, analyze and derive meaning from this information, and change the system to better address a set of pre-defined management goals.

Although one may think of management as only happening at one stage in the lifecycle – the Management stage – it is in fact an important concept at each stage [8]. As identified in [20] the choice of application deployment technique is directly linked to a.) the effort required before deployment, and b.) the ease of managing system changes after deployment. A model-based approach, for example, requires a significant application of developer effort before the application can be made available. It requires a sophisticated Planner implementation and a thoughtful specification of the acceptable range of operational parameters (such as system response time). This additional up-front effort and complexity, however, can greatly reduce the amount of time spent reacting to change: when the operational parameters leave the acceptable bounds, the current state of the system can be used by the same Planner actor to create a DeploymentPlan suitable for the current conditions of the target environment, possibly utilizing existing deployed components. This plan can then be executed (by the Executor) after which monitoring of the operational parameters can recommence.

#### 2.2.5.1 Autonomic Management

A common scenario for the management of a distributed system involves one or more system administrators actively monitoring the pre-defined measure of ‘health’ in a deployed application, identifying when something has gone wrong, devising a plan to bring the system into

a healthy state, and executing that plan. This process can be termed ‘manual troubleshooting’ and is directly linked to the ‘manual deployment’ tasks identified in the previous section.

In a closed-world view it is conceivable that, up to a certain size and complexity, a complex software system may be effectively managed by a set of humans. Beyond a point this is unrealistic, and it becomes necessary to employ a system which can make decisions in lieu of human guidance. This type of management is called ‘autonomic management’ and is part of a broader field called ‘autonomic computing’ [22] [23].

The term autonomic computing applies to a system that can monitor itself and adjust to changing demands [24]. There are four distinct characteristics of an autonomic computing system:

- Self-configuring
- Self-healing
- Self-optimizing
- Self-protecting

The goal of autonomic computing is the design and development of systems which are able to run themselves with little to no human intervention. In the context of autonomic management, these goals are achieved through a formalization of the above ‘manual troubleshooting’ steps into what is termed the ‘autonomic control loop’. Fig. 7 shows the four stages in the autonomic control loop: monitor, analyze, plan, and execute.

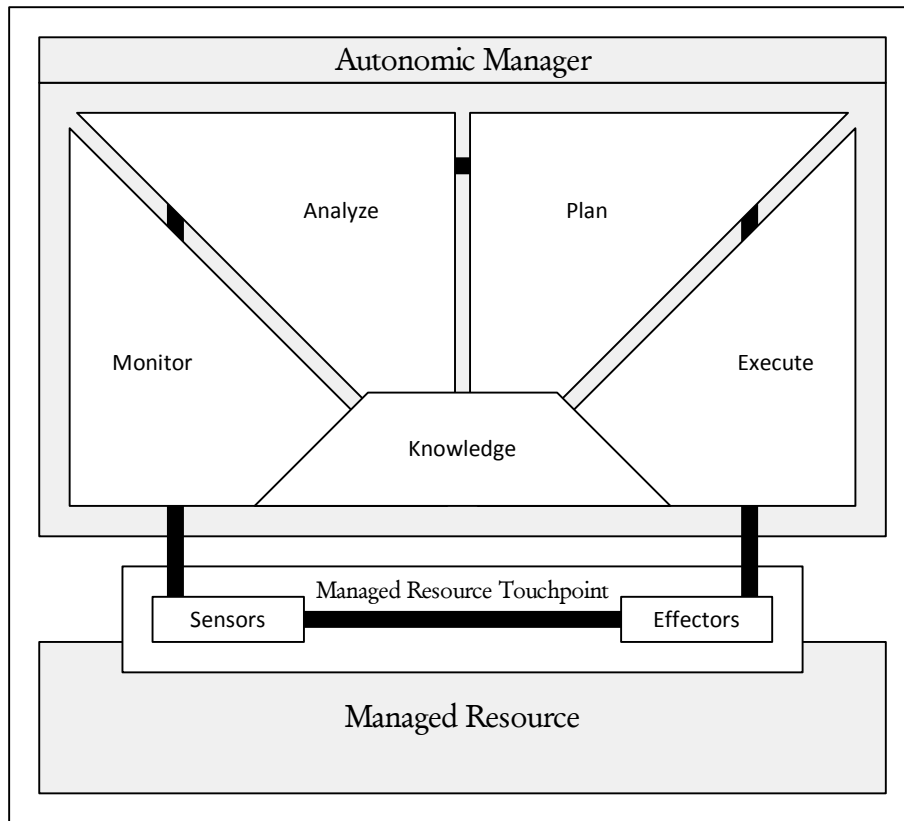


Fig. 7 The Autonomic Control Loop [25].

The monitoring and execution stages of the autonomic control loop require suitable sensors and effectors for each resource being managed – in other words, there must be ways to derive information about and exert control over manageable resources, where a manageable resource is a target environment, application, application component, client using the application, or an actor in the architecture [21]. In a distributed management architecture, the sensors and effectors linked to the manageable resource through a ‘managed resource touchpoint’, as shown in Fig. 7, may also be linked to a ‘probe’ which remotely exposes their functionality for external examination and invocation.

The two most common distributed monitoring models are called the ‘proxy’ model and the ‘agent’ model. In the proxy model, shown in Fig. 8, requests to an application are routed through an intermediary who records usage and performance statistics using a ‘collector’. This intermediary can be linked to a single application (‘Proxy 1’ in Fig. 8) or can serve as the proxy for a number of deployed applications (‘Proxy 2’ in Fig. 8). From an architectural perspective, the proxy model is vulnerable to a single point of failure if there is no infrastructure in place to cluster the proxies [8]. In the agent model, shown in Fig. 9, statistics are collected by ‘collector’ software installed at each node in the target environment and information is recorded for all components deployed on the individual node. The agent model removes the central point of failure inherent

to the proxy model, but running extra software on the same nodes as deployed application components may have an unpredictable impact on their performance.

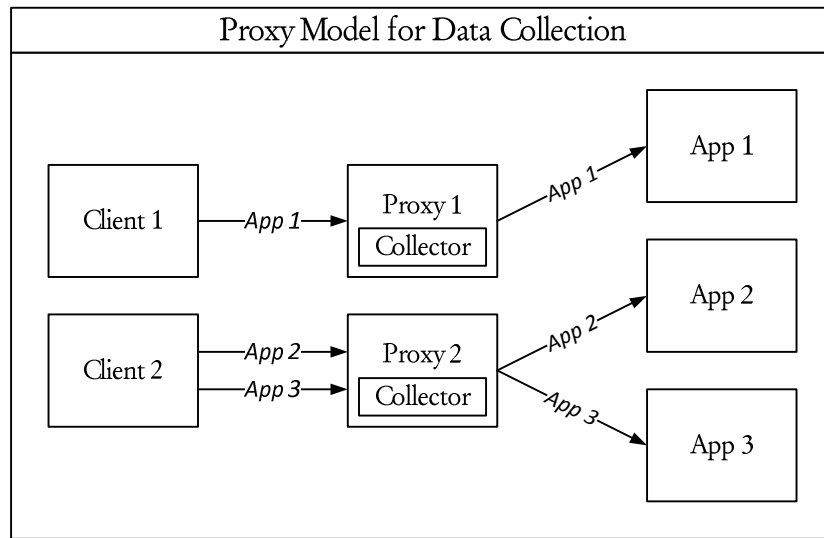


Fig. 8 The proxy model for collecting data about managed elements.

After information is collected it must be transferred to a manager for analysis. This transfer can be initiated by the collector (called reporting or 'push') or by a manager (called polling or 'pull'). Polling requires a managing entity to periodically interact with the collector via an externally exposed interface on the node (i.e. probes). Reporting requires the collector to periodically contact one or more managing entities at a well-known location. The choice of approach has implications for scalability, reliability, and bandwidth consumption. It is argued in [26] that the reporting approach is better suited to large distributed systems as it conserves bandwidth and CPU time on the management systems.

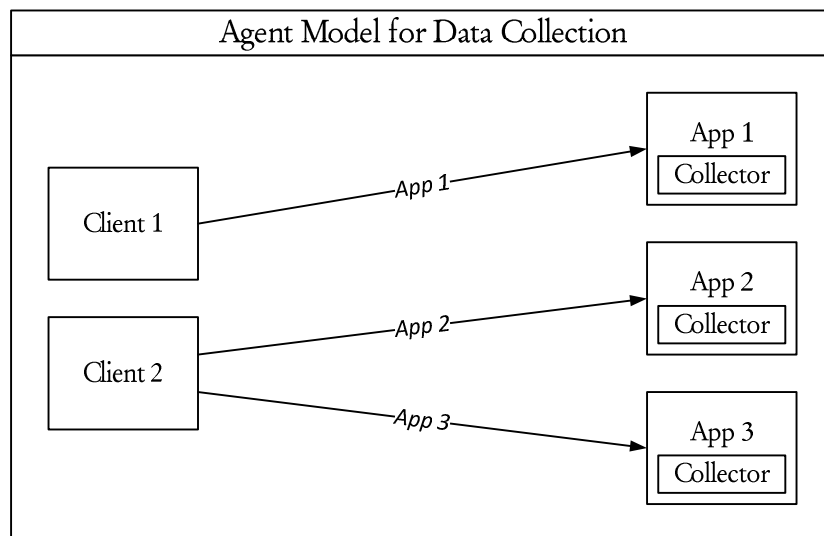


Fig. 9 The agent model for collecting data about managed elements.

Once autonomic management is introduced into a system it can be difficult to identify where the management chain ends as the introduction of each new management entity also introduces a new entity to be managed. Further, having both a deployed application and a peered set of deployed managers can be a waste of resources if the application is not being used, leading to situations where the cost of management overshadows the cost of running an application. The careful design of management architectures and the interactions between managers and managed resources is vital if system management is to be effective.

## 2.3 WEB SERVICES MODEL

Section 2.3.1 begins by describing the Web Services reference architecture and introduces the primary entities involved in all Web Service interactions. It continues with an overview of the key Web Services standards of SOAP, WSDL, and UDDI, and concludes with a discussion of efforts to create standardized Web Service interaction guidelines. Section 2.3.2 describes the Web Service actors of Service Provider, Discovery Service, and Service Consumer. It details the lifecycle activities consumed by each actor, their roles and responsibilities, and examples of technologies involved at each stage of the Web Service lifecycle. Section 2.3.3 concludes the discussion of the traditional Web Services model with an introduction to implementing Web Services using existing Web Services as building-blocks – describing, packaging and deploying them like components in an assembly in a process termed ‘Web Service Composition’.

### 2.3.1 WEB SERVICES ARCHITECTURE

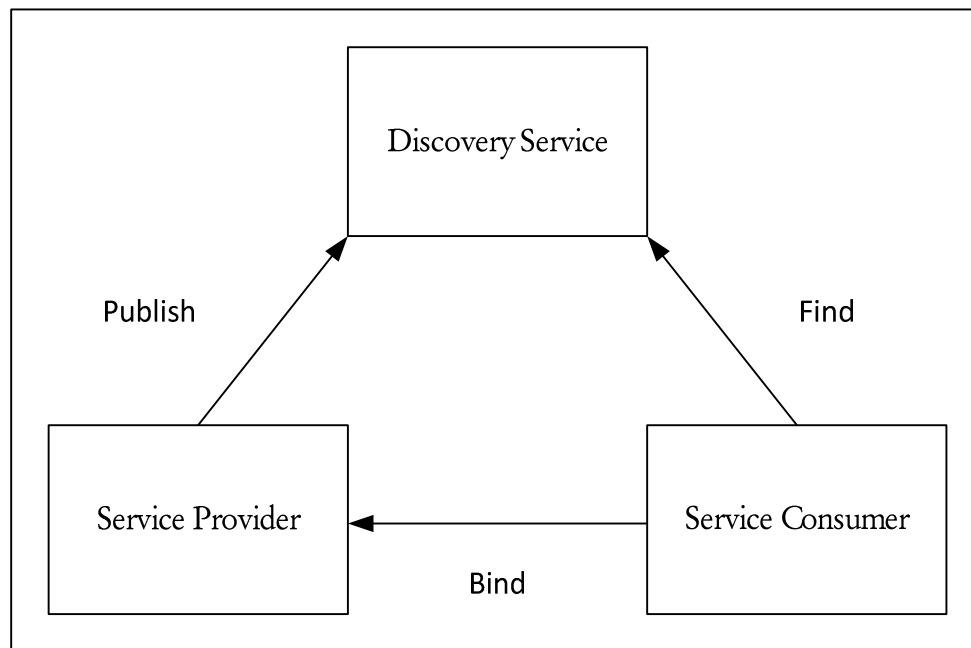
The standards introduced by Web Services provide the means to evolve distributed systems from tightly-coupled distributed applications into loosely-coupled systems of services. This evolution has led to the development of Service Oriented Architectures (SOAs), of which Services are the primary component. The guiding characteristics of SOAs are the interoperation between loosely coupled autonomous components, the promotion of code reuse at a macro (service) level, and architectural composability.

The W3C Web Services Architecture Working Group (WS-Arch) [4] defines a reference architecture for Web Services in [27] which provides a framework for interactions between participants in a service-oriented distributed system comprised of Web Services. The reference architecture describes interactions between loosely-coupled, platform-independent systems based on Web Service standards.

#### 2.3.1.1 Interaction Model

The WS-Arch specification introduces the three primary entities in Web Service interactions: the ‘Provider Entity’ (Service Provider), the ‘Requestor Entity’ (Service Consumer), and the ‘Discovery Service’. The Service Provider is responsible for deploying a Web Service and making it available to Service Consumers. Discovery Services provide a searchable directory of Web Service endpoints. Service Providers use a Discovery Service to publish information about active Web Service endpoints. Service Consumers can use a Discovery Service at design time to discover and write programs against Web Service endpoints. Service Consumers can also implement code to use a Discovery Service at run-time to dynamically locate endpoints (discussed later in section

2.4.1). The diagram in Fig. 10 below demonstrates the interaction between service providers, service consumers, and a discovery service.



**Fig. 10 The Ws-Arch Web Service interaction model.**

Individual application components exposed as Web Services are called ‘provider agents’ – as distinct from the ‘provider entity’ responsible for providing them (such as a person or organization). Similarly, ‘requestor entities’ use ‘requestor agents’ (client-side applications) to exchange messages with a provider entity’s provider agents [27]. The terms Service Provider and Service Consumer are often used ambiguously to refer to both entities and agents, and it is important to clarify this distinction.

The remainder of this thesis will use the term Service Provider to refer to the provider entity; the executable code implementing the provider agent will be called ‘service code’; deployed service code exposed as a Web Service will be called an ‘endpoint’. Because the requestor entity uses a requestor agent explicitly (rather than just being responsible for it) the term ‘Service Consumer’ will refer to both the requestor entity and the requestor agent together except where explicitly noted.

### 2.3.1.2 Web Services Description & Communication Standards

The following sections introduce the three core standards of the Web Service architecture which provide the means to describe (WSDL), advertise & discover (UDDI), and communicate (SOAP) with Web Services.

#### 2.3.1.2.1 SOAP

Web Services interact through the exchange of messages using SOAP<sup>1</sup>, a protocol designed for exchanging structured information in a decentralized, distributed environment [5]. SOAP defines a communication protocol for Web Services which is independent of programming languages and platforms and is designed to be used over a broad range of transport protocols. It is designed to be both simple and extensible. Simplicity is addressed in the messaging framework by specifying only the message constructs, without mention of common distributed system features such as reliability, security, or routing path. Extensibility is addressed by using XML to describe messages, enabling further specifications to define features which can be added to messages while preserving their interoperability.

SOAP messages are comprised of an outer XML element called `envelope` which defines the namespace(s) for the message, an optional `header` element which includes any relevant extensions to the messaging framework, and a required `body` element. The `body` element provides a 'mechanism for transmitting information to an ultimate SOAP receiver'[5] but is left intentionally unspecified beyond this role, with neither a defined structure or interpretation, nor a means to specify any processing to be done.

SOAP Messages may be exchanged over a variety of underlying protocols conforming to the SOAP Protocol Binding Framework [5]. SOAP over HTTP is used as the reference protocol binding definition and is universally adopted as the primary transport protocol for Web Services [7].

#### 2.3.1.2.2 Web Service Description Language (WSDL)

The Web Service Description Language (WSDL) is the industry standard language for describing Web Services in XML [28]. The `ComponentInterfaceDescription` of a service is realized as a WSDL document, which also includes all of the information required to locate, bind to, and interact with a Web Service endpoint.

WSDL documents are constructed from XML document elements that describe Web Service endpoints in terms of their operations, the parameters and return values of each operation (including type definitions), and the protocol and data bindings used for communication. A set of operations is called a '*Port Type*' in WSDL and the protocol and data bindings are together called a '*Binding*'. An active Web Service endpoint is called a '*Port*' and is a combination of a binding and a static physical network address where it can be contacted (realized as a URL).

---

<sup>1</sup> Commonly thought to mean 'Simple Object Access Protocol', SOAP is simply a name, not an acronym [5].

Programmers write requestor agent software against an endpoint's WSDL document. While possible to do by hand, programmers commonly use automatic code-generation tools to generate proxy or 'stub' code in their language of choice. The broad uptake and acceptance of Web Services has led to the development of a variety of such tools for generating code in a multitude of programming languages. These can be stand-alone tools, such as the popular 'WSDL to Java' [29] for Java, as well as IDE-based, such as in VisualStudio [30] and Eclipse [31], which provide a pluggable framework for generators producing code in Perl, C++, C#, PHP, and others.

#### *2.3.1.2.3 Universal Description Discovery and Integration*

Universal Description Discovery and Integration (UDDI) [32] is a Web Service standard for the Discovery Service. UDDI is developed by the OASIS consortium [33] and is based on technology originally donated by IBM, Ariva, and Microsoft. UDDI realizes the Discovery Service functionality as a searchable directory providing the means to describe, advertise, and search for information about Web Services. The standard specifies a service description format and a set of APIs through which entities can interact with the directory.

Service Providers interact with the UDDI directory through the 'Publishers' API which provides operations for adding, modifying, and deleting information about the Web Services they provide. This information includes details about the provider entity, the interfaces implemented by the Web Service (called tModels), and any other proprietary information deemed relevant to a Service Consumer (such as an endpoint location, although this is not a requirement). The Service Provider completes their publication task by storing this information in a UDDI directory under one of the three following categories:

- **WHITE PAGES** list organizations and the services they provide.
- **YELLOW PAGES** classify organizations and Web Services into standard or user-defined groups (such as 'pizza delivery' or 'payroll services').
- **GREEN PAGES** augment the standard Web Service information with pointers to service-description documents detailing how the Web Service can be invoked.

Service Consumers interact with the directory through the 'Inquiry' API which provides operations to search the various directories for Web Service information published by Service Providers. Programmers may use a UDDI directory at design time to find appropriate Web Services for their application. The resultant requestor agent software may also use the directories for run-time dynamic binding.

Reliability and performance have been widely identified as problems inherent in the UDDI registry design (e.g. [34] [35] [36] [37]). In terms of reliability and performance, the two Service Consumer usage scenarios above (static and dynamic binding) present two separate sets of requirements for UDDI registry implementations [38]: while design-time lookup has limited requirements for both reliability and performance, optimizing both of these characteristics is crucial to effective run-time binding [39] [40]. Most work addressing the scalability and reliability of UDDI registries has focused on the use of peer-to-peer (P2P) systems [41] [42] [43] [44]. Although UDDI does not require published information to be valid, P2P technologies have also been applied in [45] and [46] in an effort to maintain accurate, up-to-date, and reliably stored information about Web Services.

A full taxonomy of approaches to improving the reliability and performance of the Discovery Service (including an evaluation of UDDI's competing standard called ebXML [47]) is presented in [48]. Addressing the clear drawbacks of using UDDI registries, novel alternative approaches to providing the Discovery Service are presented in section 2.4.2 'By Discovery Services' and in section 2.5 'An Alternative Approach: Next-generation Architectures for Web Services on the Internet'.

#### 2.3.1.3 WS-I Interaction Profiles

The standards introduced by Web Services provide a flexible and extensible framework for service interactions over the Web. This freedom is necessary to ensure that Web Services can adapt to and absorb future technologies, and thus the specifications lack any concrete requirements or guidelines as to how Web Services must communicate and behave. Products for creating, deploying and interacting with Web Services may utilize an infinite number of configuration options while still conforming to the Web Services specifications. Using the term 'Web Service' to describe two different technologies which are not interoperable (due to differing version numbers, transport protocols, character sets, etc.) makes it difficult for users and developers to piece together a Web Service application which is compatible with other services and products.

The Web Services Interoperability Organization (WS-I) [49] provides guidelines for developing and identifying Web Services which are interoperable. It provides a set of 'profiles' dictating the Web Service standards, the specific revision numbers of those standards, and the communication patterns supported by profile-compliant clients and services.

The first WS-I interoperability profile, called the WS-I Basic Profile [50], requires conformance with WSDL 1.1 [6], UDDI 2.0 [51], and SOAP 1.1 [52]. The profile has been widely adopted and has become the first ISO standard for Web Services interoperability (ISO/IEC 29361:2008) [53].

### **2.3.2 WEB SERVICE ACTORS: ROLES & RESPONSIBILITIES**

The follow sections detail the roles and responsibilities of the three actors in the Web Service interaction model: Service Provider, Discovery Service, and Service Consumer. Each section outlines the generic tasks undertaken by each together with examples of supporting technologies.

#### **2.3.2.1 The Service Provider**

A Service Provider is a person or organization that is offering a Web Service [27]. Service Providers are responsible for deploying service code as Web Service endpoints which are capable of performing the operations associated with a service. Service Providers are responsible for administering the target environment and consume all of the responsibilities of the Planner and Executor deployment roles. Service Providers acquire service code, create a deployment plan, prepare the target environment, and order the execution of service code. They are responsible for describing the resultant endpoint in a WSDL document and may publish its existence using a Discovery Service.

Service Providers instantiate Web Service endpoints on target environment nodes by executing service code in environments commonly referred to as ‘containers’. Containers are software applications which provide access to the business logic implemented by the service code [54]. Web Service containers accept and translate platform-independent SOAP (from the Service Consumer) into the required language-specific data types before the request is processed by the business logic of the requested service. Any return results are translated back into SOAP-format and returned to the Service Consumer.

Web Service containers exist for a wide variety of programming languages. Apache Axis [55] for Java is the most popular stand-alone container, however containers more often represent the ‘application layer’ of a larger ‘application server’, as shown in Fig. 11. Application servers provide a Web-accessible presentation layer which coordinates access to the application layer. The application layer is responsible for executing service code and providing that code with access to a resource management layer through standard APIs (such as JDBC [56] and ODBC [57]). Software vendors use application servers to provide the presentation layer and execution environment for

a wide array of applications which use the document as the basic unit of transfer [58]. For example, commercial application servers such as Sun One [59], IBM Web Sphere [60], Microsoft .NET [61] and BEA WebLogic [62] are all capable of serving both static and dynamic content to web browsers, integrating with SMTP servers for delivering email over the Web, and all support the deployment of Web Services.

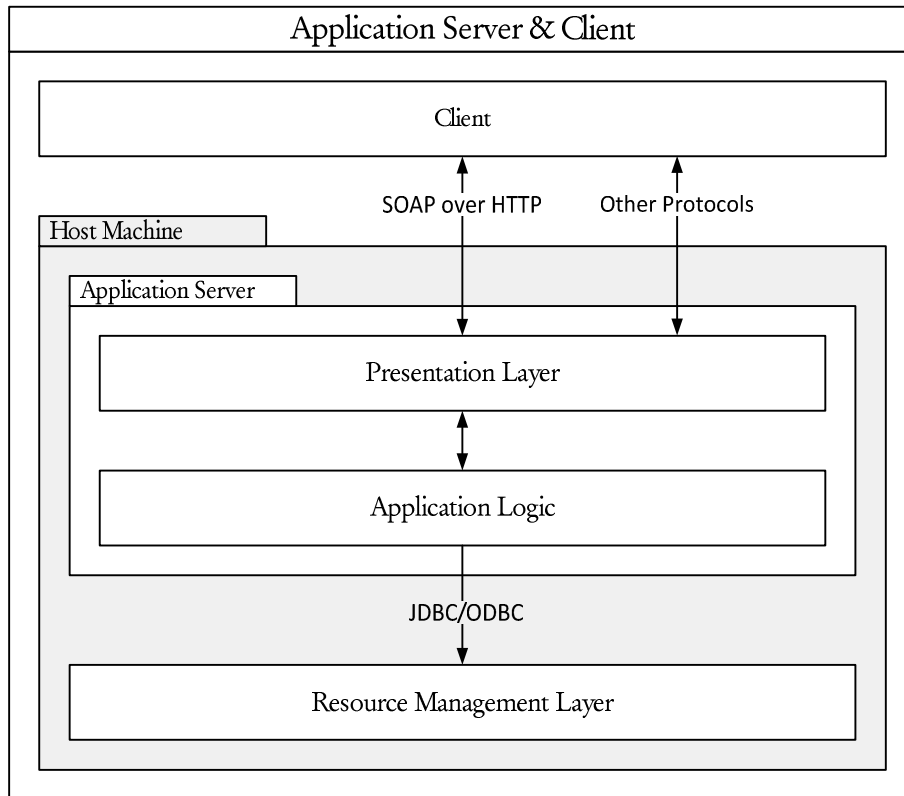


Fig. 11 The architecture of a typical application server with resource management layer

#### 2.3.2.2 The Discovery Service

The Discovery Service has a role to play in every stage of the Web Service lifecycle. It is used by the Assembler during Design & Development in order to locate information about services which can be used together to provide new composite services. The Discovery Service can also take on part of the role of TargetManager of a Domain, providing information about previously-deployed services to the Planner during Deployment. If a management infrastructure is in place, the Discovery Service plays an ongoing role in the management control-loop. It provides a source of information for the Planner and, if implemented, may have a managing entity to remove its failed entries and replace them with valid ones.

The Discovery Service is similarly crucial for requestor agent software which dynamically resolves endpoints at run-time: in order for the application to run, the Discovery Service *must* be available and, ideally, populated with correct and up-to-date information. The practicalities of

run-time lookup are discussed further in section 2.4.1 ‘Managing Change: By Service Consumers’ while approaches to improving the reliability, accuracy and performance of the Discovery Service are presented in section 2.4.2 ‘Managing Change: By Discovery Services’.

#### 2.3.2.3 The Service Consumer

Programmers of requestor agent software use a Discovery Service at design time to find information about Web Services and to retrieve WSDL documents. Service interaction software is written using the formats, protocol, and static physical endpoint specified in the Web Service’s WSDL description. More sophisticated software may contain only the interfaces of the required service, using UDDI to dynamically resolve endpoints at run-time. When executed by Service Consumers these software applications use the URI encapsulated by a Web Service’s WSDL description to contact the application server at the host address and initiate the communication protocol. Encoding the message in document/literal format, the call name, type definitions, and serialized, platform-neutral call parameters are enclosed within a SOAP-formatted XML document and sent over HTTP to the application server. Any return results are received as an HTTP response, parsed, and the return values deserialized back into language-specific objects.

The work presented in this thesis is largely undertaken for the benefit of Service Consumers: Reliable Discovery Services are important for Service Consumers and reliable Web Service endpoints are vital. Section 2.4 ‘Managing Change’ identifies the potential problems in Web Service interactions from each actor’s point of view and presents the range of approaches applied to make Web Services more reliable, high-performance, and fault tolerant.

#### 2.3.3 *WEB SERVICE COMPOSITION*

In order to implement the interface specified in its WSDL document, a Web Service’s business logic may include the invocation of operations offered by other Web Services. A Web Service implemented by combining the functionality provided by other Web Services is called a ‘composite Web Service’ and the process of developing composite Web Services is called ‘Web Service composition’ [7].

Composite Web Services can be written as assemblies or as monolithic implementations. Monolithic composite provider agent implementations directly represent requestor agent software with the exception that the provider agent implementations expose an external interface. The resultant code still suffers from all of the difficulties of writing and maintaining requestor agent software using low-level techniques (converting between XML and scalar

variables, locating and accessing Discovery Services, constructing SOAP messages, managing failure, etc.).

Languages for Web Service composition such as XL [63], WSFL [64], and BPML [65] have been developed in an effort to remove low-level technical details from the higher-level process of composition, easing the task of developing and maintaining composite Web Services. These language specifications share many similarities [7] which have converged into an industry standard language for Web Service composition: the Business Process Execution Language for Web Services (BPEL4WS or, more commonly, BPEL) [66].

BPEL documents define and coordinate Web Service invocations using a set of XML constructs called ‘activities’. Activities in BPEL can be either ‘structured’ (used for orchestration) or ‘basic’ (used for invocation). Basic activities include ‘*receive*’, for receiving a message from a client, ‘*reply*’, for replying to a client message, ‘*assign*’, for assigning data to variables, and ‘*wait*’, for blocking the process for a set period of time. Basic activities can be grouped and executed within structured activities which define their order of execution. Conditional branch logic is provided by a ‘*switch*’ activity while the ‘*flow*’ activity is used for parallel execution of basic activities (or groups of them). A ‘*while*’ activity is provided for looping and ‘*pick*’ can be used for event handling.

BPEL documents are executed within a run-time environment typically referred to as a ‘*composition engine*’. Composition engines receive and reply to invocation requests just like any other Web Service container. On receipt of an invocation request the composition engine creates a ‘composition instance’ of the composite Web Service which is responsible for executing the activities and calling any necessary external Web Services.

Composite Web Services are advertised and invoked just like any other Web Service: clients interact with composition engines in the same way that they interact with Web Service containers and application servers, using the same messages and messaging protocols as defined in the WSDL. The same service may be provided as a monolithic implementation and/or as a composite [assembly] implementation and, while their execution environments may differ, both are invoked identically. This basic abstraction is an important feature of Web Service composition as it enables services to be iteratively composed – creating composite Web Services out of other composite Web Services to provide increasingly complex and feature-rich services.

## 2.4 MANAGING CHANGE

In contrast to traditional client-server applications, clients in distributed service-oriented environments may communicate with multiple autonomous service endpoints on servers all across the Internet. These Web Service endpoints can become unreachable while their new replacements remain unlisted due to lax requirements on Discovery Services which also may be unreachable. Varying levels of demand for Web Services introduces pressure on Service Providers to implement policies to guarantee service reliability while balancing the provisioning of scarce resources. The following sections detail the sources and solutions to managing change from the perspective of each participant in the Web Service lifecycle.

### 2.4.1 *BY SERVICE CONSUMERS*

Web Service endpoint descriptions encapsulate a static endpoint which, for any number of reasons, may be temporarily (or permanently) unavailable. In order to improve the probability of successful service invocation, various techniques have been developed which aim to introduce dynamic endpoint discovery, binding, and failure recovery into the execution of requestor agent applications. The ability of a Service Consumer to adapt to changes in Web Service availability depends entirely on how and when binding to these endpoints takes place.

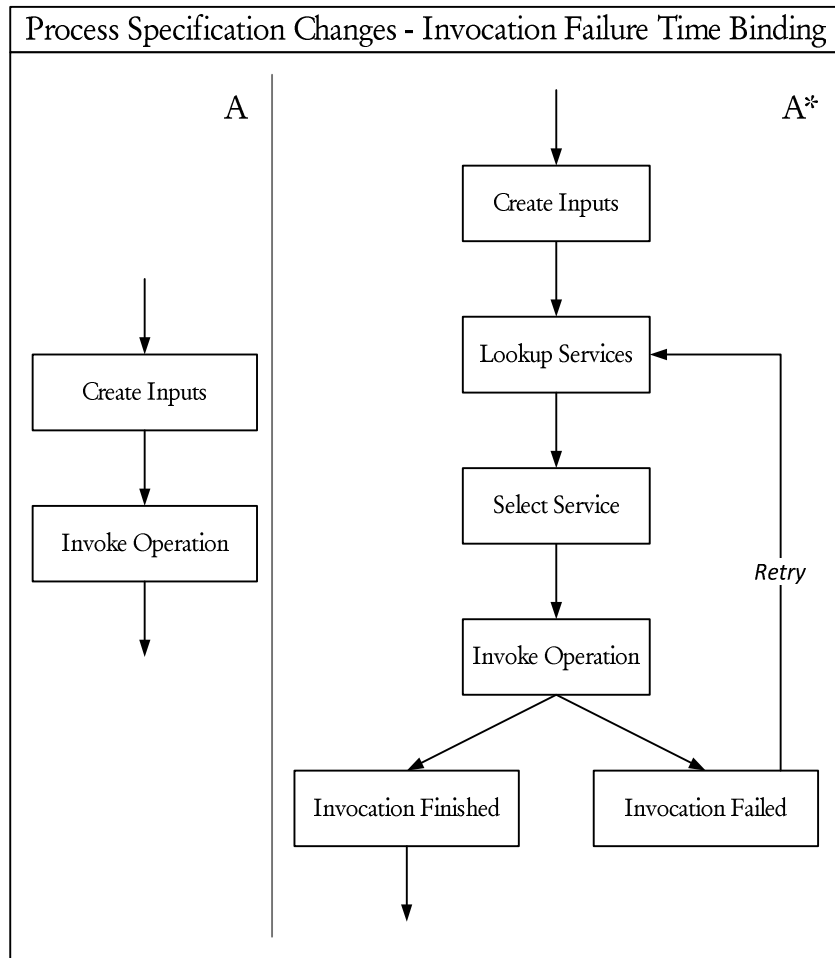
As mentioned previously in section 2.3.3 ‘Web Service Composition’, client- and server-side requestor agents are identical with respect to binding requirements and both experience the same problems when endpoints become unreachable. With the exception of informal guidance, such as that for developers of Microsoft .Net requestor agents in [67] and [68], there is little work addressing binding techniques in ‘client-side’ requestor agent software. Because a more coherent body of work exists for binding in ‘server-side’ requestor agents, this discussion will be guided by work addressing the ‘how’ and ‘when’ of service binding in Web Service compositions, using the notation and terminology introduced in [69].

A binding of a service into a composition can be defined as the reference used to choose the service to be invoked as a part of the composition [69]. Web Service requestor agent software is written against WSDL documents containing an abstract interface description (`portType`) together with a fixed endpoint (`port`). Developers can bind to a single specific `port` at design time (static binding) or to a `portType` which can be resolved to a `port` at various points in the future (dynamic binding). Although static binding is the dominant approach used in practice [7] [48] it results in code with tight-couplings between service providers and service requestors and is thus contrary to the central principles of service-orientation.

Dynamic binding is typically performed either at *startup time*, just prior to executing a composition, at *invocation time*, just prior to invoking a service operation, or at *failed invocation time*, in the case of a failed operation or unavailable endpoint. Choosing when to resolve an abstract reference into a concrete endpoint directly affects the likelihood of successful execution of a workflow and can have implications for application performance, particularly when using UDDI for service discovery [34] [36] [37]. In environments where Web Service endpoints are not very reliable, or where their location changes frequently, binding later (i.e. at invocation time) may be more desirable than in reliable environments with fixed endpoint locations. When dynamic binding is significantly expensive, however, binding earlier (i.e. at startup time) can help mitigate the risk of poor execution performance due to a slow Discovery Service [38]. Analyzing the impact of UDDI lookups on service performance, Blake et. al 2007 [38] conclude that when the percentage of services requiring re-resolution was less than 59%, *failed-invocation time* dynamic binding was most efficient when compared with dynamically binding at *startup time*.

Dynamic binding can be incorporated into requestor agent applications by internally modifying the process specification ‘in-band’, as in Fig. 12, or externally to the process using a proxy as in Fig. 13. Client-side requestor agents typically incorporate failed invocation time dynamic binding in-band, explicitly coding around every service invocation that is intended to be performed in a more robust way [70]. The primary drawback of this approach is the responsibility laid upon Service Consumers to take *all* corrective actions, requiring considerable client-side code that has fixed service-selection logic [71].

The time and effort required to introduce failure recovery techniques directly into executable code can be reduced through the use of extensions to the standard BPEL language. Developers using Self-healing BPEL (SH-BPEL) [72] include annotations in their code which they use to define recovery handlers that will be executed in the event of endpoint errors. SH-BPEL code is pre-processed prior to execution in order to replace annotations with standard BPEL constructs and yields code that can be executed in standard BPEL engines. While this code could conceivably be directly specified by the designer, the required effort to program the techniques by hand would be considerable [72]. The work described in [73] takes a similar approach but focuses on the application of their endpoint-selection algorithms, using in-band replacement only as a means to apply them at run-time.



**Fig. 12** Dynamically binding ‘in-band’ alters the structure of process specification ‘A’ by surrounding the invocation operation with lookup, binding, and failure-detection operations (‘A\*’).

While frequently used by Service Providers for endpoint brokering, the proxy model has a number of benefits when used within the Service Consumer-controlled infrastructure. SH-BPEL, for example, provides a BPEL execution engine plug-in which serves as a proxy in order to introduce dynamic binding at invocation failure time. A similar approach is taken by WS Binder [74] which pre-processes code, statically binding to infrastructure-provided proxies which are created at invocation time for each individual service in the composition. While these approaches do use a proxy they do not meet a central goal of the proxy approach which is to transparently introduce desirable properties – such as dynamic binding and failure recovery, as in Fig. 13 – into Web Service interactions while imposing limited or no special requirements on developers.

Comparatively transparent extensions to BPEL execution engines are introduced in MASC [71] and Robust Execution Layer (REL) [70] which intercept service invocation requests and apply user-defined policies to the tasks of endpoint selection and failure management. MASC policies are specified in external ‘WS-Policy4MASC’ documents which are evaluated at startup time,

invocation time and, if a language extension is used, at failed invocation time. REL takes a unique, ‘hands-off’ approach to dynamic binding and does not require any code changes. BPEL documents are written using static bindings to existing endpoints and executed in a BPEL engine with the REL proxy plugin. Service invocation requests and responses are routed and monitored through the plug-in which automatically detects service invocation failures and reacts by locating and redirecting the request to an alternative endpoint. Introduced in the next section, REL relies on a reliable, scalable Discovery Service which includes extra metadata to identify replacement Web Service endpoints which offer equivalent functionality.

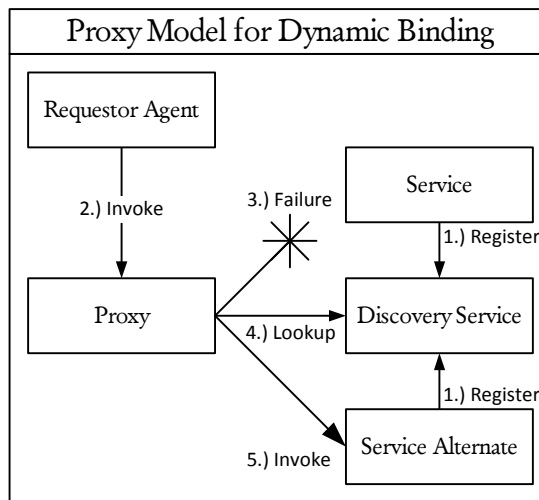


Fig. 13 Detection and rerouting of a failed invocation request using the proxy model.

Whether technology-independent or tightly bound to augmented service code, proxies are realized as distinct functional modules which are separated from the code execution itself, and thus limit the timing of dynamic binding to invocation and invocation failure time. The functional separation allows the proxy to be updated (to change the endpoint selection algorithm or update Discovery Service locations, for example) without interfering with any existing code. Despite this separation of concerns, some proxy approaches do nonetheless take a developer-oriented approach, providing programmers with the means to define how and where dynamic binding and failure recovery should occur while leaving the actual execution of these tasks to the proxy. The most radical of these approaches is jOpera [69] which enables access to system-level utilities (such as a Discovery Service) directly from the executable code, enabling programmers to write business processes which interact with the execution environment itself, reflectively introspecting and modifying their own document structure in order to control its execution.

The work described in [75] provides an API which allows for fine-grain control over the execution of a business process while abstracting over the details of the Web Service standards

involved (such as UDDI). Writing request agent software against the abstract service-oriented tasks of publish/find/bind rather than concrete implementations breaks the tight coupling to specific discovery and communication standards and reduces the amount of code maintenance required when the standards inevitably change. The application of this service-oriented principle into the actual support infrastructure is a powerful concept which is applied throughout the work introduced in this thesis.

Approaches to dynamic binding, whether performed at startup, invocation or failed invocation time and implemented in-band or by proxy, all have a critical reliance upon a Discovery Service. Without a reliable and highly available Discovery Service, dynamic binding will not work and all of the effort applied in order to make requestor agent execution more robust will have been wasted. Appropriately, much work has been applied to the investigation of architectures that can enable the provision of a reliable, scalable and fault-tolerant Discovery Service. The products of this research are presented in the following section.

#### *2.4.2 BY DISCOVERY SERVICES*

Discovery Services provide a registry where Service Providers can publish, and Service Consumers can find, information about the existence and location of Web Services and Web Service endpoints. Discovery Services should ideally be scalable, efficient, autonomic, and fault tolerant [76]. While the conceptual view of the Discovery Service role is that of a centralized service broker, as shown in Fig. 14, the architectures of Discovery Service implementations can be variously categorized as centralized, federated, or distributed. Each of these architectural models has tradeoffs of scalability, fault tolerance, performance, and administrative overhead costs and affect the Discovery Service's ability to be reliable and highly available – and thus the Service Consumers' ability to successfully complete tasks [77]. The following section details the centralized, federated, and distributed architectural models utilized by Discovery Service implementations and provides a comparison of their strengths and weaknesses.

Discovery Services are most often realized using a centralized architecture [78]. Leveraged as a benefit by SELF-SERV [79], a single server at a well-known location is simple to administer and requires no external replication or coordination with other repositories. As there is only a single point of contact, however, centralized architectures can represent a bottleneck under high load, with centralized UDDI being shown to perform poorly in general [38], and to get worse as the number of entries increases [35]. Crucially, centralized architectures represent a central point of failure for the Discovery Service.

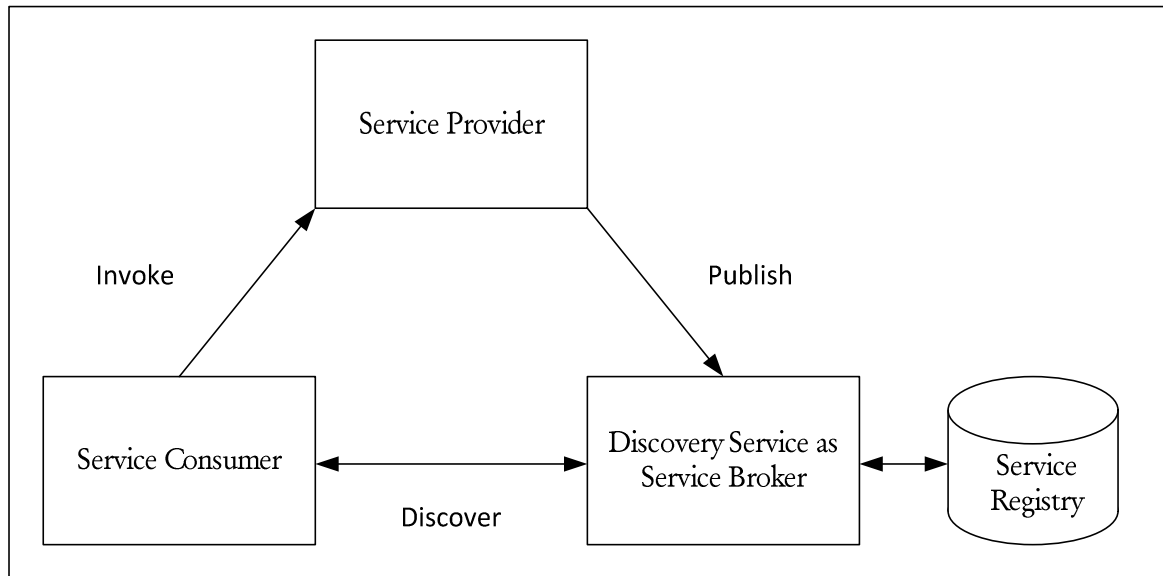


Fig. 14 Conceptual view of the Discovery Service as a centralized Service Broker.

Fault tolerance can be introduced into centralized Discovery Services by replicating registries<sup>2</sup>. While this does provide some resilience to failure, maintaining consistency between replicas requires active coordination and weakens the primary benefit of centralized repositories which is their low administrative overhead. An alternative approach is the use of a federated architecture which gathers together a number of distributed repositories under one conceptually centralized umbrella, with a master peer to maintain consistency and disseminate information among them.

Federated architectures, as shown in Fig. 15, represent a hybrid of centralized and distributed architectures: they are comprised of a distributed set of autonomous and inter-communicating registries but are still managed and administered by a centralized entity. While the administration of federated Discovery Services is centralized, the responsibility for storing entries is shared among a set of peers, which spreads the load and resource requirements and provides greater scalability. Federated architectures also provide a degree of fault-tolerance because the failure of an individual registry peer does not affect the entire network.

The METEOR [41] system implements a federated architecture, using a master gateway peer to organize and propagate information amongst different sub-peers. While the entire system remains available even if repositories fail, the entries they hold will be lost. The model can be made more resilient to failure by replicating each repository's entries in other repositories. As replicating entries on arbitrary nodes reduces the independence and autonomy of unrelated

<sup>2</sup> Hardware-based solutions can also be applied to improve reliability, such as through redundant failovers or 'hot-idling' backup servers, but these are outwith the scope and focus of this thesis.

registries, the work presented in [80] enables peers to band together and form support ‘syndicates’. These syndicates are often grouped based on the content of their registries. These groupings provide the additional benefit of enabling more efficient searches in an otherwise unstructured network.

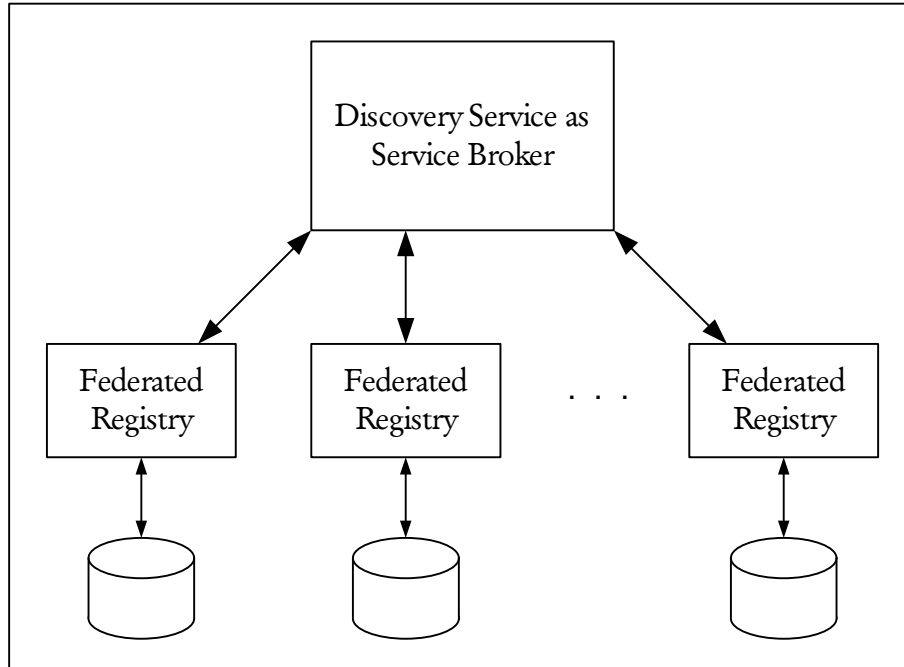


Fig. 15 The federated architecture represents a hybrid of centralized and distributed architectures.

The work presented in [45] utilizes the protocols of Sun’s JXTA [81] to enable the construction of ad-hoc and unstructured P2P networks of registry nodes with potentially restricted bandwidth and processing capacity. Peers, or ‘common nodes’, join the network using JXTA to discover a ‘master-host’ which assigns them into a peer group. Each peer group has its own master-host and an undefined number of common nodes which are responsible for monitoring each other’s liveness. Master hosts need to be capable of running a ‘light’ version of UDDI (for storing the entries of the ‘common nodes’) and are responsible for coordinating peer-group membership and routing lookups. Entries can optionally be replicated between peer groups for failure resilience.

SP2A [76] also uses JXTA to enable the formation of dynamic, ad-hoc registry ‘communities’, and can operate as either a federated registry (with only ‘supernodes’ responsible for routing messages) or as a pure distributed registry (where all nodes have equal responsibilities). WSPDS [43] can similarly operate as either a federated or distributed registry and utilizes network-flooding search protocols similar to those of Gnutella [82] and Freenet [83].

The simple client-server interaction seen in centralized approaches is also a feature of federated architectures, as the distributed nature of federated Discovery Services is not visible to Service Consumers. The centralized administration of federated architectures still represents a central point of failure, however, and reduces scalability as it becomes a communications bottleneck under increased load. Pure distributed architectures, as shown in Fig. 16, are completely decentralized and are typically designed with the aim of being both highly scalable and resilient to failure.

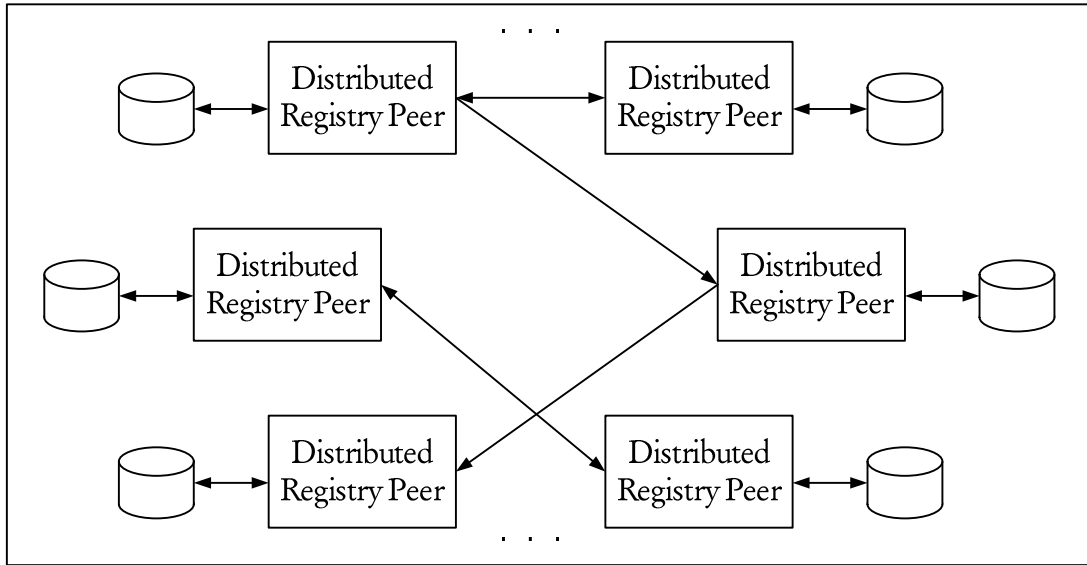


Fig. 16 A distributed network of registry peers.

Fully distributing the responsibility of storing registry entries (and responding to Service Consumer requests) spreads both load and resource requirements, similar to the federated approach. While there is an increased administrative cost associated with maintaining dynamic distributed networks, this too is spread amongst the participating repository nodes and provides the added benefit of eliminating communications bottlenecks and central points of failure.

Work on peer-to-peer (P2P) overlay networks such as Chord [84], CAN [85], Pastry [86] and Tapestry [87] provide the foundation for the majority of distributed Discovery Service implementations (e.g. [42] [44] [88]). These core technologies provide the protocols for manifesting, maintaining, and routing messages in P2P networks. Registry peers are located using content-addressable routing which introduces a structured address space into an otherwise unstructured network. Because registry peers can join and leave the network at will, replication is often used to provide more resilient mappings (as in [89]).

A primary drawback of distributed Discovery Services is the need for Service Consumers to have specialized knowledge of specific search procedures. To overcome this hurdle, distributed discovery services are often paired with the client-side invocation time dynamic binding techniques detailed previously. Proxy-based approaches in particular provide a high-level abstraction over the low-level details of endpoint resolution, and thus enable different search technologies to be transparently substituted based on the implementation of the chosen Discovery Service. Further, this approach requires no additional effort from Service Consumers who should only notice an improvement in both reliability and lookup performance.

It is also important for Discovery Services to contain accurate and up-to-date information about Web Service endpoints. Unfortunately, most service registries contain entries with broken links to WSDL documents, or WSDL documents that reference unavailable Web Service endpoints [48] with a 2005 survey [90] finding 48% of publicly-available UDDI entries unusable.

A novel method for maintaining up-to-date information is the use of the network topology to inform the removal of inaccurate registry entries. Registry information is kept up-to-date in [45], for example, by relying on the elected leader of each common-node community to send node-failure notifications to their master-hosts, who in turn remove the failed node's entries from the registry.

Instead of defending against registry peer failure, the work presented in [46] embraces it as an indicator of endpoint failure. In this approach, Service Providers themselves can join the peer network and provide the functionality of the Discovery Service, listing only the services they themselves provide. When Service Providers leave the network (whether gracefully or ungracefully) their registry entries leave with them. This presents a clean method of maintaining accurate and up-to-date registry information without requiring an external management entity.

Discovery Services which aim to be reliable and accurate – regardless of their implementation architecture – are more often operated directly by Service Providers, due to their close knowledge of the status of their domain. The following section recaps the approaches utilized by Service Providers to not only maintain an accurate and up-to-date Discovery Service, but also to ensure that domain resources are applied effectively in order to maintain the performance and availability of Web Service endpoints.

### *2.4.3 BY SERVICE PROVIDERS*

Service Providers experience change when endpoints or nodes within their domain fail, and when demand levels rise or fall. Approaches to planning, preparing, monitoring, analyzing and

correcting Web Service deployments have been presented in the previous sections which address these lifecycle stages in isolation.

The following section details work belonging to a class of next-generation architectures for Web Services on the Internet. These architectures are defined by their use of abstraction layers between various stages of Web Service deployment, location, binding and invocation for the purpose of introducing further functionality or desirable properties, such as reliability and scalability. They can be described as ‘pan-activity’ approaches and are distinct from approaches which focus on a single or small subset of Web Service lifecycle activities.

## 2.5 AN ALTERNATIVE APPROACH: NEXT-GENERATION ARCHITECTURES FOR WEB SERVICES ON THE INTERNET

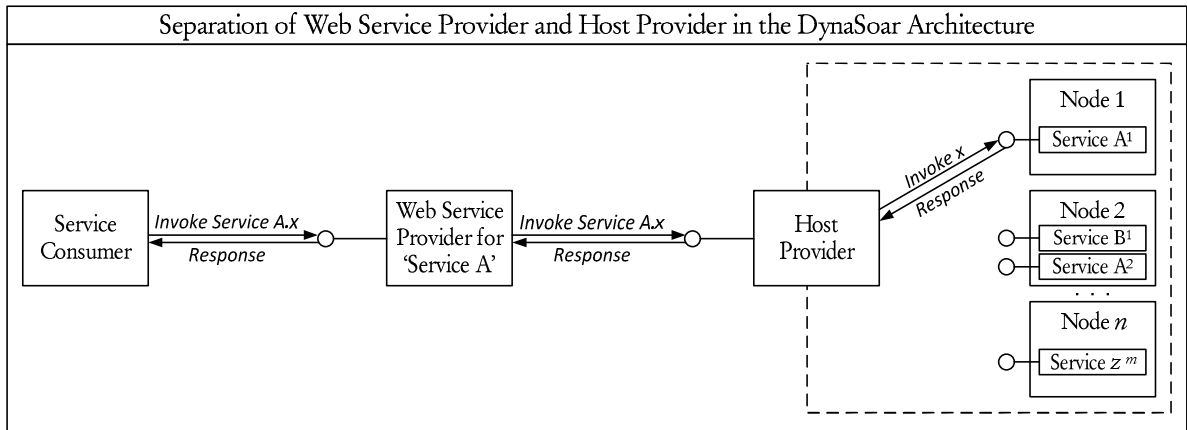
### 2.5.1 DYNASOAR

DynaSoar is an “architecture for dynamically deploying Web Services over a grid or the Internet” [91] and takes a crucial first step towards separating the concerns of the traditional Web Service role of Service Provider. DynaSoar identifies the activities of ‘providing’ a service and ‘hosting’ a service as separate tasks – a manifestation of 3rd-party outsourcing taken from industry. The overburdened Service Provider is thus separated into two distinct but interdependent roles: *Web Service Providers* and *Host Providers*.

Host Providers assume the role of Domain Administrator and control the computational resources on which services can be deployed (e.g. a cluster or grid). Host Providers also take on the Planner and Executor roles, responsible for deploying Web Service endpoints and making all decisions about service provisioning levels.

Web Service Providers are each responsible for a single Web Service and serve as a single point of contact for all actions associated with that service. They provide a Repository which holds implementations of the Web Service, and also serve as a proxy upon which Service Consumers invoke operations provided by the Web Service.

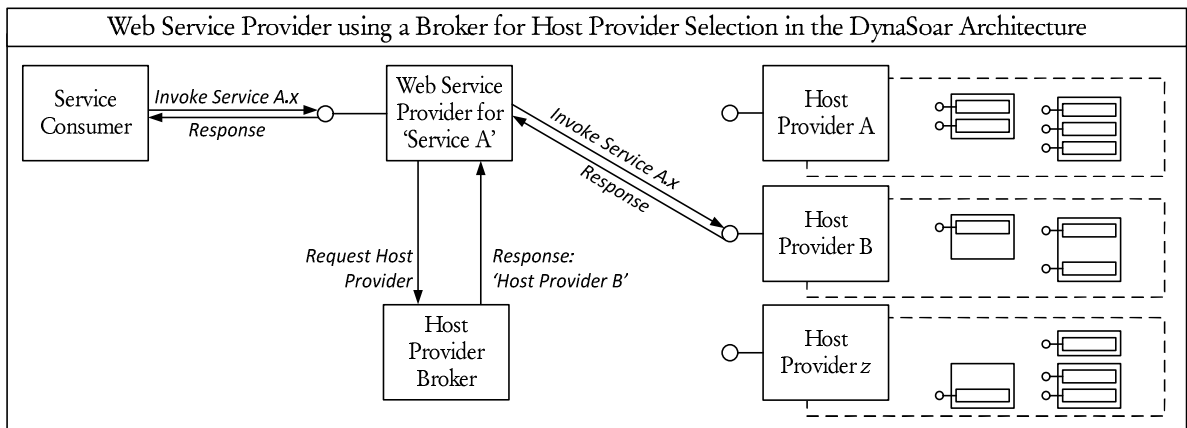
The DynaSoar system can be described as having two distinct modes of operation: ‘*standard*’ and ‘*marketplace*’. Operating in standard mode, Web Service Providers publish their location in a UDDI registry. Service Consumers locate and bind to the Web Service Provider and invoke operations on it directly, as though it were the actual Web Service endpoint. Upon receiving an invocation request, Web Service Providers must select a Host Provider and forward it the invocation request through an exposed ‘Host Provider’ interface – as shown in Fig. 17. In the same way as the Web Service Provider acts as a proxy for Service Consumers, Host Providers act as a proxy to their domain. The Host Provider fulfills requests from Web Service Providers by forwarding the SOAP message to one of potentially many active endpoints within its domain, returning any results to the Web Service Provider who in turn returns them to the Service Consumer.



**Fig. 17** Operation 'x' of Web Service 'A' can be invoked on its Web Service Provider. Invocation request are sent to a Host Provider for processing. Any results are returned to the Service Consumer via the Web Service Provider.

If there are no active endpoints deployed within the Host Provider's domain, the Host Provider must send a message back to the Web Service Provider and request a copy of the service code. In this way the Web Service Provider takes on the roles of both Repository and Repository Administrator, although it is only storing implementations of its single Web Service. Upon receiving the service code the Host Provider deploys one or more instances of the Web Service into its domain, selects one for use, forwards it the invocation request and returns any results.

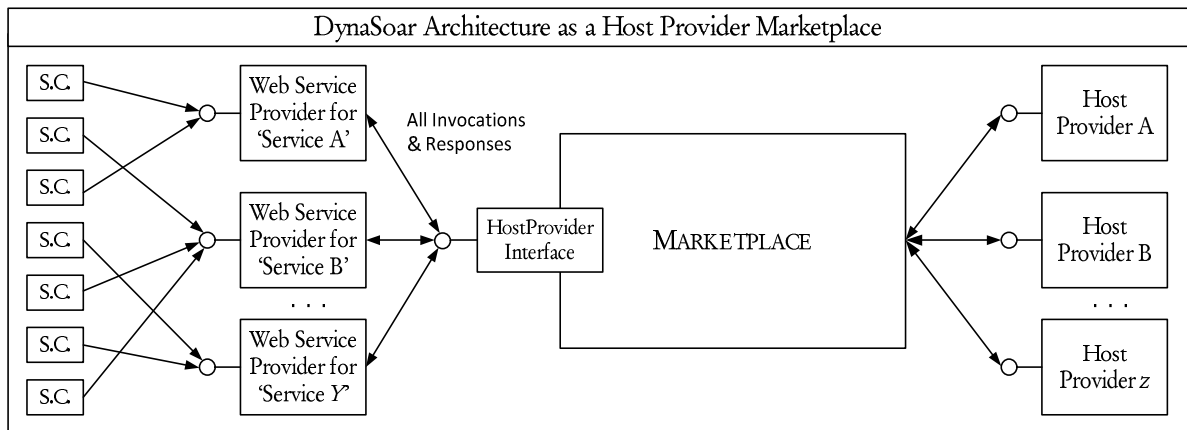
Under normal operation the Web Service Provider makes autonomous decisions about which Host Provider to use. Service Consumers may also explicitly specify a Host Provider to use (an arrangement described as "bring your own hosting" [92]). Alternatively, and as a first step towards operating as a marketplace, the Host Provider selection process can instead be assumed by a third party called a 'Host Provider Broker', shown in Fig. 18. Web Service Providers request a suitable Host Provider from the broker, who dynamically selects one based on criteria such as cost, performance, and dependability.



**Fig. 18** Web Service Providers can find a suitable Host Provider by using a Host Provider Broker.

The use of well defined interfaces between the Web Service Provider and Host Provider enables dynamic Host Provider selection, but it also creates the opportunity for markets which automatically match Web Service Provider requirements with the capabilities of the Host Providers. An example marketplace structure is described in [92] which involves Web Service Providers statically binding to a Host Provider interface on a centralized ‘Marketplace’ entity. Upon receipt of an invocation request, the Marketplace selects a Host Provider and forwards it the invocation request on behalf of the Web Service Provider, returning any results. To better facilitate the selection of Host Providers, the Marketplace entity may use the usage monitoring infrastructure described in [93] to periodically poll Host Providers for hosting-related information, such as runtime performance and service health.

Using the marketplace model shown in Fig. 19, failure can be managed at three levels. Failure of a node controlled by a Host Provider can be managed by that Host Provider, and may involve re-deploying the services previously hosted by the node. Failure of an entire Host Provider can be managed by a Web Service Provider (or the Marketplace entity) selecting an alternative Host Provider. Finally, failure of a Web Service Provider can be dealt with “through the normal Web Service mechanism of the Consumer going to a registry to find an alternative endpoint for the same service” [91]. This is assumed to mean that the Service Consumer will use a UDDI server to find the new location of the Web Service Provider once it has returned or been replaced.



**Fig. 19 Web Service Providers statically bind to a Host Provider interface exposed by a ‘Marketplace’ entity. The marketplace forwards requests to a suitable Host Provider for processing.**

An interesting use-case for this architecture is one where a researcher has a Web Service which they have written and deployed on their own machine. The researcher describes the service at a conference and people are interested in using it, but the researcher doesn’t have the computational resources to offer hosting. It could be offered for download but then new versions

(with bug fixes, etc.) wouldn't be used immediately. The researcher also can't pay for a Host Provider to host the services. The solution utilizes the ability for a Consumer to specify a Host Provider: the researcher starts up a Web Service Provider service on his own computer which receives the request from the people who want to use his service(s). These requests must, however, have specifically defined a Host Provider to use. The problem that does arise is that the researcher's computer must act as a proxy for all the messages – if the services he provides become popular his machine risks being overloaded. Further, while the service may still be available at a Host Provider, it will be unavailable if the researcher's machine goes down or is turned off.

DynaSoar conforms to the Web Service standards of the WS-I Basic Profile and the use of SOAP, in particular, is of critical importance to this architecture. The separation of message handling, service logic, and service resources is an important property of SOAP that allows the free decoupling of the service endpoint (to which message are sent) from the location at which they are executed.

### 2.5.2 WSPeer

The work of WSPeer describes a ‘high-level interface to hosting and invoking Web Services’ [94]. The work is motivated by many of the previously identified drawbacks of current Web Services systems, particularly with regard to reliability, scalability, and the management of the complex issues surrounding heterogeneity. WSPeer addresses these issues by mediating interactions between Service Consumers and Service Providers, with a focus on homogenizing the data formats and protocols used for inter-service communication. Shown in Fig. 20 below, this mediation framework is comprised of a pair of runtimes (one for Service Consumers and one for Service Providers), a proprietary high-level programming language for programming consumer agent applications, and a Web Service-based abstraction layer at the Service Provider which exposes Web Service interfaces for non-Web Service application components through the use of proxies.

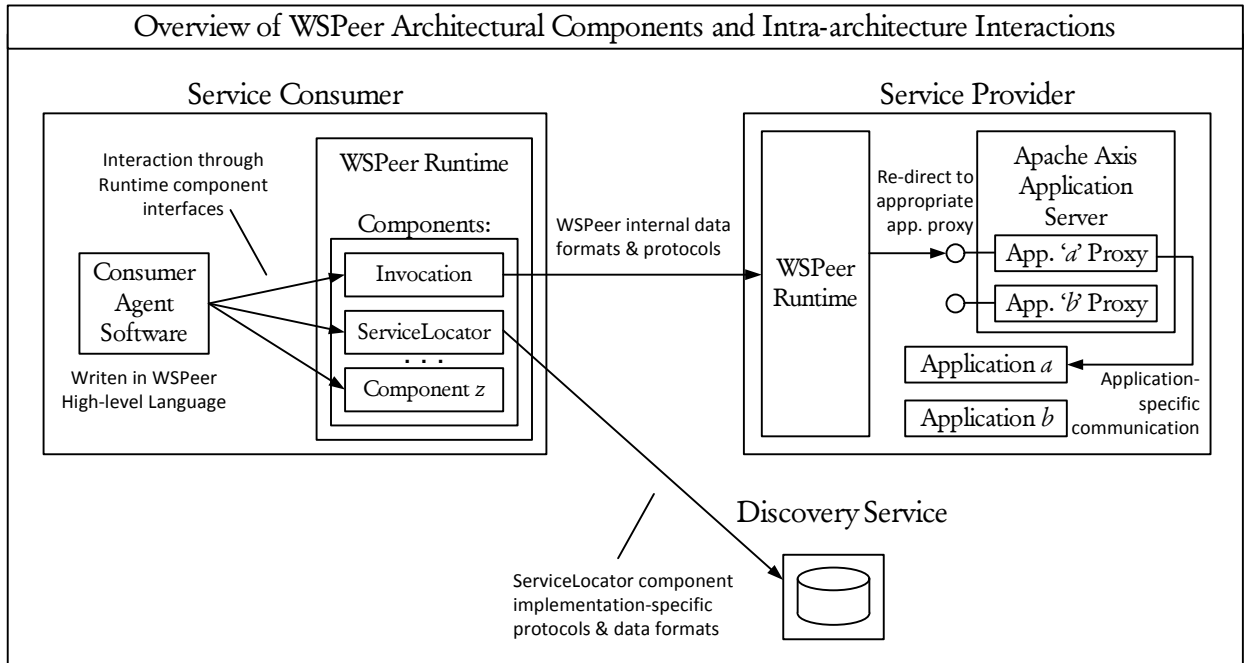
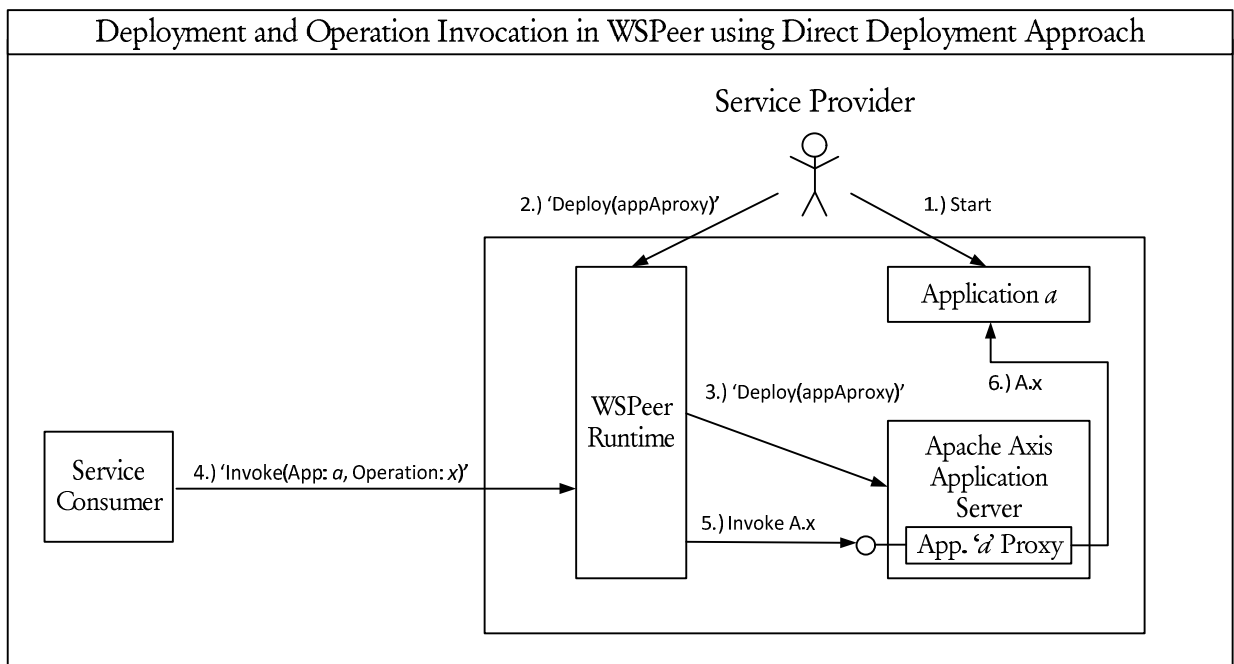


Fig. 20 WSPeer architectural components and intra-architecture interactions.

On the Service Consumer side, the WSPeer runtime is built using a pluggable architecture with interchangeable components. Each component in the runtime fulfills a distinct abstract function, such as locating service endpoints (*ServiceLocator* component) or performing remote invocations (*Invocation* component). WSPeer consumer agent software is written against these components using well-defined interfaces in a proprietary high-level language. Compiled consumer agent applications interact with the WSPeer runtime using a fixed set of ‘actions’ provided by each WSPeer component – such as the ‘*locate*’ action of the *ServiceLocator* component or the generic ‘*invoke*’ action of the *Invocation* component. When changing their

networked environment or in order to keep up with evolving standards, Service Consumers may change or upgrade their component implementations transparently, without requiring any changes to consumer agent applications.

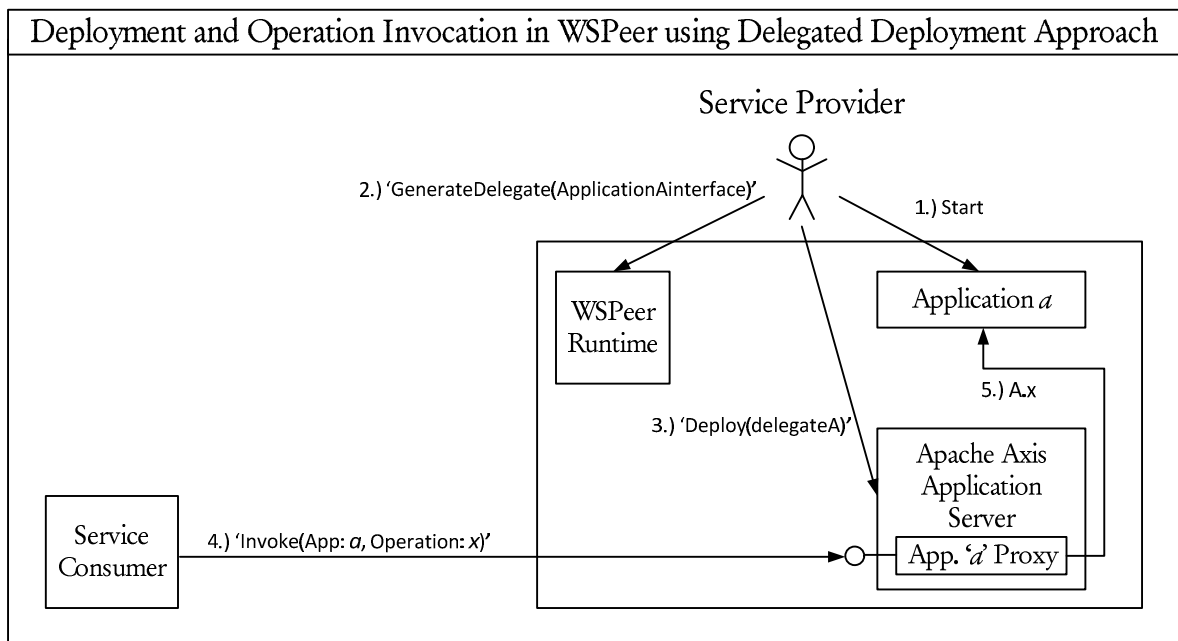
Service Providers deploy applications in WSPeer using one of two methods: 'direct' deployment or 'delegated' deployment. In direct deployment, shown in Fig. 21 below, the Service Provider first writes a custom Java-based Web Service for deployment in the Apache Axis application server. This Web Service serves as a front-end to the application, receiving incoming invocation requests and possibly performing pre-processing or basic decision-making before passing the call to the application. In order to make the application remotely accessible the Service Provider first starts the application then passes the custom front-end Java class to the WSPeer runtime which deploys it in the Apache Axis application server running on the local machine. Incoming invocation requests for applications deployed using the direct deployment method are received by the WSPeer runtime then forwarded to the correct Web Service front-end for the required application.



**Fig. 21 Direct deployment of Application '*a*' in WSPeer with subsequent invocation of operation '*x*'.**

In the delegated deployment approach, shown in Fig. 22 below, Service Providers first describe a remote interface for the application using a proprietary format. This interface is passed to the WSPeer runtime which automatically generates a Java class representing a deployable Web Service proxy for the described application. In order to deploy the application the Service Provider first starts the application then deploys the generated Web Service proxy by hand in the locally-

running Apache Axis application server. Incoming invocation requests are received directly by this proxy service which then forwards them to the corresponding running application.



**Fig. 22 Delegated deployment of Application 'a' in WSPeer with subsequent invocation of operation 'x'.**

For Service Providers, WSPeer utilizes Web Services in order to abstract over service implementation technology so that changes in the underlying application may occur transparently, without disturbing compatibility with Service Consumers. Through this abstraction WSPeer “aims to homogenize Service Consumer access to services exposed by applications which may be implemented in heterogeneous languages” [95]. This approach effectively creates a light-weight container providing remote access to stateful applications which can control their own execution environment (as opposed to the traditional approach of executing code within a container-configured environment).

For Service Consumers, WSPeer runtime components may be transparently replaced with alternative implementations in response to evolving Web Services standards or other changes in the Service Consumer’s networked environment. For example, a ServiceLocator component implementing the UDDI lookup pattern with a hard reference to a University’s private UDDI repository may be replaced with a component referencing a public system and implementing an ebXML or Gnutella search protocol. Because changing a runtime component doesn’t require modifications to consumer agent application code, WSPeer consumer agent software can be considered portable to different execution environments and resilient to changes in the standards, communication mechanisms, and indeed the transport medium used.

Requiring the adoption of a proprietary programming language forces all consumer agent application developers to learn and use a new language which is burdensome, increases the knowledge required to participate in the WSPeer framework, and may restrict the expressiveness of their applications. This rigid language requirement further renders all existing consumer agent applications unusable in the system, which appears to run contrary to the work's ideals of universal integration (though it does not claim to actually provide this). Because of their complete disconnect from the central mediation framework, the choice of deployment procedures appears only exemplary and to not provide any intrinsic value or otherwise contribute to the architecture of WSPeer.

There are, however, a number of aspects of WSPeer that are not clearly identified in its surrounding literature (nor indeed identified at all as beneficial properties of the architecture) that are of particular interest to the work presented in this thesis.

By providing service discovery as a local service, WSPeer effectively factors out service discovery code from all consumer agent applications – isolating and consuming a common and error-prone task and saving development time while reducing program size and the likelihood of programming errors. Isolating service discovery also provides a step towards the full separation of application business logic from the technological details of remoteness.

While providing a component for the fulfillment of remote invocation requests, WSPeer fails to identify the ideal placement of this 'Invocation component' for providing programmatic invocation failure detection and recovery. While DynaSoar does identify the potential for performing failure detection and recovery programmatically at a proxy, it does so using its per-service remote proxies, each of which is a central point of failure and a bottleneck for that service.

One of the most important aspects of the WSPeer design, as it relates to the work presented in this thesis, is its inarticulated consideration of designing for efficient change – that is, designing a system that can adjust to changes in its environment with a minimum of required effort in as few places as possible. WSPeer can be seen as enabling efficient change as a result of factoring out the execution of common activities that are network-dependent (e.g. service discovery and invocation) and isolating them in runtime components. Using this design, changes in a Service Consumer's networked environment can be adapted to at a single local point, within a single, isolated component, rather than by delving into, changing, and recompiling each and every consumer agent application the Service Consumer may wish to use.

Far from just saving time, the WSPeer approach is indeed the first to reduce the knowledge required to participate as a Service Consumer in a distributed software services framework through its removal of the need for Service Consumers to interact with consumer agent application code. This property of WSPeer, intentional or otherwise, is identified as being of great benefit to Service Consumers and will be addressed and improved upon in the work presented later in Chapter 4.

### 2.5.3 SERVICEGLOBE

The ServiceGlobe system is described as a “distributed, extensible e-service platform” [96]. ServiceGlobe aims to enable highly available Web Services by providing an environment that supports the existence of multiple endpoints of the same service. It includes mechanisms for dynamically selecting between available Web Service endpoints, dynamically deploying proprietary ServiceGlobe Web Services, and autonomously distributing load amongst these endpoints through the use of per-service ‘dispatcher’ proxies.

The ServiceGlobe infrastructure is composed of a set of network-accessible machines running the ServiceGlobe Runtime Engine, one or more code repositories, and a UDDI directory running at a well-known location, as shown in Fig. 23 below. This figure further introduces the remaining actors in the ServiceGlobe system: Internal Services, External Services, Service Adaptors and Dispatcher Services, each of which will be covered in the coming paragraphs.

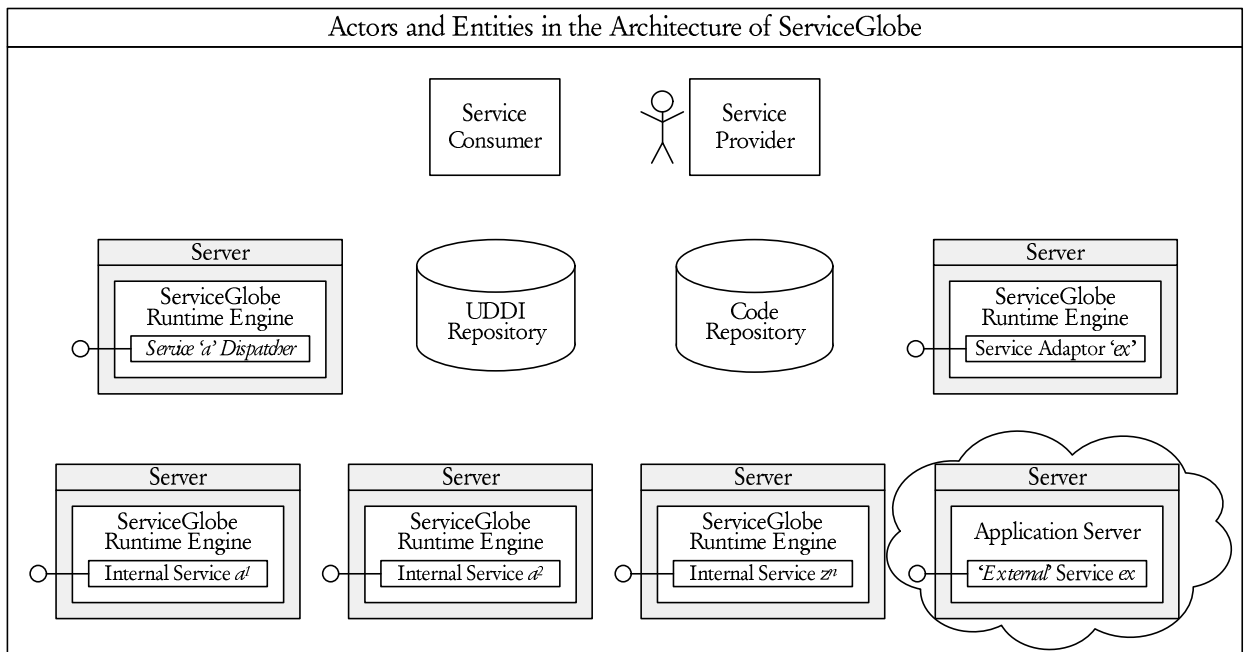


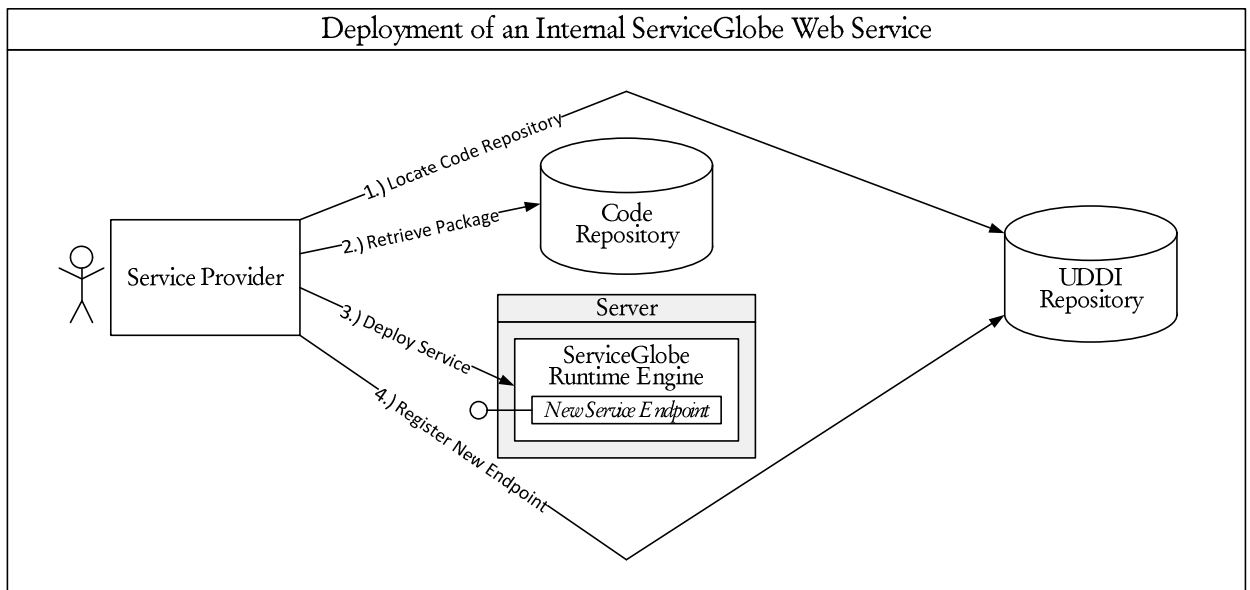
Fig. 23 The Participant Actors & Entities in the ServiceGlobe Architecture

The ServiceGlobe system has two distinct types of Web Services: *internal* services and *external* services. Internal services are written in Java against a ServiceGlobe-provided API. These service implementations are stored in code repositories (whose locations are registered with the well-known UDDI server) together with XML-based configuration files, and can be deployed on servers running the Java-based ServiceGlobe Runtime Engine. If a ServiceGlobe Web Service implementation must run on a single specific server (e.g. for access to a locally-running database) this service is further classified as a *static* internal service. If, on the other hand, the

implementation has no special requirements and can run on any server with the ServiceGlobe Runtime Engine, it is called a *dynamic* internal service. All of the novel features of the ServiceGlobe architecture are designed exclusively for dynamic internal services.

Services written in any other language cannot be used in the ServiceGlobe system. If a Service Consumer wishes to invoke the operations of an existing deployed Web Service instance (an 'external Web Service') then a Service Provider must first write and deploy a custom 'adaptor'. The role of the adaptor is to translate from the internal ServiceGlobe data formats to the externally specified ones, and vice versa.

In the basic ServiceGlobe system, the deployment of an internal ServiceGlobe Web Service (whether static or dynamic) is undertaken by a Service Provider entity as shown in Fig. 24 below. Having identified a target server (running the ServiceGlobe Runtime Engine) the Service Provider locates a code repository using the well-known UDDI server, retrieves the implementation of the desired service from the code repository, deploys the service on the target server, and registers the location of the newly deployed endpoint in the well-known UDDI repository.

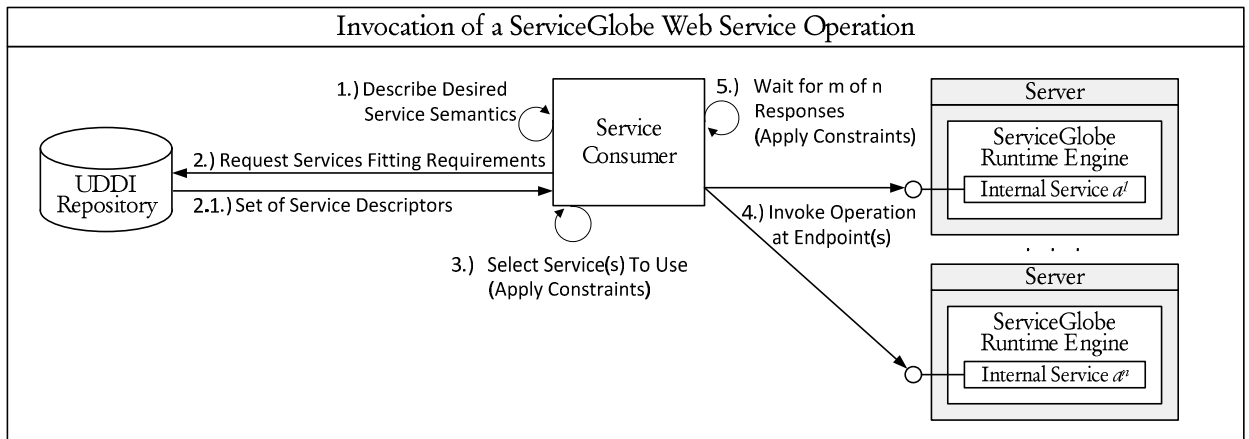


**Fig. 24 Service Provider deploying an internal ServiceGlobe Web Service**

In order to invoke Web Service operations in ServiceGlobe, Service Consumer applications must be written using the UDDI 'tModel' approach. In this approach, applications are not written to invoke a specific service, but instead include a description of the semantics that a target Web Service must provide. This description is represented with UDDI's semantic classification descriptor, called a 'tModel'.

The process begins with Service Providers first classifying a service's functionality and formally describing its interfaces before registering it with the UDDI repository. The Service Consumer then defines the desired semantics (using an agreed-upon set of terms) and 'invokes' this tModel, as shown in Fig. 25 below. Invocation of a tModel involves the Service Consumer or an intermediary contacting the UDDI repository, sending it the tModel, retrieving a list of Web Services meeting the specification, selecting one or more endpoints for use, invoking the desired operation of the specified Web Service endpoint(s), and retrieving any results.

By locally applying a series of constraints on the list of available endpoints, the number of endpoints to invoke, and the number of responses required for a successful invocation, ServiceGlobe is said to enable Service Consumers to dynamically select a Web Service endpoint at runtime.



**Fig. 25 Standard procedure for invoking an internal ServiceGlobe Web Service operation**

Simply providing multiple endpoints of a Web Service, however, does not guarantee that Service Consumers will spread their requests evenly across all endpoints. In order to provide load balancing amongst multiple deployed endpoints of a service, ServiceGlobe introduces a 'generic, modular dispatcher service' [96] which operates as a single point of reference for a single ServiceGlobe Web Service. Similar to a ServiceProvider in DynaSoar, the dispatcher acts as a proxy for arbitrary service calls: invocation requests are sent to the dispatcher service which forwards them to one of potentially many identical instances of the requested service – shown below in Fig. 26.

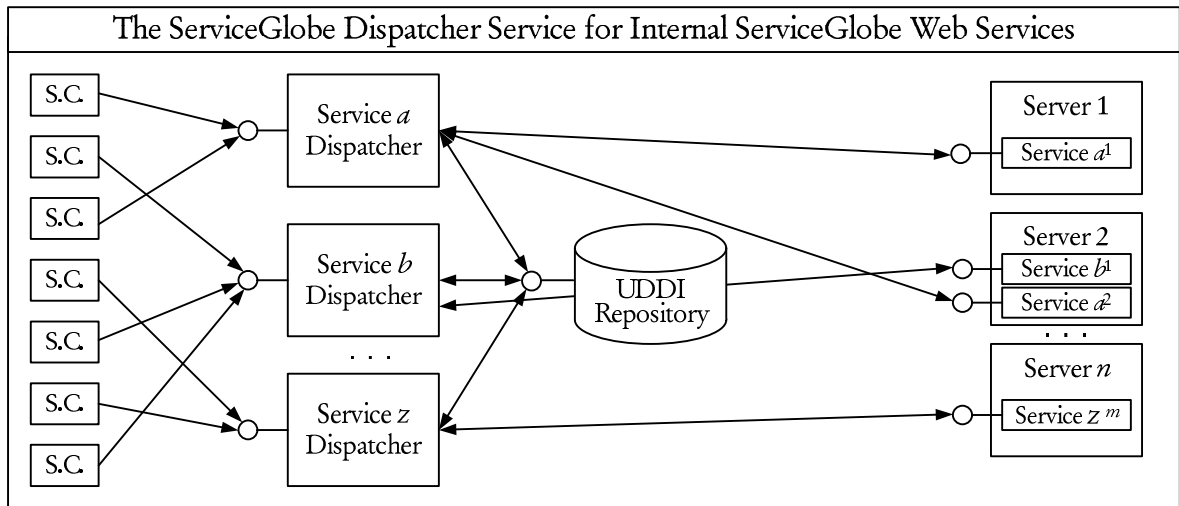


Fig. 26 Three Dispatch Services proxying invocation requests from Service Consumers

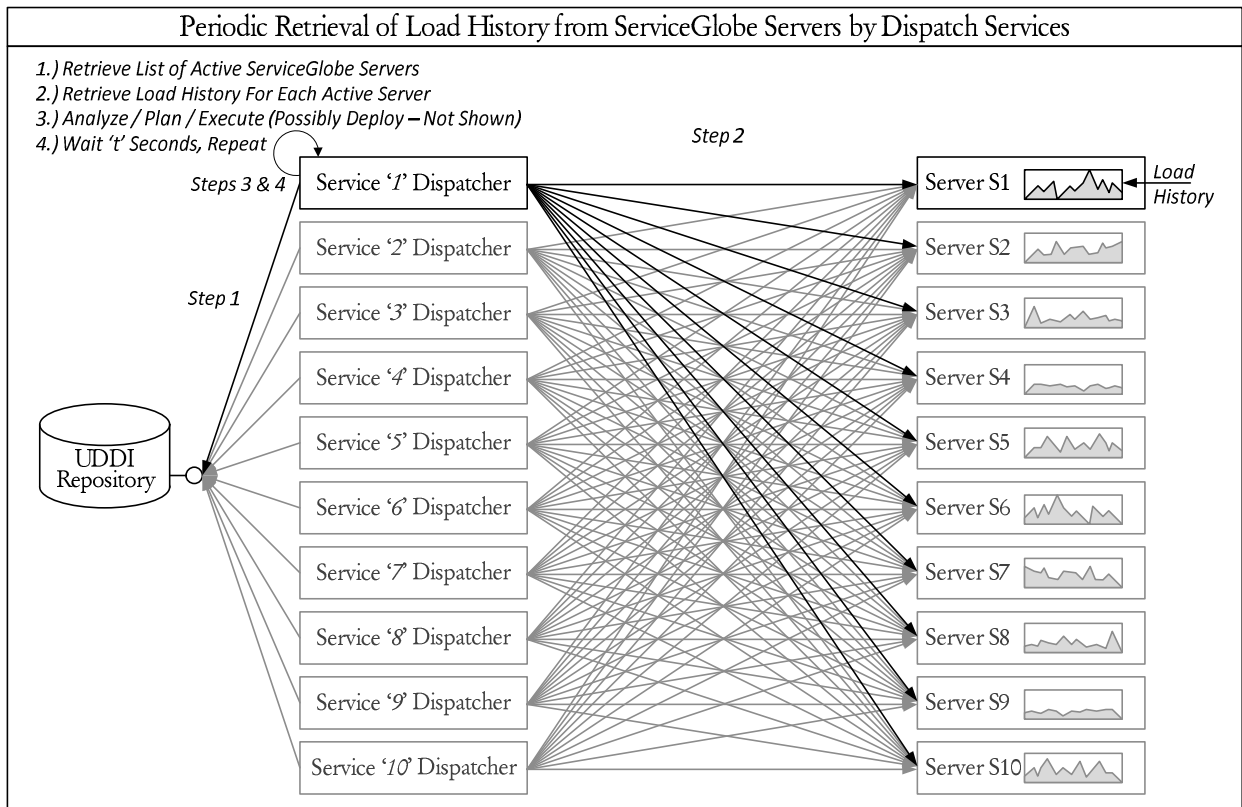
Due to its participation in all invocations of a single Web Service's operations, a dispatcher is well-placed to make informed adjustments to the number of available endpoints of that Web Service. In order to make these decisions a dispatcher must first obtain information about the state of the system – in this case, the load history of the set of servers capable of hosting dynamic internal services. If a dispatcher concludes that its Web Service is currently under-served, and there is a server which is under-utilized, the dispatcher may decide that it is appropriate to deploy an additional endpoint of its Web Service.

The goal of automating the management of Web Service provisioning levels is achieved in ServiceGlobe through the introduction of a standard autonomic control loop in the dispatcher entity. Placing the responsibility of monitoring, analyzing, and acting upon a Web Service's availability levels with regard to an arbitrarily large system makes this decision unscalable for even a small number of services and, as detailed below, counter-effective due to the similarity of decisions reached through isolated analysis of identical data.

Service dispatchers operate autonomously for a single dynamic internal Web Service. They periodically<sup>3</sup> retrieve the list of available servers from the UDDI repository, then retrieve the load history of all available servers running the ServiceGlobe Runtime Engine. By definition, dynamic internal Web Services must be capable of being deployed on any server running the ServiceGlobe Runtime Engine, so each of their dispatchers must always query every available server. For even a small number of services (each of which requires an active dispatcher, regardless of the existence of demand for the service) the data collection, analysis, decision-making and possible deployment

<sup>3</sup> No further information is provided regarding the polling-periods used

processes could easily consume significant resources – as shown in Fig. 27 below for a system of just ten Web Services. Further, because these dispatchers are operating independently using the same set of data, dispatchers may work against one another by coming to and acting upon the same decisions. Acting on identical information in isolation from one another, multiple dispatchers which deploy endpoints and shift load to the ‘least burdened server’ may only create a further heavily-burdened server, prompting another round of tandem analysis and decision-making, leading to thrashing in the autonomic control mechanism.



**Fig. 27 Dispatch Services for 10 dynamic internal ServiceGlobe Web Services periodically retrieving load history data from all active ServiceGlobe servers.**

The ServiceGlobe authors do identify the dispatcher as a critical single point of failure in [97] and present possibilities for reducing the risk of failure. Redundant hardware is rejected as out of the scope of an otherwise all-software solution. A “watchdog mechanism” borrowed from the field of database systems is proposed as further work. In this scheme the ServiceGlobe system would run two identical dispatcher services with only one registered at the UDDI server. The second one would watch the first one and detect any failures. In the event of failure the second server would update the UDDI record to point to itself. Clients would be expected to detect a failed invocation and go back to the UDDI directory, search for the dispatcher service, update their references and retry the invocation with the updated location. There is no procedure

described for handling the failure of a service's backup dispatcher or of the UDDI server. Resource consumption issues surrounding the need for two active dispatcher services for every Web Service are not addressed but appear significant.

While no data is presented regarding the various merits and ramifications of the monitoring, analysis and deployment systems, it is ServiceGlobe's successful use of multiple endpoints to provide high availability that is of greatest value to the work at hand. It is shown in [96] that service replication is an effective approach to providing very high availability and decreasing average invocation response time, for which a simple analytical investigation is presented below – note that it relies on a reliable dispatcher which is not present in the ServiceGlobe architecture:

“Assuming that the server running the dispatcher itself... [is] highly reliable, the availability of the entire system depends only on the availability  $\alpha_{\text{ServiceHost}} = \alpha$  of the service hosts. The availability of a pool of service hosts can be calculated as follows:

$$\alpha = \frac{\text{MTBF}}{(\text{MTBF} + \text{MTTR})(1)} \quad (1)$$

$$\alpha_{\text{pool}} = \sum_{i=1}^N \alpha^i (1 - \alpha)^{(N-i)} = 1 - (1 - \alpha)^N \quad (2)$$

Equation 1 calculates the availability of a single service host based on its MTBF (mean time between failures) and MTTR (mean time to repair). The availability of a pool of  $N$  service hosts can be calculated using Equation 2. Even assuming very unreliable service hosts with MTBF = 48hrs and MTTR = 12h a pool with 8 members will only be unavailable about 1.5 minutes a year.”

Based upon the observation that “using several instances of a service greatly increases its availability and decreases the average response time” [96], ServiceGlobe approaches the goals of high-availability and reliable execution by providing a multi-endpoint environment – that is, one in which multiple endpoints of the same Web Service may be deployed in order to serve demand. Motivated by the prospect of providing high-availability while cognizant of the challenges of creating an flexible and reliable Web Services architecture, this multi-endpoint approach is embraced and these challenges addressed in the work presented in the next chapter: “A Next-generation Architecture for Web Services on the Internet”.

## 3 A NEXT-GENERATION ARCHITECTURE FOR WEB SERVICES ON THE INTERNET

### 3.1 INTRODUCTION

This chapter introduces a next-generation architecture for Web Services on the Internet. It begins in section 3.2 by defining a set of requirements for a next-generation Web Service architecture and continues in section 3.3 by presenting an architecture that meets these requirements. The description of the architecture begins in section 3.3.1 with a high-level structural overview. The actors of *Publisher*, *ServiceHost*, *Manager*, *ActiveServiceDirectory*, and *PointOfPresence* are introduced next in section 3.3.2, followed by a set of descriptors and two new architectural entities: repositories called the *ServiceLibrary* and *HostDirectory*. Finally, section 3.4 presents a framework for interaction between these actors and entities and details the functionality provided by each.

### 3.2 REQUIREMENTS FOR A NEXT-GENERATION WEB SERVICES ARCHITECTURE

This section defines a set of requirements that a next-generation architecture must meet, based on the previously-identified drawbacks of current approaches. These requirements aim to encourage a separation of concerns between participants in the architecture, to advance the creation of mechanisms which can accomodate heterogeneous technologies, and to promote the development of architectures which are resilient to various types of failure. Although some of the existing architectures address some of the individual requirements, the satisfaction of this complete set of requirements is unique to this next-generation model.

#### 3.2.1 ACCOMODATING TECHNOLOGICAL HETEROGENITY

There currently exist multiple, competing Web Services technologies for which a large number of provider agent and consumer agent applications have been developed, and it can be reasonably assumed that provider agent and consumer agent applications will continue to be developed using more than one set of technologies. If the goal of universal interoperation is to be realized, then technological heterogeneity must be taken into account in the design of any next-generation architecture. The first three requirements address the accommodation of heterogeneity in the technology used for developing, deploying, and hosting provider agent Web Service applications:

*In order to promote the integration of existing closed-world systems and foster the development of new, open-world environments, it must be possible to include hosting resources with various configurations and application server technologies.*

**Requirement 1:** The architecture must enable the incorporation and utilization of heterogeneous Web Service hosting resources.

*Providing a model which enables the incorporation of existing Web Service implementations with minimal effort can encourage existing and future provider agent-publishers to participate in an open market for software services. This is argued to be of benefit to the Service Consumer as it increases access to services previously trapped in proprietary environments, which may in turn lower costs (through competition) and save on development time.*

**Requirement 2:** It must be possible for existing Web Service provider agent application implementations to be utilized in the architecture without any modifications to their code.

*Generic deployment mechanisms are argued to be of critical importance to any architecture which aims to support heterogeneous technologies. Different Web Service application hosting technologies, for example, require different sets of steps to be executed in order to deploy and expose a provider agent application as a Web Service. Identifying common traits amongst the deployment procedures of Web Service application hosting environments will allow generic mechanisms to be developed for Web Service deployment and undeployment, which in turn will enable the isolation (and possibly automation) of the responsibility of managing the provisioning of Web Service endpoints, independent of the specific technologies in use.*

**Requirement 3:** The model must provide technology-neutral mechanisms for instigating the deployment and undeployment of Web Services.

### 3.2.2 SEPARATING ENTANGLED CONCERNS

The Service Provider role, as currently defined, involves high cognitive complexity and has high barriers to entry, discouraging participation by individuals due to the cost and complexity involved in fulfilling all the responsibilities of the role. Having identified the need for a separation

of concerns and a more fine-grained approach to assigning actors' responsibilities, requirements 4 and 5 address hosting resources, while requirements 6 through 8 pertain to the development of provider agent applications (i.e. deployable Web Service implementations).

**Requirement 4:** The model must lower the barriers to participation in the provision of Web Services.

*Implementing the business logic of a Web Service is very different from hosting a Web Service, requires a different skill-set, and a different set of software and hardware technologies. Separating the tasks of implementing and deploying a Web Service also protects the hosts by leaving the process of deployment within their control – and further enables the development of a marketplace for hosting resources.*

**Requirement 5:** The tasks of developing and publishing a Web Service must be separate from the task of deploying and hosting a Web Service.

*Separating the basic responsibilities of a 'host' (exposing interfaces for remote access to applications deployed within its domain) from activities surrounding 'hosting' (deploying and undeploying Web Service endpoints on host machines) allows hosts to be viewed as a consumable resource. Because statically provisioning resources can be wasteful, any new architecture should enable available hosts to be added to a pool of available hosting capacity which can then be applied to suit the goals of the architecture as and when necessary. This separation further encourages a 'market-based' approach to pairing provider applications with suitable hosts.*

**Requirement 6:** It must be possible to exploit the latent computational resources of hosts in a distributed computing environment.

*Because Web Service standards do not specify the procedures for re-locating a migrated Web Service endpoint (or an alternative endpoint in the case of failure), ad-hoc methods must currently be developed & implemented, taking time and introducing complexity and the potential for failure into consumer agent applications.*

**Requirement 7:** Mechanisms must be provided which disentangle distribution-related details from the business logic of consumer agent applications.

*Monitoring and reporting of Web Service usage data is an important aspect of the Web Service lifecycle which enables managing entities to respond to failure and make decisions on the required level of resource provisioning. In the same way that implementing the business logic of a consumer agent application must be separated from handling distribution-related details, Web Service implementations must not be required to perform any environment-specific tasks which are unrelated to providing and fulfilling requests for its advertised operations.*

**Requirement 8:** The architecture must provide means for the monitoring and management of Web Service provisioning which place no requirements on Web Service provider agent implementations to record or report usage data.

### 3.2.3 DESIGNING FOR FAILURE

*For developers of consumer agent applications, programmatically detecting and recovering from failure is an onerous, complex, and time-consuming task, fraught with difficulty and opportunities for error. It is argued that there is an expectation of failure in distributed systems and, knowing this, that any next-generation system must be designed to gracefully accommodate failure within the architecture.*

**Requirement 9:** The architecture must be resilient to the failure of its components and the Web Services which rely upon them.

*Finally, it is important to take into account the intrinsic fallibility of hosts on the Internet. Designing for dynamically changing levels of hosting resources will enable systems to evolve to changing user demands and provide more reliable and highly available services.*

**Requirement 10:** The model must gracefully handle the addition and removal of hosting resources.

### 3.3 A NEXT-GENERATION ARCHITECTURE

#### 3.3.1 INTRODUCTION

The following sections define each component of the next-generation architecture described in this thesis. Section 3.3.2 begins by introducing the core concepts and abstractions in the architecture and presents an overview of its structure. The roles and responsibilities of each actor in the architecture are defined next, together with their descriptors and the registries used to hold these descriptors. The framework for interaction between each actor is detailed last, together with deployment and invocation use-cases.

Throughout this chapter the term ‘model’ should be understood to mean ‘architectural model’; the term ‘infrastructure’ is used to refer to the abstract notion of a ‘running’ architecture – that is, a collective reference to the described architectural components existing in an operational state, independent of any particular implementation.

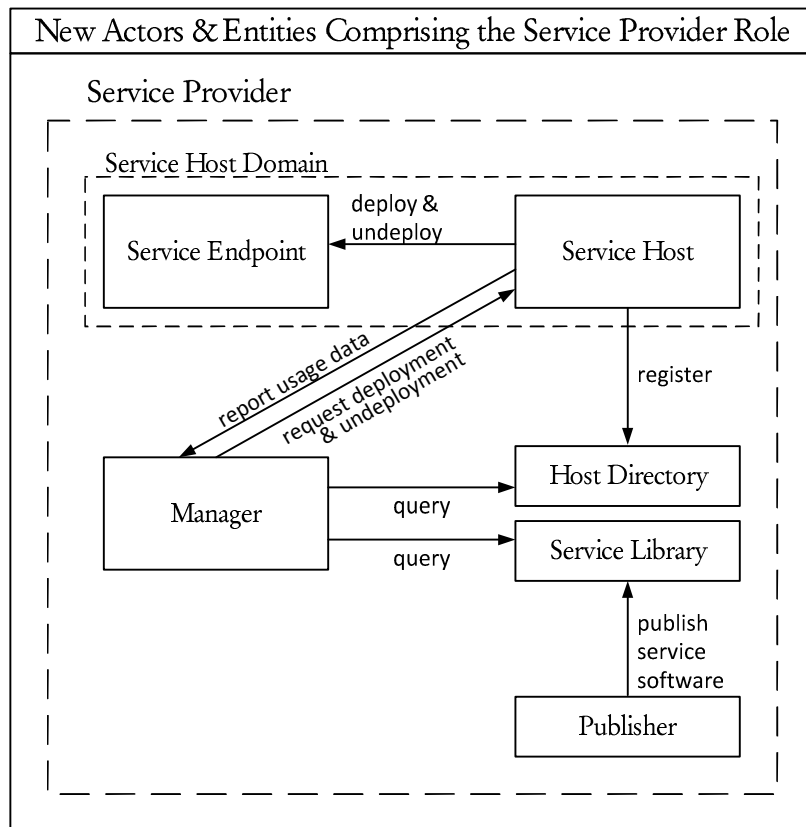
#### 3.3.2 OVERVIEW

The architecture described in this chapter takes an end-to-end or ‘holistic’ approach to addressing the previously-identified shortcomings of the traditional Web Services model. The architecture presents a multi-endpoint Web Service environment which abstracts over Web Service location and technology and enables the dynamic provision of highly-available Web Services. The model describes mechanisms which provide a framework within which Web Services can be reliably addressed, bound to, and utilized, at any time and from any location.

The presented model aims to ease the task of providing a Web Service by consuming deployment and management tasks. It eases the development of consumer agent applications by letting developers program against what a service does, not where it is or whether it is currently deployed. It extends the platform-independent ethos of Web Services by providing deployment mechanisms which can be used independent of implementation and deployment technologies. Crucially, it maintains the Web Service goal of universal interoperability, preserving each actors’ view upon the system so that existing Service Consumers and Service Providers can participate without any modifications to provider agent or consumer agent application code. Lastly, the model aims to enable the efficient consumption of hosting resources by providing mechanisms to dynamically apply and reclaim resources based upon measured consumer demand.

##### 3.3.2.1 Providing Web Services

The presented model addresses the goal of reducing complexity for participants in the Web Service lifecycle by partitioning the responsibility of providing a Web Service into multiple independent roles, reducing the amount of domain-specific knowledge required by each actor and lowering the barriers to participation in the provision of Web Services. This process begins by treating the tasks of publishing, deploying and hosting a Web Service as distinct, independent activities. The traditional Service Provider role is thus decomposed into three autonomous actors: *Publisher*, *Manager*, and *ServiceHost*. In collectively fulfilling the responsibilities of the traditional role of Service Provider, these actors are supported by two new architectural entities: repositories called the *ServiceLibrary* and *HostDirectory*. These actors and entities are shown in Fig. 28 below together with labeled edges indicating the abstract interactions carried out between them. Each of these interactions is covered in the more detail in the paragraphs to follow.



**Fig. 28 Interactions between the new actors of ServiceHost, Manager and Publisher, and the two new architectural entities of HostDirectory and ServiceLibrary which together provide the functionality of the traditional Service Provider role.**

Rather than being deployed explicitly, a Web Service provider agent implementation is instead described by a Publisher who then ‘publishes’ it into the infrastructure by storing it in a repository called the ServiceLibrary. This approach represents a break from the traditional approach to Web Service deployment by separating service substantiation from actual realization:

in the presented model, the lifecycle of a Web Service begins when it is published, not when it is deployed.

In order to participate in the infrastructure, ServiceHosts register their willingness to host Web Services by describing their available resources and registering with a directory called the HostDirectory. ServiceHosts indicate their available Web Service deployment containers and specify the list of Publishers whose Web Service provider agent implementations they are willing to deploy. ServiceHosts thus participate in an infrastructure not by advertising their statically deployed services, but by advertising their hosting capabilities, joining a shared pool of latent hosting resources which can be dynamically consumed (and reclaimed) by Managers as necessary to meet changing levels of demand.

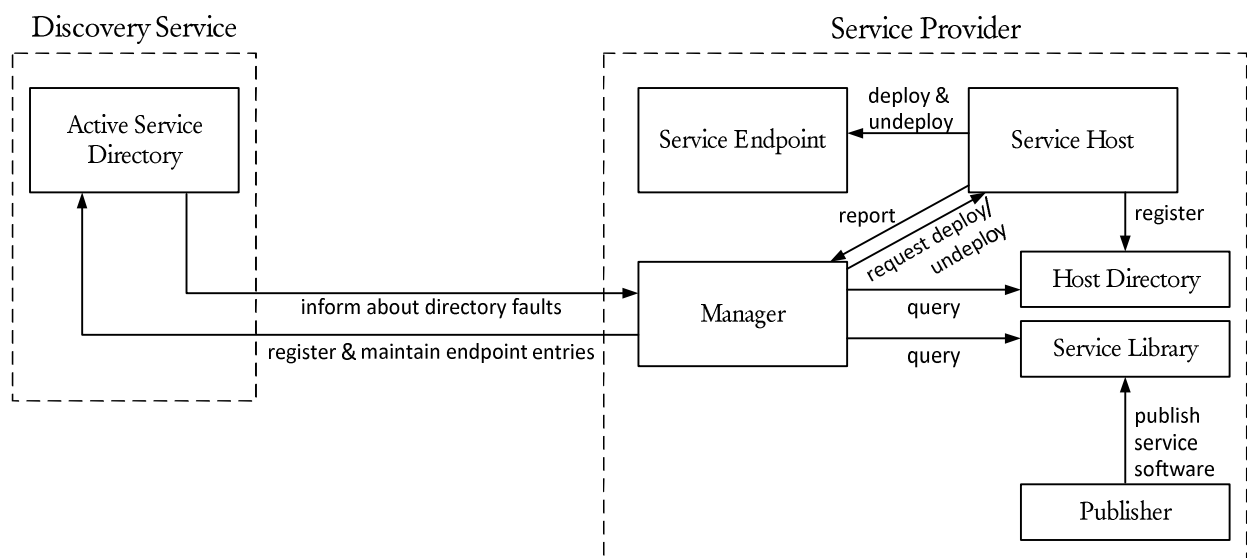
Managers are responsible for managing the provisioning level of a single Web Service (for which there may be zero or more endpoints at any given time). In order to enact deployment, Managers first query the ServiceLibrary and HostDirectory, then create deployment plans by pairing Web Service implementations with a suitably capable ServiceHost. Managers send deployment requests to ServiceHosts who are then responsible for instantiating an endpoint of the Web Service (or denying the request). ServiceHosts provide information about the usage of each Web Service endpoint deployed within their domain to each Web Service's Manager. A Manager can use this usage data to make decisions about the necessary level of provisioning of the Web Service they manage.

#### 3.3.2.2 Activating the Discovery Service Role

The responsibilities of the traditional Discovery Service role are consumed by a new actor called the *ActiveServiceDirectory*. The *ActiveServiceDirectory* holds mappings between a single Web Service and a set of active endpoints of the Web Service. A central entity in the architecture, the collective mappings held in the *ActiveServiceDirectory* represent the current state of an infrastructure from all participants' perspectives. Because all infrastructure participants rely on the *ActiveServiceDirectory* to locate endpoints of their desired Web Services, this directory is ideally placed to instigate autodeployment procedures for Web Services that have no currently deployed endpoints.

Detailed in the upcoming section 3.3.3.4, the *ActiveServiceDirectory* provides operations to add, remove and locate active endpoints of Web Services. If the *ActiveServiceDirectory* receives a lookup request for the active endpoints of a particular Web Service, but no such entries exist, the *ActiveServiceDirectory* is responsible for proactively locating and informing the Manager of the

requested Web Service. The Manager can dynamically initiate the deployment of a new endpoint using the previously described deployment procedure. Managers are responsible for inserting and maintaining all ActiveServiceDirectory entries for any newly deployed endpoints of the Web Service it manages. If demand for its Web Service drops to zero, a Manager may decide to undeploy one (or all) of the deployed endpoints. If an endpoint is undeployed the Manager removes the endpoint entry from the ActiveServiceDirectory, ensuring that the directory remains up-to-date and as accurate a reflection as possible of the current state of the infrastructure. The interactions between the ActiveServiceDirectory (as Discovery Service) and Managers are shown in Fig. 29 below.



**Fig. 29 The interactions between the ActiveServiceDirectory (as Discovery Service) and the actors & entities comprising the Service Provider role.**

### 3.3.2.3 Consuming Web Services

The presented architecture does not use URLs to describe Web Services since URLs may become invalid over time. Web Services are instead identified with a URI, abstractly describing a service which at any point in time may have zero or more active endpoints. The *ServiceConsumer* actor is relieved from the tasks of locating and binding to Web Service endpoints through the introduction of a mechanism which robustly performs these tasks on their behalf.

The architecture presents a limited-mediation framework for the consumption of Web Services which transparently resolves a live endpoint URL from the URI contained in an invocation request. This framework is realized as proxy mechanism, residing at the Service Consumer, which acts as a gateway into an instantiation of the architecture – a ‘point of presence’, as shown in Fig. 30. Consumer agent software is written to bind to this local point of presence and invoke Web

Service operations using the desired Web Service's URI. The point of presence is responsible for transparently resolving a URL from the URI by retrieving a list of active endpoints of the Web Service from the ActiveServiceDirectory, selecting an endpoint for use, invoking the requested operation on behalf of the Service Consumer, and returning any results. It is also responsible for transparently detecting and recovering from the failure of Web Services and the ServiceHosts on which they are deployed, and for proactively recovering from these failures by retrying alternative endpoints (according to local policy). Only unrecoverable errors are returned to Service Consumers, indicating that given the available resources of the infrastructure, it is not currently possible to fulfill the request.

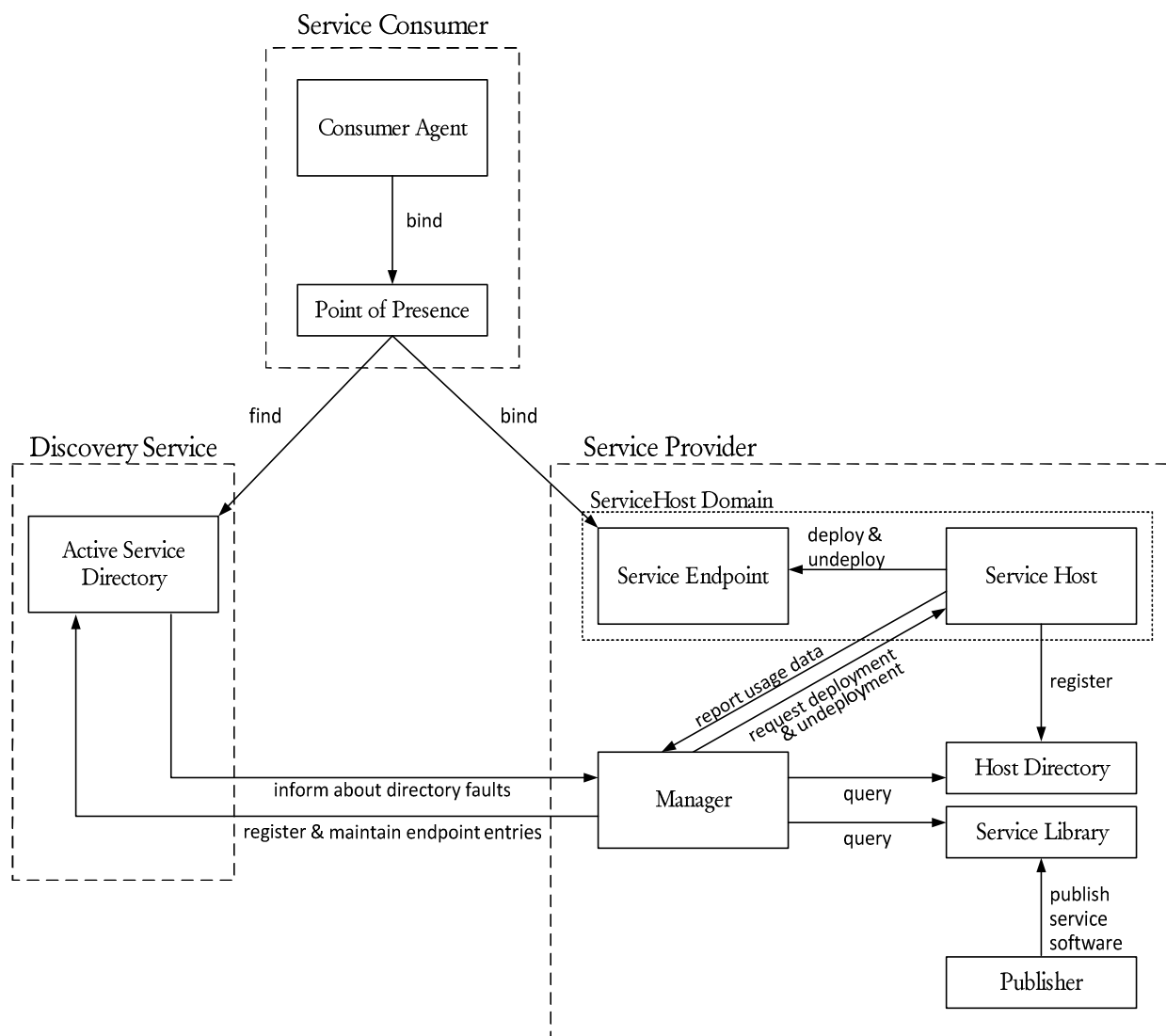


Fig. 30 Actors, entities and interactions in the newly presented architecture.

The point of presence abstracts over the location of a service while still preserving the current Service Consumer actor's view upon the system. It simplifies the creation of consumer

agent applications by allowing developers to program against what a Web Service does, not where it is or whether it is currently deployed. Further, because it consumes all tasks which require interaction with the Discovery Service, the point of presence provides a layer of abstraction over the particular standards versions used in an infrastructure (e.g. UDDI version). This provides a barrier to obsolescence in the face of evolving standards while enabling consumer agent applications to be portable between environments which use different standards.

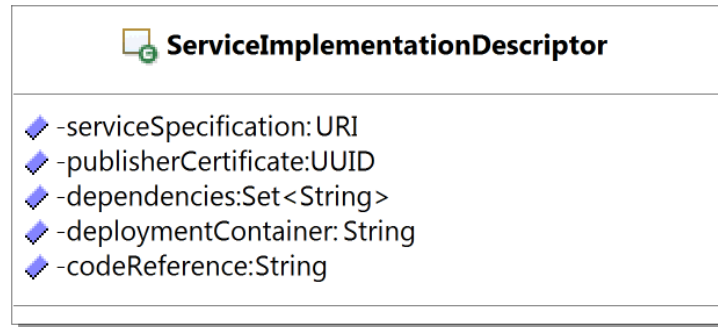
The process of transparently resolving endpoint URLs from the provided URI also factors out the redundant failure-detection and recovery code necessary in the traditional Web Services usage pattern. Multi-endpoint environments such as DynaSoar [91] and WS-Peer [94] provide alternative methods of approximating these properties by offering an external single point of contact responsible for each individual Web Service. By moving the proxy indirection out of the infrastructure and onto the client, the presented architecture not only conserves resources, but removes the bottleneck and central point of failure introduced by remote proxies, assuming that if the local point of presence dies, the client has died too.

The diagram in Fig. 30 presents the structure and relationships between the newly introduced actors, with directed links indicating the actions carried out between them. The full responsibilities of each actor are detailed in the upcoming section, “Next-Generation Actors: Roles & Responsibilities”.

### 3.3.3 NEXT-GENERATION ACTORS: ROLES & RESPONSIBILITIES

#### 3.3.3.1 Publisher

The *Publisher* actor is responsible for bringing an implemented Web Service into a repository: it acquires the location of the provider agent code from a developer, describes its deployment container and hosting requirements using a `ServiceImplementationDescriptor` (shown in Fig. 31 below), and publishes it into a library of implementations called a *ServiceLibrary*.



**Fig. 31** The '`ServiceImplementationDescriptor`' is used by a Publisher to identify a single implementation of a Web Service provider agent application.

A `ServiceImplementationDescriptor` is used to describe an implementation of a Web Service. It includes the URI of the implemented Web Service, a UUID identifying the Publisher, the location of the service code, a set of hosting environment requirements as Strings, and the String identifying the required deployment container. The uniquely-identifiable *PublisherID* identifies a single Publisher and can be used to guide decisions on service deployment and invocation-time endpoint selection. At any point in time there may be multiple published implementations of a Web Service, each published by a different Publisher, which implement the same Web Service (as identified by the 'serviceSpecification' element of the `ServiceImplementationDescriptor`). A major benefit of this approach is the increased likelihood of a Web Service implementation being compatible for deployment on one of the set of currently available hosts.

#### 3.3.3.2 ServiceHost

`ServiceHosts` control one or more Web Service deployment containers which are capable of hosting and exposing Web Services. Each `ServiceHost` must expose a remotely-accessible Web Service interface for the provision of Web Service endpoints conforming to the interface '`IServiceHost`' shown below in Fig. 32. `ServiceHosts` are required to fulfill deployment & undeployment requests received through this interface. For every Web Service endpoint deployed in their domain, `ServiceHosts` are required to record and periodically report endpoint

usage data to the service's managing entity (described next in section 3.3.3.3 'Manager'). For each deployed Web Service, ServiceHosts report usage data at a frequency specified in the 'reportPeriod' parameter of the 'deployServiceInstance' operation. This data is used by the managing entity to make decisions on endpoint provisioning levels.

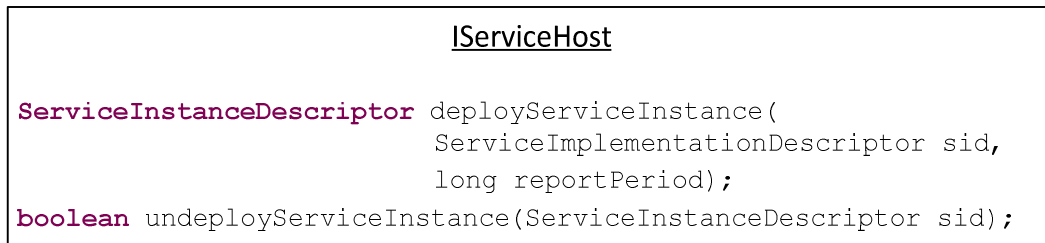


Fig. 32 The `IServiceHost` interface for managing the deployment and undeployment of Web Service endpoints.

Shown in Fig. 33 below, a *HostDescriptor* is used to describe a ServiceHost. Each HostDescriptor includes the ServiceHost's uniquely-identifiable *hostID*, a reference to the location of their `IServiceHost` Web Service, and a description of their hosting capabilities (such as the available deployment containers). In order to indicate their willingness to participate in the hosting of Web Services, a ServiceHost stores their HostDescriptor in a directory called the *HostDirectory* (detailed in the upcoming section 3.3.4).

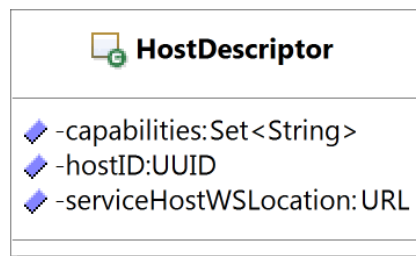
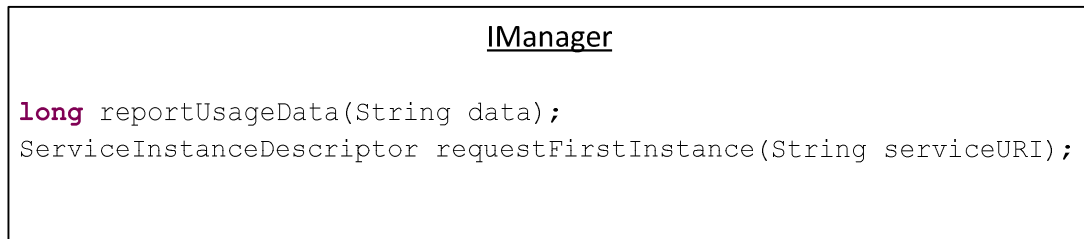


Fig. 33 A `HostDescriptor` describes the capabilities, location and identity of a single ServiceHost.

### 3.3.3.3 Manager

Managers are responsible for managing the provisioning level of a particular Web Service. Managers create and execute deployment plans in order to ensure that an adequate number of endpoints are deployed in order to meet the current level of demand for the Web Service being managed. Managers interact directly with ServiceHosts in order to request the deployment and undeployment of endpoints and are responsible for registering and maintaining Web Service endpoint entries in a directory called the *ActiveServiceDirectory* (described next in section 3.3.3.4).

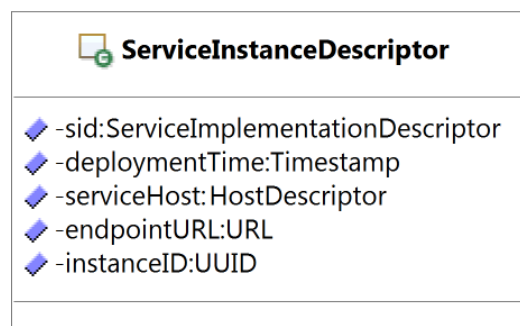


**Fig. 34 Each Manager must expose a Web Service implementing the IManager interface. The 'reportUsageData' operation is used by ServiceHosts to report usage data for the Web Service under management.**

Each Manager is required to expose a remotely-accessible Web Service conforming to the *IManager* interface, shown in Fig. 34 above. ServiceHosts report usage data to Managers through the `reportUsageData` operation of the *IManager* interface; in return, Managers return ServiceHosts a numerical value indicating the length of time the ServiceHost should wait before next reporting. Decisions about endpoint provisioning can be made based on the usage data returned from the ServiceHosts which are currently hosting endpoints of the Web Service under management. Detailed later in section 3.5.3 'Demand-driven Dynamic Deployment', the `requestFirstInstance` operation is invoked by the *ActiveServiceDirectory* in the event that it receives a lookup request for the list of all active endpoints of a particular Web Service and it finds that no endpoints are currently deployed.

#### 3.3.3.4 ActiveServiceDirectory

The *ActiveServiceDirectory* realizes the Discovery Service role, providing a directory that maps from Web Service URI to a set of active endpoints of the Web Service, each described using *ServiceInstanceDescriptor* (shown in Fig. 35 below).



**Fig. 35 A ServiceInstanceDescriptor describes a single deployed Web Service endpoint**

A *ServiceInstanceDescriptor* describes a single deployed endpoint of a Web Service. It includes a *HostDescriptor* identifying the endpoint's *ServiceHost*, a *ServiceImplementationDescriptor* describing the implementation behind the deployed Web

Service, the URL of the Web Service endpoint, and a timestamp indicating when it was deployed. These details can be used by Service Consumers to select a preferred endpoint from the set of all deployed instances of the desired Web Service. A unique instance identifier is also included in the descriptor and is used in situations where undeployment is requested on a Service Host with multiple instances of the same Web Service.

If there are no active endpoints of a Web Service the `ActiveServiceDirectory` is responsible for initiating the autodeployment of a new endpoint by contacting the Manager of the Web Service. By proactively addressing this ‘directory fault’, the `ActiveServiceDirectory` provides a crucial mechanism for efficient resource utilization by allowing for endpoints to be deployed in response to demand and remain ‘available’ without any pre-provisioned capacity. As detailed in the next chapter, Managers themselves are realized as Web Services and are published and deployed using the same mechanisms as the Web Services they manage. When demand drops to zero and a Manager undeploys the final endpoint of the Web Service, the Manager will no longer receive data from the Service Hosts on which the endpoints were previously deployed. Due to lack of demand (this time from Service Host to Manager) the Manager itself will be undeployed and the newly-released hosting capacity added back to the shared resource pool. This recursive management model enables an infrastructure with no demand for its Web Services to progressively wind down its deployments until it reaches a level of zero resource consumption.

#### 3.3.3.5 Point of Presence

Service Consumers bind to and invoke Web Service operations on a local entity called the Point of Presence (POP). The POP is a transparent indirection mechanism – a proxy – that interacts with the infrastructure on behalf of the Service Consumer. When it receives an invocation request from a consumer agent application, the POP is responsible for using the `ActiveServiceDirectory` to resolve an active endpoint of the requested Web Service, invoking the requested operation on the endpoint, and returning any results to the requesting consumer agent application. It is responsible for transparently detecting and recovering from the failure of Web Service endpoints and the hosts on which they are deployed.

### 3.3.4 ARCHITECTURAL ENTITIES

#### 3.3.4.1 Service Library

The Service Library serves as a repository for Web Service implementations. The Service Library provides a mapping from Web Service URI to the set of `ServiceImplementationDescriptors` describing implementations of the Web Service, each of which may be written in a different programming language for a different target deployment container. Publishers use the Service Library to publish and unpublish Web Service implementations. Managers use the lookup operations of the Service Library during the Planning phase in order to select an implementation for deployment on a Service Host.



Fig. 36 The `ServiceLibrary` interface through which Publishers and Managers interact with the repository.

#### 3.3.4.2 Host Directory

The `HostDirectory` is a repository which holds a mapping from `PublisherID` to a set of `HostDescriptors` describing the `ServiceHosts` registered as willing to host Web Services from the specified Publisher. Managers use the Host Directory during the Planning stage of deployment in order to select a Service Host on which to deploy a new Web Service endpoint.



Fig. 37 The `HostDirectory` interface is used by `ServiceHosts` and Managers to interact with the `HostDirectory` repository.

### 3.4 INTRA-ARCHITECTURE INTERACTIONS

#### 3.4.1 SERVICE CONSUMER TO LOCAL POINT OF PRESENCE (POP)

Service Consumers do not interact directly with Web Service endpoints. In order to use a Web Service a Service Consumer always binds to and invokes operations upon an entity known as the ‘local point of presence’, shown in Fig. 38. POPs are co-located with Service Consumers; consumer agent software is written to statically bind to the URI of a Web Service, prefixed with the protocol, hostname and port of a well-known local URI (e.g. `http://localhost/WebServiceURI`) – thus creating a URL as per the W3C standard.

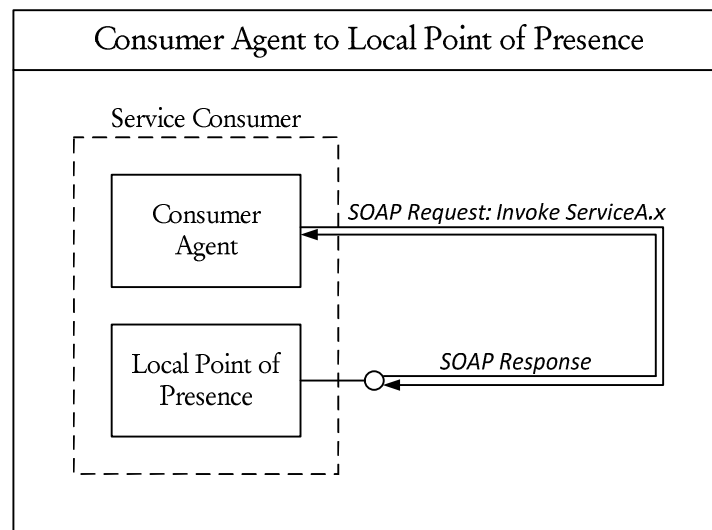


Fig. 38 Consumer Agent binding to and invoking Web Service operations on the local point of presence.

The local point of presence entity transparently locates, binds to, and invokes the operations of Web Services on behalf of the Service Consumer (shown in Fig. 39 below). It is responsible for detecting and attempting to recover from the failure of Web Service endpoints and the hosts on which they are deployed (between Fig. 39 steps 5 and 7). As with all architectural entities, the local point of presence exerts a ‘best effort’ to complete the requested operation. All errors returned by the local point of presence are non-recoverable, indicating that, given the currently available resources of the system, it is simply not possible to carry out the requested operation.

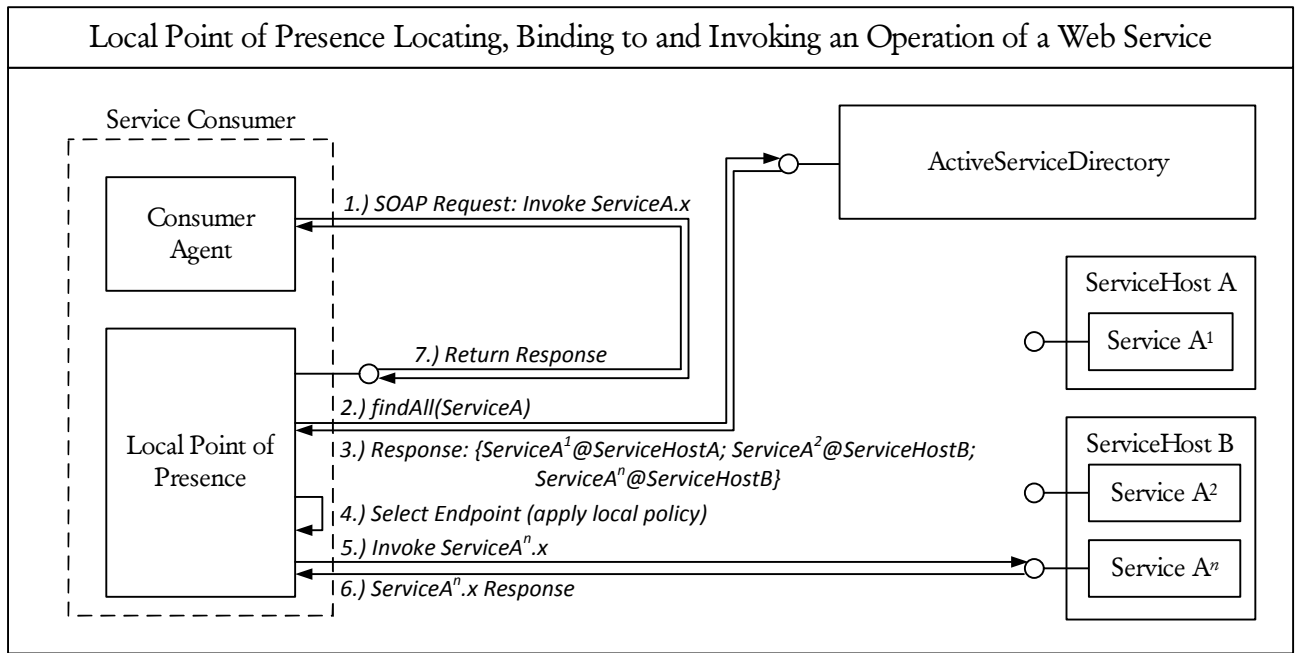


Fig. 39 The seven basic steps for invoking a Web Service operation via the local Point of Presence

The diagram in Fig. 39 above outlines the process of a Service Consumer invoking operation 'x' of the Web Service identified by the URI 'ServiceA'. The consumer agent application binds to and invokes the operation on the POP (step 1), which then must locate an endpoint of the requested Web Service. The POP sends a lookup request to the ActiveServiceDirectory by invoking its 'findAll' operation with the URI 'ServiceA' as a parameter (step 2) and receives back a list of ServiceInstanceDescriptors describing the currently active endpoints of ServiceA (step 3). The POP applies local policy to select which endpoint to use (step 4) before connecting to the endpoint and invoking operation 'x' on behalf of the Service Consumer (step 5). The results of the operation are returned to the POP (step 6) and finally returned to the consumer agent application (step 7).

It is important to note that Service Consumers, as commonly understood, are not the only entities in the architecture which use the facilities provided by a local point of presence. Entities commonly considered to be solely provider agents – such as Managers and Service Hosts – also use a local point of presence to carry out intra-architecture interactions. By factoring out the location and failure recovery mechanisms of both intra- and extra-architecture Web Service interactions, both consumers and providers of Web Services are freed from the responsibility of implementing their own proprietary mechanisms for these tasks. Thus a single application of effort toward the development of a reliable, robust, and efficient point of presence implementation will benefit all actors and entities which both use *and provide* the infrastructure.

### 3.4.2 PUBLISHER TO SERVICE LIBRARY

Publishers are Service Consumers who interact with the ServiceLibrary in order to publish and unpublish implementation of a Web Service. As with all Service Consumers in the architecture, Publishers do not interact directly with any external Web Services, instead invoking the desired operations via the local point of presence.

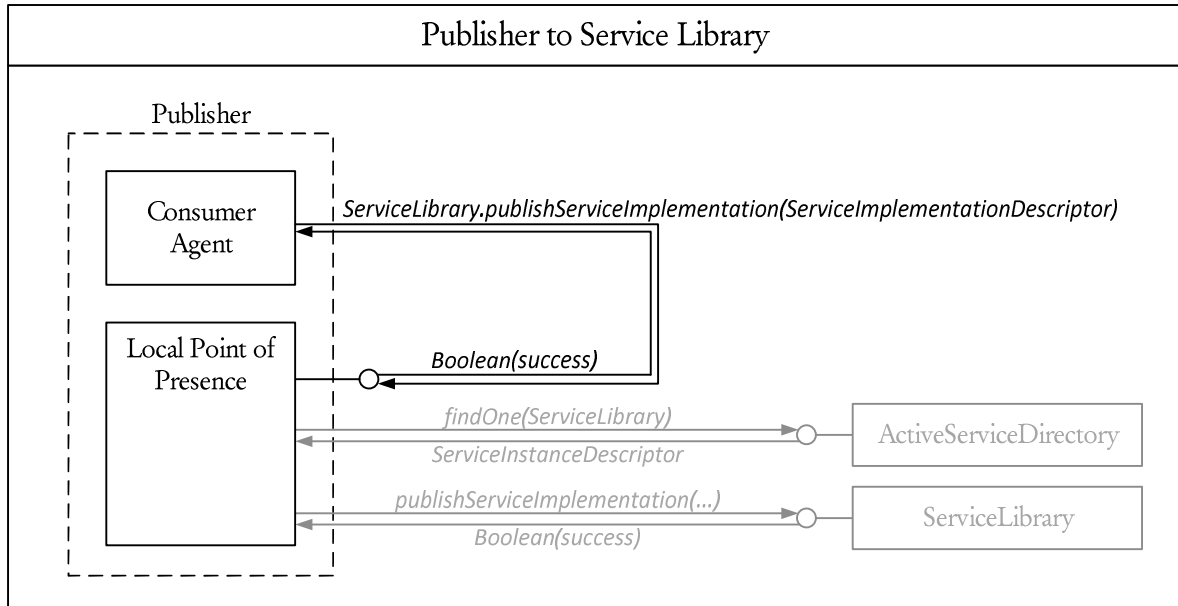
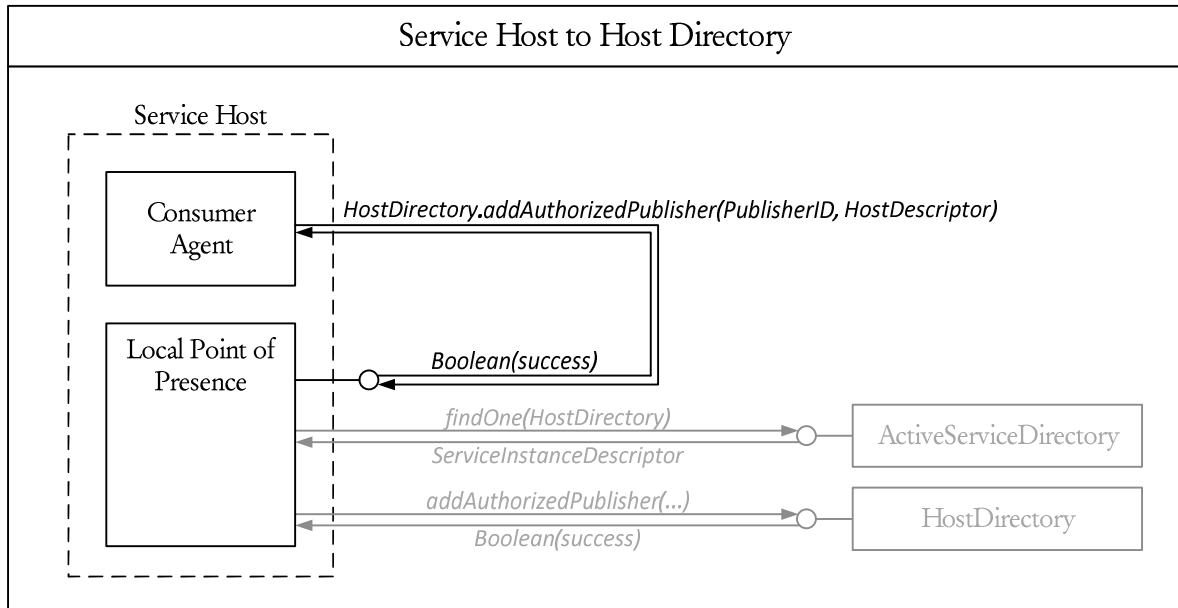


Fig. 40 Interaction between a Publisher and the ServiceLibrary in order to publish a Web Service implementation.

In order to publish an implementation of a Web Service (described with a *ServiceImplementationDescriptor*), Publishers bind to the local point of presence and invoke the 'publishServiceImplementation' operation of the *ServiceLibrary* with the desired *ServiceImplementationDescriptor*, as shown in Fig. 40 above. Once an implementation has been published it becomes immediately available for use, able to be deployed onto capable *ServiceHosts* as necessary to meet demand.

### 3.4.3 SERVICE HOST TO HOST DIRECTORY

While ServiceHosts contribute to the role of Service Provider, they may also act as Service Consumers. In order to register themselves with the Host Directory, ServiceHosts first describe themselves using a HostDescriptor, bind to the local point of presence, and invoke the 'addAuthorizedPublisher' operation of the HostDirectory, as shown in Fig. 41.

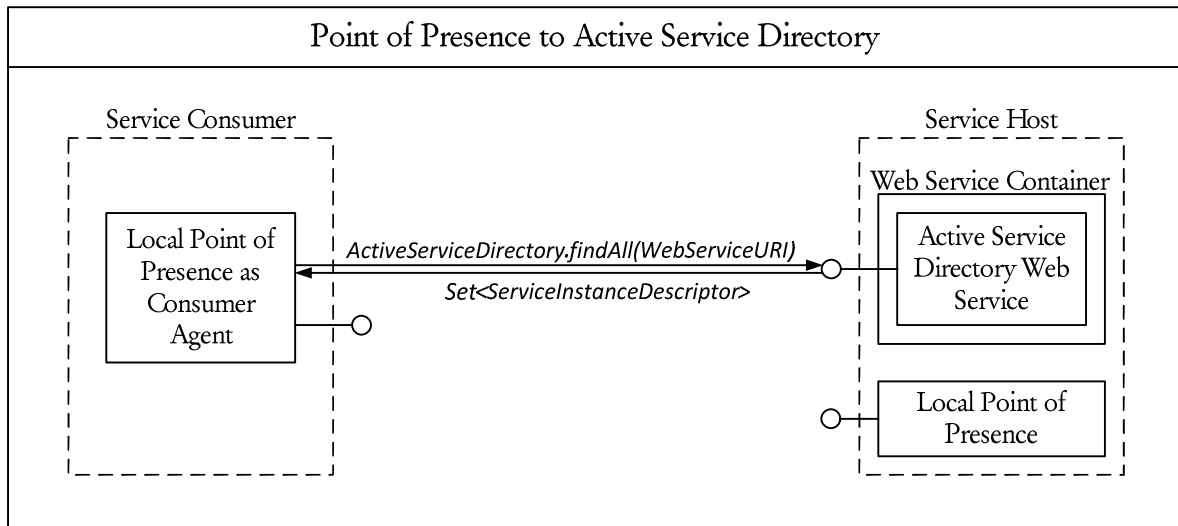


**Fig. 41 ServiceHosts register their HostDescriptor with the HostDirectory, indicating the Publishers whose Web Service implementations they are willing to deploy and host.**

By operating through the local point of presence the ServiceHost registration process is thus performed with the same robust invocation procedures provided by the infrastructure to all Service Consumers. Upon registration, a Service Host's HostDescriptor is added to a shared pool of hosting resources, ready to be consumed as necessary to meet demand.

### 3.4.4 POP TO ACTIVE SERVICE DIRECTORY

The point of presence is a transparent endpoint resolution and failure recovery mechanism which locates, binds to, and invokes operations upon Web Services on behalf of the Service Consumer. In order to locate an active instance of a Web Service the point of presence interacts with a Web Service called the 'ActiveServiceDirectory', shown in Fig. 42 below, invoking either its 'findOne' or 'findAll' operations with the URI of the desired Web Service as a parameter.



**Fig. 42 Point of Presence interacting with the ActiveServiceDirectory in order to retrieve a set of all known endpoints of a Web Service**

In order to participate in an instantiation of the architecture, the local point of presence must have a-priori knowledge of at least one ActiveServiceDirectory endpoint. Any alternative ActiveServiceDirectory endpoints are listed in the ActiveServiceDirectory under the Web Service URI 'ActiveServiceDirectory' and can be located using the same `findOne` and `findAll` operations. It is recommended that the POP periodically retrieve and store a list of alternative ActiveServiceDirectory endpoints to use in the event of failure.

#### 3.4.5 ACTIVE SERVICE DIRECTORY TO ACTIVE SERVICE DIRECTORY

When the ActiveServiceDirectory receives a lookup request for a Web Service with zero active endpoints, it must locate an instance of that Web Service's Manager so that an endpoint may be deployed. The ActiveServiceDirectory utilizes its own lookup facilities by binding to its local point of presence and invoking the ActiveServiceDirectory 'findOne' or 'findAll' operation using the URI of the Managing entity responsible for the originally requested Web Service. This URI is constructed by concatenating the well-known 'Manager' URI to the URI of the requested Web Service in the pattern 'WebServiceURI\_ManagerURI' (shown below in Fig. 43). The ActiveServiceDirectory is returned a ServiceInstanceDescriptor describing a Web Service endpoint which implements the 'IManager' interface described earlier.

Although the ActiveServiceDirectory is looking up a service in the ActiveServiceDirectory, there are no restrictions on how the ActiveServiceDirectory Web Service may be implemented, and thus no guarantee that the required service listing will be held by the requesting ActiveServiceDirectory. For this reason it is important that the operation be invoked using the POP, rather than the ActiveServiceDirectory simply looking in its local map.

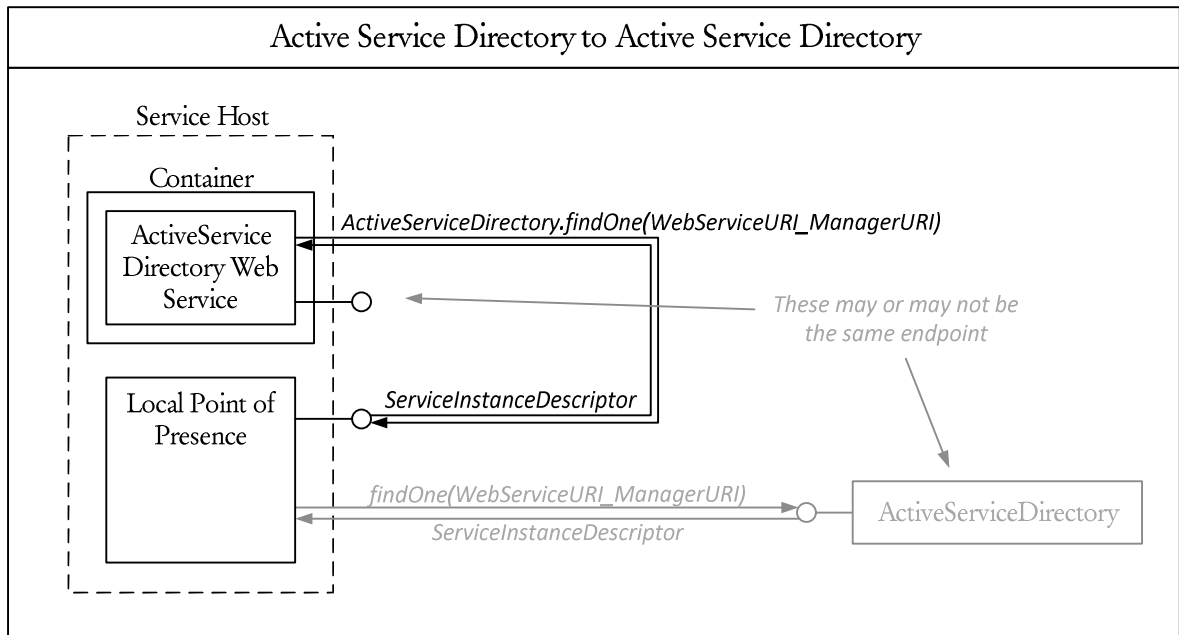


Fig. 43 ActiveServiceDirectory invoking the findOne operation of the ActiveServiceDirectory via the local POP.

### 3.4.6 ACTIVE SERVICE DIRECTORY TO MANAGER

In order to request deployment of the first endpoint of a Web Service, the ActiveServiceDirectory invokes the 'requestFirstInstance' operation of the requested Web Service's managing entity. As shown in Fig. 44 below, the URI of the requested Web Service is used as a parameter of this call which returns a boolean value indicating whether deployment was successful.

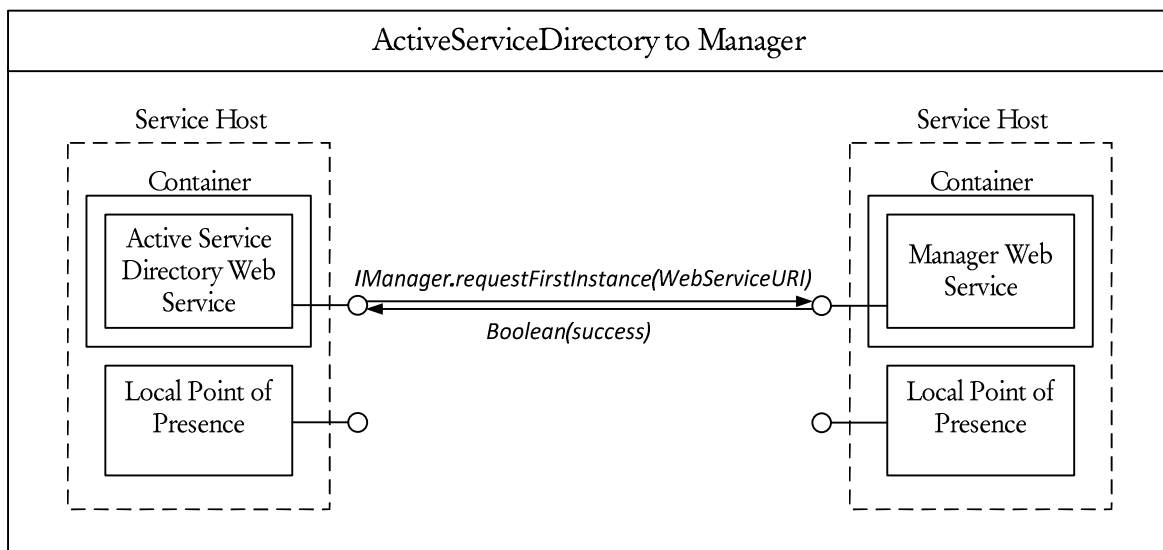


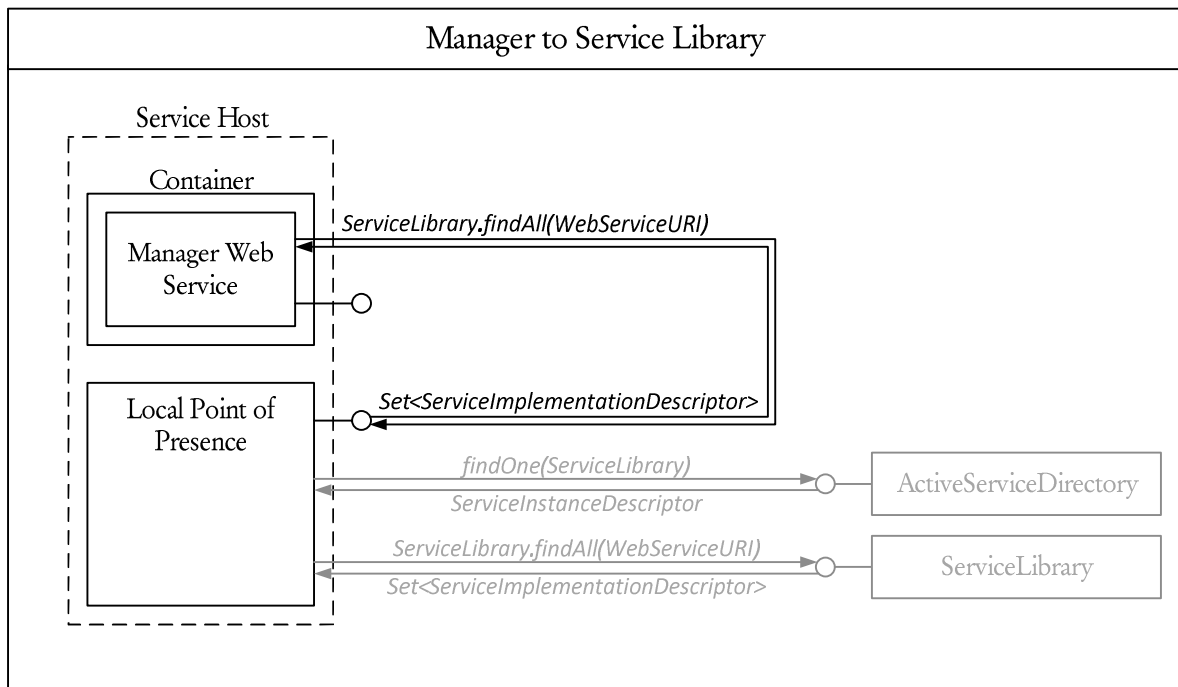
Fig. 44 The ActiveServiceDirectory invoking the 'requestFirstInstance' operation of an IManager Web Service.

As a Manager is responsible for maintaining the directory records of the Web Service they manage, Managers must insert a record of any endpoint they deploy into the ActiveServiceDirectory. Managers perform this operation in-band with the 'requestFirstInstance' invocation – returning 'true' if the entire deployment and registration process completed successfully, or 'false' otherwise. If the registration procedure is instead performed out of band there is a risk of a 'resource leak': a Manager returning 'true' after deployment but failing to register the new endpoint in the ActiveServiceDirectory will cause the newly deployed endpoint to become 'stranded', permanently consuming resources without being of use.

The 'requestFirstInstance' operation returning 'false' indicates that, given the current state of the system, it is simply not possible to deploy a new instance of the Web Service. This situation represents an unrecoverable error which is returned to the requesting entity (i.e. the Service Consumer). If the operation returns 'true' the ActiveServiceDirectory re-performs the originally-requested lookup operation (as described in section 0) and returns the newly-inserted ServiceInstanceDescriptor to the requesting entity.

### 3.4.7 MANAGER TO SERVICE LIBRARY

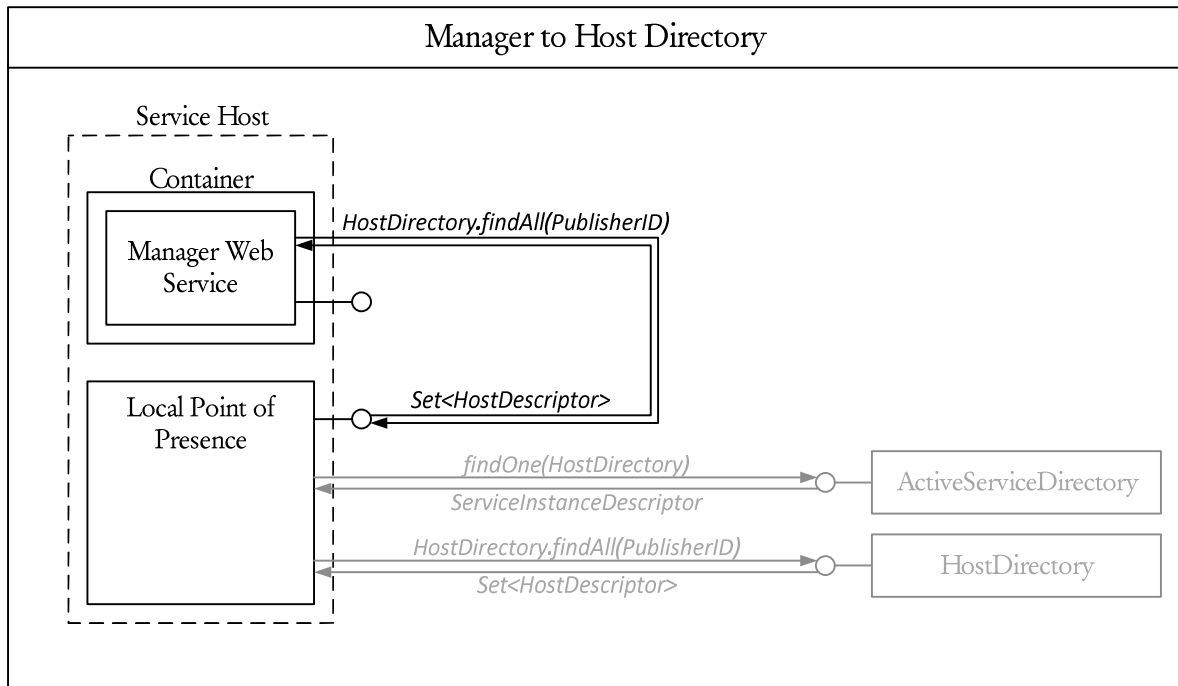
During the deployment process, Managers invoke the operations of the Service Library in order to retrieve a set of implementations of the Web Service being deployed (shown below in Fig. 45). These implementations are each described with a *ServiceImplementationDescriptor*; the elements of this descriptor can be used later by the Manager in order to select a suitable candidate implementation for deployment.



**Fig. 45 Managers invoke the operations of the ServiceLibrary during deployment in order to retrieve the set of implementations of a Web Service.**

### 3.4.8 MANAGER TO HOST DIRECTORY

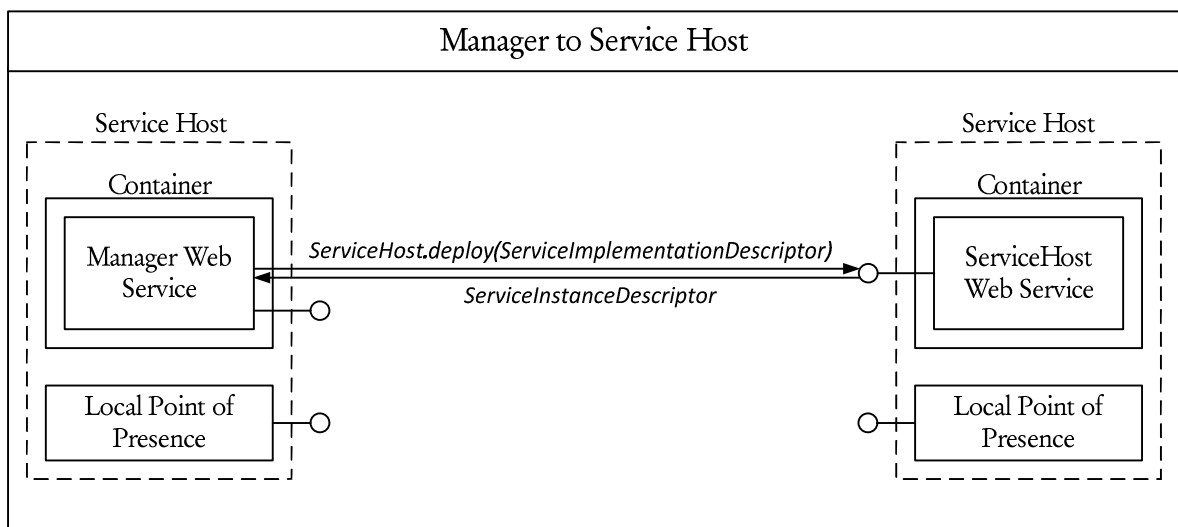
Much like their interaction with the ServiceLibrary, Managers invoke the operations of the HostDirectory during the deployment process (shown in Fig. 46 below). Managers use the HostDirectory to retrieve a set of HostDescriptors identifying ServiceHosts willing to deploy Web Service implementations published by the indicated publisher (identified with a unique PublisherID). Managers compare the *ServiceImplementationDescriptors* retrieved from the ServiceLibrary with the *HostDescriptors* retrieved from the HostDirectory in order to craft suitable candidate deployment plans based on local policy. Once a plan is selected the Manager may contact the selected ServiceHost and initiate the deployment process, as detailed next.



**Fig. 46 Managers interact with the HostDirectory during the deployment process in order to retrieve a list of ServiceHosts willing to deploy Web Service implementations published by various Publishers.**

### 3.4.9 MANAGER TO SERVICE HOST

After selecting a suitable Web Service implementation (described with a *ServiceImplementationDescriptor*) for deployment on a selected Service Host (described with a *HostDescriptor*) a Manager binds to the ServiceHost included in the *HostDescriptor* and invokes the 'deploy' operation directly (shown below in Fig. 47). Upon successful deployment the Service Host returns the Manager a *ServiceInstanceDescriptor* describing the newly deployed Web Service endpoint. The Manager then registers this *ServiceInstanceDescriptor* in the *ActiveServiceDirectory*, completing the deployment process.



**Fig. 47 Managers initiate the deployment of a Web Service endpoint on a ServiceHost by invoking the 'deploy' operation with a *ServiceImplementationDescriptor* describing the Web Service implementation to be deployed**

### 3.4.10 SERVICE HOST TO MANAGER

For each Web Service endpoint deployed within their domain (i.e. the containers under their control), a Service Host must report usage data to that Web Service's managing entity. Shown in Fig. 48 below, the Service Host binds to the local point of presence and invokes the 'reportUsageData' operation of the Web Service identified by the concatenation of a well-known 'Manager' URI to the URI of the Web Service for which data is being reported (i.e. 'WebServiceURI\_ManagerURI').

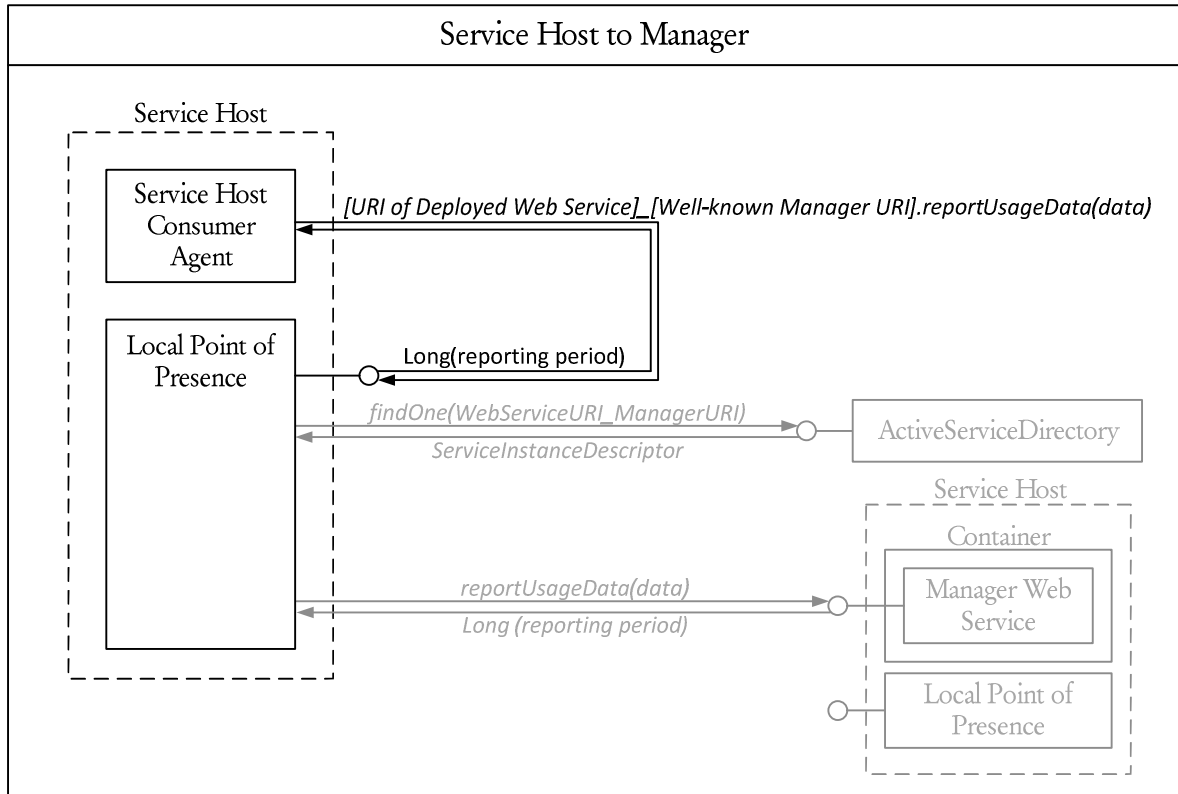
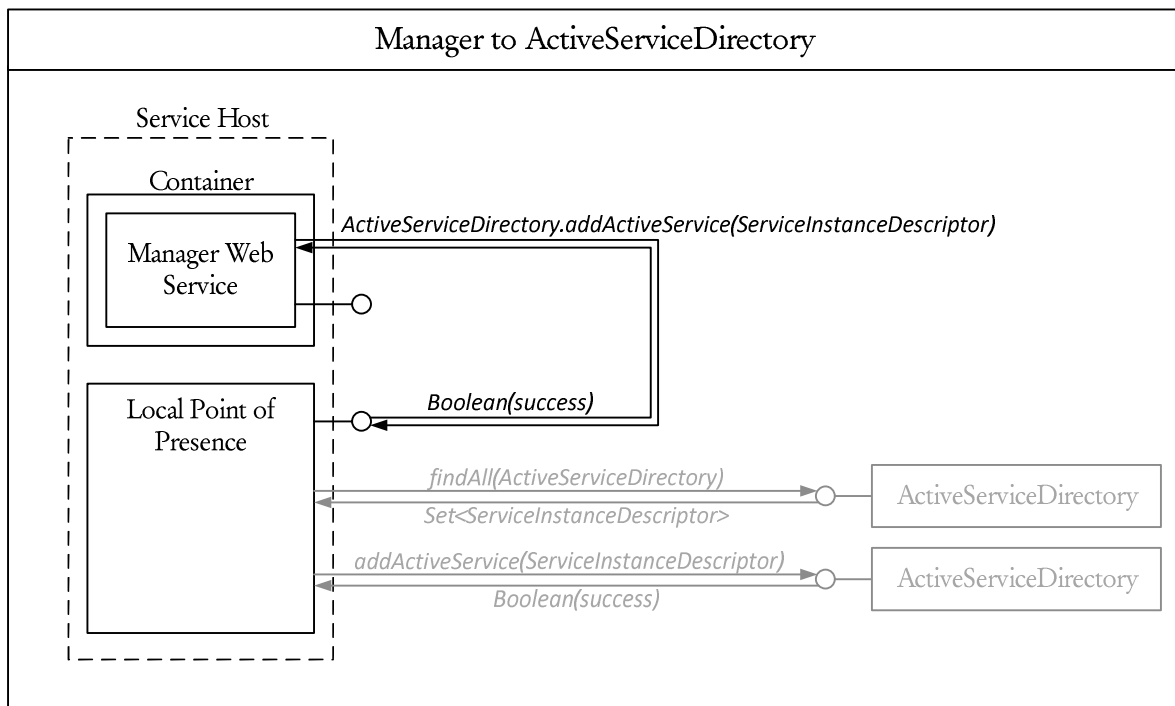


Fig. 48 ServiceHosts report usage data for each Web Service deployed within their domain by invoking the 'reportUsageData' operation of each Web Service's Manager.

The IManager 'reportUsageData' operation returns a numerical value indicating the length of time the ServiceHost should wait before next reporting usage data for this particular Web Service. A ServiceHost's local policy may dictate that usage data be returned earlier than requested (e.g. due to local resource constraints, such as working memory); returning data significantly later than the requested period may indicate to the Manager that there is a problem with the ServiceHost – information which a Manager may act upon in order to effectively manage the availability of its service.

### 3.4.11 MANAGER TO ACTIVE SERVICE DIRECTORY

When a Manager of a Web Service successfully deploys or undeploys endpoints of that Web Service it must add or remove the endpoint reference in the ActiveServiceDirectory. Managers bind to the local point of presence and invoke the 'addActiveService' or 'removeActiveService' operation of the ActiveServiceDirectory Web Service using the relevant ServiceInstanceDescriptor (shown below in Fig. 49).



**Fig. 49 Managers use the operations of the ActiveServiceDirectory to maintain the entries of the Web Service they manage.**

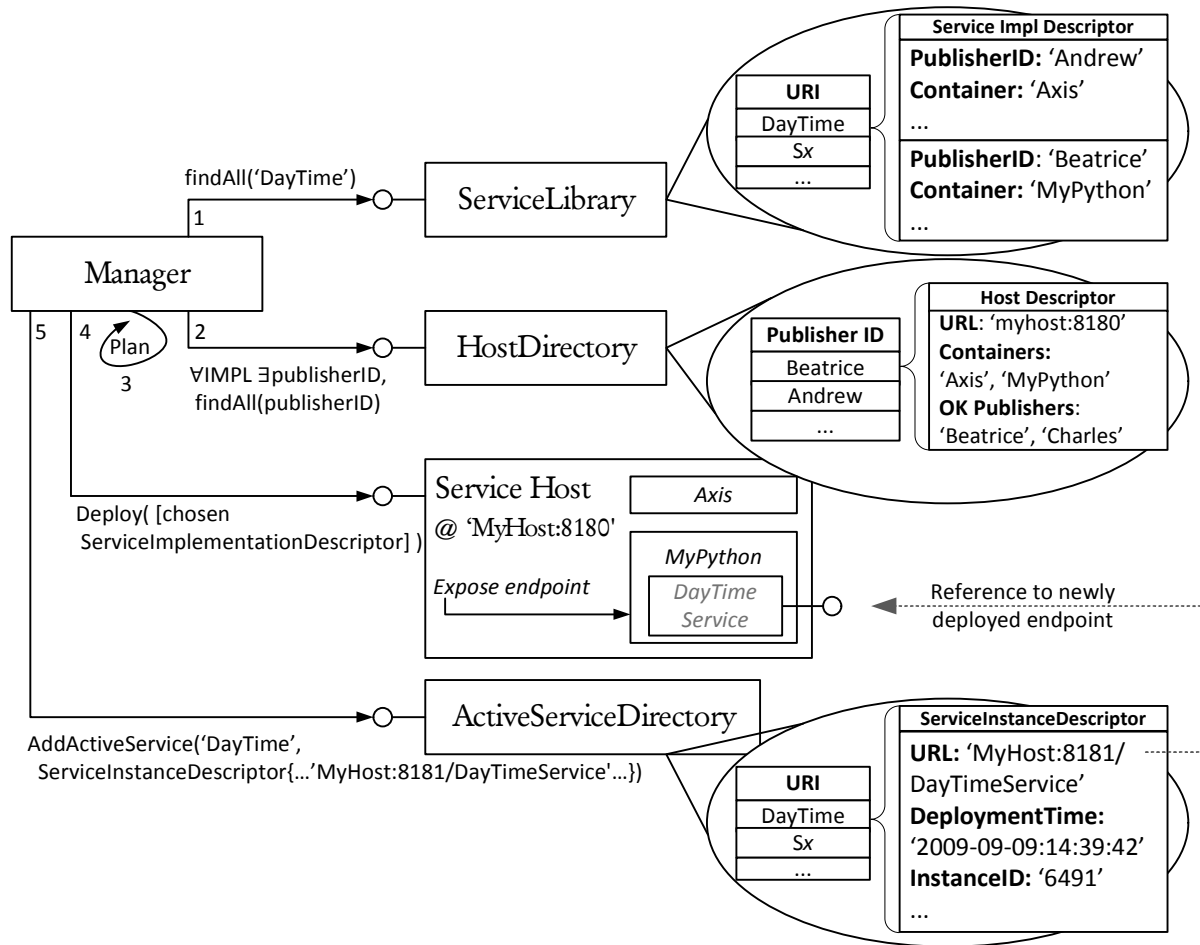
Managers are responsible for maintaining the ActiveServiceDirectory entries of the Web Service they manage. Of all entities fulfilling the responsibilities of the Service Provider, Managers are deemed to be most interested in maintaining an accurate public record of the Web Service they each manage as it provides them with accurate usage data from which they may make more informed decisions in the fulfillment of their responsibility to manage the provisioning level of a Web Service. If the ActiveServiceDirectory entries are not maintained then active endpoints of the Web Service may not be available to Service Consumers (wasting resources due to idle deployments) and the effectiveness of load-balancing techniques will be diminished. Alternatively, if a list including many failed endpoints is returned to a ServiceConsumer's local point of presence, the requesting consumer agent application may be significantly slowed as the point of presence tries each failed endpoint in turn.

### 3.5 PROCEDURES

This section presents the high-level procedures of the architecture (e.g. deployment) by using the previously described intra-architecture interactions as single coarse-grain steps. The examples are presented in step-by-step walkthroughs of the procedures together with supporting diagrams. The following sub-sections detail the intra-architecture interactions involved in the processes of endpoint deployment, service invocation, and demand-driven dynamic deployment, using the provision and invocation of a 'DayTime' Web Service as an example.

#### 3.5.1 *ENDPOINT DEPLOYMENT*

The deployment of a new Web Service endpoint is always carried out by a Manager. As shown in Fig. 50 below and described in the paragraphs to follow, the Manager interacts with the ServiceLibrary, HostDirectory, one or more ServiceHosts, and the ActiveServiceDirectory in order to make a new 'DayTime' Web Service endpoint available for use.



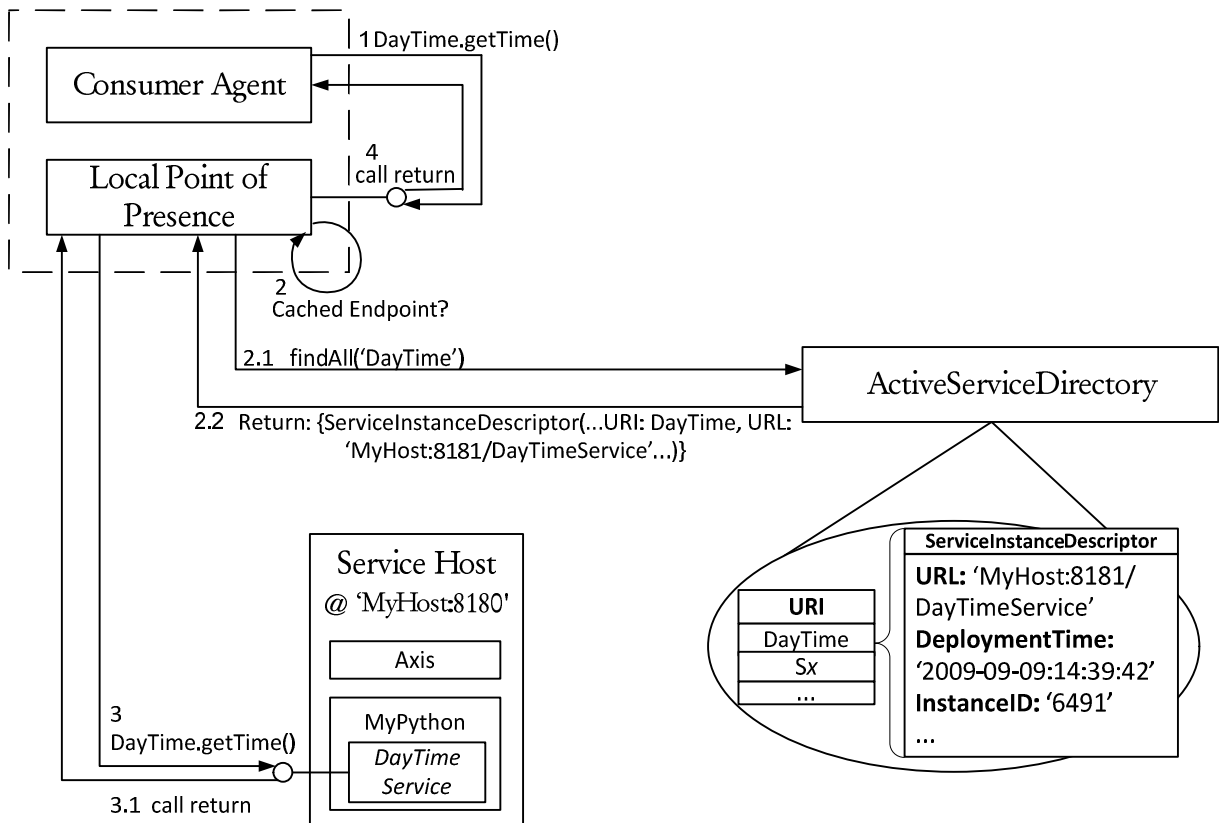
**Fig. 50 Managers enact deployment by querying the ServiceLibrary and HostDirectory, creating a deployment plan, requesting deployment on a ServiceHost, and registering any newly deployed endpoint with the ActiveServiceDirectory.**

In order to deploy a new Web Service endpoint a Manager first creates and selects a valid deployment plan (recall that a deployment plan consists of a ServiceImplementationDescriptor whose 'requirements' are matched by the 'capabilities' described in a HostDescriptor). Shown in Fig. 50, above, the Manager begins by contacting the ServiceLibrary and retrieves the set of all published implementations of the 'DayTime' Web Service (Step 1). Each implementation returned is described with a ServiceImplementationDescriptor, each of which has a 'PublisherID' element. For each implementation, the Manager contacts the HostDirectory using the PublisherID (Step 2) and retrieves the set of ServiceHosts willing to deploy provider agent applications written by the specified publisher. The Manager then creates a set of valid deployment plans and selects one from amongst the candidates based on local policy (Step 3). Next the Manager executes the deployment plan, contacting the ServiceHost described by the selected HostDescriptor and invoking its 'deploy' operation with the selected ServiceImplementationDescriptor (Step 4). Upon

successful deployment the target ServiceHost returns the Manager a ServiceInstanceDescriptor describing the endpoint. The Manager completes deployment by inserting the ServiceInstanceDescriptor into the ActiveServiceDirectory (Step 5). Once the newly deployed endpoint is listed in the ActiveServiceDirectory it is deemed to be deployed, ready to be located and its operations invoked by Service Consumers.

### 3.5.2 SERVICE INVOCATION

Service Consumers bind to and invoke Web Service operations on the local point of presence. Once an invocation request is received, the local point of presence is responsible for locating an active endpoint of the target Web Service and robustly invoking the requested operation on behalf of the Service Consumer. The process of invoking the 'getTime' operation of the 'DayTime' Web Service is shown below in Fig. 51 and described in the paragraphs that follow.



**Fig. 51** The process of invoking the 'getTime' operation of the 'DayTime' Web Service using the Local Point of Presence, including location of active endpoints of the Web Service using the ActiveServiceDirectory, selecting an endpoint from the list of available endpoints, invoking the operation on a remote host, and finally return of the results to the Service Consumer

Upon receiving an invocation request from a consumer agent application (Step 1) the local point of presence first extracts the desired Web Service URI from the incoming request. If the point of presence implements an endpoint cache it may first check locally for a previously retrieved list of endpoints (Step 2). If no such entries exist, a new list must be retrieved using the

ActiveServiceDirectory (Step 2.1). Once a list of endpoints is retrieved the point of presence selects one for use according to local selection policy (Step 2.2). The point of presence must then prepare the invocation request (including possible modifications to the message), bind to the selected Web Service endpoint, and forward the invocation request (Step 3). If the invocation fails for any reason the same procedure is attempted for the remainder of the active endpoints (Steps 2 and 3) and, if all endpoints prove unavailable, a generic error returned to the Service Consumer (as the result in Step 4). If the invocation is successful the resulting response is first prepared (again, possibly requiring modification to the message) before being returned to the Service Consumer (Step 4) and concluding the invocation process.

### 3.5.3 DEMAND-DRIVEN DYNAMIC DEPLOYMENT

In a dynamic, demand-driven system, endpoint deployment may occur in response to an existing or anticipated event – in this case, the invocation of an operation of a Web Service for which there are currently zero active endpoints. The process presented below will be familiar – it is a composition of two processes presented previously: Web Service endpoint deployment and the invocation of a deployed endpoint's operations.

When the ActiveServiceDirectory receives a request for all endpoints of the 'DayTime' Web Service, it finds there are none currently deployed. In this case it is up to the ActiveServiceDirectory to initiate (though not enact) the deployment of the first instance of the requested Web Service by contacting that Web Service's Manager and invoking the 'requestFirstInstance' operation. At this point the deployment process is executed identically as previously presented; when the Manager returns from the 'requestFirstInstance' operation the ActiveServiceDirectory can re-perform the lookup and return either a reference to the newly deployed endpoint, or a meaningful error indicating that, given the currently available resources in the infrastructure, it is not possible to fulfill the request at the given time.

The process represented in Fig. 52 below details the steps from the initial Service Consumer request through the demand-driven dynamic deployment of a 'DayTime' endpoint, its subsequent registration in the ActiveServiceDirectory, the return of a list of active 'DayTime' endpoints to the local point of presence, the performance of the original invocation request on the newly deployed endpoint and, finally, the return of results to the Service Consumer.

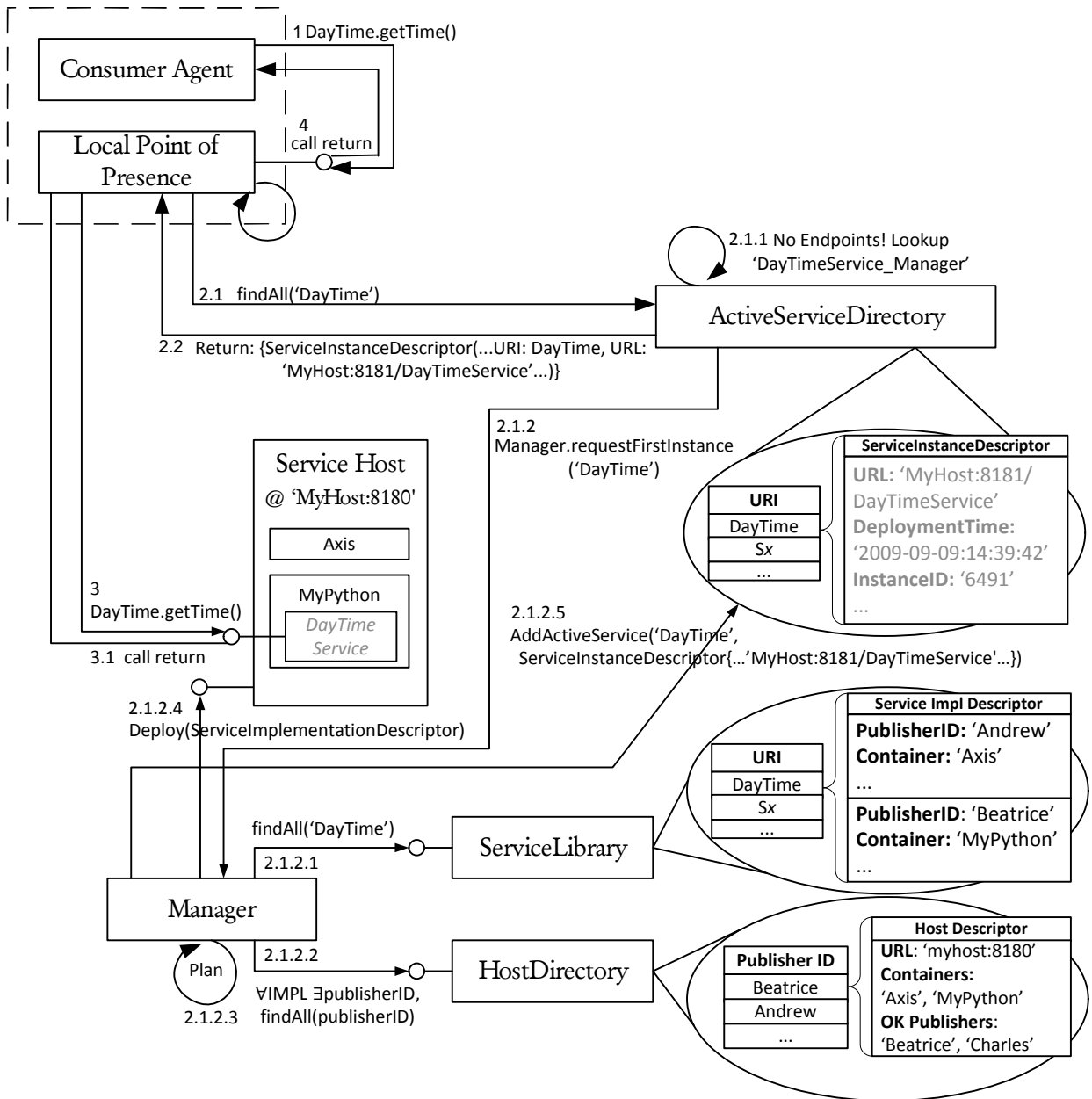


Fig. 52 Demand-driven dynamic deployment of the first 'DayTime' Web Service endpoint. Deployment is carried out in response to demand for a service which initially has no deployed endpoints.

It should be mentioned again at this point that the Manager for a Web Service does not need to be currently deployed in order for that Web Service to be deployable. If implemented as a Web Service – and with generic and/or custom, service-specific Manager implementations published in the ServiceLibrary – the exact same deployment mechanisms can be used to deploy and manage a Web Service's Manager as those Managers use to manage their particular Web Service. This recursive, collapsible model enables an infrastructure to be 'wound up' to provide enough Web Service endpoints to meet demand, and then 'wound down' to a state of zero resource consumption when demand falls to zero. (Note that the infrastructure must start somewhere

and, even in a period of zero demand, a passive bootstrapping process [or similar mechanism] capable of deploying the first 'ManagerManager' is necessary; resource consumption can thus never be said to be 'completely' zero).

In pursuit of a design where an idle infrastructure consumes low levels of hosting resources, the described mechanisms and framework for the publication, invocation, deployment and management of Web Services can also be used for the provision of the infrastructure components themselves. Detailed in the next chapter, entitled "Reference Implementation", the `ActiveServiceDirectory`, `HostDirectory`, and `ServiceLibrary` are all published as Web Services, each deployed and managed by a generic Manager capable of detecting and reacting to changes in consumer demand, deploying and undeploying endpoints as necessary to balance resource consumption while providing Service Consumers with highly available Web Services. The properties provided to Service Consumer (reliability, robust invocation techniques, highly available Web Services) themselves become properties of the architecture itself – a reflective design that both simplifies and improves the quality of interactions for all participants in the Web Service lifecycle.

## 4 REFERENCE IMPLEMENTATION

### 4.1 INTRODUCTION

This chapter details a reference implementation of the presented Web Services architecture using the development, provision, and consumption of a ‘TimeService’ Web Service as an example. It begins with a system overview followed by an introduction to the technology used to implement the individual infrastructure components. It continues with a section dedicated to each newly defined actor and details the implementation techniques utilized to develop mechanisms which enable them to fulfill the responsibilities of their individual roles. Each section includes diagrams representing the interactions between these various actors and entities, provides examples of the decision-making processes which take place during these interactions, and identifies points where custom policy decisions may be implemented.

### 4.2 SYSTEM OVERVIEW

The following sections describe the implementation of a distributed, service-oriented system of loosely-coupled autonomous components. These components work together to provide a robust infrastructure for the reliable, on-demand provision and consumption of Web Services. The system is designed to be both flexible and simple to use, and aims to minimize complexity for all users by isolating and consuming complex and redundant tasks into infrastructure components. It automates the majority of the role of Service Provider and completely automates Web Service deployment and management. It allows (but does not require) custom management and deployment techniques to be implemented and used in lieu of the generic techniques, as desired by Publishers, on a service-by-service basis. Importantly, the infrastructure components are themselves realized as Web Services and published into the infrastructure: after a bootstrapping process is used to deploy a ServiceLibrary and ActiveServiceDirectory instance, the generic deployment, monitoring, management and maintenance mechanisms provided by the infrastructure are utilized, automatically, for the provision and management of the infrastructure itself.

As described in the previous chapter, communication between infrastructural components is also carried out using the same mechanisms used to deliver services to Consumers (i.e. the POP). Thus a single application of effort towards a robust and reliable point of presence implementation benefits Service Consumers and Service Providers simultaneously and introduces the same properties of reliability and robustness to all intra-infrastructure communications, and thus to the system as a whole.

The described system has been implemented with clear boundaries between independent functional components – not simply as an exercise in good design, but in order to more easily isolate components during testing. The system described in this chapter will be used to validate the claims made of the architecture and evaluate the efficacy of its mechanisms through a rigorous set of experiments presented in Chapter 5: ‘Evaluation’.

### 4.3 DEVELOPMENT TECHNOLOGIES

The presented implementation has been completely developed in Java and utilizes the Apache Software Foundation’s Tomcat [99] and Axis [55] deployment environments. Although it is not utilized in the implementation of the architecture, the Web Service middleware platform RAFDA [100, 101] is provided as an additional example deployment container.

### 4.4 DEVELOPING WEB SERVICES

Web Services implementations to be published into an instantiation of the presented architecture do not need to be developed with any more special considerations than if they were to be deployed by hand on an individual machine. Implementations are encouraged to provide idempotent operations and utilize mechanisms to maintain shared state between different instances of the service.

The Java source code for an example ‘TimeService’ Web Service implementation is shown below in Fig. 53 and is developed for the Apache Axis ‘Hot Deploy’ [102] deployment container. This container monitors a directory on the host machine and automatically deploys Web Service endpoints from Java source files written into the directory. The container exposes a Web Service interface based on the declared public methods of the Java class in each source file. For the ‘TimeService’ these are: ‘getCurrentTime:long’ and ‘getCurrentTime(int timezoneOffset):long’.

```
public class TimeService {

    /** Returns the current time, represented as milliseconds between
     *  the current time and midnight, January 1, 1970 UTC
     */
    public long getCurrentTime(){
        return System.currentTimeMillis();
    }

    /** Returns the current time in the timezone given as an offset
     *  from GMT (e.g. -5 for New York) in ms between
     *  the current time and midnight, January 1, 1970 UTC
     */
    public long getCurrentTime(int timezoneOffset){
        ...
    }
}
```

**Fig. 53** Java source file for the 'TimeService' class.

Using a tool such as Java2WSDL on the Java source file of the 'TimeService' class produces a WSDL document such as the one in Fig. 54 below. Alternatively, WSDL files may be retrieved from any exposed Web Service by concatenating "?WSDL" to the Web Service URL (e.g. 'http://www.example.org/TimeService?WSDL'). In the traditional Web Services model, Service Consumers bind to the URL included in the 'location' attribute of the WSDL document's 'address' element (found towards the bottom of Fig. 54). The treatment of the WSDL document in general, its referenced namespaces, and this 'location' attribute in particular will be discussed in upcoming sections detailing the implementation of the POP.

```

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://xyz.com:8080/rfmthesis/services/examples/"
xmlns:apache:soap="http://xml.apache.org/xml-soap" xmlns:impl="http://xyz.com:8080/rfmthesis/services/
examples/" xmlns:intf="http://xyz.com:8080/rfmthesis/services/examples/" xmlns:soapenc="http://
schemas.xmlsoap.org/soap/encoding/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:wsdlsoap="http://
schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <wsdl:message name="getCurrentTimeResponse1">
    <wsdl:part name="getCurrentTimeReturn" type="xsd:long" />
  </wsdl:message>
  <wsdl:message name="getCurrentTimeRequest">
  </wsdl:message>
  <wsdl:message name="getCurrentTimeResponse">
    <wsdl:part name="getCurrentTimeReturn" type="xsd:long" />
  </wsdl:message>
  <wsdl:message name="getCurrentTimeRequest1">
    <wsdl:part name="timezoneOffset" type="xsd:int" />
  </wsdl:message>
  <wsdl:portType name="TimeService">
    <wsdl:operation name="getCurrentTime">
      <wsdl:input message="impl:getCurrentTimeRequest" name="getCurrentTimeRequest" />
      <wsdl:output message="impl:getCurrentTimeResponse" name="getCurrentTimeResponse" />
    </wsdl:operation>
    <wsdl:operation name="getCurrentTime" parameterOrder="timezoneOffset">
      <wsdl:input message="impl:getCurrentTimeRequest1" name="getCurrentTimeRequest1" />
      <wsdl:output message="impl:getCurrentTimeResponse1" name="getCurrentTimeResponse1" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="TimeServiceSoapBinding" type="impl:TimeService">
    <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="getCurrentTime">
      <wsdlsoap:operation soapAction="" />
      <wsdl:input name="getCurrentTimeRequest">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://xyz.com:8080/rfmthesis/services/examples/" use="encoded" />
      </wsdl:input>
      <wsdl:output name="getCurrentTimeResponse">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://xyz.com:8080/rfmthesis/services/examples/" use="encoded" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="getCurrentTime">
      <wsdlsoap:operation soapAction="" />
      <wsdl:input name="getCurrentTimeRequest1">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://xyz.com:8080/rfmthesis/services/examples/" use="encoded" />
      </wsdl:input>
      <wsdl:output name="getCurrentTimeResponse1">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://xyz.com:8080/rfmthesis/services/examples/" use="encoded" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="TimeServiceService">
    <wsdl:port binding="impl:TimeServiceSoapBinding" name="TimeService">
      <wsdlsoap:address location=
        "http://xyz.com:8080/axis/services/rfmthesis/services/examples/TimeServiceExampleEndpoint" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>

```

**Fig. 54 A standard WSDL document. This document describes the ‘TimeService’ Web Service, detailing its provided operations and including a static URL in the ‘location’ attribute of its ‘address’ element.**

Within the described system, in order to make a Web Service implementation available for deployment and consumption it must be first described and then published into an instantiation of the presented architecture. The descriptor, called a ‘ServiceImplementationDescriptor’, includes the URI of the implemented Web Service, the target deployment container, a publisher identification number, a reference to the source code, and a set of dependencies as Strings. A

Publisher with ID '12345' would describe the preceding implementation of the 'TimeService' using a ServiceImplementationDescriptor such as that in Fig. 55 below.

```
<multiRef soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xsi:type="ns3:ServiceImplementationDescriptor" xmlns:soapenc="http://schemas.xmlsoap.org/
  soap/encoding/" xmlns:ns3="http://util.rfmthesis.cs.standrews.ac.uk">
  <serviceURI xsi:type="xsd:string">TimeService</serviceURI>
  <publisherID xsi:type="xsd:integer">12345</publisherID>
  <deploymentContainer xsi:type="xsd:string">AXIS_HOT_DEPLOY</deploymentContainer>
  <dependencySet soapenc:arrayType="xsd:string[0]" xsi:type="soapenc:Array">
  </dependencySet>
  <codeReference xsi:type="xsd:string">
    http://myRepository.com/codeRef
  </codeReference>
</multiRef>
```

**Fig. 55 XML representation of a ServiceImplementationDescriptor describing an implementation of the 'TimeService' Web Service published in the described system by a publisher with ID '12345'.**

During the deployment planning process it is desirable to be able to compare implementations to one another without having to download the actual software artifact, and thus ServiceImplementationDescriptors specify the implementation code 'location' in a 'codeReference' element. In order to support existing repositories which do not expose Web Services interfaces, this code reference is formatted as a URL. Before deployment a ServiceHost simply resolves the 'codeReference' URL to retrieve the implementation code; in the next section we will see how this code reference field can in fact reference a Web Service invoked via the local point of presence (POP) to store and retrieve implementation code (and other data), independent of any fixed repository location, using an infrastructure-provided storage service.

#### 4.5 LOCAL POINT OF PRESENCE

The local point of presence (POP) is a transparent indirection mechanism – a proxy – which runs on Service Consumer machines. Consumer agent software statically binds to and invokes Web Service operations as though they were hosted by the POP, which then interacts with the infrastructure on their behalf. An example request header for the 'TimeService' operation 'getCurrentTime' is shown in Fig. 56 below with the 'Host' value as a reference to the local machine.

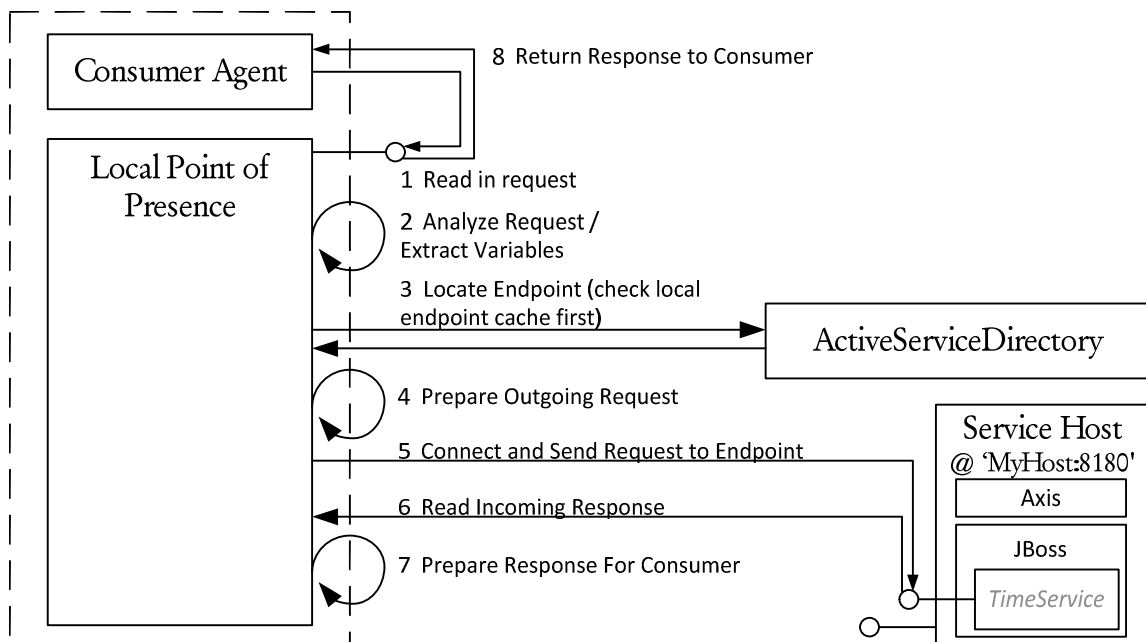
```

GET /TimeService?method=getCurrentTime HTTP/1.1
Host: localhost:8888
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.0.1) Gecko/2008070208 Firefox/3.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive

```

**Fig. 56** A HTTP header received by the local POP requesting invocation of the TimeService getCurrentTime operation.

The POP is responsible for receiving incoming invocation requests from a Service Consumer's consumer agent applications, extracting the requested Web Service URI from the message, locating an active endpoint of the service, performing any necessary modifications to the outgoing message, and sending the message to the endpoint. These steps are shown in Fig. 57 below. The POP is also responsible for detecting the failure of both a Web Service endpoint and the machine upon which it is deployed, and for attempting to recover from this failure. Described in detail in the upcoming sections, the POP recovery routine involves contacting the ActiveServiceDirectory, retrieving a list of alternative endpoints of the failed Web Service (using the URI contained in the HTTP request header), and retrying these alternative endpoints until all options have been exhausted.



**Fig. 57** A visual representation of the POP's 8-step process for fulfilling incoming invocation requests from Service Consumers.

The implementation of the mechanisms used to carry out the POP's role are detailed in the remainder of this section, beginning with the POP initialization and startup steps. The section continues with a description of the invocation and failure recovery routines and finishes with details and examples of the specific modifications necessary to proxy SOAP messages.

#### 4.5.1 *STARTUP*

The POP is initialized and started by invoking the 'main' method entry point of an executable Java archive ('JAR' file) called 'LocalPointOfPresence.jar' which contains the described POP implementation. Three run-time arguments are passed to the POP:

1. the URL of a well-known ActiveServiceDirectory endpoint, which is used as the entry-point into a running instantiation of the architecture. This URL is the single piece of 'well-known' location information required by the POP.
2. the local port used to receive SOAP requests from the Service Consumer, and,
3. a 'lease duration' parameter (in seconds) specifying the period of validity for any locally cached endpoints.

Upon initialization, the well-known ActiveServiceDirectory endpoint is queried for a list of all other alternative ActiveServiceDirectory endpoints by invoking its 'findAll' operation with the ActiveServiceDirectory URI. Even in the absence of Service Consumer demand, the POP proactively refreshes the list of ActiveServiceDirectory endpoints in order to maintain contact with the infrastructure. The POP maintains a local record of these and other service endpoints in a map called the 'EndpointCache'. This map and its application of local endpoint-selection policy are detailed next.

##### 4.5.1.1 The EndpointCache

A central component of the POP implementation, the EndpointCache maintains a local mapping from Web Service URI to a set of ServiceInstanceDescriptors describing [recently] active endpoints of the service. When the POP receives an invocation request from a Service Consumer it extracts the URI and requests an active endpoint of the service from the EndpointCache. If no endpoints of the desired Web Service exist in the map (because the Web Service has not been requested before) or the entries are stale (as defined by the 'lease duration' parameter passed to the POP at startup) then the EndpointCache will contact the ActiveServiceDirectory and request a new list. By caching and sharing the results of recent lookups, the EndpointCache amortizes the cost of ActiveServiceDirectory queries over a number of requests for the same service.

The EndpointCache uses the 'lease duration' parameter to decide when requests will be serviced using an endpoint in the local map and when to query the ActiveServiceDirectory for a new list. It makes this decision by calculating the difference between the current time and the time when the service's endpoint entries were last updated, and comparing this value with the 'lease duration' parameter: if the calculated age of the entries is less than the lease duration then the existing list is used; if it is greater, the ActiveServiceDirectory is queried with the requested Web Service URI and the resulting list of ServiceInstanceDescriptors inserted into the local map before being used to fulfill the request. A detailed specification of the invocation and failure-handling logic (such as what happens when there are no available endpoints of a Web Service and none can be deployed) is covered in the upcoming section 4.5.2 'Request Handling, Failure Detection & Recovery'.

In the event that there are multiple available endpoints of a Web Service, a decision must be made about which to use first, and the order in which to try the rest should that endpoint fail or prove unreachable. This decision-making process may be considered the 'local endpoint selection policy' and it is implemented in the EndpointCache. Implementations may order at random or on any of the information contained in a ServiceInstanceDescriptor, which intentionally includes both the ServiceImplementationDescriptor and the HostDescriptor of each deployed endpoint of a Web Service.

```
public List<ServiceInstanceDescriptor> getEndpointList(String serviceURI)
    throws ResourceUnavailableException{

    List<ServiceInstanceDescriptor> endpointList = uriToEndpointListMap.get(serviceURI);

    /**
     * Refresh the endpoint entries for this URI if:
     *     1.) This service has not been used before (endpointList == null)
     *     2.) There are no cached endpoints
     *         possibly due to removal (endpointList.isEmpty())
     *     3.) The lease has expired (leaseExpired(serviceURI))
     */
    if(endpointList == null || endpointList.isEmpty() || this.leaseExpired(serviceURI)){
        this.refreshEndpointCache(serviceURI);
        endpointList = uriToEndpointListMap.get(serviceURI);
    }
    // If there are still no endpoints throw a ResourceUnavailableException
    if(endpointList.isEmpty()){
        throw new ResourceUnavailableException("No endpoints available for requested "
            + "service: '"+serviceURI+"'.");
    }else{
        // Returns a list of endpoints
        // ordered by 'priority' as dictated by the setEndpointOrder method
        return this.setEndpointOrder(endpointList);
    }
}
```

**Fig. 58** The 'EndpointCache' centralizes the local knowledge of endpoint locations and manages the processes of updating and sorting endpoint location information and retrieving fresh locations from the ActiveServiceDirectory.

In the described implementation, when an EndpointCache retrieves the first list of endpoints for a particular URI, as shown in Fig. 58 above, the order of the results is first randomized before they are stored. While this may appear to be a purely local decision, it can have broader implications for the functionality of the infrastructure. For example, because the infrastructure's generic mechanism for handling growing demand for a Web Service is to deploy a new endpoint and list it in the ActiveServiceDirectory, it is important to the effectiveness of the infrastructure's generic load balancing techniques that the POP select an endpoint randomly from all currently available endpoints of a service. If all POPs instead ordered on the remote IP address, for example, then a single endpoint would be used by all Service Consumers. In this case, a Manager detecting increased load and deploying an additional endpoint of the Web Service would not reduce the average invocation response time experienced by Service Consumers because all of their local endpoint-selection processes would be identical. While the POPs would detect any failure of the first endpoint and transparently re-direct requests to the next available endpoint, they would again select the same secondary endpoint. Thus if the the local endpoint selection policy is to be changed is should be done either with a full understanding of its broader implications, or together with a change to the generic management and load-balancing techniques.

After randomizing the original endpoint list, the local order of the list is then retained by the EndpointCache. This decision to give priority to the *most*-recently-used endpoint is based on evidence in [REF] showing that the natural impulse to select by least-recently-used can result in processing delays at the host, often due to the need to page the requested service into working memory. In order to absorb the identified benefits of temporal locality, the original order of the list is maintained when the list is updated, with failed listing removed and fresh listings randomly dispersed throughout.

#### 4.5.1.2 Starting the POP

Once the list of ActiveServiceDirectory endpoints has been received and the EndpointCache initialized, the POP is then ready to service Service Consumer requests. The portion of the POP application that Service Consumers interact with is realized as a generic multi-threaded server which listens for incoming connections on a port specified in the second run-time parameter (the default is '8888'). For each new connection received the server spawns an 'InvocationRequestHandlerThread'. These handlers are responsible for analyzing an incoming request, locating an endpoint of the requested Web Service, performing any necessary transformations on the outgoing call, sending the outgoing call, receiving and transforming

results, and returning them to the consumer. These handlers contain all of the client-side fault-detection and recovery logic which is presented in the next section.

#### 4.5.2 REQUEST HANDLING, FAILURE DETECTION & RECOVERY

An `InvocationRequestHandlerThread` is responsible for the entire process of handling a single invocation request received from a service consumer. There are eight steps in the request handling process, each of which has associated recoverable and non-recoverable errors (from the endpoint, endpoint host, and the consumer agent) all of which must be dealt with appropriately. These steps are, colloquially:

1. 'read in request',
2. 'extract variables',
3. 'locate endpoint',
4. 'prepare outgoing call',
5. 'connect and send call to endpoint',
6. 'read incoming response',
7. 'prepare response for consumer', and,
8. 'return response to consumer'.

Together with their associated failure-handling routines, each of these steps is detailed in the following eight sub-sections.

##### 4.5.2.1 Reading Requests

Upon initialization, the `InvocationRequestHandlerThread` is passed a socket object which serves as an abstraction over the underlying connection. This `Socket` provides a '`java.io.InputStream`' object which has methods for reading data out of the incoming read buffer. This `InputStream` object is wrapped by a '`java.io.reader.BufferedReader`' object which provides coarse-grain operations for more convenient reading of large portions of data (such as a '`readLine()`' operation).

The HTTP headers are read first and the value of the '`Content-length`' header extracted. This header specifies the number of valid bytes contained in the body of the message. In the absence of failure, the specified number of bytes is read off the incoming data stream and the entire request (including headers) is returned to the request handler. This read operation is susceptible to interrupted and failed connections, or other failures of the network stack. In Java, these errors are thrown as `java.io.IOExceptions` and are considered unrecoverable. In the event of an `IOException` the '`readRequest`' method aborts and the request handling thread terminates.

Further, a timeout error may occur if the specified number of bytes has not been received before the client-specified timeout period. In this case an HTTP error is returned to the consumer agent before throwing an IOException and terminating the thread. The error returned to the consumer agent is an HTTP '408' error indicating that a timeout has occurred.

#### 4.5.2.2 Extracting HTTP Request Variables

Upon successfully reading a request, the request handler thread must extract a pair of variables from the first line of the HTTP request. The first line of an HTTP request is called the 'request line' and contains the 'method' to be applied, the identifier of the object (URI), and the protocol version in use. This line can be seen in Fig. 59, below.

The method name is extracted first and its value stored. The value must be either 'GET' or 'POST' as these are the two methods supported by SOAP – in the event that any other method value is received an HTTP 405 'Method Not Allowed' error is returned and the connection closed. The method value indicates the method to be performed on the object identified by the given URI: a 'GET' means retrieve whatever data is identified by the URI, where the URI refers to a data-producing process; the 'POST' method creates a new object linked to the object specified in the message content. While the POP does not perform these actions (and is not actually an active participant in the HTTP protocol) the value of the request method dictates the kind of pre- and post-processing that must be performed on a message before it can be sent to an endpoint or returned to a consumer.

```
GET /TimeService?method=getCurrentTime HTTP/1.1
Host: localhost:8888
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.0.1) Gecko/2008070208 Firefox/3.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

**Fig. 59 A HTTP 'GET' request with the URI '/TimeService' identified in bold.**

The second variable extracted is the request URI. In the HTTP request shown in Fig. 59, 'TimeService' is the URI and is shown in bold (note that although a URI does include the leading slash ('/') character, these will be omitted in this document's text for ease of reading). The URI contained in the HTTP request is the same URI value used to identify the Web Service in the ActiveServiceDirectory and ServiceLibrary. The URI is used by the POP in the next step to locate

an active endpoint of the requested service to which the current message can be sent. As we will see later, the URI used to identify the Web Service may not correspond to the URI used to identify the destination Web Service endpoint on a remote host.

In the event of formatting errors an HTTP 400 'Bad Request' error is returned to the consumer agent and the connection closed. Barring any errors, the extracted values are stored as private variables and control returned to the request handler.

#### 4.5.2.3 Invocation Handler Routine

Having read in the invocation request and extracted the service URI, the POP is ready to initiate the process of performing the invocation on behalf of the Service Consumer. Invoking an operation on behalf of the Service Consumer involves locating an active endpoint of the requested Web Service, preparing the outgoing request based on the endpoint details, binding to and sending the request to the endpoint, reading the response, checking the response for errors, preparing the response for return and, if all these steps completed successfully, returning the response to the consumer. As there are many possible failures to detect and recover from, these steps must be tightly interwoven in order for invocation to be performed reliably. Dictating that participants in the architecture apply an earnest 'best effort' to the fulfillment of requests and specifying a simple and clear-cut failure model allows the invocation and failure-recovery procedures to remain clean, simple, and un-cluttered by rarely-used exception-handling routines.

To start the invocation process, an ordered list of active endpoints (each represented as a `ServiceInstanceDescriptor`) is first requested from the `EndpointCache`. Because locating the set of available endpoints takes place in the `EndpointCache`, the invocation routine is relieved from the added complication of detecting and handling errors of the `ActiveServiceDirectory`. If the `EndpointCache` fails to locate any endpoint of the requested Web Service, a `ResourceUnavailableException` will be thrown and the invocation process will end. As a `ResourceUnavailableException` is indicative of the state of the infrastructure at the moment (there are no available endpoints for the requested Web Service URI and it is not possible to deploy one given the available resources) it will propagate up to the request handler thread and be returned to the consumer agent before the connection is terminated.

Having successfully retrieved a list of endpoints of the target Web Service, the invocation request handler then begins the process of fulfilling the invocation request, as shown in Fig. 60 below. Beginning with the first endpoint in the list, the invocation request message is first modified as necessary, a connection to the endpoint opened, the modified request sent to the

endpoint, the response read, parsed and analyzed. If the request is successful the response message may again be modified before finally being returned to the service consumer. Any intermittent errors or exceptions will result in the same process being performed upon the next endpoint in the list until the invocation succeeds or the end of the list is reached. Upon reaching the end of the list, a new list will be requested from the EndpointCache only once, for reasons explained in the next paragraph. Upon the failure of that list the operation will be considered to have failed and a `ResourceUnavailableException` returned to the service consumer.

```
public String invokeOperation(String originalRequest, String serviceURI)
    throws ResourceUnavailableException, IOException {

    /** Retrieve endpoint list
    allow ResourceUnavailableException to propagate up to handler */
    List<ServiceInstanceDescriptor> endpointList =
        endpointCache.getEndpointList(serviceURI);

    /** Set up local variables */
    String request = originalRequest;
    String response;
    Socket connection;

    for(ServiceInstanceDescriptor endpoint : endpointList){
        try{
            /** Connect, prepare, and send request to endpoint */
            connection = connectToEndpoint(endpoint);
            request = prepareForOutgoingCall(request, endpoint);
            sendRequestToEndpoint(request, connection);

            /** Read, validate, prepare and return response to consumer */
            response = readIncomingResponse(connection);
            validateResponse(response);
            response = prepareResponseForClient(response, endpoint);

            return response;

        }catch(IOException ioe){
            /** In the event of connection-level failure, signal failure to the cache */
            endpointCache.signalFailure(endpoint);
        }catch(ResourceUnavailableException rue){
            /** Various endpoint-specific failures, including HTTP-related exceptions.
            * Possibly temporal but still considered failures. Signal to cache. */
            endpointCache.signalFailure(endpoint);
        }
    }
    /** All attempts unsuccessful - throw ResourceUnavailableException */
    throw new ResourceUnavailableException("The requested resource '" + serviceURI
        + "' is currently unavailable. Please try again later.");
}
```

Fig. 60 The 'invokeOperation' method of the invocation request-handling routine

If all of the endpoints held in a POP's EndpointCache have failed, a final additional query to the ActiveServiceDirectory will reflect whether the requested resource (a Web Service endpoint) is able to be deployed or not, given the current resources of the infrastructure. If there are currently no deployed endpoints of a particular Web Service, this request for an endpoint will block in the ActiveServiceDirectory while the Web Service's Manager is contacted and

autodeployment attempted. If it is possible to deploy an endpoint of the requested Web Service then the Manager will do so, returning the resultant ServiceInstanceDescriptor to the ActiveServiceDirectory which returns it to the requesting Service Consumer's POP. If it is not possible to deploy an endpoint of the requested Web Service its Manager will return the ActiveServiceDirectory a ResourceUnavailableException, which is then returned to the requesting Service Consumer's POP. Because a 'ResourceUnavailableException' carries a temporal supposition it can be meaningfully returned to the consumer if this second attempt fails, even though a resource fitting the requirement may in fact be available at some point in the future.

Each individual step of the invocation procedure shown in Fig. 60 above is covered in an upcoming sub-section.

#### 4.5.2.4 Locating Active Endpoints

Having extracted the URI of the requested Web Service from the incoming invocation request, the invocation request handler requests the list of active endpoints of the service from the EndpointCache. The EndpointCache uses the previously-detailed invocation routine to invoke the 'findAll' operation of the ActiveServiceDirectory using the 'TimeService' URI as a parameter. An example request for this operation is shown below in Fig. 61.

```
POST /rfmthesis/services/ActiveServiceDirectory HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/1.4
Host: abc.com:8080
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: ""
Content-Length: 582

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:findAll soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns1="http://abc.com:8080/rfmthesis/services">
      <serviceURI xsi:type="soapenc:string"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">TimeService</serviceURI>
    </ns1:findAll>
  </soapenv:Body>
</soapenv:Envelope>
```

**Fig. 61** An example of an outgoing lookup request for the list of all active endpoints of the Web Service identified by the URI 'TimeService' sent to the ActiveServiceDirectory endpoint at host abc.com on port 8080.

The request in Fig. 61 will result in a response similar to the one shown below in Fig. 62 which indicates that a single endpoint is available for the 'TimeService' and that it is hosted at 'xyz.com'. Because the returned XML is very verbose, the single line of information relevant to this discussion is highlighted in bold. The remainder of the information contained in the returned ServiceInstanceDescriptor (such as the HostDescriptor and the ServiceImplementationDescriptor) is nonetheless worthwhile to observe as this information will be used by other entities in the infrastructure for purposes covered in the upcoming sections.

Upon receiving the ActiveServiceDirectory response the EndpointCache stores the returned ServiceInstanceDescriptor in its local map under the 'TimeService' URI entry, ready to be provided when required for subsequent invocations.

```

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/xml;charset=utf-8
Date: Mon, 26 Jan 2009 17:26:04 GMT
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:findAllResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns1="http://abc.com:8080/rfmthesis/services">
      <findAllReturn soapenc:arrayType="ns2:ServiceInstanceDescriptor[1]"
        xsi:type="soapenc:Array" xmlns:ns2="http://util.rfmthesis.cs.standrews.ac.uk"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
        <findAllReturn href="#id0"/>
      </findAllReturn>
    </ns1:findAllResponse>
    <multiRef id="id0" soapenc:root="0" soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/
      encoding/" xsi:type="ns3:ServiceInstanceDescriptor" xmlns:soapenc="http://
      schemas.xmlsoap.org/soap/encoding/" xmlns:ns3="http://util.rfmthesis.cs.standrews.ac.uk">
      <endpointURL xsi:type="xsd:string">
        http://xyz.com:8080/axis/rfmthesis/services/examples/TimeServiceExampleEndpoint
      </endpointURL>
      <hostDescriptor href="#id1"/>
      <implDescriptor href="#id2"/>
      <instanceID xsi:type="xsd:integer">
        1451057475215743030728584773222206527584438429144</instanceID>
      <timestamp xsi:type="xsd:dateTime">2009-01-26T17:25:42.255Z</timestamp>
    </multiRef>
    <multiRef id="id1" soapenc:root="0" soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/
      encoding/" xsi:type="ns4:HostDescriptor" xmlns:ns4="http://util.rfmthesis.cs.standrews.ac.uk"
      xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
      <serviceHostID xsi:type="xsd:integer">
        797226840858717413927154300516025213427171252934</serviceHostID>
      <serviceHostWSLocation xsi:type="xsd:string">
        http://xyz.com:8080/axis/services/ServiceHostEndpoint</serviceHostWSLocation>
      <capabilities soapenc:arrayType="xsd:string[4]" xsi:type="soapenc:Array">
        <capabilities xsi:type="xsd:string">AXIS_AAR</capabilities>
        <capabilities xsi:type="xsd:string">AXIS_HOT_DEPLOY</capabilities>
        <capabilities xsi:type="xsd:string">RAFDA</capabilities>
        <capabilities xsi:type="xsd:string">POP</capabilities>
      </capabilities>
    </multiRef>
    <multiRef id="id2" soapenc:root="0" soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/
      encoding/" xsi:type="ns5:ServiceImplementationDescriptor" xmlns:ns5="http://
      util.rfmthesis.cs.standrews.ac.uk" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
      <serviceURI xsi:type="xsd:string">TimeService</serviceURI>
      <publisherID xsi:type="xsd:integer">
        570486094678203678964019527945730647279773233338</publisherID>
      <deploymentContainer xsi:type="xsd:string">AXIS_AAR</deploymentContainer>
      <codeReference xsi:type="xsd:string">
        http://localhost:8888/StoreService/TimeService.aar</codeReference>
      <dependencySet soapenc:arrayType="xsd:string[0]" xsi:type="soapenc:Array"/>
    </multiRef>
  </soapenv:Body>
</soapenv:Envelope>

```

**Fig. 62** An example response from an `ActiveServiceDirectory` 'findAll' operation using the URI 'TimeService'. This response identifies a single endpoint of the service. The URL of this endpoint is shown in bold.

In communication between the Service Consumer and the POP, the URL referencing the location of the 'TimeService' Web Service is `http://localhost:8888/TimeService`. However the `ServiceInstanceDescriptor` in Fig. 62 describes an endpoint of the 'TimeService' Web Service which is not a simple concatenation of its URI to the remote hostname. In communication between the POP and an active endpoint of a Web Service, the URI used to identify the Web Service on the remote host is not necessarily the same URI used to identify the Web Service within the infrastructure. In this case the remote URI included in the `ServiceInstanceDescriptor` is `'/axis/services/rfmthesis/services/examples/TimeServiceExampleEndpoint'`, which is similar but not identical to the URI 'TimeService'. By translating between the URI used to identify a Web Service within the architecture and the URL of a remote endpoint offering that service, the POP allows for flexibility in the naming of remote endpoints (which are usually burdened by strict naming schemes) and enables Web Services to be published and deployed which offer a super-set of the functionality required by the requested Web Service. The steps necessary to perform these transformations on outgoing invocation request are detailed next.

#### 4.5.2.5 Preparing Outgoing Calls

Having read the invocation request from the Service Consumer, extracted the service URI, and retrieved a list of active endpoints of the Web Service (using the `EndpointCache` and possibly the `ActiveServiceDirectory`), the POP must now perform some alterations on the outgoing request before it can be sent to an endpoint for processing. These modifications are necessary in order to be compliant with both SOAP and HTTP protocols.

For requests coming from a consumer, the HTTP headers of both 'POST' and 'GET' requests require alterations before being sent to a remote endpoint. Both request methods require changes to the 'Host' value, as well as the URI in the request line. Further, 'POST' requests may require alterations to the body of the request. Because these alterations may change the length of the body, the new length must also be reflected in the 'Content-length' header value. Because 'GET' requests require a subset of the modifications required by 'POST' requests, the 'POST' request in Fig. 63 will be used as an example. This request is for the second `'getCurrentTime'` operation of the 'TimeService' which takes an integer parameter indicating the desired time-zone offset from GMT (e.g. -5 for Eastern Standard Time).

```

POST /TimeService HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/1.4
Host: localhost:8888
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: ""
Content-Length: 680

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:getCurrentTime soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns1="http://localhost:8888">
      <timezoneOffset href="#id0"/>
    </ns1:getCurrentTime>
    <multiRef id="id0" soapenc:root="0"
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="xsd:int"
      xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">-5</multiRef>
  </soapenv:Body>
</soapenv:Envelope>

```

**Fig. 63 A 'POST' request received by the POP for the 'getCurrentTime' operation of the 'TimeService' using a timezone offset parameter of '-5'**

Upon receipt of the request in Fig. 63 the POP will extract the desired URI ('TimeService') and use this URI to retrieve a list of ServiceInstanceDescriptors from the EndpointCache. In the current example, as shown previously in Fig. 62, a single endpoint of the 'TimeService' Web Service is currently deployed. According to the ServiceInstanceDescriptor in Fig. 62 the endpoint is located at the URL: `http://xyz.com:8080/axis/services/rfmthesis/services/examples/TimeServiceExampleEndpoint`. For the purposes of the transformations this URL is separated into its hostname and port (`xyz.com:8080`) and remote URI (`/axis/services/rfmthesis/services/examples/TimeServiceExampleEndpoint`).

The POP uses the remote hostname, port, and URI values to modify the invocation request shown in Fig. 63 into the fully-prepared invocation request shown in Fig. 64. To do this the request URI is changed to the remote URI, the 'Host' header value is changed to indicate the remote hostname and port, any namespaces referencing the localhost are changed to indicate the remote host, and, finally, the 'ContentLength' header value is changed to reflect the new length of the request.

```

POST /axis/services/rfmthesis/services/examples/TimeServiceExampleEndpoint HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/1.4
Host: xyz.com:8080
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: ""
Content-Length: 693

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://
www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:getCurrentTime soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns1="http://xyz.com:8080/axis/services/rfmthesis/services/examples">
      <timezoneOffset href="#id0"/>
    </ns1:getCurrentTime>
    <multiRef id="id0" soapenc:root="0"
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="xsd:int"
      xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">-5</multiRef>
  </soapenv:Body>
</soapenv:Envelope>

```

**Fig. 64 The fully prepared outgoing request for the 'getCurrentTime' operation of the 'TimeService' Web Service hosted at xyz.com on port 8080 with URI /axis/services/rfmthesis/services/examples/TimeServiceExampleEndpoint, with modifications shown in bold.**

#### 4.5.2.6 Binding to Endpoints and Invoking Operations

Once the outgoing invocation request is prepared for its remote destination, the 'getCurrentTime' request must next be sent to the target endpoint by the POP on behalf of the Service Consumer. The requested operation is invoked by opening a connection to the identified host on the specified port and sending the newly prepared request in its entirety.

This operation is susceptible to the same set of failures as reading the original request from the service consumer during the 'read request' phase. Any connection-level faults experienced while sending an invocation request to a remote endpoint are thrown as IOExceptions and caught by the surrounding invocation routine, which may try again using another endpoint. The EndpointCache is notified of any failed endpoints by invoking its 'signalEndpointFailure' method with the offending ServiceInstanceDescriptor, which is removed from its list. Note that because it is the sole responsibility of a Web Service's Manager to maintain the service's entries in the ActiveServiceDirectory, the offending entry may still be included in subsequent lists returned by the ActiveServiceDirectory until the service's Manager identifies the problem.

#### 4.5.2.7 Reading and Validating Incoming Responses

After sending an invocation request to an endpoint the resultant response must be read, parsed, and checked for HTTP errors and SOAP faults before being returned to the consumer agent. The process of reading a response from an endpoint is the same as reading a request from a consumer, and all of the same errors may occur. Any resultant `IOExceptions` are thrown, caught by the invocation request handler routine, and the offending endpoint removed from the `EndpointCache`. The invocation routine will then attempt the invocation process again using the remaining alternative endpoints of the service. Upon successfully reading a response the connection to the remote host is closed and the response is returned to the invocation handler routine. The response must now be checked for errors.

Errors of the HTTP protocol are indicated on the first line of the HTTP response header in a three-digit token called the 'status code'. While there are many status codes which indicate errors, the only status code indicating success is '200' or "200 – OK". An HTTP '200' response can be seen in Fig. 65 below, and is considered to signal a successfully invoked operation.

Responses with a HTTP '200' status code are returned to the invocation handler routine for post-processing, as covered in the next section, before finally being returned to the consumer agent. SOAP faults are returned as HTTP '500' errors and are processed as detailed in the next paragraph. All other response codes are considered to be endpoint-specific failures and are wrapped and thrown as `IOExceptions`, to be caught by the invocation handler routine, which removes the offending endpoint from the `EndpointCache` before attempting the operation again using any remaining alternative endpoints. The full list of status codes and their meanings can be found in [103].

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/xml;charset=utf-8
Date: Mon, 26 Jan 2009 18:39:37 GMT
Connection: close

<?xml version="1.0" encoding="utf-8"?><soapenv:Envelope xmlns:soapenv="http://
schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
  <ns1:getCurrentTimeResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/
encoding/" xmlns:ns1="http://xyz.com:8080/axis/services/rfmthesis/services/examples">
    <getCurrentTimeReturn href="#id0"/>
  </ns1:getCurrentTimeResponse>
  <multiRef id="id0" soapenc:root="0"
    soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="xsd:long"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">1233088598496</multiRef>
</soapenv:Body>
</soapenv:Envelope>
```

**Fig. 65 A response to an invocation of the 'getCurrentTime' operation of the Web Service identified by the URI 'TimeService'. The HTTP response code is '200' which indicates a successful request.**

In order to detect and handle SOAP faults the entire response message must be parsed and analyzed. The implementation defines three categories for SOAP faults: generic, application-specific, and wrapped `ResourceUnavailableException` SOAP faults. Application-specific faults are thrown intentionally by a Web Service in order to indicate an application-specific exception (such as, for example, a referenced 'customerID' not existing in a database). These faults are considered to be information-bearing messages rather than failures and are returned to the consumer as the result of the invocation.

A wrapped '`ResourceUnavailableException`' SOAP fault is an example of an application-specific SOAP fault thrown from infrastructure-provided services, such as the `ActiveServiceDirectory` or `ServiceLibrary`. These are intentionally thrown faults that signal a fundamental incapacity of the infrastructure to provide the requested resource – an unrecoverable error containing information about the requested operation. All other SOAP faults are considered to be endpoint-specific faults and are wrapped and thrown as `IOExceptions`. The invocation routine will catch these exceptions, signal to the `EndpointCache` that the endpoint is considered to have failed, and will try the invocation again using any remaining alternative endpoints.

#### 4.5.2.8 Preparing Responses for Consumers

If the incoming response passes the validation step it must then be post-processed before being returned to the Service Consumer. For the vast majority of responses it is only necessary to reverse the previous namespace changes. Returned messages also require no changes to HTTP header values. One particular type of 'GET' request, however, produces a response which requires more significant post-processing. These are termed 'WSDL requests' and return a WSDL document as the result of 'GET' requests whose request line contains a URI value which ends in the string "?WSDL" (e.g. "GET /TimeService?WSDL HTTP/1.1").

Because WSDL documents are endpoint-specific – that is, they encapsulate a static physical endpoint described with a URL – an incoming WSDL document for the 'TimeService' will contain a remote target namespace and a remote URL as the 'location' value in its 'PortType' element. As part of the process of achieving location-independent Web Service addressing, these values must be changed to reference the local POP and the in-infrastructure URI value identifying the 'TimeService' Web Service. For example, before being returned to the consumer, the incoming WSDL document shown previously in Fig. 56 for the 'TimeService' Web Service endpoint hosted at 'example.org' with the URI 'TimeServiceEndpoint' will require the modifications indicated in bold in Fig. 66 below. Responses from Web Services which use 'complex-type' elements will require a

greater number of individual alterations but their scope remains limited to the same manipulations of the namespace and endpoint location attribute values.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://localhost:8888/" xmlns:apachesoap="http://xml.apache.org/
xml-soap" xmlns:intf="http://localhost:8888/" xmlns:soapenc="http://schemas.xmlsoap.org/soap/
encoding/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:wsdlsoap="http://schemas.xmlsoap.org/
wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <wsdl:message name="getCurrentTimeResponse1">
    <wsdl:part name="getCurrentTimeReturn" type="xsd:long"/>
  </wsdl:message>
  <wsdl:message name="getCurrentTimeRequest">
  </wsdl:message>
  <wsdl:message name="getCurrentTimeResponse">
    <wsdl:part name="getCurrentTimeReturn" type="xsd:long"/>
  </wsdl:message>
  <wsdl:message name="getCurrentTimeRequest1">
    <wsdl:part name="timezoneOffset" type="xsd:int"/>
  </wsdl:message>
  <wsdl:portType name="TimeService">
    <wsdl:operation name="getCurrentTime">
      <wsdl:input message="intf:getCurrentTimeRequest" name="getCurrentTimeRequest"/>
      <wsdl:output message="intf:getCurrentTimeResponse" name="getCurrentTimeResponse"/>
    </wsdl:operation>
    <wsdl:operation name="getCurrentTime" parameterOrder="timezoneOffset">
      <wsdl:input message="intf:getCurrentTimeRequest1" name="getCurrentTimeRequest1"/>
      <wsdl:output message="intf:getCurrentTimeResponse1" name="getCurrentTimeResponse1"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="TimeServiceSoapBinding" type="intf:TimeService">
    <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="getCurrentTime">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="getCurrentTimeRequest">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://localhost:8888/" use="encoded"/>
      </wsdl:input>
      <wsdl:output name="getCurrentTimeResponse">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://localhost:8888/" use="encoded"/>
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="getCurrentTime">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="getCurrentTimeRequest1">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://localhost:8888/" use="encoded"/>
      </wsdl:input>
      <wsdl:output name="getCurrentTimeResponse1">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://localhost:8888/" use="encoded"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="TimeServiceService">
    <wsdl:port binding="intf:TimeServiceSoapBinding" name="TimeService">
      <wsdlsoap:address location="http://localhost:8888/TimeService/">
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Fig. 66 A WSDL document for the 'TimeService' hosted by 'localhost' on port 8888 using the URI '/TimeService'. The data in bold represents changes applied to the WSDL document from Fig. 54.

#### 4.5.2.9 Returning Responses to Consumers

Upon completion of the preceding steps, a valid and successfully post-processed invocation response will be returned to the Service Consumer. The response is returned in its entirety to the consumer agent application using the output stream of the socket originally used to read the invocation request. Upon successfully writing the invocation response the connection is closed and the invocation request handler thread terminated. The POP's task is complete.

#### 4.6 PROGRAMMING CONSUMER AGENT APPLICATIONS

A programmer of a consumer agent application using the 'TimeService' Web Service can retrieve its WSDL at 'http://localhost:8888/TimeService?WSDL'. The POP consumes the location and translation tasks using the previously described mechanisms. If the programmer creates a proxy class (using WSDL2JAVA, for example), the proxy will contain a static reference to the URL 'http://localhost:8888/TimeService'. Invocations of TimeService operations will result in SOAP messages being sent to the POP, which uses the mechanisms described in the previous sections to carry out the requested operation on behalf of the Service Consumer. If the operation succeeds the consumer will receive their results as though the operation were carried out at the POP. The consumer agent application will only receive a 'ResourceUnavailableException' in the event that it is simply not possible to carry out the requested operation given the current available resources in the infrastructure. This exception is non-recoverable at the given moment and should be considered fatal.

As shown in the previous chapter, provider agents also behave as consumer agents when carrying out intra-architecture interactions. In order to abstract over their implementation and hosting technology, architectural entities such as the ServiceLibrary expose Web Service interfaces which allow for remote invocation of their operations using platform-independent mechanisms. In order to abstract over the location of these entities, this reference implementation utilizes the POP for all intra-architecture interactions. This means that communication between infrastructural components is carried out in the same way as a Service Consumer invokes the operation of any Web Service, using all of the previously-described invocation, communication, failure-detection and recovery routines.

For example, a publisher can statically bind to and invoke the 'publishServiceImplementation' operation of the ServiceLibrary Web Service located at 'http://localhost:8888/services/ServiceLibrary'. Similarly, a ServiceHost or Manager may also invoke the 'findOne' or 'findAll' operations of the ServiceLibrary using the same static reference to the POP, which then contacts the ActiveServiceDirectory to locate active endpoints of the service and attempts to fulfill the request using the same robust invocation techniques

described previously. Thus the actors fulfilling the role of Service Provider use the same fault-tolerant invocation mechanisms as the Service Consumers they serve.

Further, implementations of the `ActiveServiceDirectory`, `ServiceLibrary`, and `HostDirectory` (as well as a `StoreService` detailed later) are themselves published into the infrastructure. In the face of rising demand for these services a new endpoint will simply be deployed upon an available `ServiceHost` and the location of the new endpoint registered with the `ActiveServiceDirectory`. The provision of the support architecture is thus achieved and maintained using its own mechanisms.

## 4.7 SERVICEHOSTS

### 4.7.1 INTRODUCTION

Having detailed the process of describing and publishing Web Service implementations into a running instantiation of the architecture, and how Service Consumers may invoke Web Service operations via their local POP, this section details the processes through which ServiceHosts join and participate in the infrastructure. While Managers make decisions about when and where a Web Service endpoint should be deployed, ServiceHosts enact deployment into a local container under their control and provide the computational resources used to host Web Service endpoints. Because all Web Services and all infrastructure components are deployed upon them, ServiceHosts provide the most basic and critical resource in the infrastructure.

Managers and their decision-making processes are covered later in section 4.8 ‘Managers’, and for the time being it is sufficient to recall that Managers are responsible for requesting the deployment of a Web Service endpoint at a ServiceHost. Managers request deployment by invoking the ‘requestDeployment’ operation of a ServiceHost’s exposed ‘IServiceHost’ Web Service, using as a parameter the ServiceImplementationDescriptor describing the Web Service to be deployed. This interaction between a Manager and ServiceHost is shown below in Fig. 67 (originally presented in Chapter 3 Fig. 50 and duplicated here for ease of reference). The remainder of this section details first how a ServiceHost is set up and configured by an administrator, and how it is subsequently described and registered in the HostDirectory. The internal structure of the ServiceHost Web Service is detailed next, including descriptions of the mechanisms used to deploy Web Service endpoints into containers on the local machine. The section continues with a step-by-step example of the deployment of a ‘TimeService’ Web Service endpoint, and finally concludes with a presentation of the processes undertaken by a ServiceHost to extract endpoint usage data from containers on its machine and report this data to each deployed Web Service’s corresponding Manager.

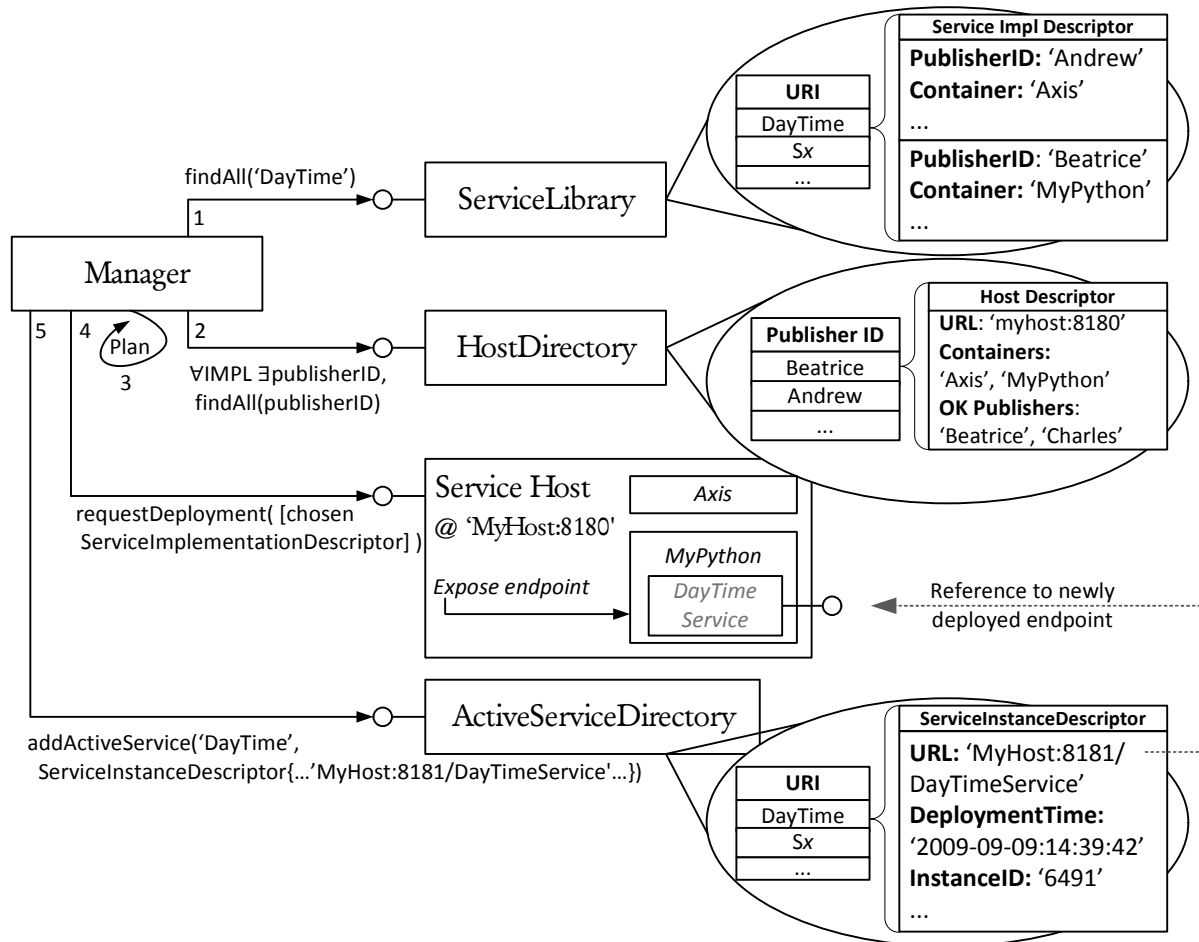


Fig. 67 Managers enact deployment by querying the ServiceLibrary and HostDirectory, creating a deployment plan, requesting deployment on a ServiceHost, and registering any newly deployed endpoint with the ActiveServiceDirectory.

#### 4.7.2 ROLE OF THE ADMINISTRATOR

A ServiceHost system administrator is responsible for setting up and configuring a ServiceHost machine, describing the ServiceHost using a HostDescriptor, and registering this HostDescriptor in the HostDirectory. The administrator begins the setup process by launching the application servers and Web Service containers which will be used to host Web Service endpoints. These containers will be listed in the ServiceHost's HostDescriptor to indicate the types of Web Service implementations the ServiceHost is capable of deploying. Because a ServiceHost must periodically send usage data to the Manager of each of its deployed Web Service endpoint, system administrators must also ensure that these containers are properly configured to record endpoint usage data.

Having configured and launched all of the supported containers, the administrator then launches the ServiceHost Web Service itself. As well as exposing the requisite 'IServiceHost'

interface, this Web Service implements all of the functionality required to fulfill the role of a ServiceHost, including deploying and undeploying endpoints and periodically extracting and reporting endpoint usage data. On startup, the administrator provides the ServiceHost Web Service with parameters that indicate the containers to be supported. Detailed in the next section 4.7.3 'The ServiceHost Web Service', the administrator also provides a set of 'container adapters' (realized as Java classes) which enable the ServiceHost to communicate with each supported container through a generic interface. Completing the setup and configuration process the system administrator starts a local POP, as described in the previous sections, which will be used for all intra-architecture interactions, including the ServiceHost's registration in the HostDirectory.

Once the ServiceHost machine is configured the administrator is responsible for describing the ServiceHost using a HostDescriptor and adding this HostDescriptor to the HostDirectory. The HostDescriptor includes an entry for each container that the ServiceHost is capable of deploying a Web Service endpoint into. An example HostDescriptor for a ServiceHost capable of deploying implementations written for the 'Axis HotDeploy' and the 'RAFDA' deployment environments is shown below in Fig. 68. Once configured and registered, the remaining ServiceHost tasks of responding to deployment requests and reporting usage data are fully automated and require no further human input.

```

...
<multiRef id="id1" soapenc:root="0" soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/
encoding/" xsi:type="ns4:HostDescriptor" xmlns:ns4="http://util.rfmthesis.cs.standrews.ac.uk"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
  <serviceHostID xsi:type="xsd:integer">
    797226840858717413927154300516025213427171252934</serviceHostID>
  <serviceHostWSLocation xsi:type="xsd:string">
    http://xyz.com:8080/axis/services/ServiceHostEndpoint</serviceHostWSLocation>
  <capabilities soapenc:arrayType="xsd:string[3]" xsi:type="soapenc:Array">
    <capabilities xsi:type="xsd:string">AXIS_HOT_DEPLOY</capabilities>
    <capabilities xsi:type="xsd:string">RAFDA</capabilities>
    <capabilities xsi:type="xsd:string">POP</capabilities>
  </capabilities>
</multiRef>
...

```

**Fig. 68 HostDescriptor describing a ServiceHost which supports the AXIS\_HOT\_DEPLOY and RAFDA deployment environments.**

#### 4.7.3 THE SERVICEHOST WEB SERVICE

The reference implementation's ServiceHost provider agent application is written in Java as Web Service for the Apache Axis deployment environment. The implementation encapsulates all of the functionality required to fulfill the role of ServiceHost, including the deployment and undeployment of Web Service endpoints, and the periodic retrieval and reporting of endpoint

usage data to each of the deployed Web Service's Managers. Each of these functions is carried out by an individual functional component within the provider agent application, as described below.

The ServiceHost provider agent implementation can be used to deploy Web Service endpoints in any container under the ServiceHost's control by providing each supported container's name as a runtime argument together with a corresponding software 'container adapter'. A container adapter is a Java class implementing the 'IContainerAdapter' interface shown below in Fig. 69. ServiceHosts deploy endpoints into containers on their local machine by interacting with these intermediaries which serve to abstract over the container-specific techniques required to deploy and undeploy Web Services.

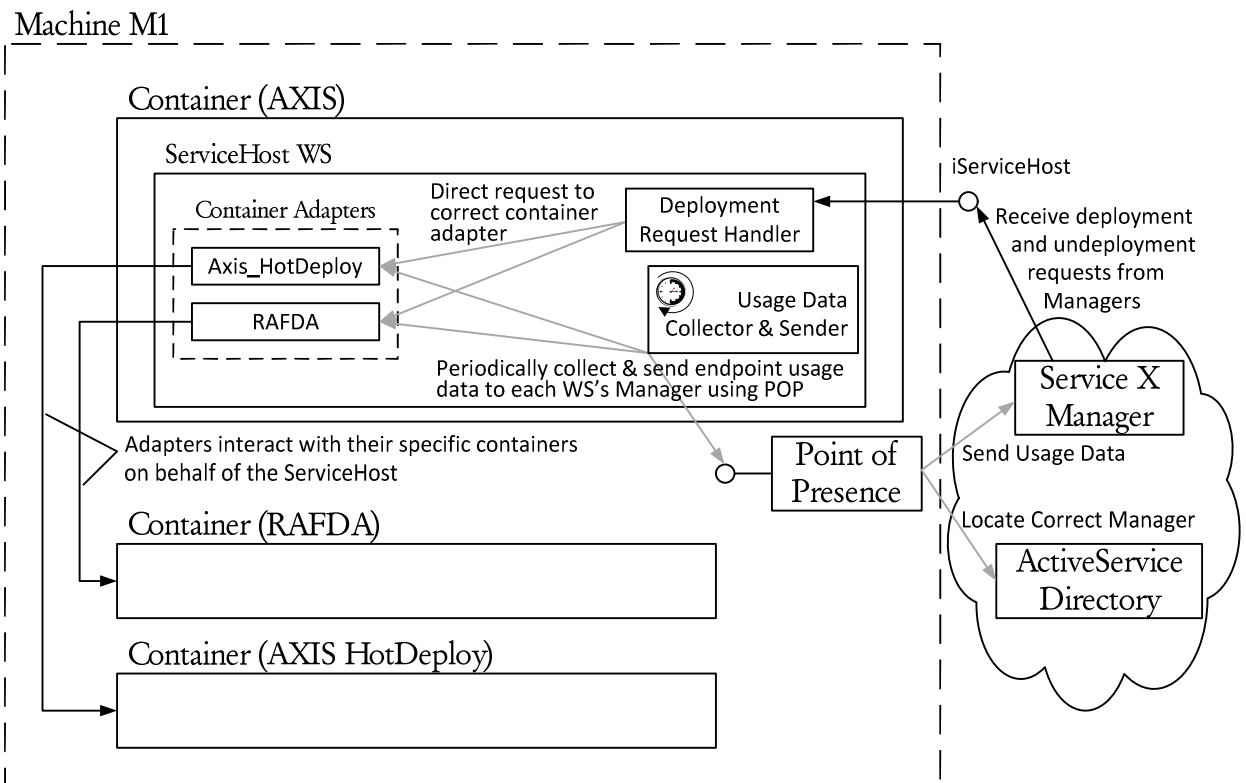
```
public interface IContainerAdapter {  
  
    public ServiceInstanceDescriptor deploy(  
        ServiceImplementationDescriptor serviceImplDescriptor,  
        Object[] args, byte[] code, Long deploymentID)  
        throws ResourceUnavailableException;  
    public void undeploy(ServiceInstanceDescriptor instanceDescriptor);  
    public String getContainerName();  
    public String getUsageData(String URI, Long deploymentID);  
}
```

**Fig. 69 The IContainerAdapter Java interface. Implementations of this interface provide ServiceHosts with the ability to deploy and undeploy Web Service endpoints into various Web Service containers on the ServiceHost's machine.**

Container adapters also interact with their specified container on behalf of the ServiceHost in order to retrieve endpoint usage data and convert it into a generic format. The process of collecting and reporting endpoint usage data is covered in more detail in the upcoming section 4.7.5 'Collecting & Reporting Usage Data'. Using container adapters allows the exposed ServiceHost Web Service to perform its generic role accepting deployment requests in a platform-independent manner, leaving the platform- and container-specific actions to the individual adapters. This design also allows containers to be added in the future without requiring changes to the ServiceHost implementation itself.

The diagram in Fig. 70 below shows an example ServiceHost hosted on a single machine with two deployment containers ('Axis HotDeploy' and 'RAFDA', as described by the HostDescriptor shown previously in Fig. 68) and this diagram will be used in the upcoming sections to demonstrate the deployment of a Web Service endpoint on a ServiceHost. It includes the ServiceHost provider agent application deployed as a Web Service in the Axis container, with

separate functional units implementing each of its primary tasks (called the ‘Deployment Request Handler’ and ‘Usage Data Collector & Sender’ respectively). The two supported containers are labeled with their respective names and are shown empty as they do not initially have any endpoints deployed in them. Each container’s corresponding container adapter is shown as an instantiated object in the ServiceHost provider agent application. Finally, directed links between all components are labeled to indicate the interactions that take place between them, including the receipt of deployment and undeployment requests from remote Managers and the reporting of endpoint usage to Managers via the local POP.

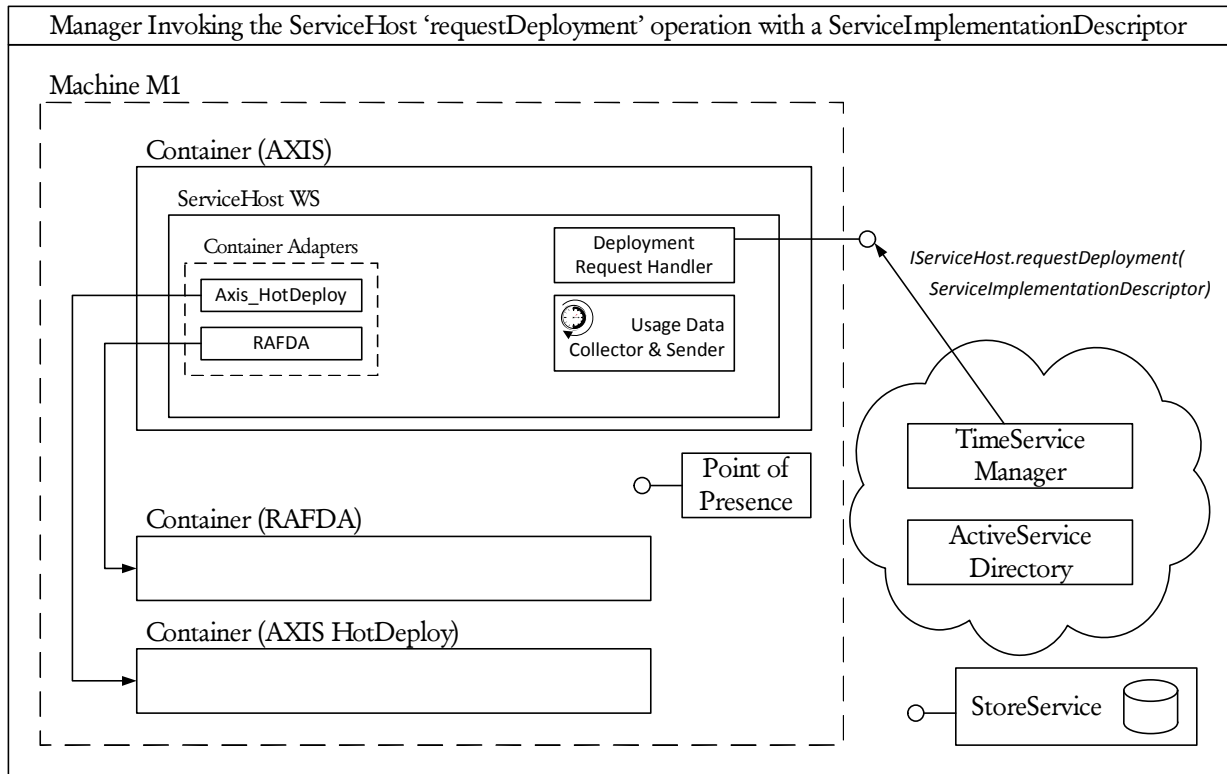


**Fig. 70 The structure of a ServiceHost machine, including the ServiceHost provider agent application deployed as a Web Service, its internal components, supported deployment containers and corresponding container adapters, and interactions with Managers.**

#### 4.7.4 FULFILLING PROVISIONING REQUESTS

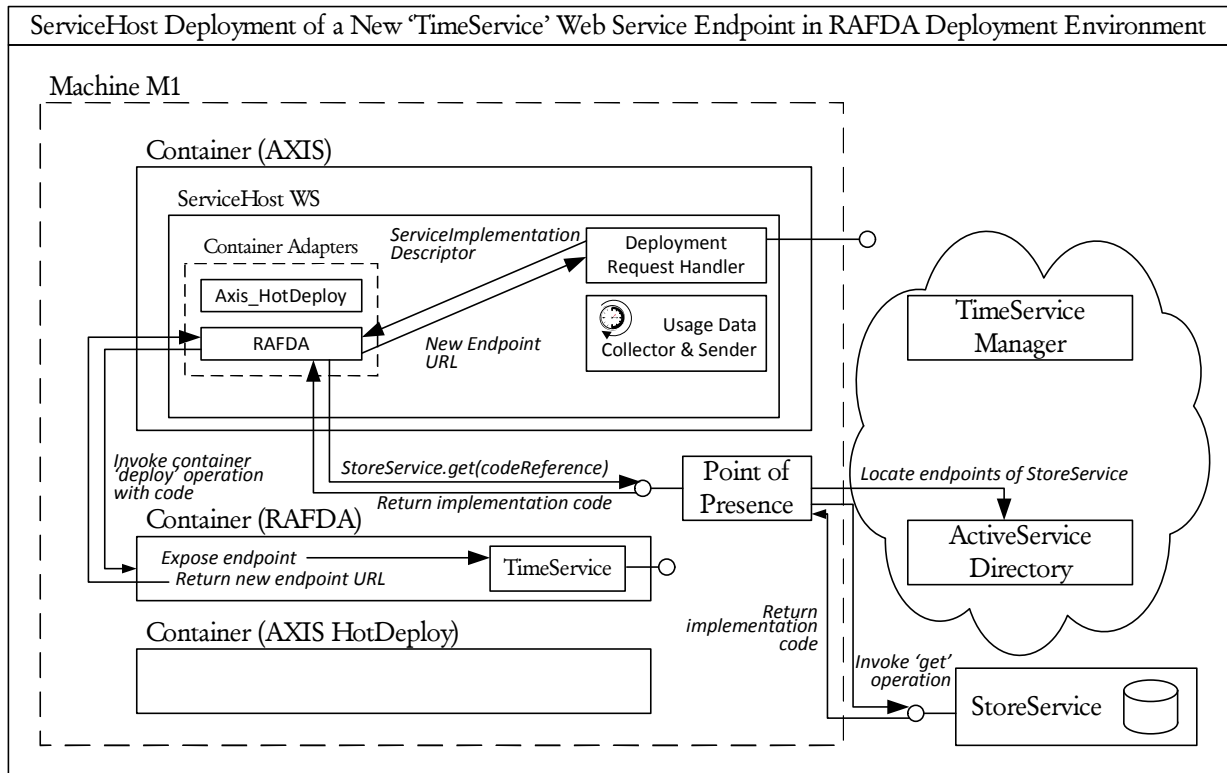
ServiceHosts receive deployment requests from Managers through the ‘requestDeployment’ operation of their exposed ServiceHost Web Service, as shown in Fig. 71 below. This operation takes a *ServiceImplementationDescriptor* as a parameter, which includes a reference to the target container and the code location. The first step a ServiceHost takes upon receiving a deployment request is to generate the deployment ID number (a random Long value). This value identifies the unique individual endpoint that results from a successful deployment process and is included in

the resultant `ServiceInstanceDescriptor`. Because it is important to respond accurately to future undeployment requests, this value is necessary for identifying the correct endpoint to undeploy in the event that there are multiple endpoints of the same Web Service deployed at the same `ServiceHost`.



**Fig. 71 Manager invoking the ServiceHost 'requestDeployment' operation with a `ServiceImplementationDescriptor`.**

After generating the deployment ID number, the `ServiceHost` next retrieves the implementation code from the URL included in the `ServiceImplementationDescriptor`'s 'CodeLocation' element. If successful, the 'ContainerName' element is retrieved from the `ServiceImplementationDescriptor` and a reference to the corresponding container adapter is retrieved from a local map. Container adapters implement the 'IContainerAdapter' interface, shown previously in Fig. 69, which includes a 'requestDeployment' operation. This operation is invoked with the `ServiceImplementationDescriptor` and retrieved code as parameters. Once invoked, it is up to the container adapter to interact with its corresponding container in order to effect the deployment and exposure of the indicated Web Service. The process of deploying a `TimeService` endpoint using the `RAFDA` container adapter is shown in Fig. 72 below.



**Fig. 72 ServiceHost deployment of a new 'TimeService' Web Service endpoint in the RAFDA deployment environment.**

If the endpoint is successfully deployed, the container adapter returns the URL of the newly deployed endpoint to the deployment request handler. If the endpoint cannot be deployed for any reason the container adapter instead returns null, which causes the deployment request handler to return a 'ResourceUnavailableException' to the Manager that requested deployment.

Before the URL of a successfully deployed endpoint is returned to the requesting Manager, the endpoint is first tested by the ServiceHost which retrieves the endpoint's WSDL by resolving the provided URL followed by the String '?WSDL'. If a WSDL document is retrieved then the deployment is deemed to be a success (note that although no conformance checks are performed on the WSDL, this would be a suitable place for testing to occur). A successful deployment of an endpoint of the TimeService Web Service, shown previously in Fig. 72, would be described by a ServiceHost using a ServiceInstanceDescriptor, such as that shown below in Fig. 73.

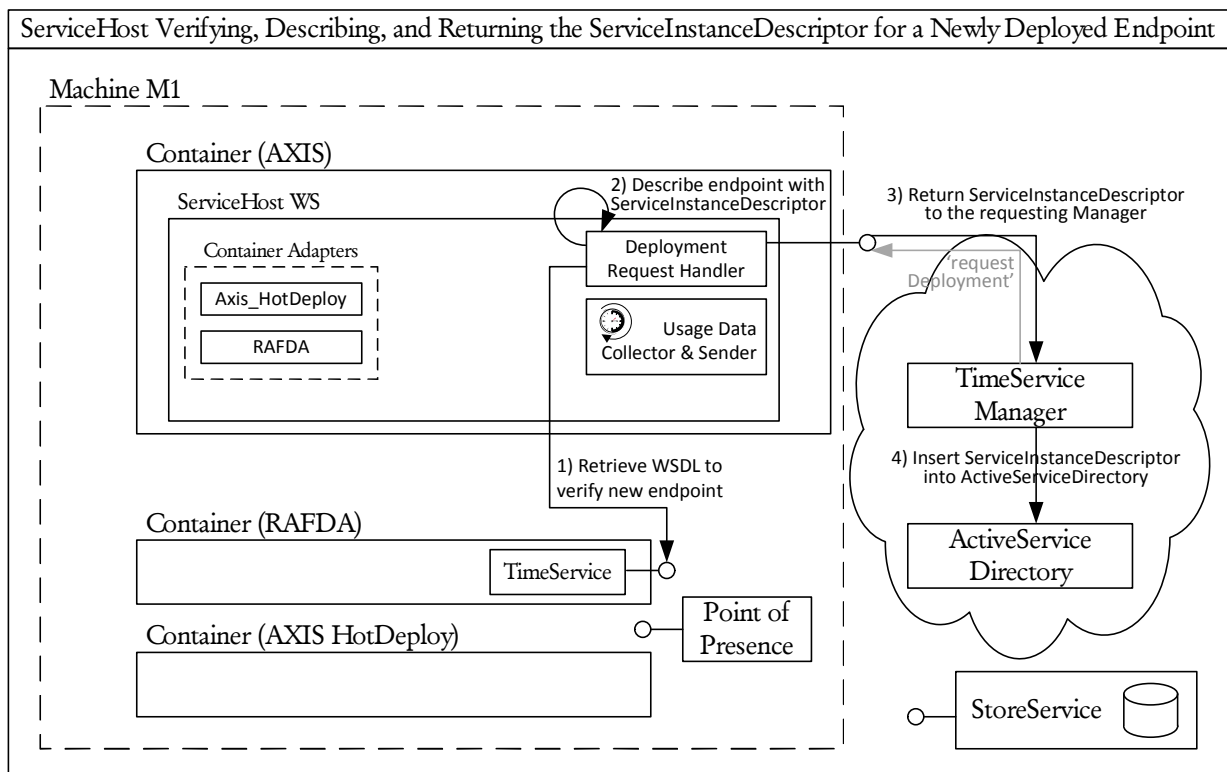
```

...
<multiRef id="id0" soapenc:root="0" soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/
encoding/" xsi:type="ns3:ServiceInstanceDescriptor" xmlns:soapenc="http://
schemas.xmlsoap.org/soap/encoding/" xmlns:ns3="http://util.rfmthesis.cs.standrews.ac.uk">
  <endpointURL xsi:type="xsd:string">
    http://xyz.com/RAFDA/services/TimeService
  </endpointURL>
  <hostDescriptor href="#id1"/>
  <implDescriptor href="#id2"/>
  <instanceID xsi:type="xsd:integer">
    1451057475215743030728584773222206527584438429144</instanceID>
  <timestamp xsi:type="xsd:dateTime">2009-01-26T17:25:42.255Z</timestamp>
</multiRef>
...

```

**Fig. 73 Example ServiceInstanceDescriptor describing an endpoint of the TimeService Web Service deployed at <http://XYZ.com/RAFDA/services/TimeService>.**

In the final step of deployment, shown in Fig. 74 below, the ServiceInstanceDescriptor is returned to the requesting Manager who inserts it into the ActiveServiceDirectory, making this newly deployed Web Service endpoint available to Service Consumers and completing the endpoint deployment process.



**Fig. 74 ServiceHost verifying, describing, and returning the ServiceInstanceDescriptor for a newly deployed endpoint.**

#### 4.7.5 COLLECTING & REPORTING USAGE DATA

ServiceHosts are responsible for extracting usage data for each Web Service endpoint deployed within their domain and reporting it to the Web Service's Manager. When a ServiceHost invokes the Manager's 'reportUsageData' operation they are returned a numerical value indicating the number of seconds the ServiceHost should wait before next reporting usage data for the same Web Service. To retrieve and report usage data the ServiceHost invokes the 'getUsageData' operation of the container adapter corresponding to the container in which the indicated Web Service is deployed, contacts that Web Service's Manager and invokes its 'reportUsageData' operation.

Container adapters implement the IContainerAdapter interface and are responsible for interacting with the container in order to retrieve the usage data corresponding to the indicated Web Service. Different containers may hold this data in different formats, so it is up to the container adapter to process the data and ensure that it conforms with the XML schema shown below in Fig. 75. Using this schema ServiceHosts may report endpoint usage as a full set of individual 'invocation' elements, each represented by a start and end time. Because representing large quantities of usage data with XML can result in an enormous document, and parsing this document can consume a large quantity of memory and processor resources, the schema also provides for more consolidated reports. These reports, of element type 'report', indicate the total number of invocations performed during the reporting period together with the average invocation duration, in milliseconds. A report can be in the form of a single 'serviceReport' representing the total number of invocations handled by the endpoint during the last reporting period, and the average duration of all invocations. Alternatively, this information may be reported at a finer grain using a collection of 'operationReports', one for each individual operation provided by the service that was invoked during the last reporting period. The 'report' type may also be extended to include additional fields which may provide Managers with greater insight into the behaviour of the service.

```

<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://localhost:8888/rfm_thesis/uk/ac/standrews/cs/rfmthesis/management">

<xs:element name="usageData">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="ServiceURI" type="xs:string"/>
      <xs:element name="Host" type="tns:HostDescriptor"/>
      <xs:element name="serviceReport" type="report"/>
      <xs:element name="operationReport" type="report" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="call" type="invocation" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:complexType name="report">
  <xs:sequence>
    <xs:element name="numInvocations" type="xs:int"/>
    <xs:element name="averageDurationMilliseconds" type="xs:int"/>
    <xs:element name="name" type="xs:string"/>
    <xs:any minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="invocation">
  <xs:sequence>
    <xs:element name="startTime" type="xs:time"/>
    <xs:element name="endTime" type="xs:time"/>
    <xs:any minOccurs="0"/>
  </xs:sequence>
</xs:complexType>

</xs:schema>

```

**Fig. 75 The XML schema for endpoint usage data reports. ServiceHosts send usage data to Managers in reports which conform to this schema.**

Once it is properly formatted the endpoint usage data is ready to be sent to the Web Service's Manager. To do this, the ServiceHost invokes the 'reportUsageData' operation of the Web Service identified by the concatenation of the Web Service's URI with the well-known 'Manager' suffix: "\_MANAGER", located at 'localhost' on port '8888'. This operation is received by the POP, which locates the correct Manager endpoint using the ActiveServiceDirectory, invokes the operation and returns the results to the ServiceHost. The diagram in Fig. 76 shows the interaction with the POP and its resultant endpoint-resolution and operation-invocation activities.

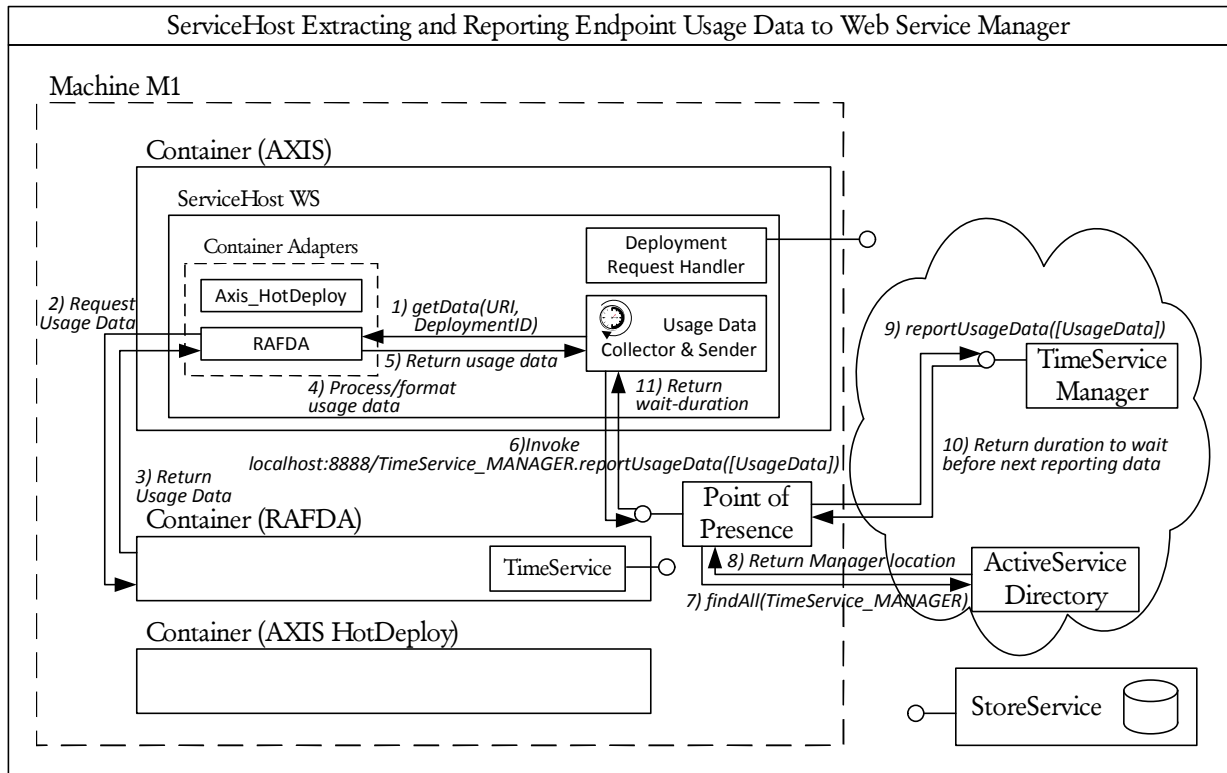


Fig. 76 ServiceHost extracting endpoint usage data via a container adapter and returning the results to the endpoint's Manager

As mentioned earlier, the 'reportUsageData' operation returns a value, in seconds, which specifies the length of time the ServiceHost should wait before next reporting usage data for the specified Web Service. After each cycle a new timer is set using this value and, when triggered, the process of collecting and reporting Web Service usage data is repeated until the endpoint is undeployed.

## 4.8 MANAGERS

All Managers are realized as Web Services and are published into and deployed by the infrastructure identically to the Web Services which they manage. A generic Manager implementation is provided for the management of all Web Services, but custom Managers may also be provided by Publishers for Web Services which require special management or deployment techniques. In keeping with the isolation of 'domain-specific knowledge' to specific actors, Publishers are responsible for publishing custom Managers – which are just Web Services implementing a particular interface, as described in the next section 'Creating Custom Managers'.

When a Web Service in the infrastructure experiences a period of zero demand, endpoints of the Web Service will gradually be undeployed by its Manager. If all endpoints of a Web Service

are undeployed, it becomes unnecessary for an endpoint of its Manager to exist until the next time the Web Service is used, and its Manager can therefore be undeployed as well. Just like any other Web Service in the infrastructure, Managers are thus actively managed by a Manager, and are undeployed when demand for them drops to zero – indicating that there are zero deployed endpoints of the Web Service they manage. This occurs because any endpoints of the Web Service they manage must be deployed on ServiceHosts, which are responsible for reporting data to the Manager by invoking the ‘reportUsageData’ operation; when the target Manager no longer receives invocation requests for this operation, no usage data will be logged by the container the Manager is deployed in. When the Manager’s Manager receives empty usage reports, it can reliably infer that there are no deployed endpoints of the Web Service under management, that this Manager is thus no longer needed, and that it can be undeployed and the resources it is consuming used for other services.

These Manager-Managers (which we call ‘MasterManagers’) use the same generic Manager implementation provided by the infrastructure, but are named differently in the ActiveServiceDirectory and are initially deployed by hand as part of the initial infrastructure setup process described in the next section. Before covering this bootstrapping process, however, the following sub-sections first describe the creation and publication of a custom Web Service Manager, followed by the generic manager’s implementation of the deployment planning process, and how endpoint usage data is used to make provisioning decisions.

#### *4.8.1 CREATING CUSTOM MANAGERS*

A custom Manager is simply a deployable Web Service implementing the IManager interface (Fig. 34 page 72) and published into the ServiceLibrary using a URI ending in ‘\_MANAGER’. For example, a custom Manager for the TimeService Web Service would be published under the URI ‘/TimeService\_MANAGER’. By implementing custom techniques for handling ‘reportUsageData’ invocations the custom Manager can utilize alternative methods of evaluating the required provisioning level of a Web Service based upon a closer understanding of the Web Service’s behaviour and the meaning of various changes in the returned usage data. The custom Manager can also implement a custom deployment planning process, matching ServiceImplementationDescriptors with HostDescriptors and selecting an optimal pair for deployment.

When demand arises for a Web Service with no currently deployed endpoints, and no currently deployed Manager, a new Manager first needs to be deployed for the Web Service. That Manager is then used to enact deployment of an endpoint of the requested Web Service. In

deploying this Manager, the MasterManager first looks in the ServiceLibrary for a custom Manager for the service, published under the well-known scheme of Web Service URI+‘\_MANAGER’. If a custom Manager is found, it will be deployed and used as the Manager for the Web Service. If a custom Manager is not found, the generic Manager will be used, which is published under the URI ‘/MANAGER’.

Regardless of whether a custom Manager or the generic Manager implementation is used for a Web Service, the resultant endpoint ServiceInstanceDescriptor describing the deployed Manager is stored in the ActiveServiceDirectory under the well-known URI scheme of Web Service URI + ‘\_MANAGER’. When the Manager needs to be contacted (such as to report usage data as described in the previous section) the POP locates the Manager by querying the ActiveServiceDirectory using this compound URI. As it is the Manager of the Manager Web Service, the MasterManager is registered and located in the ActiveServiceDirectory using the URI ‘/MANAGER\_MANAGER’.

#### *4.8.2 MANAGING SERVICE STATE*

Web Service provider agent applications can be broadly categorized as either stateful or stateless services. Stateful Web Services maintain some set of private information (‘state’) which is used to affect the results of invoked operations; because individual endpoints of a stateful Web Service may hold different sets of information, stateful Web Services often require Service Consumers to continually interact with the same service endpoint in order for the service to function properly. Stateless Web Services, on the other hand, either operate solely on the content of invocation request messages, or operate over dynamic state which is accessible to all instances of the service (‘shared state’).

The presented architecture does not provide any facilities or mechanisms for maintaining the consistency or durability of the shared state of stateless Web Services, and the provision of a shared ‘data plane’ is proposed for this purpose in chapter 6, section 5: ‘Future Work’. The architecture does nonetheless have three stateful directory services which, like all Web Services in the architecture, may have one or more endpoints deployed at any one time. Because having more than one endpoint of a stateful directory service could result in different endpoints holding different information, it was necessary to implement some means of maintaining consistency between the endpoints of the directory services.

When Managers deploy a new endpoint of a Web Service, they are responsible for describing the endpoint using a ServiceInstanceDescriptor and inserting this descriptor into the

ActiveServiceDirectory. Because there may be multiple endpoints of the ActiveServiceDirectory, Managers must invoke the 'addServiceInstance' operation of each active ActiveServiceDirectory endpoint in order to ensure that entries are consistent across all instances of the service. When Publishers and ServiceHosts register with the HostDirectory and ServiceLibrary, it is the directory endpoints themselves which take responsibility for ensuring that their directory entries are consistent – locating other endpoints of the directory and invoking the same add or remove operation on each one. When a new endpoint of a directory service must be deployed, the directory's custom Manager takes responsibility for locating an existing directory endpoint in the ActiveServiceDirectory and using it to pre-populate the maps of the newly deployed endpoint, ensuring that the entries are consistent across all instances of the directory before the newly deployed directory endpoint is listed in the ActiveServiceDirectory.

Even with multiple endpoints, concurrent failures are possible and, in the case of stateful services, would result in complete loss of the Web Service's operational state. While the described methods effectively maintain consistency across multiple endpoints of a stateful service, no mechanism is provided to ensure the long-term durability of the directory entries. Further, because there is no mechanism provided for writing the entries into non-volatile storage, the infrastructure directory services are initially empty upon deployment, requiring both Publishers and ServiceHosts to re-register with their respective directories. As this design results in an identical, 'clean slate' every time the infrastructure is deployed, it is ideal for testing and evaluating the infrastructure (as described in the following chapter 'Evaluation'). However, this arrangement would be inconvenient for real Publishers and ServiceHosts, and thus any 'real-world' infrastructure deployment would benefit from the introduction of a mechanism for durable, long-term storage.

#### **4.8.3 DEPLOYMENT PLANNING**

The generic deployment planning process involves first retrieving the list of implementations of the desired Web Service from the ServiceLibrary by invoking its 'findAll' operation. This operation returns a list of ServiceImplementationDescriptors describing implementations of the desired Web Service, each of which contains a 'PublisherID' element. The HostDirectory is next queried using each unique PublisherID, and the set of all ServiceHosts willing to deploy Web Services written by each Publisher is compiled. Next, the code shown below in Fig. 77 is used to generate the set of all valid deployment plans by comparing the 'requirements' of each implementation with the 'capabilities' of each host provider. A valid deployment plan is

represented using a 'DeploymentPair' object which consists of a ServiceImplementationDescriptor and a HostDescriptor (describing a compatible ServiceHost).

```
public Set<DeploymentPair> generateDeploymentPairs(  
    ServiceImplementationDescriptor[] impls, HostDescriptor[] hosts) {  
    Set<DeploymentPair> deploymentPairs = new HashSet<DeploymentPair>();  
    String deploymentContainer;  
    for(int i=0; i<impls.length; i++){  
        deploymentContainer = impls[i].getDeploymentContainer();  
        for(int j=0; j<hosts.length; j++){  
            String[] capabilities = hosts[j].getCapabilities();  
            for(int k=0; k<capabilities.length; k++){  
                if(deploymentContainer.equalsIgnoreCase(capabilities[k])){  
                    deploymentPairs.add(new DeploymentPair(hosts[j],impls[i]));  
                }  
            }  
        }  
    }  
    return deploymentPairs;  
}
```

**Fig. 77 Java source code used to generate the set of all valid deployment plans from a set of ServiceImplementationDescriptors and a set of HostDescriptors**

While the process of generating and selecting a deployment plan must be generically applicable, there is no universal criteria for ranking and selecting one plan above another, so we must simply choose one at random (using the 'selectRandomDeploymentPair' method). If a plan fails it is removed from the list and an alternative plan selected and executed until deployment is successful or all options have been exhausted. The deployment process is implemented in the 'deploy' method, shown below in Fig. 78.

```

public ServiceInstanceDescriptor deploy(String serviceURI)
    throws ResourceUnavailableException{

    ServiceLibrary serviceLibrary = new ServiceLibraryProxy();
    HostDirectory hostDirectory = new HostDirectoryProxy();

    ServiceImplementationDescriptor[] impls = serviceLibrary.findAll(serviceURI);
    HostDescriptor[] hosts = getHostsForPublishers(hostDirectory, impls);

    Set<DeploymentPair> deploymentPairs = generateDeploymentPairs(impls, hosts);
    DeploymentPair selectedPair;
    ServiceInstanceDescriptor instance;

    while(!deploymentPairs.isEmpty()){
        selectedPair = selectRandomDeploymentPair(deploymentPairs);
        instance = safeDeployEndpoint(selectedPair);
        if(instance != null){
            return instance;
        }else{
            deploymentPairs.remove(selectedPair);
        }
    }
    throw new ResourceUnavailableException(
        "Unable to deploy a new endpoint of service with URI '"+serviceURI
        + "'.\n All options have been exhausted in the fulfillment of this request,"
        + " please try again at a later date.");
}

```

**Fig. 78 Java source code implementing the deployment planning and execution process in the generic Manager implementation.**

#### 4.8.4 THE GENERIC MANAGER

The design of the generic manager was approached with the goal of producing a simple, straight-forward decision-making process that could be used to gain insight into the behaviour of a [fairly complex] system and to programmatically deduce whether more or fewer endpoints of a service were necessary to meet the measured demand. The reference implementation of the generic manager is designed to be able to be reliably used for testing the ability of the system to balance resource consumption with demand, as will be evaluated in the upcoming chapter.

The utility of any universally-applicable generic manager rests on the selection of a metric whose fluctuations can be trusted to accurately indicate the availability of an arbitrary Web Service. As there is no knowing what kind of target system one is communicating with, or over which protocols, one of the only reliable measures of acceptable performance in network communications is simply ‘how long it takes’. Growth in the average round-trip-time for invocations of a Web Service’s operations can indicate that some component of the target system is being stressed beyond capacity, that requests are beginning to queue up, and that additional load should be directed elsewhere (i.e. that an additional endpoint of the Web Service should be deployed and its location added to the ActiveServiceDirectory).

To that end, the generic manager employs a very simple comparison process based upon the average invocation response time of a Web Service, averaged over all endpoints of the service within the current reporting period: if this value is more than twice its average over the past three reporting periods then a new endpoint will be deployed and the historical average reset; if the average invocation response time for a period is less than half of the historical response time then an endpoint will be undeployed until there are none remaining. While there are a multitude of alternatives, the selection of a historical response-time as the metric of choice came down to a few important factors. First, the value is simple to calculate, and it was clear that a decision-making process with clear inputs and discernable, easily explainable reactions was desirable. Secondly, tests showed that the value was consistent – a crucial factor for producing repeatable results, and thus enabling a quantitative evaluation to arrive at meaningful conclusions.

This simple choice of metric does not come without its drawbacks and the generic Manager clearly has its limitations. For example, it does not consider the simple fact that different operations of the same Web Service can take different lengths of time, or indeed that a single operation may have wildly different response times under normal operating conditions. Under low call volume, an anomalously long response time can skew the analysis and may trigger unnecessary deployments. Similarly, failed calls are not indicated in the results, and thus a number of short response times may be mistaken as indicative of an overly-provisioned service and an endpoint of the service erroneously undeployed. Despite these limitations, however, the generic Manager implementation more than adequately demonstrates the capabilities of the system and, importantly, can reliably be used for testing the ability of the system to balance resource consumption with demand, as will be evaluated in the upcoming chapter. As an alternative measure of demand, a manager implementation which makes provisioning decisions based simply on the number of invocation requests will also be included in the evaluation.

#### *4.8.5 BOOTSTRAPPING PROCESS*

The process of initially setting up an instantiation of the infrastructure requires a small number of manual steps to be carried out before it is ready for service. These steps may be completed individually by a human administrator or automated through the use of a script (as is done in the next chapter for the set up of the experimental test environment).

The first step of the bootstrapping process is the deployment of an ActiveServiceDirectory endpoint at a well-known location. This ActiveServiceDirectory endpoint needs to be described using a ServiceInstanceDescriptor and inserted into the ActiveServiceDirectory itself using the ‘addActiveService’ operation. The ServiceLibrary is deployed next, and implementations of the

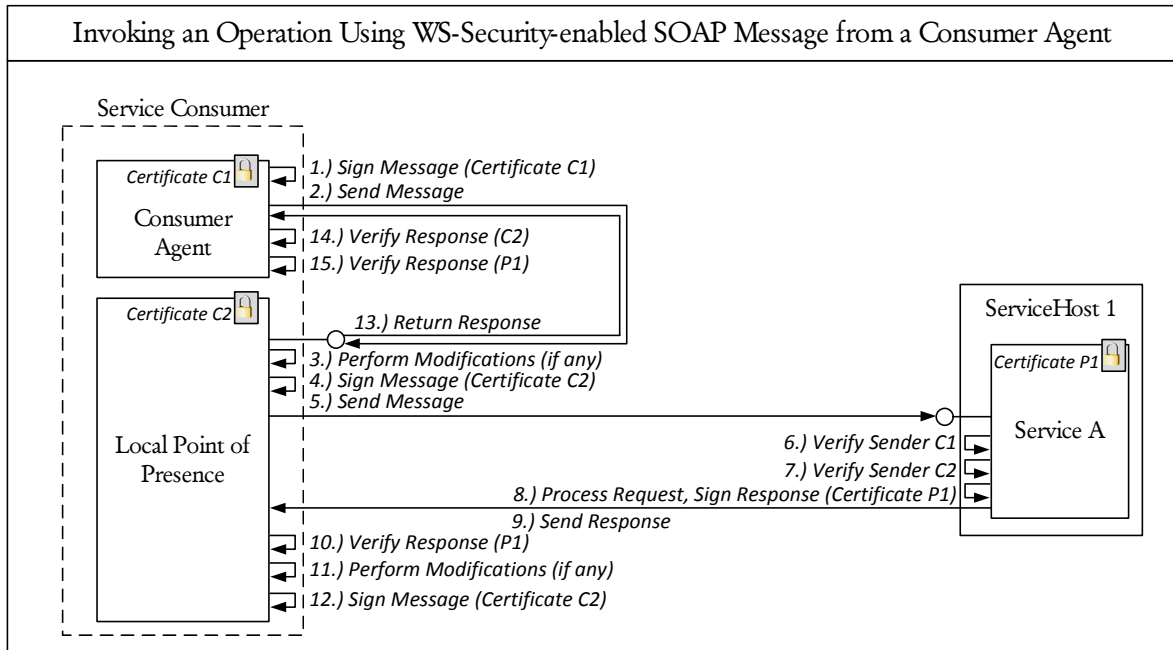
ActiveServiceDirectory and ServiceLibrary published into it. Next a HostDirectory endpoint is deployed and its implementation also published in the ServiceLibrary. The generic Manager implementation is published in the ServiceLibrary next and at least one ServiceHost capable of deploying it is registered in the HostDirectory. Lastly, an endpoint of the generic Manager implementation is deployed with a runtime parameter indicating that it is a MasterManager, and registered in the ActiveServiceDirectory under the URI `’/MANAGER_MANAGER’`. The Manager is written to maintain a minimum of two endpoints of the Web Service identified by the URI `’/MANAGER_MANAGER’` (of which it is one). When it is first deployed, this Manager first retrieves the list of master managers from the ActiveServiceDirectory, identifies itself as the sole MasterManager, and promptly initiates the process of deploying an additional one. Because this autodeployment process requires successful interactions with the ActiveServiceDirectory, the ServiceLibrary, the HostDirectory, and a ServiceHost, the success or failure of deployment is taken to indicate the success or failure of the infrastructure startup process. The MasterManager successfully executing the autodeployment process signals the end of the bootstrapping procedure and indicates that the infrastructure is up, operational, and ready to serve the needs of its users.

#### 4.9 SECURITY CONSIDERATIONS

Secure communication between consumer agent applications and Web Services is commonly introduced at the message level, using WS-Security [107], and/or at the transport level, using HTTPS [108]. WS-Security is a set of standards provided by the OASIS foundation as extensions to the SOAP specification which enable Service Consumers and Service Providers to verify the integrity and authenticity of SOAP messages. Senders of WS-Security-enabled SOAP messages sign outgoing messages and append a digest element representing a content hash of the message; the receiver of a WS-Security enabled SOAP message can use these elements to verify the identity of the sender and determine whether the message has been tampered with.

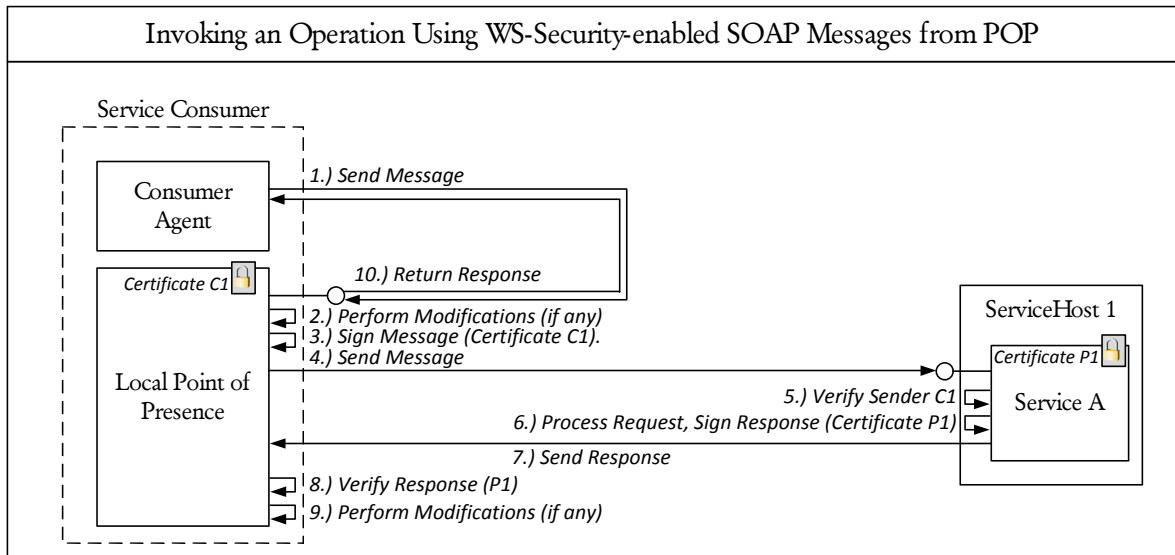
The body element of a WS-Security-compliant SOAP message is not encrypted, so it remains possible for an intermediate party to observe and, if desired, alter and re-sign a SOAP message. This is considered a benefit of the specification because it allows transactions to take place between multiple parties asynchronously – that is, for multiple parties to be involved in the exchange of a document before it is returned to the originating Service Consumer. By interacting with a Web Service using the WS-Security extensions, both Service Consumer and Service Provider can verify the integrity of SOAP documents and the identity of each party involved in the transaction.

WS-Security can be introduced into the presented architecture either exclusively at the Service Consumer's POP, or in both consumer agent applications and the POP. Because the POP performs minor modifications to outgoing SOAP requests, consumer and provider agent applications which utilize the WS-Security SOAP extensions would have to authorize the Service Consumer's POP as an additional party in each service interaction. In these cases the POP would sign each modified WS-Security-enabled message, and would thus require its own set of security credentials, as shown in Fig. 79 below.



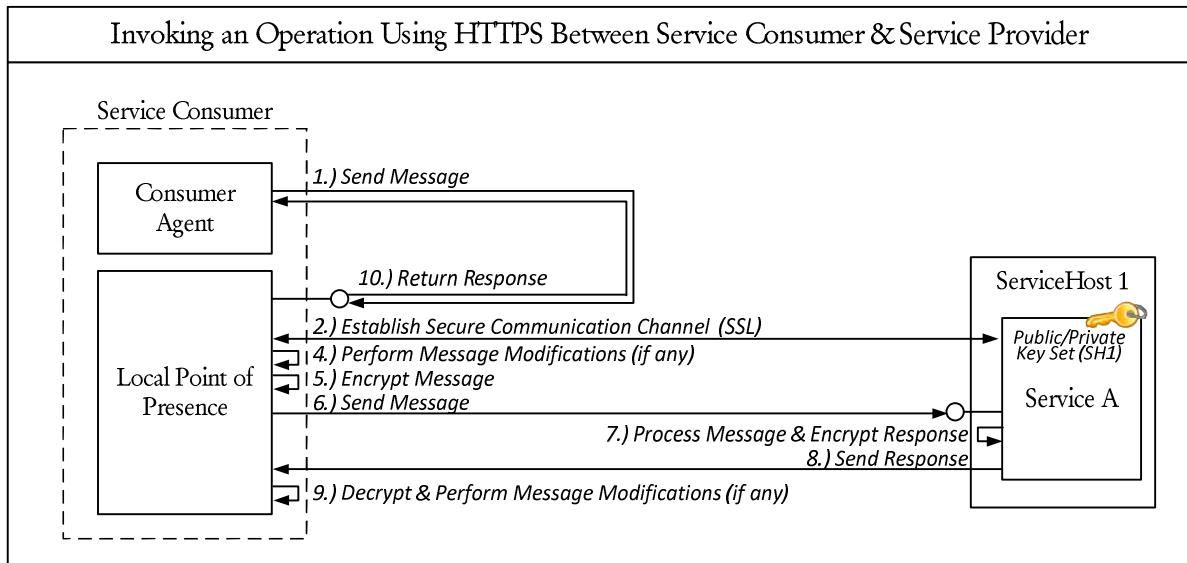
**Fig. 79 The process of invoking an Web Service operation using WS-Security-enabled SOAP message originating from the Service Consumer's consumer agent application, including intermediate signing steps and additional POP credentials.**

Alternatively, if the Service Consumer considers the communication channel between their consumer agent applications and their local POP to be secure, the POP could itself be endowed with the credentials of the Service Consumer and could consume all security considerations on their behalf. This arrangement saves developers from implementing the WS-Security specifications, and saves the computational effort required to compute an additional digest for each proxied message, while still providing the means to verify message authenticity and integrity between the Service Consumer and a service endpoint. This alternative arrangement is shown in Fig. 80 below.



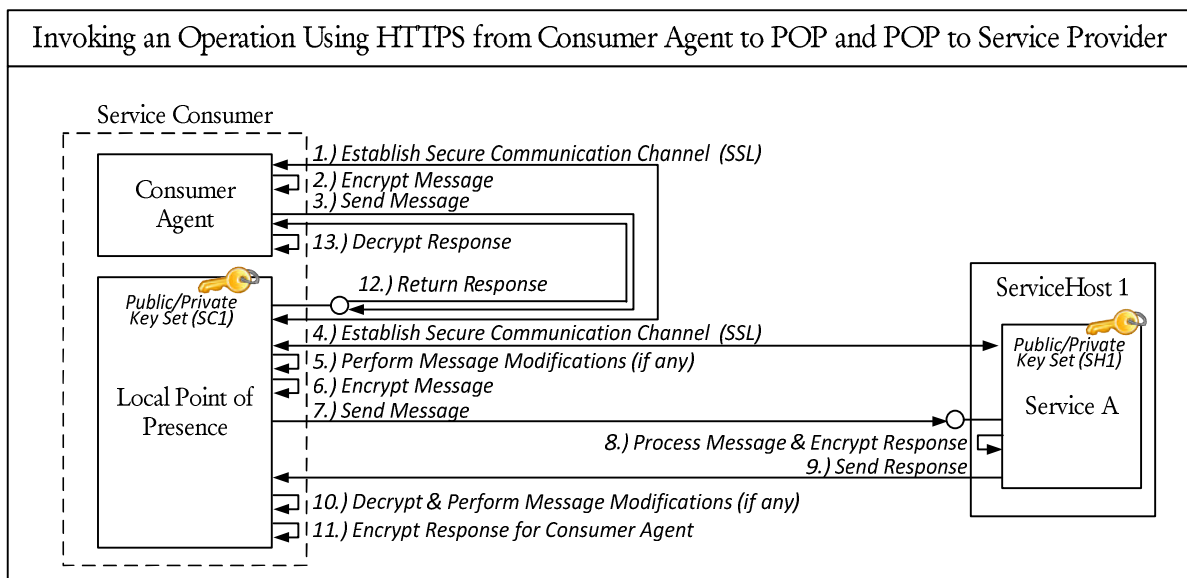
**Fig. 80** The steps involved in signing and verifying WS-Security-enabled SOAP messages when Service Consumer security provisions are isolated in the local POP.

Service Consumers and Service Providers may alternatively (or additionally) utilize HTTPS for secure communication if there is a need to protect the message content from being observed. HTTPS provides point-to-point, transport-level encryption of a communication channel established using the SSL [109] protocol. Once established, a Service Consumer and a Service Provider communicate across the channel using the HTTP protocol, sending messages in an encrypted format that can only be decrypted using the recipient's private key. As with WS-Security, HTTPS is most elegantly introduced into the presented architecture at Service Consumer's local POP as this arrangement requires only one set of security credentials per-Service Consumer. Further, this arrangement isolates the Service Consumer's security concerns into a single entity, thus saving developers from redundant effort and enabling existing consumer agent applications to use secure communication without requiring any changes to their code. Introducing security at the POP also provides a single point of evolution if new or different security protocols are required in the future, again saving developers from re-implementing every consumer agent application to adjust to these changes. The introduction of HTTPS communication into the architecture at the Service Consumer's local POP is shown in Fig. 81 below.



**Fig. 81** The steps involved in sending a message between a Service Consumer and a service endpoint over a secure HTTPS communication channel established by the POP.

If a consumer agent application is already written using HTTPS, the Service Consumer's local POP would not be able to function correctly without its own set of credentials. The only way to integrate end-to-end HTTPS into the presented architecture is to secure first the channel between an executing consumer agent application and the local POP, and secondly between the local POP and any remote server that it communicates with. This arrangement requires a separate set of credentials for the Service Consumer's POP, as shown in Fig. 82 below.



**Fig. 82** The steps involved in establishing and communicating using HTTPS between a consumer agent application and the POP, and between the POP and a service endpoint.

Introducing security into an implementation of the presented architecture will require a number of practical considerations. Using HTTPS, for example, has the benefit of simplicity and secrecy (because the content of the messages is encrypted) although the encryption and decryption processes can be computationally intensive [110]. Additionally, in the case of Service Hosts whose domain is comprised of multiple systems, decrypting incoming SOAP requests at the point of entry into their domain before routing the request to the final endpoint may result in the message being sent through the internal network in an insecure format. The benefit of WS-Security in this regard is that message integrity can be verified right up until final processing. Finally, HTTPS and WS-Security can also be used in conjunction with one another if an application requires both point-to-point secrecy and verifiable message integrity.

## 5 EVALUATION

### 5.1 INTRODUCTION

The following sections detail the specifications and results of an experimental evaluation of the presented architecture. These experiments have been carried out using the described reference implementation and a collection of 53 identically configured machines connected in an isolated 1Gbit Ethernet network. A single leader node was used to direct the execution of the experiments and two machines were left aside as replacements for any machines that failed during testing. The machine specifications are presented first below, followed by a description of the target Web Service used for testing and the architecture of the test-rig. An overview of the starting-state of the instantiated infrastructure services is presented next, followed by the experiment specifications, methodologies, results and analysis.

#### 5.1.1 THE TEST ENVIRONMENT

All of the machines used for testing are identical in terms of their hardware and installed software. The name of the head node is “blub” and the remainder of the machines are named using the scheme ‘compute-0- $n$ ’ where  $n$  is a unique value between 0 and 52. The value of  $n$  used in the name of the machine corresponds directly to the IP address of the machine, with the format ‘10.1.255. $n$ ’. The hardware and software specification of the machines is presented below.

#### 5.1.2 HARDWARE

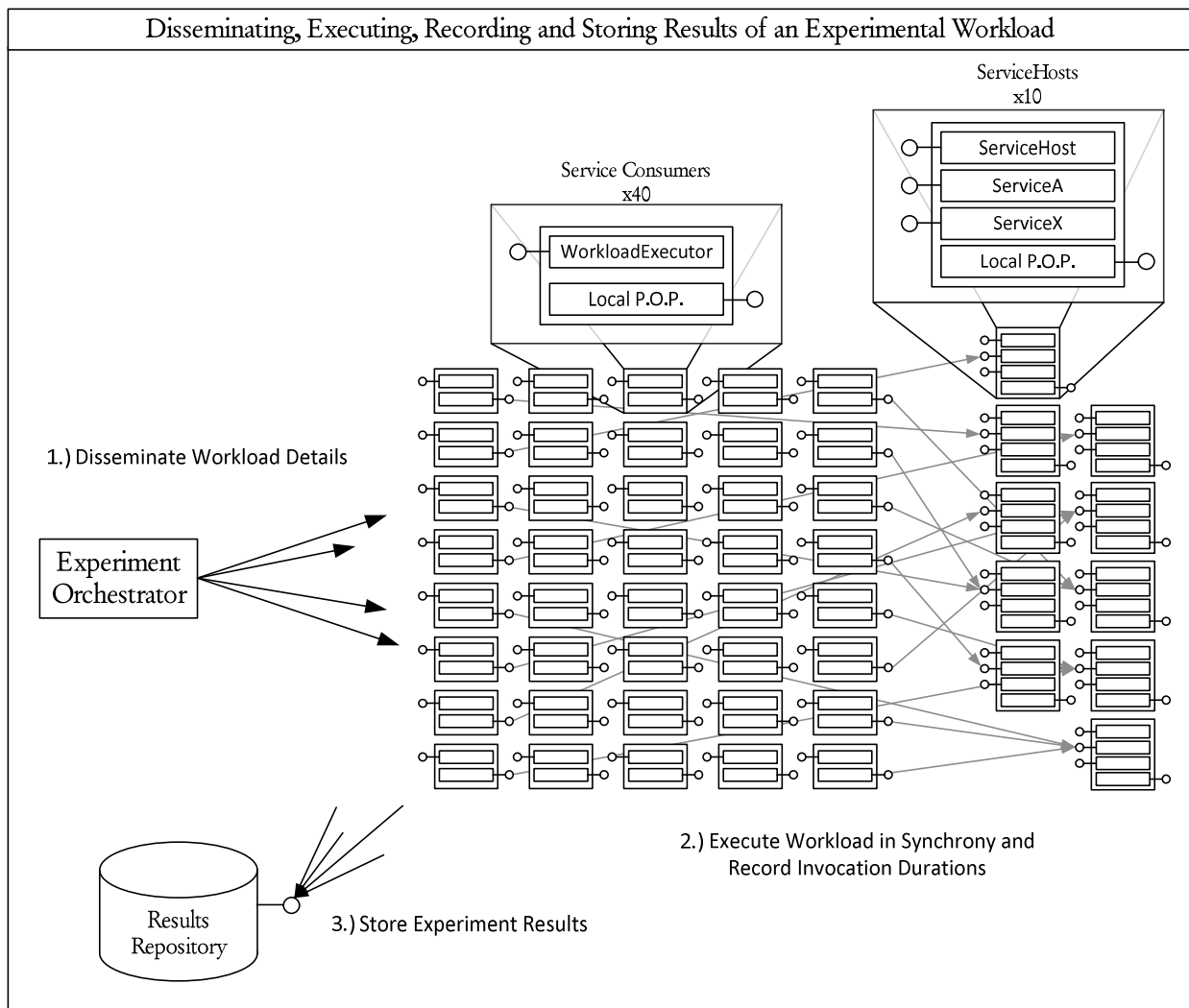
- Number of CPUs: 2
  - Model: Intel® Zeon™
  - CPU Clock: 2.34GHz
  - Cache Size: 512KB
  - CPU Cores: 1
- Memory (RAM): 2GB
- Hard Disk: 20GB IDE @ 5400RPM
- NIC: 1Gbit Ethernet (Intel 82544GC rev.02)

#### 5.1.3 SOFTWARE

- Operating System: Linux version 2.6.18-92.1.10.el5 x86 (CentOS 5.2)
- Java™ SE Runtime Environment version 1.6.0 (build 1.6.0\_07-b06)
- Application Servers:
  - Apache Tomcat-5.5.17
    - Including Apache Axis 1.4 Final
  - JBoss-4.2.1.GA
  - RAFDA 2.0

#### 5.1.4 OVERVIEW OF EXPERIMENT EXECUTION PROCESS

The fifty machines in the experiment test-rig are divided into two groups: ten to provide an infrastructure for evaluation, and forty to participate as Service Consumers. Experiments begin with an experiment orchestrator describing the experimental workload and disseminating this workload to the Service Consumer machines. Each Service Consumer is responsible for executing the workload at the specified time, recording the round-trip time for each invocation, and reporting the results to a repository where they can later be retrieved and analyzed. This process is shown in Fig. 83 below.



**Fig. 83 Disseminating, executing, recording and storing the results of an experimental workload**

The specifics of describing, disseminating and executing an experiment are covered in each of the next sections. The 'TargetExperimentEndpoint' Web Service is described first, followed by the 'WorkloadDetails' document used to describe the experiment workload, the

'ResultsRepository' Web Service, the pre-experiment check-list of required deployments and directory entries, and finally the experiments themselves.

#### 5.1.5 THE 'TARGETEXPERIMENTENDPOINT' WEB SERVICE

A Web Service called 'TargetExperimentEndpoint' is used in the experiments as the target Web Service whose operations are invoked by Service Consumers. This Web Service serves as a generic placeholder, used to demonstrate the techniques of the architecture without displaying any service-specific behaviour. Service Consumers interact with the service by invoking a single 'doNothing' operation, shown in the Fig. 84 below. This operation takes no parameters and returns immediately, returning no value. The only computation performed upon invoking this operation occurs in the HTTP- and SOAP-stack layers.

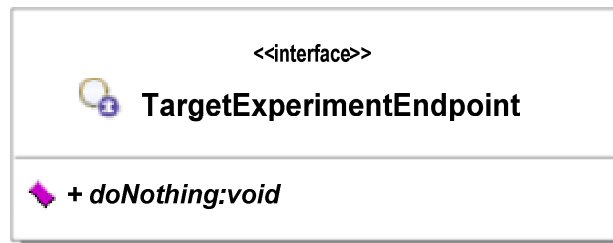


Fig. 84 Interface for the 'TargetExperimentEndpoint' Web Service used in the upcoming experiments, indicating the single 'doNothing' operation.

#### 5.1.6 DESCRIBING WORKLOADS WITH 'WORKLOADDETAILS'

Before an experiment can begin the Experiment Orchestrator first describes the experiment workload to be executed using a 'WorkloadDetails' descriptor. The 'WorkloadDetails' descriptor describes the duration, target Web Service, and invocation workload pattern for a single experiment. The descriptor includes the name of the TargetExperimentEndpoint operation to invoke, the number of times per-second to invoke the operation, and instructions on how the invocation rate should vary during the experiment in order to simulate various usage patterns. The descriptor further specifies the number of worker threads each RemoteOperative should use to execute the workload, a number of parameters used for invoking the TargetExperimentEndpoint operations, a reference to the ResultsRepository Web Service, and a unique 'jobID' used to store the results once the experiment is complete.

The application in Fig. 85 is used by the Experiment Orchestrator to craft and launch experiment workloads. Each of the GUI fields corresponds to a single WorkloadDetails attribute, each of which are described in the bullet list directly following Fig. 85. When the Experiment Orchestrator clicks the 'Launch' button the application creates a WorkloadDetails descriptor and distributes it to the specified number of Service Consumer nodes for execution.

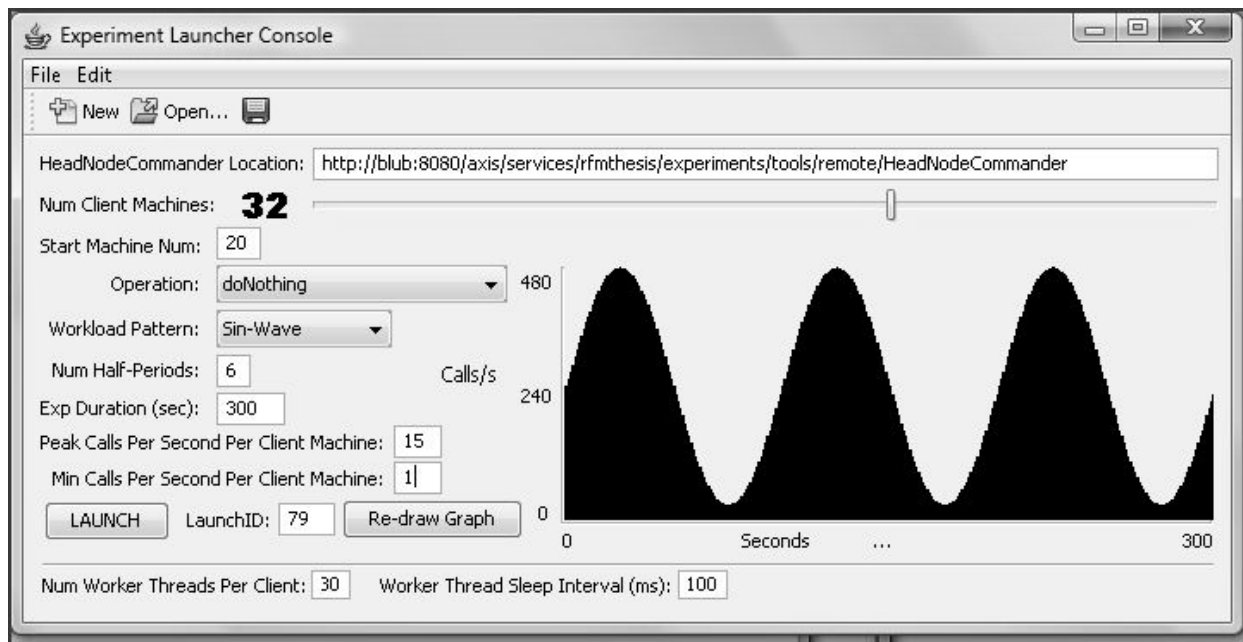


Fig. 85 The Experiment Launcher Console is used to describe and launch experiment workloads

- **experimentDuration:Integer**
  - Running time of the experiment, in seconds.
- **operationName:String**
  - Specifies which TargetExperimentEndpoint operation to invoke.
- **jobID:Integer**
  - Uniquely identifies this particular job. The jobID is used when storing results in the ResultsRepository.
- **endpointURL:String**
  - The URL of the TargetExperimentEndpoint Web Service
    - This URL is not included as a field in the GUI in Fig. 85 because it does not change during the experiments.
    - For all experiments the location is:

`http://localhost:55555/rfmthesis/experiments/TargetExperimentEndpoint`

- **reportServiceURL:String**
  - URL of the ResultsRepository Web Service used to store the results of this experiment.
- **workloadPatternID:Integer**
  - RemoteOperatives are capable of executing a number of workload patterns, each of which has a unique 'workloadPatternID'. The workload patterns are:
    0. Linear
    1. Linear Increasing/Decreasing
    2. Square-wave
    3. Sin-wave

- **numThreads:Integer / workerSleepTime:Integer**
  - These values dictate the number of independent threads (numThreads) that each RemoteOperative should use to execute the specified workload, and the length of time, in milliseconds, that the threads should sleep before fetching a new job (workerSleepTime).
- **startCPS:Integer / endCPS:Integer / waveHalfPeriods:Integer**
  - These values dictate the minimum and maximum calls-per-second (CPS) to perform at the start (startCPS) and the end (endCPS) of an experiment. For square- and sin-wave workloads the startCPS defines the lowest CPS value and the endCPS defines the highest CPS value. The 'waveHalfPeriods' value defines the number of half-waves to perform during the experiment (i.e. the number of times to cycle between the startCPS value and the endCPS value, or vice-versa).

Fig. 86 below shows an example WorkloadDetails descriptor describing a 600-second linearly increasing workload to be performed at UTC 1241961077693 from machines compute-0-10 through compute-0-29, each using 30 worker threads to invoke the 'doNothing' operation of the 'TargetExperimentEndpoint' Web Service at a starting rate of 5 CPS. The response-time is measured for each invocation as the call rate increases linearly, reaching a maximum value of 80 CPS at the end of the experiment at which time the results are to be stored under jobID number 318.

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <multiRef id="id1" ... xsi:type="xsd:long" ... >1241961077693</multiRef>
    <multiRef id="id0" ... xsi:type="ns2:WorkloadDetails"
      xmlns:ns2="http://util.experiments.rfmthesis" ... >
      <endCPS>80</endCPS>
      <endpointURL xsi:type="soapenc:string">
        http://localhost:55555/rfmthesis/experiments/TargetExperimentEndpoint
      </endpointURL>
      <experimentDuration>600</experimentDuration>
      <jobID>318</jobID>
      <numThreads>30</numThreads>
      <operationName xsi:type="soapenc:string">doNothing</operationName>
      <resultsRepositoryURL xsi:type="soapenc:string">http://localhost:55555/
        rfmthesis/experiments/tools/remote/ResultsRepository
      </resultsRepositoryURL>
      <startCPS>5</startCPS>
      <workerSleepTime>100</workerSleepTime>
      <workloadPatternID>0</workloadPatternID>
    </multiRef>
  </soapenv:Body>
</soapenv:Envelope>
```

**Fig. 86 Example 'WorkloadDetails'.** These details are passed to each of the specified 'RemoteOperative' endpoints which are responsible for executing the workload and reporting the results to the 'ResultsRepository' Web Service located at the URL specified in the 'ResultsRepositoryURL'.

The execution of this workload results in the participating Service Consumers performing a number of invocations of the 'doNothing' operation of the 'TargetExperimentEndpoint' Web Service, an example of which is shown in Fig. 87 below. For ease of reading, some content unrelated to the current discussion has been replaced by ellipses.

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:doNothing ... xmlns:ns1="http://localhost:55555/rfmthesis/experiments">
    </ns1:doNothing>
  </soapenv:Body>
</soapenv:Envelope>
```

**Fig. 87 Invocation of the 'doNothing' operation of the 'TargetExperimentEndpoint' Web Service using the POP.**

#### 5.1.7 REPORTING AND RETRIEVING EXPERIMENT RESULTS

Once an experiment is complete, each Service Consumer reports their results using a Web Service called the 'ResultsRepository' which exposes two operations: 'reportResults(Integer jobId, String[] results):void' and 'retrieveResults(Integer jobId):String[]'. The start and end time of every invocation is stored by each Service Consumer and they do not perform any pre-processing over this data. A sample set of results returned from an invocation of the 'retrieveResults' operation of the ResultsRepository Web Service is shown in Fig. 88 below:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope ... >
  <soapenv:Body>
    <ns1:retrieveResultsResponse xmlns:ns1="http://localhost:55555/rfmthesis/experiments/tools/remote">
      <retrieveResultsReturn soapenc:arrayType="xsd:string[1372]" xsi:type="soapenc:Array" ... >
        <retrieveResultsReturn xsi:type="xsd:string">compute-0-1</retrieveResultsReturn>
        <retrieveResultsReturn xsi:type="xsd:string">START_TIMES</retrieveResultsReturn>
        <retrieveResultsReturn xsi:type="xsd:string">1241972687375</retrieveResultsReturn>
        ...
        <retrieveResultsReturn xsi:type="xsd:string">END_TIMES</retrieveResultsReturn>
        <retrieveResultsReturn xsi:type="xsd:string">1241972696736</retrieveResultsReturn>
        ...
      </ns1:retrieveResultsResponse>
    </soapenv:Body>
  </soapenv:Envelope>
```

**Fig. 88 A results set returned from an invocation of the 'retrieveResults' operation of the ResultsRepository Web Service.**

ServiceHosts are used throughout the experiments to provide hosting capacity as necessary to meet demand for both the 'TargetExperimentEndpoint' Web Service and the directory and

managerial services of the infrastructure. The start-up steps for instantiating the infrastructure components and registering these Service Hosts in the Host Directory is presented next.

#### **5.1.8 PRE- AND POST-EXPERIMENT CHECK-LISTS**

In order to provide a consistent environment for running experiments, a fixed list of actions are performed before and after each experiment. The first list of actions contain descriptions of the steps necessary to configure the environment by deploying and publishing the necessary services that comprise a running 'infrastructure'. The second list describes the steps required to remove any infrastructure or infrastructure-related processes once the experiment is complete.

##### **Pre-Experiment:**

1. Deploy 'ServiceHost' Web Service on each ServiceHost node [0-9]
2. Deploy 'WorkloadExecutor' Web Service on each Service Consumer node [10-50]
3. Deploy 'ActiveServiceDirectory' Web Service at 'compute-0-0'
  - a. This is the 'well-known location' provided to each POP instance
4. Register ActiveServiceDirectory location in ActiveServiceDirectory
5. Launch POP on all nodes [0-50]
6. Deploy 'ServiceLibrary' Web Service on any ServiceHost
  - a. Register ServiceLibrary location in ActiveServiceDirectory
  - b. Publish implementations of ActiveServiceDirectory, ServiceLibrary, Manager, and HostDirectory Web Services into ServiceLibrary
7. Deploy 'HostDirectory' Web Service on any ServiceHost
  - a. Register HostDirectory location in ActiveServiceDirectory
8. Describe and register ServiceHost nodes [0-9] in HostDirectory
9. Deploy 'Manager' Web Service at any ServiceHost
  - a. Register location in ActiveServiceDirectory under URI '/MANAGER\_MANAGER'
10. Publish 'TargetExperimentEndpoint' Web Service in ServiceLibrary

##### **Post-Experiment:**

1. Undeploy 'ServiceHost' Web Service at ServiceHost nodes [0-9]
2. Undeploy 'WorkloadExecutor' Web Service at Service Consumer nodes [10-50]
3. Stop any running application servers at all nodes [0-50]
  - a. Clear temporary directories
4. Stop POP at all nodes [0-50]

## 5.2 EFFICACY OF MECHANISM TO REDUCE AVERAGE INVOCATION RESPONSE-TIME

### 5.2.1 INTRODUCTION

The purpose of this experiment is to test whether provisioning an additional endpoint of a Web Service and registering it in the ActiveServiceDirectory is an effective means of reducing the average invocation response-time experienced by Service Consumers. Increasing or decreasing the level of provisioning of a Web Service is the primary means by which a Manager can control the availability of the service it is managing, so it is important to test whether this mechanism is actually effective in practice. This experiment aims to demonstrate the effectiveness of increasing the provisioning level of a service, and is followed by an experiment testing the effectiveness of reducing the provisioning level of a service (in order to reclaim resources and increase the average invocation response-time experienced by Service Consumers). These two experiments will be followed by an experiment demonstrating the effectiveness of the generic Manager in using these two mechanisms to autonomically maintain the provisioning level under various workload patterns. In the final experiment the number of invocation requests handled by endpoints during these workloads will be compared with the number of invocations performed on infrastructure-provided services (e.g. the ActiveServiceDirectory and Managers) in order to assess the scalability of the architecture.

### 5.2.2 HYPOTHESIS

In the presence of an actively-managed endpoint directory and a client-side endpoint-selection mechanism, increasing the number of available endpoints of a Web Service is an effective means of reducing the average response time for invocations of its operations. This mechanism is effective in systems populated by heterogeneous Web Service implementations and hosting technologies.

### 5.2.3 CRITERIA FOR SUCCESS

The hypothesis of this experiment will be validated if deploying an additional endpoint of a Web Service and listing it in the ActiveServiceDirectory is shown to reduce the average response-time for invocations of its operations. This result must be evident using three different implementations of the 'TargetExperimentEndpoint' Web Service deployed using three different Web Service container technologies.

#### 5.2.4 METHODOLOGY

An outline of the methodology used for this experiment is presented below. More specific details (such as exact workloads) are presented in the sections that follow.

In preparation for the experiment, three implementations of the 'TargetExperimentEndpoint' Web Service were developed, each for deployment in a different Web Service container. The three technologies chosen were Axis 1.4, JBoss 4.2.1, and RAFDA 2.0, and were selected based upon familiarity with the products. Next, a custom Manager Web Service for the 'TargetExperimentEndpoint' Web Service was developed which automatically deploys additional endpoints at regular intervals.

A single run of the experiment begins with the publication of one of the 'TargetExperimentEndpoint' Web Service implementations into the ServiceLibrary under the URI 'TargetExperimentEndpoint', followed by the publication of its custom Manager under the URI 'TargetExperimentEndpoint\_MANAGER'. Next, a significantly taxing workload is executed from a number of Service Consumer machines which invoke the operations of the 'TargetExperimentEndpoint' Web Service at an increasing rate. The custom Manager is automatically deployed by the infrastructure during the first invocation, after which it periodically deploys additional endpoints of the 'TargetExperimentEndpoint' Web Service and registers them in the ActiveServiceDirectory. Each Service Consumer records the response time of each invocation for the duration of the experiment and stores them using the 'ResultsRepository' Web Service when the experiment is complete. The experiment is run ten times using each of the three 'TargetExperimentEndpoint' implementations for a total of 30 executions. The results of the experiments are compared to the baseline behaviour of each implementation (as measured in the upcoming section 'Background Work'). The results of these comparisons are analyzed and the conclusions judged against the stated criteria for success.

#### 5.2.5 BACKGROUND WORK

In preparing for this experiment it was found that each of the target deployment containers exhibited vastly different response-times under identical workloads. Before a suitable workload could be selected for the experiments it was necessary to first pinpoint the number of invocations per-second each implementation (and thus deployment container) could handle before response times began growing non-linearly relative to the number of invocations.

For these preliminary tests a workload with 15 Service Consumer nodes is executed against a single endpoint of the TargetExperimentEndpoint Web Service. The workload begins at one call-

per-second and increases linearly to 30 calls per-second, per Service Consumer node, over a period of two minutes. Shown graphically in Fig. 89, this produces an effective workload of between 15 and 450 calls-per-second (CPS), which is shown to be suitable for identifying the maximum CPS for each container type. The workload is run 10 times for each container type, and the results averaged and graphed. It is expected that the graphs will show consistent response-times up until a ‘tipping point’ – when the number of calls per-second grows too high, causing response times to spike and remain high. Results are presented below for each container type followed by analysis.

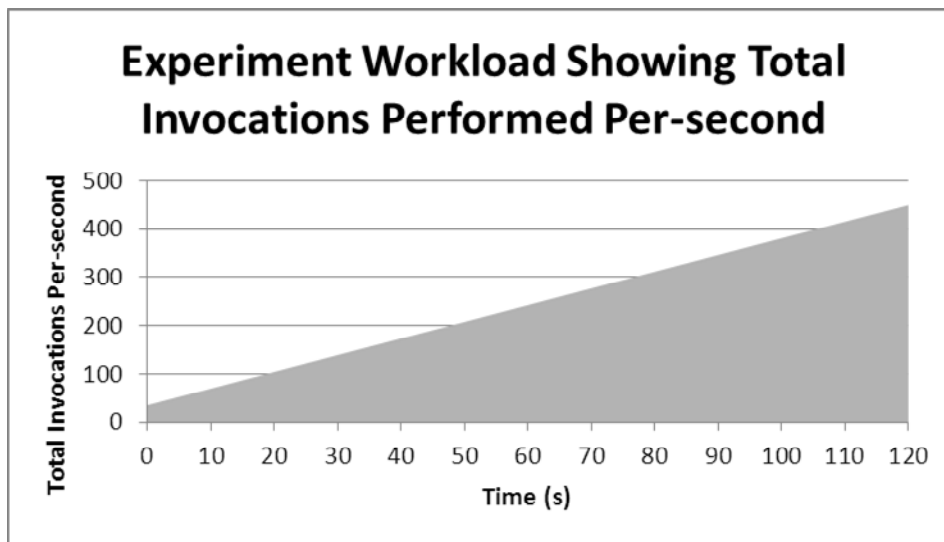
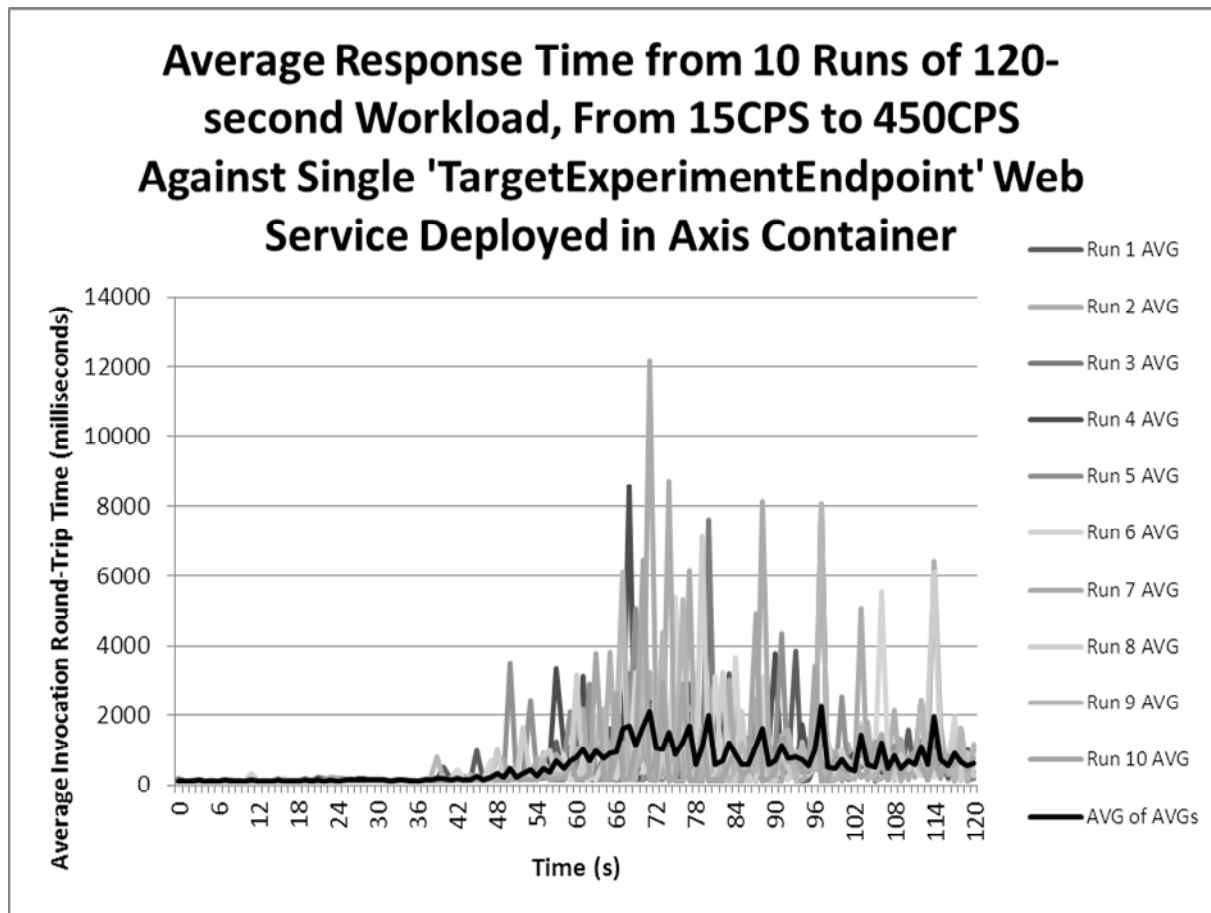


Fig. 89 Graph of the number of invocations performed per-second during the next experiment.

#### 5.2.5.1 AXIS RESULTS

Shown in Fig. 90 below, the ‘TargetExperimentEndpoint’ endpoint deployed in the Axis container performed consistently for the first forty seconds of the experiment until the workload reached roughly 150cps, at which point response times began to approach the one-second mark. A three-second spike after fifty seconds (195cps) was indicative of the upcoming delays – with the average staying above one second after sixty-five seconds (240cps). Many response-times became catastrophic after 255cps, with averages reaching above 12 seconds and some individual nodes experiencing 30- and 40-second response times. These results are shown in the graph in Fig. 90 below.



**Fig. 90 Graph of average response time from 10 runs of linear increasing workload, from 15CPS to 450CPS, against 'TargetExperimentEndpoint' Web Service endpoint deployed in the Axis Web Service container.**

Because the very high response-times in Fig. 90 obscure the pre-disaster growth rates we're trying to observe, a second graph is provided below in Fig. 91 which provides a closer look at response times during the first seventy seconds of the experiment. This second graph shows details of the early response-time growth and enables us to identify a 'tipping point' at roughly 220cps.

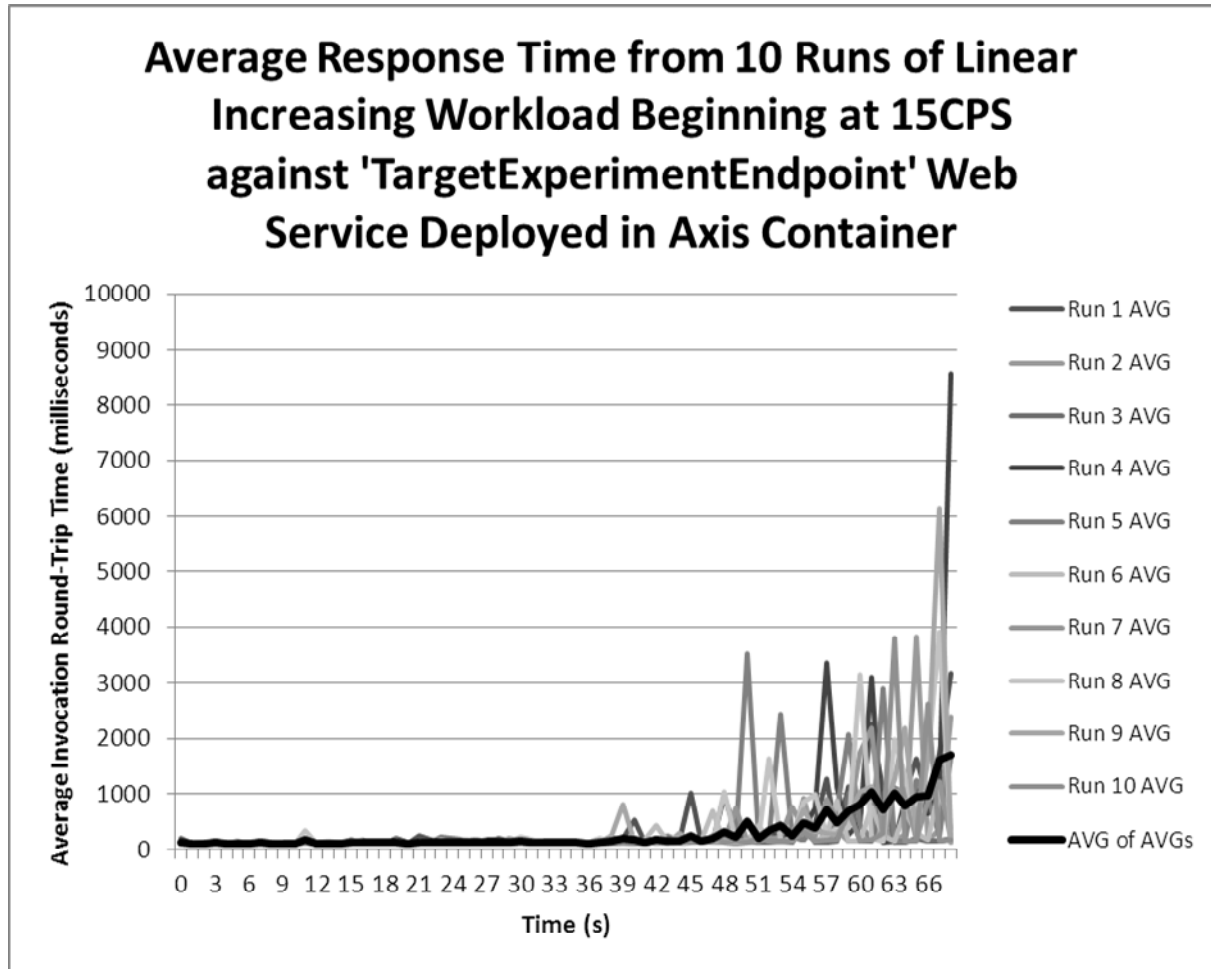


Fig. 91 Graph of the first seventy seconds of average response-times from 10 runs of linear increasing workload, from 15CPS to 450CPS, against 'TargetExperimentEndpoint' Web Service endpoint deployed in the Axis Web Service container.

#### 5.2.5.2 RAFDA RESULTS

The response times for the same workload carried out against the RAFDA 'TargetExperimentEndpoint' Web Service overloaded the endpoint much more swiftly than with Axis. Once again the response times remain fairly stable up until a portentous blip at the twenty-second mark (90cps). Response time doubles five seconds later at 105cps with steady, steep increases thereafter. The frenzied graph in Fig. 92 below ends after 50 seconds (200cps) because the RAFDA container consumes too much memory and crashes under loads in excess of 200cps (the RAFDA Web Service container has been analyzed using a Java application profiler and the problem is related to XML parsing and object marshalling). The highest call rate achieved with RAFDA was 240cps and the results contained many response-times of over a minute in duration. For the RAFDA container the 'tipping point' identified in the graph below occurs after twenty-five seconds, or roughly 105cps.

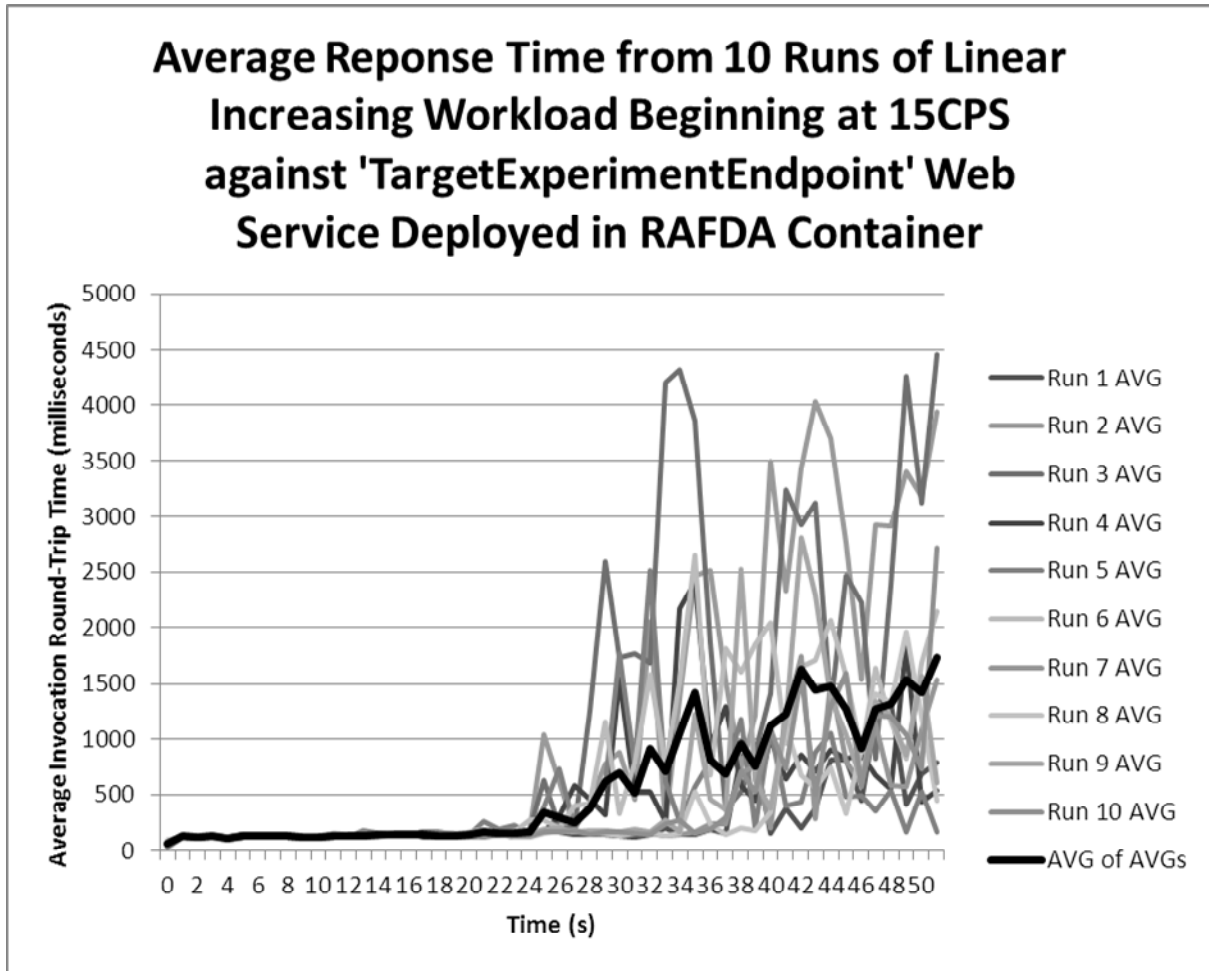
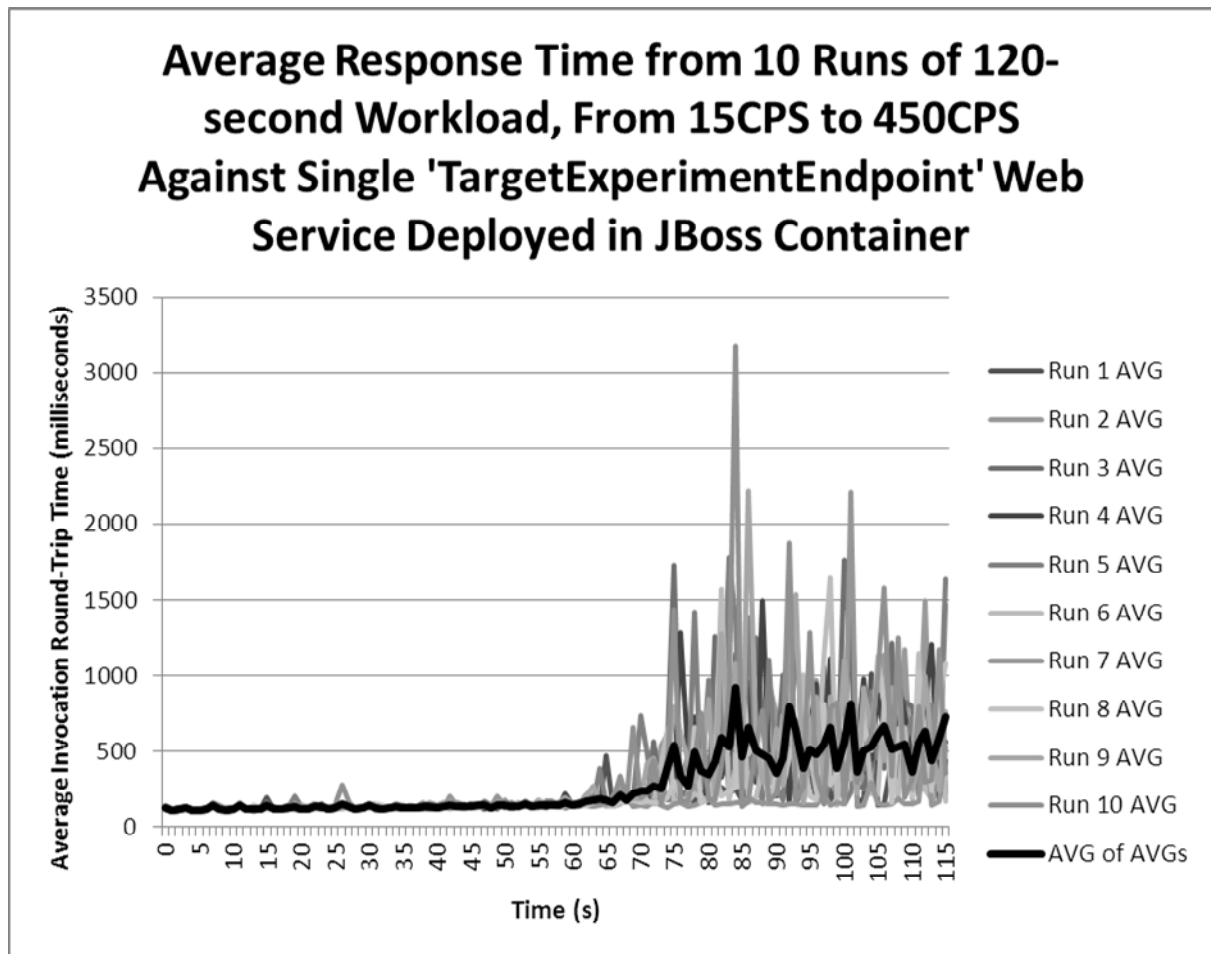


Fig. 92 Average Reponse Time from 10 Runs of Linear Increasing Workload Beginning at 15CPS against 'TargetExperimentEndpoint' Web Service Deployed in RAFDA Container.

#### 5.2.5.3 JBOSS RESULTS

The 'TargetExperimentEndpoint' Web Service endpoint exposed in the JBoss container proved to be the most consistent performer, and could also handle the greatest number of calls per-second. While RAFDA toppled at 105cps and Axis more than doubled its performance at 220cps, the JBoss container managed to consistently serve over 285cps before any of the individual response times hit the one-second mark. Indeed, overall *average* response time never once reaches above one-second – even at 450cps. While this is very good performance indeed – and importantly, there is no grand cataclysm as with the other two containers – there is still a point beyond which average response time accelerates rapidly, a 'tipping point' for the JBoss 'TargetExperimentEndpoint' implementation at around 275cps.



**Fig. 93 Graph of Average Response Time from 10 Runs of Linear Increasing Workload Beginning at 15CPS  
Against 'TargetExperimentEndpoint' Web Service Endpoint Deployed in JBoss Container.**

#### 5.2.5.4 CONCLUSIONS

The JBoss container was not only able to handle the heaviest workload, but also exhibited the most consistent behaviour. The Axis container was also a consistent performer but was unable to handle the same workloads as the JBoss container. Both of these containers supported loads well over twice that of RAFDA, although this is not surprising as RAFDA is a research platform and not intended as a commercial-grade deployment technology. As the maximum load varies depending on the container used, the workloads used for each of the upcoming experiments will vary depending on the container. The list below shows the 'tipping points' derived from the previous three experiments:

- Max CPS for Single TargetExperimentEndpoint:
  - Axis: 220CPS
  - RAFDA: 105CPS
  - JBoss: 275CPS

#### 5.2.5.5 COST OF POP

The overhead cost of proxying Service Consumer invocation requests through the local point of presence used in this evaluation adds an additional 96ms per-request. This is very expensive and the primary cost-driver in the POP implementation is XML parsing which consumes 98% of the overall non-network-related time. Because of the complexity involved in the rest of the system, the POP was implemented very conservatively so as to be able to rule it out as a source of error when implementing the rest of the system. The implementation uses the full Xerces DOM parser [98] and full schema validation. In hindsight a stream-parser, or even basic string manipulation would have been sufficient as only individual document element attributes are modified and there is no manipulation of the document tree. Network-related costs contribute an average of 9ms for each ActiveServiceDirectory lookup, although this cost is amortized over the total number of requests received for the Web Service during the ‘lease duration’ period. A comprehensive evaluation of the cost contribution of this intra-architecture communication is conducted in Chapter 5 Section 5.5 ‘Cost & Scalability of Infrastructure Services’ and alternative POP implementation approaches are discussed in the ‘Future Work’ section of Chapter 6.

#### 5.2.6 EXPERIMENTS

Having established the maximum CPS value that each container can handle, experiment workloads can be crafted that approach each container’s individual peak CPS before deploying an additional endpoint. The goal is to demonstrate that, while the CPS value continues to rise, the presence of an additional endpoint of the Web Service successfully controls the rise of average invocation response-time experienced in the previous single-endpoint tests. Since we know that rapid rises in response time normally occur at the ‘tipping points’ identified above, we can judge the experiments successful if the presence of an additional endpoint successfully averts this out-of-control growth. Graphs will indicate where an additional endpoint was deployed by using a ☆ icon.

#### 5.2.7 CUSTOM MANAGER BEHAVIOUR

A custom Manager is used in this experiment in order to deploy endpoints at a consistent, repeatable rate. The custom Manager is written to deploy additional endpoints of the TargetExperimentEndpoint Web Service at 30-second intervals, up to a maximum of 5 deployed endpoints, beginning from the first invocation request. This implementation is published in the ServiceLibrary under the URI ‘/rfmthesis/experiments/TargetExperimentEndpoint\_MANAGER’.

#### 5.2.7.1 EXPERIMENT 1.1: 'TARGETEXPERIMENTENDPOINT' IN AXIS

In the previous tests, the peak load capacity of a single TargetExperimentEndpoint Web Service deployed in the Axis container exhibited a 'tipping point' at 220cps. In order to observe the effect on response-time of provisioning an additional endpoint of a Web Service, the workload for this experiment is designed to hit 220cps after only 25 seconds. Because the custom Manager published for the TargetExperimentEndpoint only deploys an additional endpoint every 30 seconds, the results are expected to show periods of response-time growth before an endpoint is deployed, followed by a slowing or reversal of growth.

In order to test this behaviour, the workload for this experiment will increase linearly at a constant rate of 7.4CPS per-second for 150 seconds, beginning at 35CPS from 35 Service Consumers (1CPS from each Service Consumer) and ending at 1120CPS (32CPS from each Service Consumer). This workload is shown graphically below in Fig. 94.

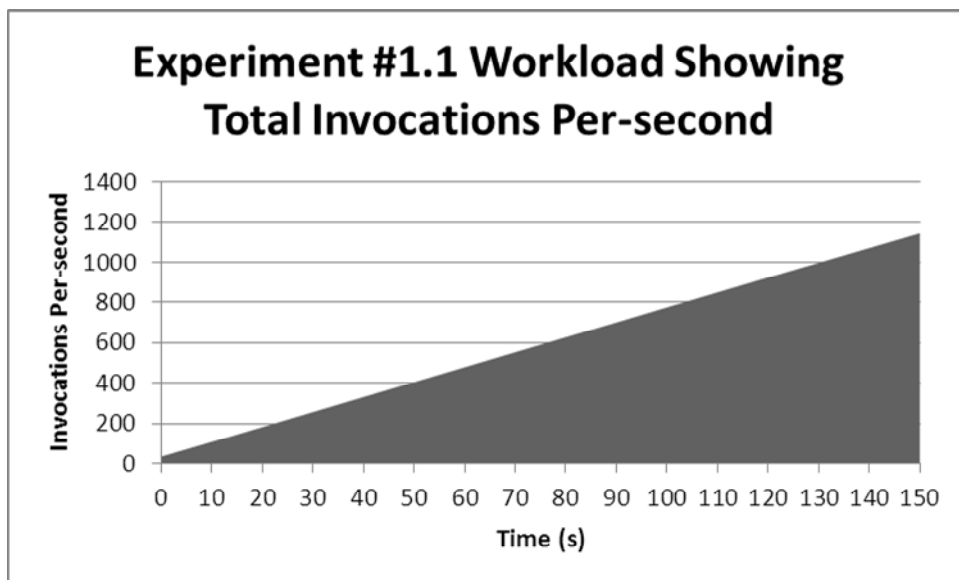
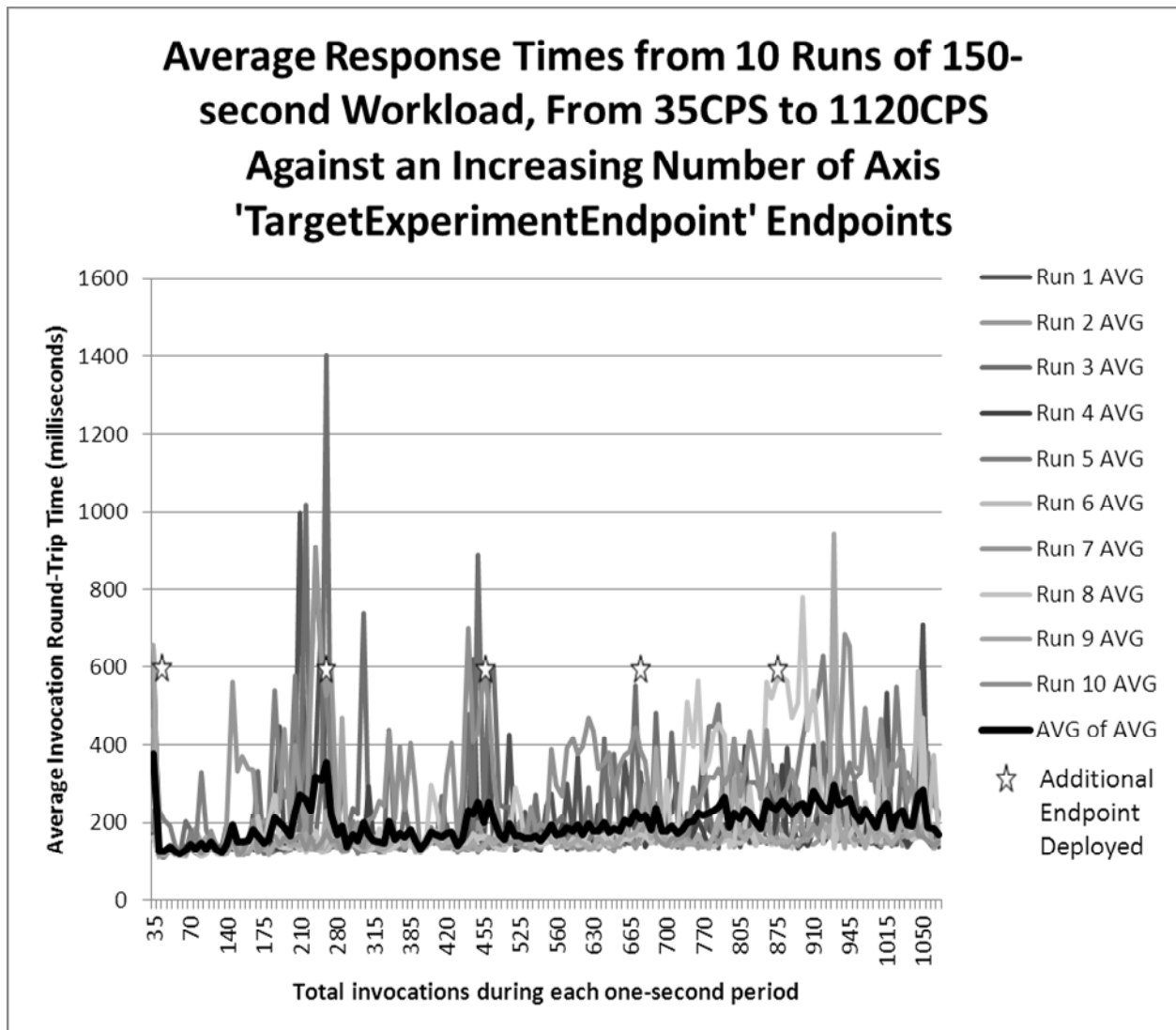


Fig. 94 Graph of the workload executed in experiment 1.1 using 35 Service Consumer machines. This graph shows the number of invocations performed per-second on TargetExperimentEndpoint Web Service endpoints deployed in the Axis container.

## 5.2.7.1.1 Results

The results of this experiment are shown in Fig. 95 below, with stars indicating where additional endpoints were deployed and registered in the ActiveServiceDirectory.



**Fig. 95** The results of Experiment 1.1, detailing the average invocation response time experienced by Service Consumers over a 150-second experiment with a steadily increasing workload and endpoint provisioning level.

It can be clearly seen in Fig. 95 that response-time begins to grow steeply at 200CPS, until an additional endpoint is provisioned, at which point the growth is reversed and response-time returns to a stable level, even as the load continues to rise. This behaviour is in stark contrast to the single-endpoint test shown previously, where response-time continued to grow out of control. This effect is repeated at 440CPS, 660CPS, and 880CPS, where the number of invocations received per endpoint is 220CPS. These results strongly suggest that deploying an additional endpoint of a Web Service and listing it in the ActiveServiceDirectory is an effective means of

reducing the average invocation response time experienced by Service Consumers. The following two experiments will repeat this experiment using different deployment containers and slightly different maximum workloads.

#### 5.2.7.2 EXPERIMENT 1.2: 'TARGETEXPERIMENTENDPOINT' IN RAFDA

A single endpoint exposed in RAFDA was shown previously to be capable of handling around 110CPS. The workload for this experiment will thus mirror the one run against the Axis endpoint in the previous experiment, but with a 'tipping point' of 110CPS reached after 25 seconds. The workload will start at 1CPS from 35 machines (35CPS total) and increase at a rate of 4.2CPS per-second for 150 seconds, reaching a total of 665CPS, as shown in Fig. 96 below. The custom TargetExperimentEndpoint Manager will again be published which deploys an additional endpoint of the TargetExperimentEndpoint Web Service every 30 seconds following the first invocation.

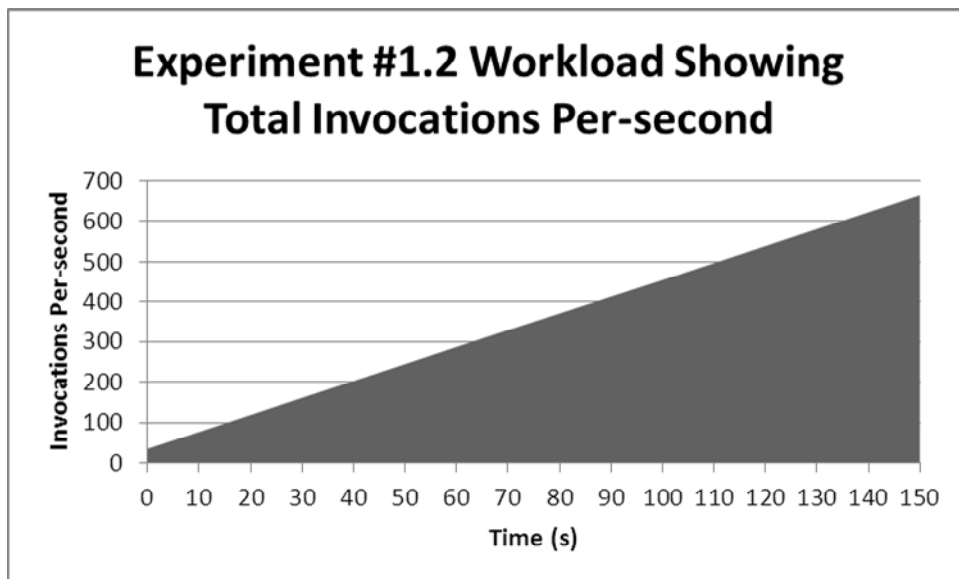
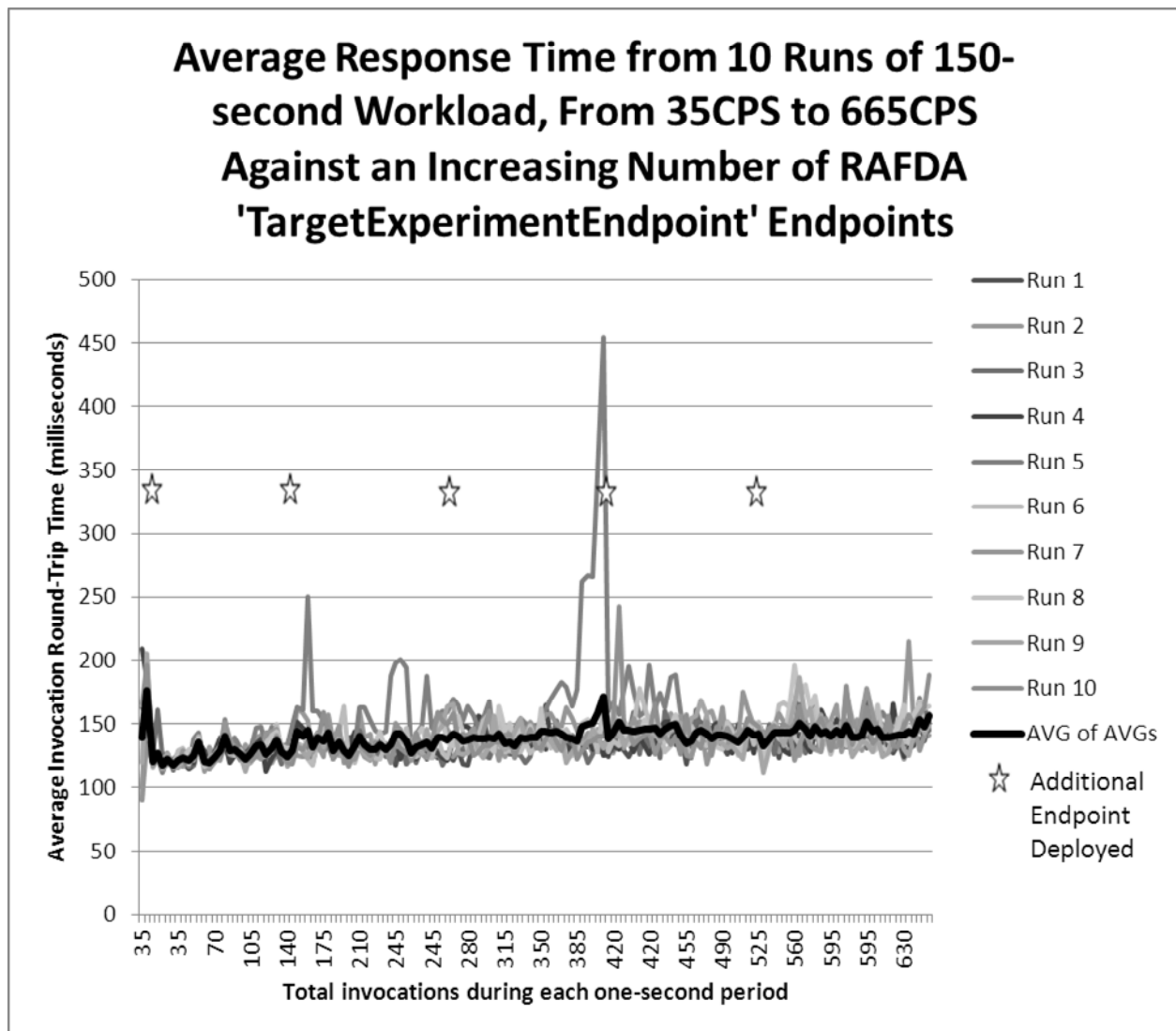


Fig. 96 Graph of the workload executed in Experiment 1.2 using 35 Service Consumer machines. This graph shows the number of invocations performed per-second on TargetExperimentEndpoint Web Service endpoints deployed in the RAFDA container.

##### 5.2.7.2.1 Results

The graph in Fig. 97 below shows the results of running the described workload against an increasing number of TargetExperimentEndpoint endpoints deployed using the RAFDA Web Service container and listed in the ActiveServiceDirectory. When compared with the results of the previous RAFDA experiment in which a single endpoint was tested and response-time grew significantly after just 110CPS, the results below demonstrate only a slight overall increase in response-time over the duration of the experiment, with none of the rapid growth experienced previously. These results closely mirror the results of experiment 1.1 and support the notion that

providing an additional endpoint of a Web Service and listing it in the ActiveServiceDirectory is an effective means of reducing the average invocation response-time experienced by Service Consumers. The third and final test will be run using the JBoss container before any conclusions are drawn.

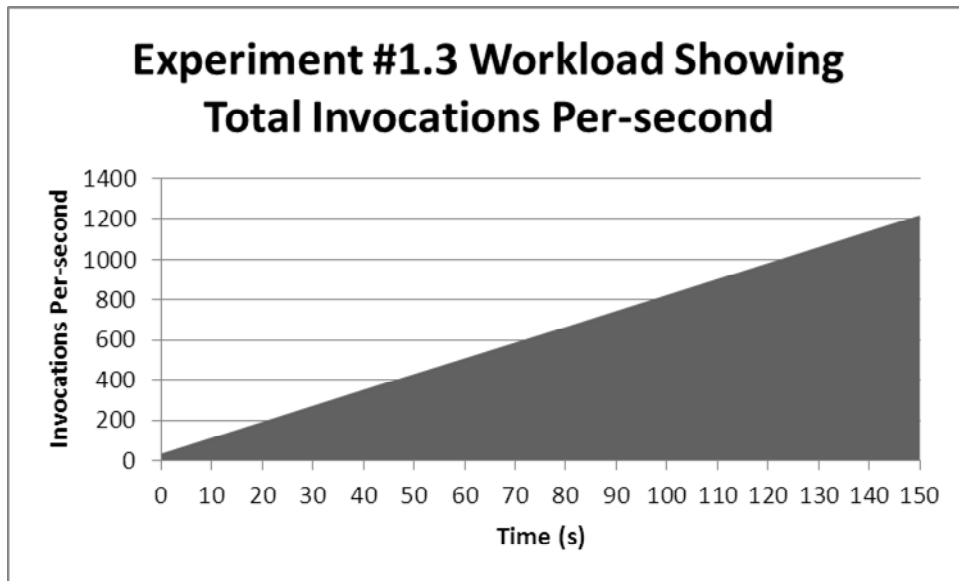


**Fig. 97** The results of Experiment 1.2, detailing the average invocation response time experienced by Service Consumers over a 150-second experiment with a steadily increasing workload and endpoint provisioning level.

#### 5.2.7.3 EXPERIMENT 1.3: 'TARGETEXPERIMENTENDPOINT' IN JBOSS

The JBoss Web Service deployment container will be used in this last of three experiments testing whether providing an additional endpoint of a Web Service and listing it in the ActiveServiceDirectory is an effective mechanism for reducing the average invocation response-time experienced by Service Consumers. A single TargetExperimentEndpoint deployed in the

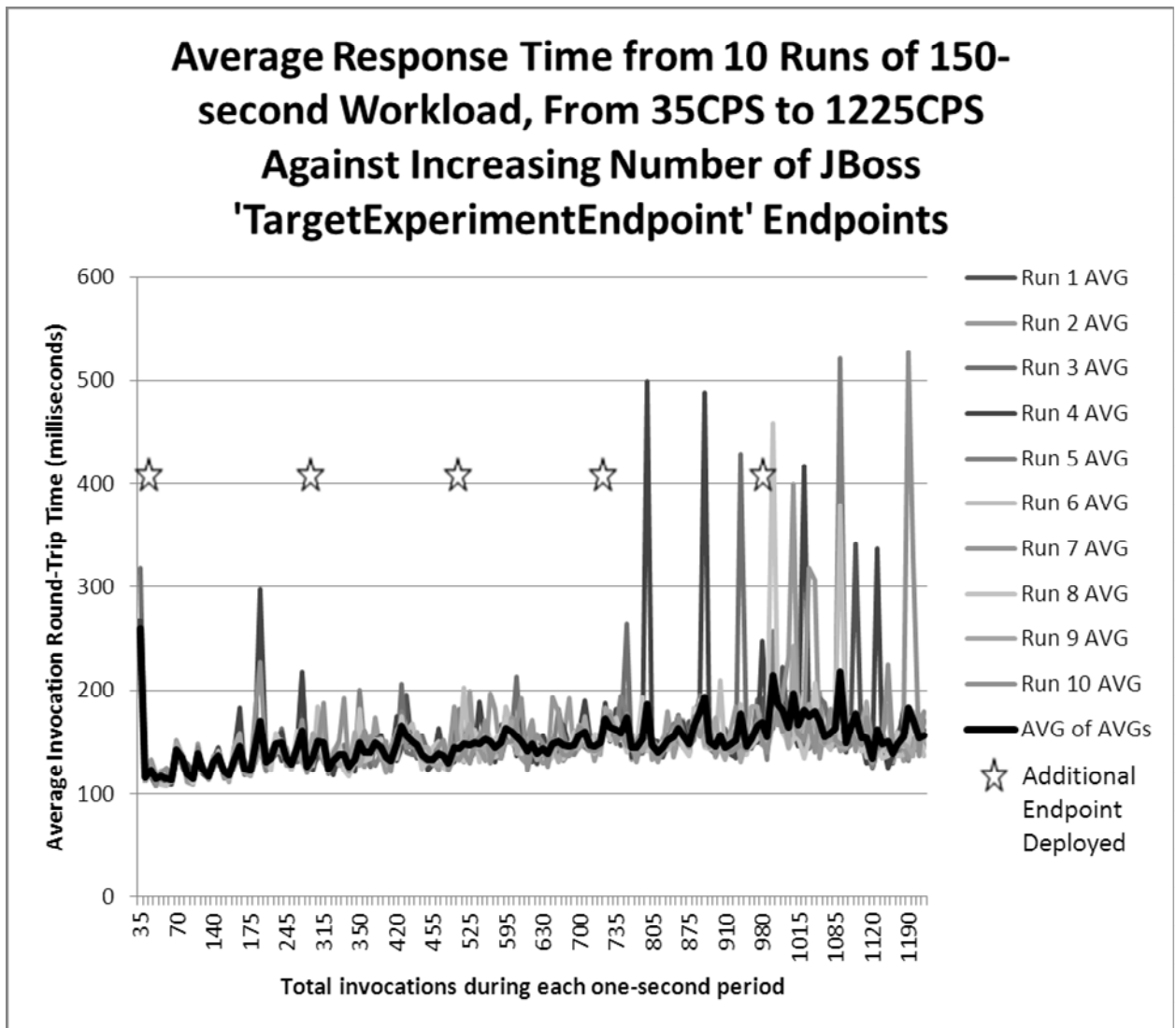
JBoss container was shown previously to be capable of handling around 275CPS before response-times began to grow rapidly. As in the previous two experiments, the workload in this experiment is designed to hit this ‘tipping point’ after 25 seconds. The workload will begin at 1CPS from each machine and increase at a rate of 7.9CPS per-second for 150 seconds, reaching a total of 1225CPS, as shown graphically in Fig. 98 below. The same custom Manager will be published which automatically deploys an additional endpoint of the TargetExperimentEndpoint Web Service and lists it in the ActiveServiceDirectory every 30 seconds, following the first invocation.



**Fig. 98 Graph of the workload executed in Experiment 1.3 using 35 Service Consumer machines. This graph shows the number of invocations performed per-second on TargetExperimentEndpoint Web Service endpoints deployed in the JBoss container.**

#### 5.2.7.3.1 Results

The graph in Fig. 99 below shows the results of running the described workload against an increasing number of TargetExperimentEndpoint endpoints deployed in JBoss container and listed in the ActiveServiceDirectory. This increased level of provisioning is shown to contain the rapid growth in response-time experienced in the single-endpoint experiment after 275CPS; response times in the current experiment remain below 200ms even at 1225CPS. This result is identical to the past two experiments in which the response times are controlled to well below their post-‘tipping point’ levels, while supporting levels of demand many times greater than possible with a single endpoint. The results of all three experiments are discussed in the section immediately following Fig. 99.



**Fig. 99** The results of Experiment 1.3, detailing the average invocation response time experienced by Service Consumers over a 150-second experiment with a steadily increasing workload and endpoint provisioning level.

### 5.2.8 ANALYSIS

During the tests in 'background work', each of the three demonstrated deployment technologies was used to host a single endpoint of the TargetExperimentEndpoint Web Service and each endpoint was subjected to an increasing workload from a number of Service Consumers. The response-times measured by Service Consumers were used to establish a baseline level of performance for each technology and to identify its 'tipping-point' – an invocation rate at which response-times rose sharply and continued to grow.

In the next set of experiments a custom Manager was published which deployed an additional endpoint of the TargetExperimentEndpoint Web Service and listed it in the ActiveServiceDirectory every 30 seconds. Having established the maximum invocation rate that

each container could handle, a second set of experiment workloads were executed which reached each container's individual 'tipping-point' just before each additional endpoint was deployed. By comparing the invocation-response times measured by Service Consumers executing the first workload against the results of executing a significantly greater workload with a steadily increasing number of endpoints, we were able to show that the presence of an additional endpoint of the Web Service successfully controls the rising response-times observed in the single-endpoint tests. In all three cases the response-times were controlled to the stable levels experienced before the 'tipping-points', even while servicing Service Consumer demand many times greater than was possible with a single endpoint. These experiments have shown conclusively that deploying an additional endpoint of a Web Service and listing it in the ActiveServiceDirectory is an effective means of reducing the average invocation response-time experienced by service consumers, and that the mechanism is effective independently from the technology used to host the endpoint (which only has a bearing on the maximum sustainable load).

Another equally important result is that by binding to and invoking operations via the local point of presence, the Service Consumer application did not need to be modified once during testing, nor did the POP, even though the endpoints were deployed on multiple different machines and using multiple different technologies. That this mechanism was successful in transparently distributing load amongst all of the currently available endpoints of the desired service is an important success that will be tested again in the next experiment when the number of available endpoints is gradually reduced.

## 5.3 EFFICACY OF MECHANISMS TO INCREASE AVERAGE INVOCATION RESPONSE-TIME AND RE-DISTRIBUTE LOAD

### 5.3.1 INTRODUCTION

Having shown that deploying an additional endpoint of a Web Service and listing it in the ActiveServiceDirectory is an effective means of reducing the average invocation response-time experienced by Service Consumers, this experiment will test whether removing an endpoint is an effective means of increasing the average invocation response-time, and whether Service Consumers' local POPs can effectively re-direct load to the remaining endpoints. As Managers may reclaim hosting resources from over-provisioned (or under-demanded) services by undeploying an endpoint of the service and removing it from the ActiveServiceDirectory, it is important to test whether the POP can effectively re-distribute load once an endpoint is removed – this will be shown as an increase in the average invocation response time experienced by Service Consumers, similar to the drop in response time experienced when an additional endpoint was provided in the previous experiment.

### 5.3.2 HYPOTHESIS

Removing an endpoint entry from the ActiveServiceDirectory causes all Service Consumer's local points of presence to re-direct invocation requests to one of the remaining endpoints. Within a sufficiently large population of Service Consumers, selecting an alternative endpoint is an effective means of re-distributing load amongst all remaining registered endpoints of a Web Service.

### 5.3.3 CRITERIA FOR SUCCESS

The average invocation response-time experienced by a Service Consumer invoking operations of a Web Service can be shown to measurably increase as the level of provisioning of that Web Service is reduced.

### 5.3.4 METHODOLOGY

The preparation for this experiment included the creation of another custom Manager for the 'TargetExperimentEndpoint' Web Service. This Manager initially deploys a number of endpoints of the TargetExperimentEndpoint Web Service and registers them in the ActiveServiceDirectory, then removes a single entry at regular intervals until there is only one endpoint remaining. This implementation is published in the ServiceLibrary under the URI 'TargetExperimentEndpoint\_MANAGER'. A number of ServiceHosts capable of hosting both the

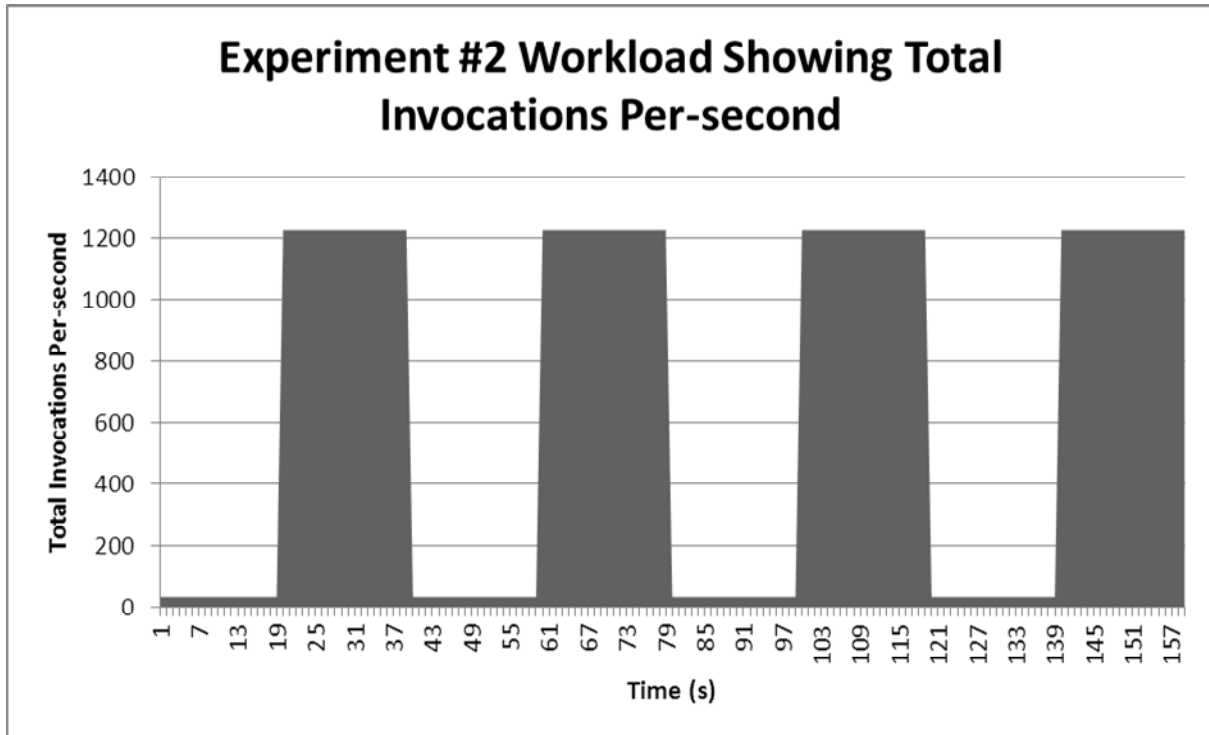
TargetExperimentEndpoint implementation and its custom Manager were also started up and their HostDescriptors registered in the HostDirectory.

Specified in detail in the upcoming section, this experiment entails the execution of a four-period square-wave workload from a number of Service Consumer machines against a steadily decreasing number of TargetExperimentEndpoint Web Service endpoints. A square-wave workload was chosen because the low cps periods can be handled by even a single endpoint, whereas the high cps periods will show the difference between having various numbers of endpoints under identical loads. The Manager is designed to remove endpoints during the low-cps periods, so subsequent high-cps periods will be handled by progressively fewer and fewer endpoints.

The custom manager will be automatically deployed during the first invocation and will automatically deploy a number of endpoints of the 'TargetExperimentEndpoint' Web Service and register them in the ActiveServiceDirectory. The Manager will thereafter remove one entry from the list of available endpoints at a regular interval for the duration of the experiment. Service Consumers record the response time of each invocation for the duration of the experiment and store the results in the ResultsRepository.

#### *5.3.4.1 WORKLOAD AND CUSTOM MANAGER DETAILS*

A graph of the workload for this experiment is shown below in Fig. 100. It is a 4-period, square-wave workload with a duration of 160 seconds, executed from 35 Service Consumer machines, with peak loads of 1225CPS and troughs of 35CPS. The custom Manager will initially deploy five endpoints of the Axis TargetExperimentEndpoint implementation and register them in the ActiveServiceDirectory. It will thereafter begin to undeploy and unregister a single endpoint every forty-five seconds, so that each peak of the workload is handled by progressively fewer endpoints, beginning at five and ending with two. The experiment stops at two because, as shown previously, the results of running a 1225CPS workload against a single endpoint produce results that, when graphed, completely obscure the changes in response-time experienced between the higher levels of provisioning.

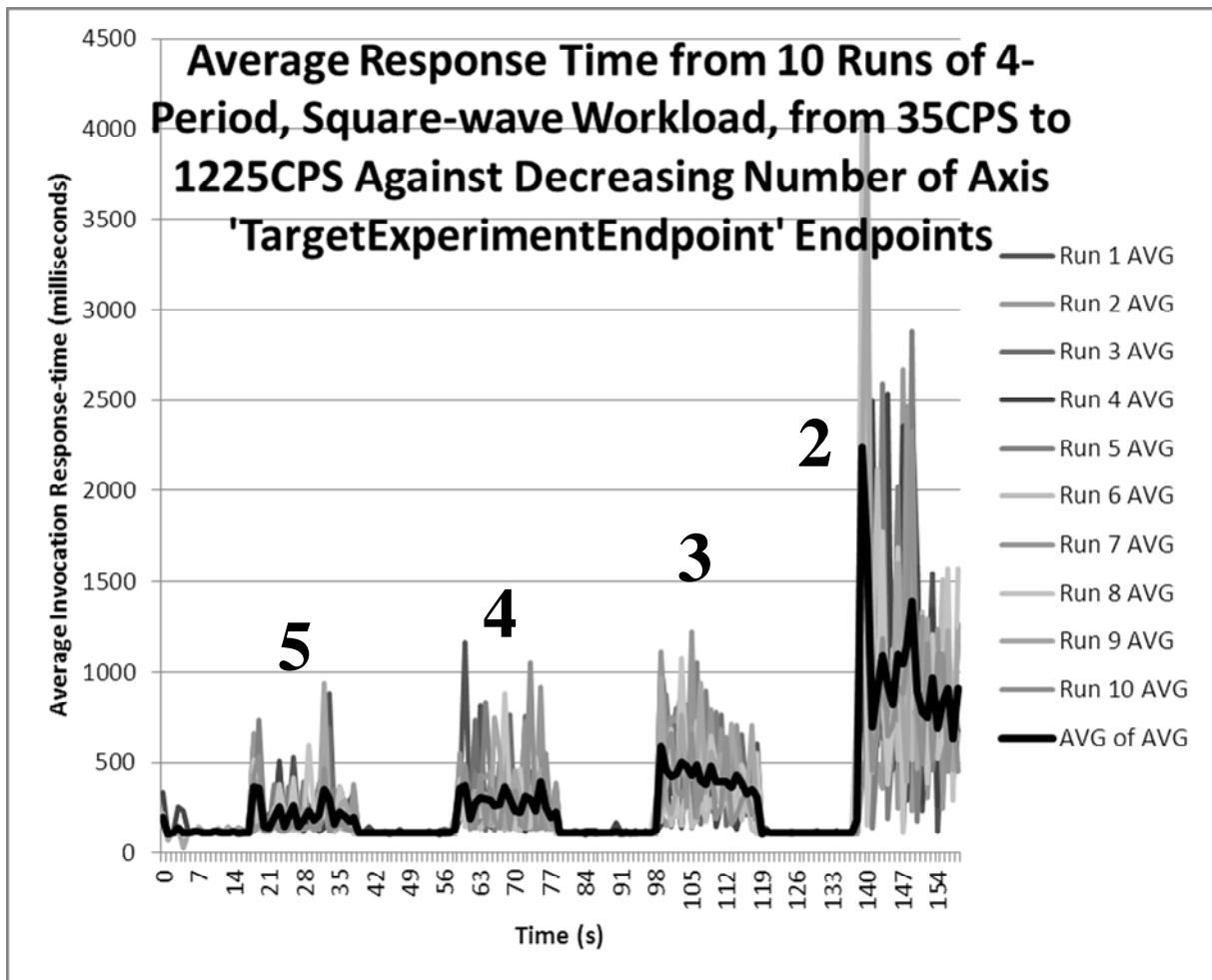


**Fig. 100 Graph of the workload executed in Experiment 2 using 35 Service Consumer machines. This graph shows the number of invocations performed per-second on TargetExperimentEndpoint Web Service endpoints deployed in the Axis container.**

We have shown in the previous experiment that the infrastructure is capable of deploying Web Service endpoints developed for various hosting technologies, and that the selection only has a bearing on the endpoint's maximum sustainable load. In order to introduce some consistency for later comparison, the remainder of the experiments will be performed using a single deployment technology. Because of its consistent performance and predictable 'tipping-point' the Axis container will be used for the remainder of the experiments.

### 5.3.5 RESULTS & ANALYSIS

The results of performing the described experiment ten times are displayed below in Fig. 101, together with an average of all ten executions represented by the solid black line. The y-axis indicates the average invocation response-time as measured by Service Consumers, the x-axis is time (in seconds), and large numbers indicate the number of endpoints available during that peak period of the workload. As the experiment proceeds the number of deployed endpoints gradually decreases from five to four, to three, and finally two deployed endpoints. A clear increase in the average invocation response-time can be observed in response to this decline. This increase is attributed to the average number of calls per-endpoint increasing as the number of endpoints decreases.



**Fig. 101 Results of experiment 2, detailing the average invocation response-time experienced by Service Consumers over the 160-second experiment. The large numbers indicate the number of TargetExperimentEndpoint endpoints available during each high-cps period**

It is interesting to observe the consistency with which the Axis endpoints handled a comparatively large workload with a small number of endpoints, notably with three and two. While a number of four-second invocations can be observed during the two-endpoint period, the average response time actually remains close to 1-second – far from ideal, but less than expected. While an expanded tcp window frame may be a contributing factor, this behaviour is more likely the result of the Java VM ‘warming up’ and expanding the size of the VM heap. This behaviour will have a bearing on the applicability of response-time as a suitable measure of endpoint-performance, versus a metric based upon other aspects of demand, such as the numbers of invocations per-second. These decisions are explored in the next experiment when the performance of two Managers is compared under identical workloads.

### 5.3.6 EXPERIMENT 2 CONCLUSIONS

By servicing identical periods of high consumer demand with progressively fewer endpoints of the requested Web Service, this experiment has allowed us to observe that fewer deployed endpoints of a Web Service leads to an increase in the average invocation response time experienced by Service Consumers. Further, we have demonstrated that the mechanisms of the Service Consumer's local POP are capable of redirecting invocation requests to alternative endpoints of a Web Service in order to redistribute load.

Together with the results of Experiment 1, we can conclude that changing the provisioning level of a Web Service is an effective tool for managing the average invocation response time experienced by Service Consumers. We can also conclude that Service Consumer points of presence, together with the infrastructure-provided services, are capable of automatically and transparently distributing invocation requests amongst a dynamic number of Web Service endpoints without requiring any action on the part of the Service Consumer or Service Provider. The next experiment tests whether the architecture enables Managers to dynamically manage the availability of a Web Service under various workloads by analyzing the usage data returned from ServiceHosts and deploying or undeploying endpoints as necessary to meet demand.

## 5.4 SCALABILITY AND DYNAMIC ADAPTATION TO CHANGES IN CONSUMER DEMAND

### 5.4.1 INTRODUCTION

Having shown that changing the provisioning level of a Web Service changes the average invocation response-time experienced by Service Consumers, it can now be tested whether a Manager capable of deploying and undeploying endpoints can make provisioning decisions based upon usage data provided by Service Hosts. For this experiment, three Managers will be demonstrated which make provisioning decisions based upon two distinct metrics: response-time as measured at the ServiceHost, and the load experienced by ServiceHosts as measured by the total number of invocation requests received for a Web Service during each reporting period. The generic Manager described in Chapter 4 (which evaluates based on response-time) will be tested and compared to two custom managers which implement two separate provisioning policies based upon the quantity of invocation requests.

For this experiment the Service Consumers will continue to execute workloads which invoke operations of the TargetExperimentEndpoint Web Service. The behaviour of each Manager will be evaluated as they respond to two distinct workload patterns: linearly increasing demand, as seen previously, as well as demand that fluctuates in a sin-wave pattern. Response times will be measured by Service Consumers and graphed, as in the previous experiments, with icons on the graphs indicating where an endpoint was either deployed or undeployed by a Manager.

### 5.4.2 HYPOTHESIS

Managers are capable of identifying and dynamically responding to changes in the level of Service Consumer demand by increasing or decreasing the number of available endpoints of a Web Service.

### 5.4.3 CRITERIA FOR SUCCESS

The experiment will validate the hypothesis if each Manager implementation is shown to respond to measured changes in demand by altering the provisioning level of the Web Service being managed. In the presented architecture, ServiceHosts periodically report invocation-rate and response-time usage data to the Manager of each Web Service deployed within their domain. In experiment 1, increases in demand are shown to result in increases in the time taken to service each request by a deployed endpoint. The generic Manager described previously is expected to be able to analyze the reported usage data and identify the rise in response-time, prompting a

deployment which results in a drop in the response time. Similarly, reductions in demand are expected to result in steadily low response-times, prompting the Manager to undeploy an endpoint of the service which, depending on the load, may or may not result in a significant increase in the average invocation response time experienced by Service Consumers. For Managers making deployment decisions using a count of the invocation requests received during each reporting period, the provisioning decisions are expected to reflect a direct response to the reported workload being executed.

#### 5.4.4 METHODOLOGY

The generic Manager implementation described in Chapter 4 is published into the ServiceLibrary with the URI 'Manager'. This Manager periodically analyzes usage data reported by ServiceHosts and, based upon the measured demand, decides if the provisioning level needs to be changed. In the described generic Manager implementation, an additional endpoint is deployed if the average measured response time doubles over the past three reporting periods, and an endpoint is undeployed if the measured average response time drops by half or remains flat for three periods.

A number of Service Consumer machines will invoke operations of the 'TargetExperimentEndpoint' Web Service in a specified workload pattern. The response-time of each invocation will be measured by Service Consumers for the duration of the experiment and the results stored in the ResultsRepository. The experiment will be performed first with a linear increasing workload and then with a sin-wave workload. These workloads are repeated for each of the additional Manager implementations described in the upcoming sections. Service Consumers record the response time of each invocation for the duration of the experiment and store the results in the ResultsRepository.

#### 5.4.5 EXPERIMENTS AND RESULTS

##### 5.4.5.1 *MANAGING BY RESPONSE-TIME*

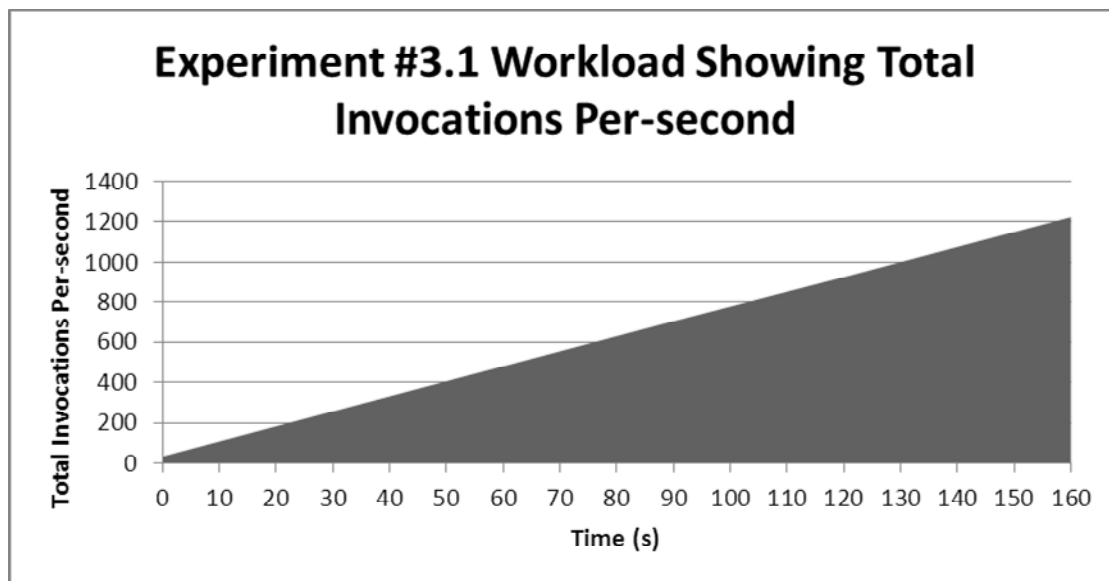
###### 5.4.5.1.1 Experiment 3.1 Workload, Manager and Configuration Details

It is the task of the Manager to analyze and make decisions on the necessary level of provisioning based upon the usage data reports returned from ServiceHosts currently hosting an endpoint of the TargetExperimentEndpoint Web Service. In the current experiment the generic Manager is configured to analyze these reports every five seconds. In order to have usage reports ready to evaluate during each period, the Manager must somehow communicate to ServiceHosts that they should also report their usage data with the same frequency (i.e. every five seconds). Every time a ServiceHost invokes a Manager's 'reportUsageData' operation they are returned a

value indicating the length of time, in milliseconds, to wait before next reporting; during this experiment the Manager will constantly return the value 5000.

As before, the POPs at all Service Consumer nodes are initialized with a parameter specifying an endpoint lease duration of three seconds. This value dictates the frequency with which queries are sent to the ActiveServiceDirectory during the experiments, and thus the length of time before a newly deployed endpoint will begin to be used by Service Consumers. We will see later how this behaviour influences the timing and content of the endpoint usage reports coming from ServiceHosts and how this in turn affects the information that is available to a Manager during the reporting period immediately following a provisioning event. This topic will be discussed at length later once we have more experimental results to reference.

A graph of the first workload executed in this experiment is shown in Fig. 102 below. It is a linear increasing workload, executed for 160 seconds from 35 Service Consumer machines, beginning at 35CPS and ending at 1225CPS.



**Fig. 102 Experiment #3.1 linear increasing workload to be executed from 35 Service Consumer machines by invoking operations of the TargetExperimentEndpoint Web Service at the specified rate.**

#### 5.4.5.1.2 Experiment 3.1 Results

The results of performing the described experiment using the generic Manager are presented in Fig. 103 on page 164. The average invocation response-time experienced by Service Consumers is represented by the solid blue line and is presented against a light blue background representing the workload being executed as the experiment progresses. Note that this workload is plotted on the secondary axis to the right. Based upon the usage data returned from

ServiceHosts, the Manager made seven provisioning decisions during this experiment, as indicated by the vertical black lines. Deployment events are represented by a red square icon while undeployment events are represented with a black 'X'. The number above each of these icons indicates the number of endpoints available in the ActiveServiceDirectory after the provisioning event.

All of the provisioning events shown in Fig. 103 are executed in response to fluctuations in the invocation response-time as measured at the ServiceHosts and reported to the Manager. While the first three deployments (marked '1', '2', and '3') do not appear to be preceded by a significant rise in the average invocation response-time experienced by Service Consumers, the measured changes in response time at the ServiceHosts are significant enough to prompt deployment. During periods of low call volume this can happen because a small number of longer invocations have a greater impact on the calculated average response time when there are fewer calls to average.

After the third deployment event at the 19-second mark, Service Consumers experience an extended period of relatively stable response time. When the average response time measured and reported by the ServiceHosts remains stable for three complete periods the Manager undeploys first one endpoint at the 55-second mark followed by another at the 60-second mark. These two undeployments have the effect of shifting all Service Consumer demand to the single remaining endpoint, increasing the measured response time and prompting the Manager to correct its previous actions and deploy an additional endpoint in response.

Two deployed endpoints support all of the Service Consumer demand between the 70- and 100-second mark before the Service Consumers begin to experience increased response times. While the average invocation response times do grow, they do not do so as sharply as they have been shown to do in previous experiments when confronted with a high call volume shortly after being deployed. This is further evidence of the 'warming up' effect identified previously in experiment 3.2, where two long-deployed endpoints support nearly 400CPS each before response times are shown to increase significantly. Even so, response times do not grow steeply enough to prompt a deployment for another 30 seconds, as the invocation rate approaches 500CPS per endpoint. Similar to the first 20-second of the experiment where a small number of long calls influenced the measured average response-time at the ServiceHost, a small number of particularly long calls that adversely affect individual Service Consumers during periods of high call volume do not necessarily impact the average response time measured by ServiceHosts

enough to prompt a deployment (recall that the average response time at the ServiceHosts must double over the previous three periods for a deployment to be triggered).

As the invocation request rate approaches 1225 invocations per-second two additional endpoint are deployed at the 135- and 140-second mark in response to the measured increase in demand at the ServiceHosts. As expected, these deployments are shown to halt the growth and measurably reduce the average invocation response time experienced by Service Consumers. The experiment concludes with the four deployed endpoints handling the remainder of the demand.

### Average Invocation Response-time Experienced by Service Consumers Invoking Operations of Managed TargetExperimentEndpoint Web Service in a Linearly Increasing 35-1225CPS Workload, Indicating Provisioning Events

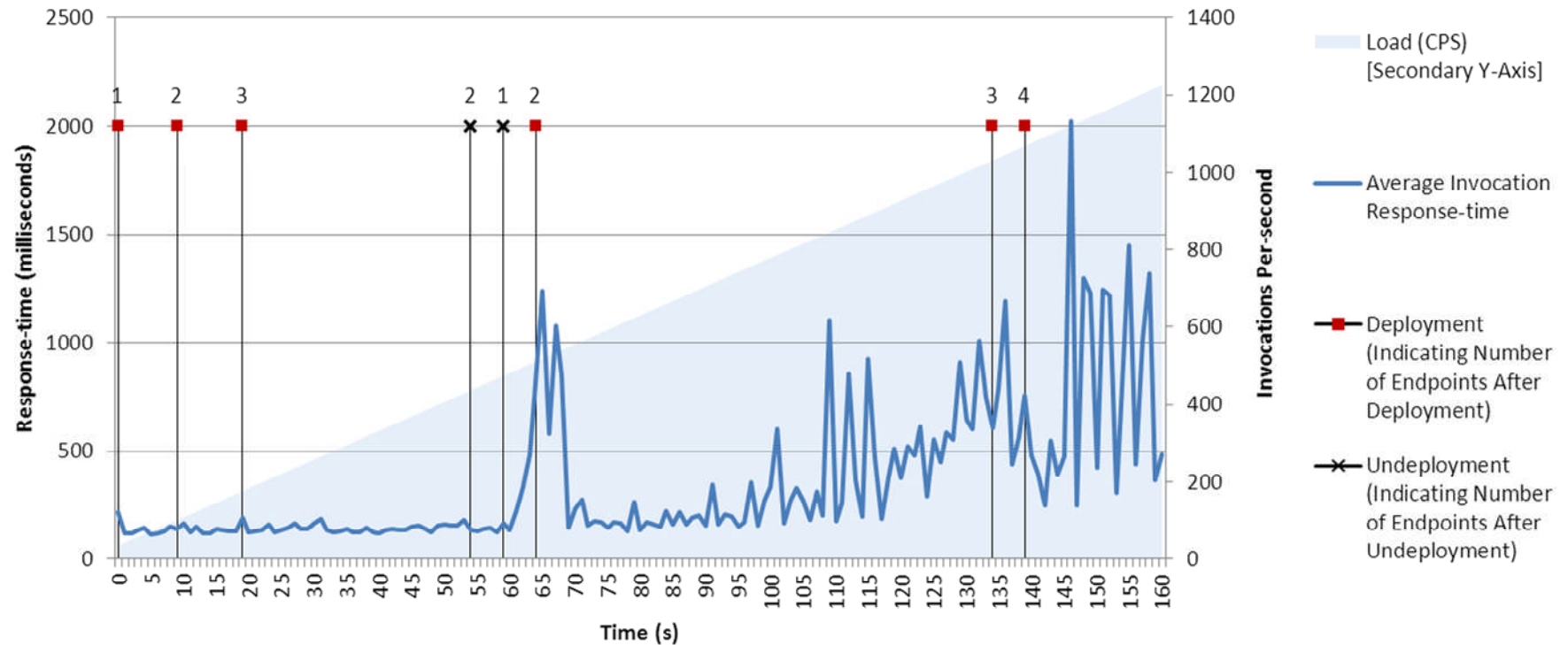


Fig. 103 Results of Experiment 3.1 showing the average invocation response time experienced by Service Consumers, and with vertical lines indicating when the generic Manager changed the provisioning level of the TargetExperimentEndpoint Web Service.

#### 5.4.5.1.3 Experiment 3.1 Conclusions

We have observed in previous experiments that a Manager is capable of deploying and undeploying endpoints of a Web Service and of updating the Web Service's entries in the ActiveServiceDirectory. We have also demonstrated that altering the provisioning level of a Web Service has a measurable effect on the average invocation response time experienced by Service Consumers invoking operations of that Web Service. The results of this experiment indicate that the generic Manager is capable of making these provisioning decisions based upon an analysis of the usage data returned by ServiceHosts, and that these provisioning decisions are shown to be in response to known changes in demand.

The remaining experiments are designed to provide more insight into the functionality of the management framework and its suitability to the provision and management of a dynamic, multi-endpoint Web Service environment. The generic Manager will be tested with a more taxing workload in the next experiment, and two further Managers will be demonstrated which make provisioning decisions based on a different measure of demand. The final section of this chapter presents an analysis of the overhead cost of providing the infrastructure services and the per-endpoint cost of managing a Web Service in the infrastructure.

#### 5.4.5.2 EXPERIMENT 3.2 WORKLOAD, MANAGER AND CONFIGURATION DETAILS

The following experiment uses the same generic Manager as the previous experiment, this time with the Service Consumers executing a more dynamic workload. Shown in Fig. 104 below, the workload describes a sin-wave pattern, with swift changes between periods of high and low call volume. The workload begins at 560CPS and dips down to 35CPS in the first ten seconds before rising again. The entire experiment includes 7 half periods (or 3.5 full wave periods) of 28 seconds each, cycling from troughs of 35CPS to peaks of 1225CPS for a total of 200 seconds. The Manager will make provisioning decisions using identical criteria to the previous experiment and will also evaluate using the same five second interval.

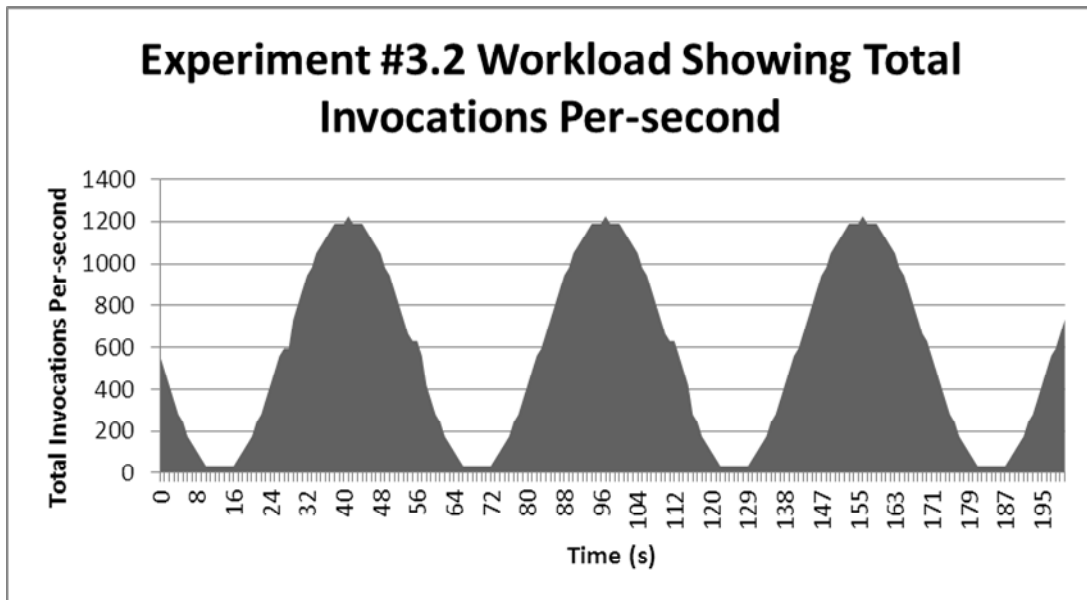


Fig. 104 Experiment #3.2 sin-wave workload to be executed from 35 Service Consumer machines by invoking the 'doNothing' operation of the TargetExperimentEndpoint Web Service at the specified rate.

##### 5.4.5.2.1 Experiment 3.2 Results

The results of executing the described experiment are shown in Fig. 105 on page 168. The number of invocations performed per-second by Service Consumers is represented by the light-blue wave which is again rendered on the secondary axis to the right. The provisioning events executed by the Manager are represented with vertical black lines with red boxes indicating deployments and black 'X's indicating undeployments. The number of endpoints available after the provisioning event is indicated with a number above each event icon. The average invocation response-time experienced by the Service Consumers is indicated with the blue line.

The results of this experiment show deployments in each of the first three periods as the ServiceHost(s) struggle to support the immediate burst of invocations. Although the invocation rate drops from 560CPS to 35CPS within the first 15 seconds, the initial response-times of newly deployed endpoints has been shown in experiment 3.1 to be high enough to prompt deployments even at low rates of demand. This effect is demonstrated here in the deployment labeled '4'. The next three deployments are in response to rising response-times at the ServiceHosts due to the swiftly increasing invocation rate which climbs from 35CPS to 1225CPS over a period of 25 seconds. As the invocation rate drops off again at the 45-second mark there are no additional deployments as the Service Consumer demand is able to be absorbed by the existing endpoints. After descending into the second low-cps trough, three consecutive periods of flat response-time prompt the Manager to undeploy an endpoint (marked '6') at the 80-second mark.

As the second peak of 1225CPS is approached, the increasing invocation rate does result in a slight increase in the average invocation response-time experienced by the Service Consumers – indicating that the six endpoints did experience slight increases in the average time taken to service requests, but not enough to prompt the Manager to deploy an deployment. What this slight increase does do, however, is set a higher historical value for the average call duration as measured at the ServiceHosts during those high-cps periods. When demand begins to drop towards a 35CPS trough at the 100-second mark, the average call duration also drops significantly enough that it prompts the Manager to execute undeployments in each of the next three periods. After undergoing a 'warming up' period, as we observed with the square-wave workload in experiment 2 and experiment 3.1, the remaining three endpoints are capable of absorbing the final 1225CPS peak without requiring an additional endpoint to be deployed. As the demand rate declines towards 35CPS for the last time the Manager performs one additional undeployment, leaving two active endpoints to meet the Service Consumer demand for the remainder of the experiment.

### Average Invocation Response-time Experienced by Service Consumers Invoking Operations of Managed TargetExperimentEndpoint Web Service in a Sin-wave Workload from 35-1225CPS, Indicating Provisioning Events

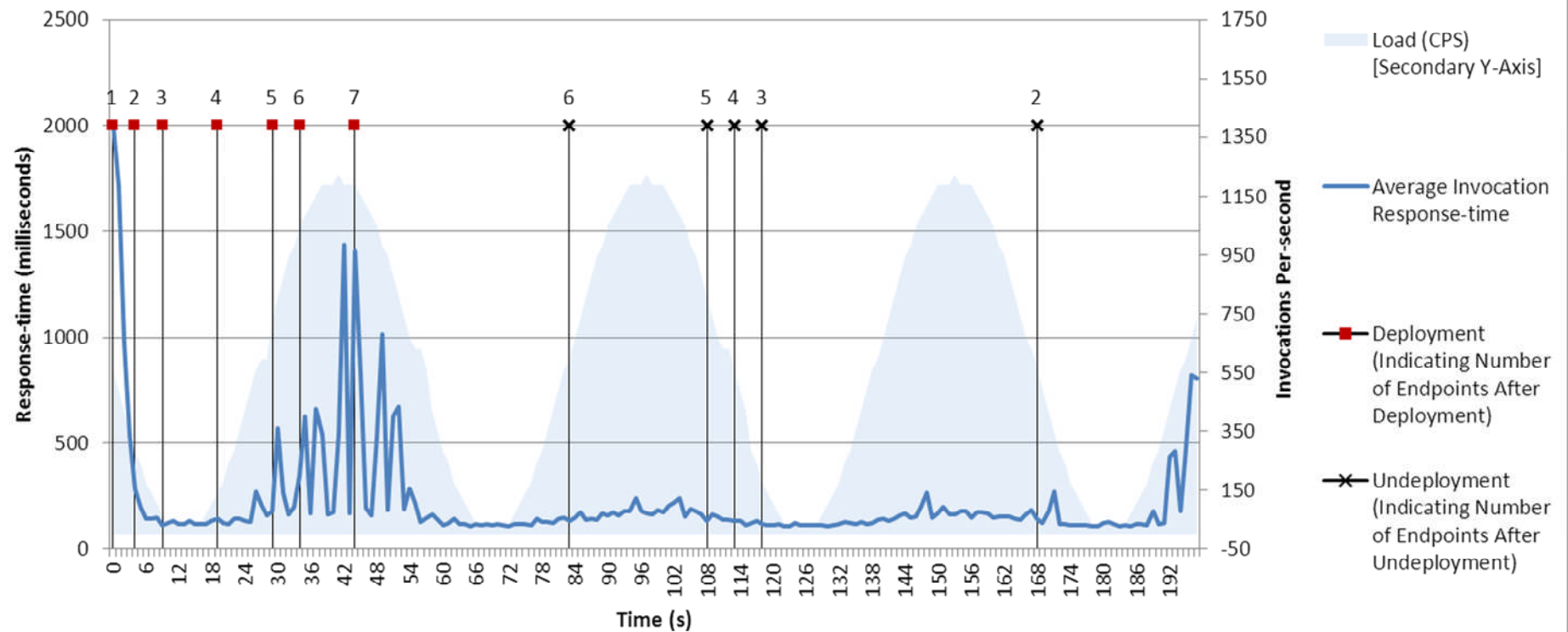


Fig. 105 The results of executing experiment 3.2, indicating the load imposed by Service Consumers (secondary axis), the provisioning events carried out by the Manager, and the response-time experienced by Service Consumers.

#### 5.4.5.3 *MANAGING BY INVOCATION-COUNT*

Experiment 3.1 and Experiment 3.2 have shown that the generic Manager is capable of analyzing usage data reported by ServiceHosts and making effective decisions that are in direct response to a measured state of the system. Their actions are shown to materially resolve imbalances between resource-allocation and demand: excessive increases in response-time are met by the deployment of additional capacity, while the resources of over-provisioned services are reclaimed so that they may be applied elsewhere as necessary.

The next set of experiments will attempt to address the same issues using a different measure of demand for a Web Service. In the following two experiments the number of invocation requests for the TargetExperimentEndpoint Web Service will be used to guide the provisioning decisions of a custom Manager. This approach is clearly not appropriate for a generic Manager and can only be applied in this circumstance because we have already measured the performance characteristics of the Web Service to be managed (TargetExperimentEndpoint) using a specific deployment technology (Axis) on a known set of ServiceHost machines. The purpose of this experiment then is not to provide a supposedly ‘generic’ technique for managing the availability of a Web Service, but to provide an example of one of the many other possible ways that a Manager can use the usage data returned by ServiceHosts to make decisions on exactly what level of provisioning is adequate for the Web Service being managed.

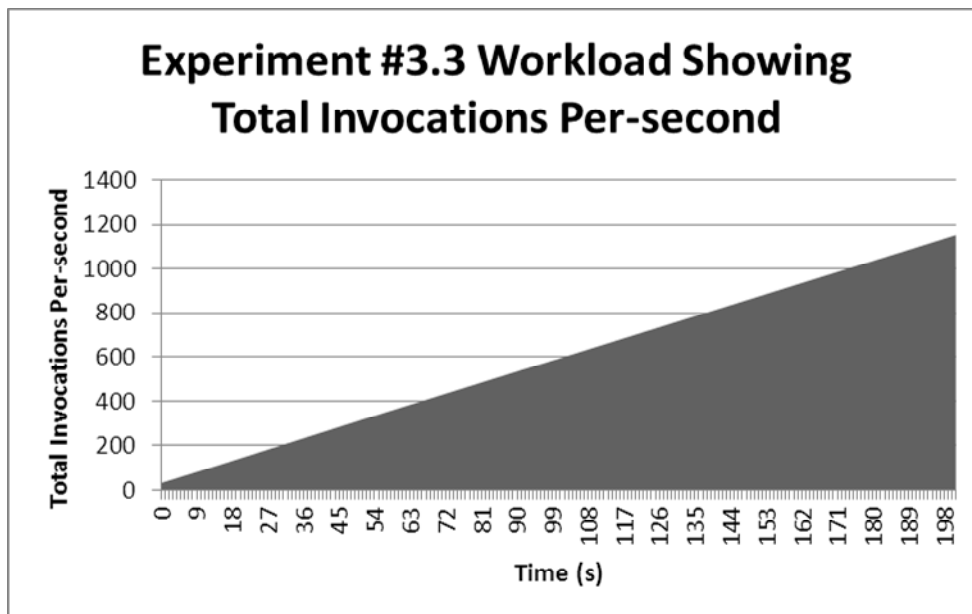
##### 5.4.5.3.1 Experiment 3.3 Workload, Manager and Configuration Details

The custom Manager implemented for this experiment evaluates reported usage data every five seconds, just like the Manager demonstrated previously. Rather than compare the contents of the current usage reports to the reports of the previous periods, this Manager simply counts the number of invocations received by each deployed endpoint of the TargetExperimentEndpoint Web Service, as indicated in the reports returned by ServiceHosts. The call count is divided by the number of reports received during the current evaluation period to give a value that represents the average number of invocations handled by each active endpoint during the period. Based upon the results of previous experiments using the Axis implementation of the TargetExperimentEndpoint Web Service, this Manager will deploy an additional endpoint if the average number of invocations per-endpoint is greater than 200 invocations per-second – that is, if the total number of invocation requests received over the past five seconds averages to 1000 invocations per endpoint.

Due to the failure of a number of systems (including the backups), 32 Service Consumer nodes will be used for the remainder of the experiments, as opposed to the 35 used previously.

Because of a desire to maintain the same performance characteristics of the Service Consumer machines they will continue to execute the same individual workloads, with the collective total in this experiment beginning at 32CPS and ending at 1150CPS. Fortunately, the provisioning decisions being made by Managers in this and the remainder of the experiments are based upon the total number of invocations received at the ServiceHosts and not the number of invocations performed per-Service Consumer. Due to this fact the loss of three machines will not negatively impact our ability to evaluate the behaviour of the custom Managers as they will continue to respond to measured fluctuations in demand.

In the first experiment the set of 32 Service Consumers will execute a linear increasing workload for a period of 200 seconds, imposing a load ranging from 32CPS to 1150CPS. The custom Manager will evaluate usage data every five seconds and ServiceHosts will correspondingly report usage data every five seconds. The workload for experiment 3.3 is shown below in Fig. 106.



**Fig. 106** Workload executed by 32 Service Consumer nodes during experiment 3.3, indicating the total number of invocations per-second executed during each one-second interval during the 200-second duration of the experiment.

#### 5.4.5.3.2 Experiment 3.3 Results

The results of executing the described experiment are presented in Fig. 107 on page 172. The load imposed by Service Consumers is represented in light-blue and plotted on the secondary y-axis to the right. Provisioning events are indicated with vertical black lines with red square icons representing deployments. The numbers above the deployment events indicate the total number of endpoints available after the deployment event. There were no undeployments performed

during this experiment. The average invocation response-time experienced by Service Consumers is represented by the blue line.

The results of this experiment show that the custom Manager deployed additional endpoints of the TargetExperimentEndpoint Web Service at regular intervals as the number of invocations grew at a steady rate. This is the expected behaviour of a Manager making provisioning decisions based upon the reported number of invocations received by endpoints of the Web Service it is managing. The results show that the ServiceHosts can accurately measure and report usage data to a Manager and that the Manager can analyze and act upon the reported data. Deployments are shown to be in direct correlation to the known workload being executed during the reporting period directly preceeding the deployment event.

Selecting an average of 200CPS as the trigger point for deployment may be a conservative measure but it does serve to keep the response time experienced by Service Consumers very consistent, with the average response time staying below 200ms for the duration of the experiment, even under high call volume. Having demonstrated the ability of the custom Manager to make decisions based upon the average number of invocations received by the deployed endpoints, the Manager will now be evaluated using the more taxing sin-wave workload used previously in experiment 3.2.

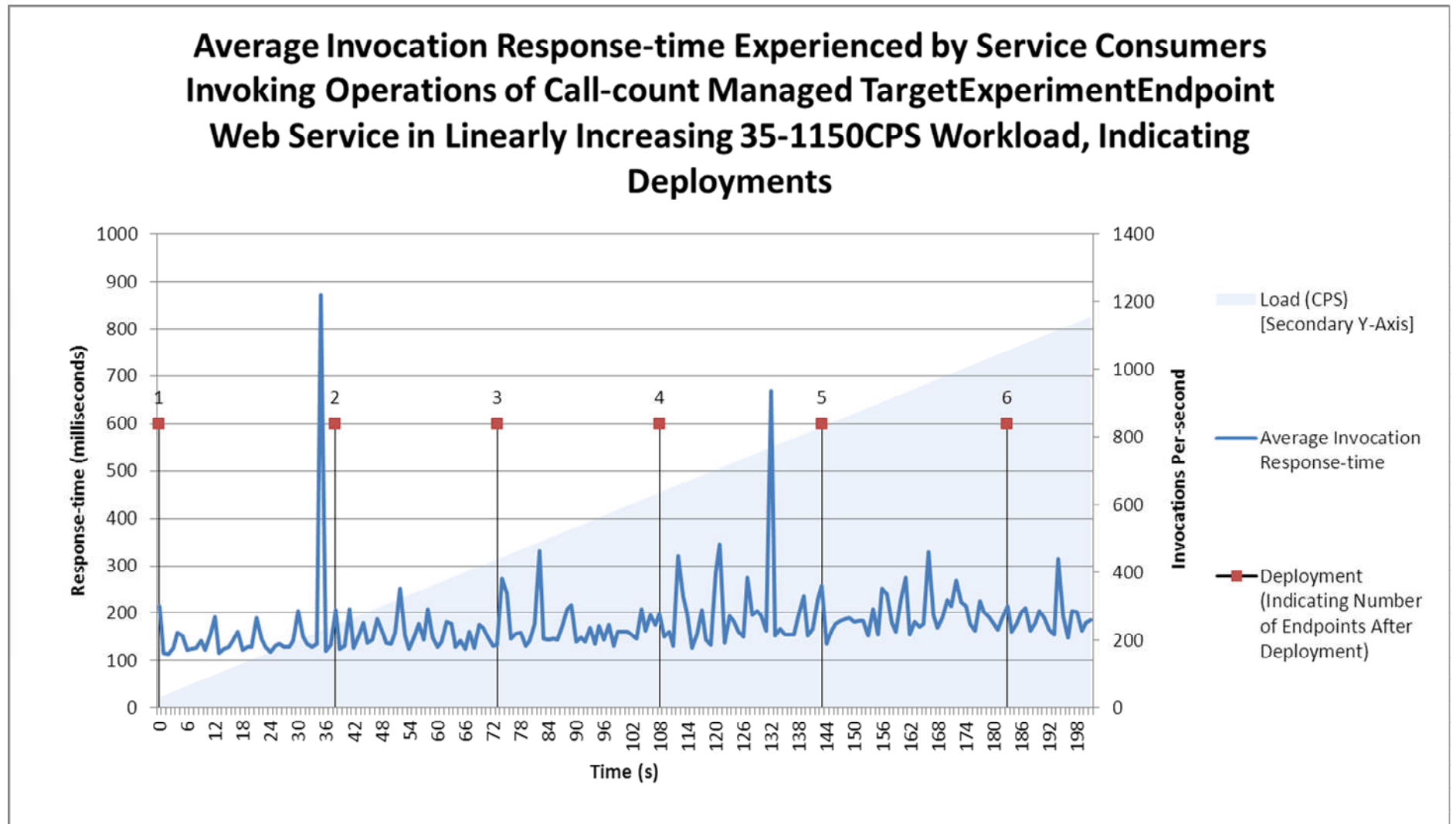


Fig. 107 The results of executing experiment 3.3, indicating the load imposed by Service Consumers (secondary y-axis), the provisioning events carried out by the Manager, and the response-time experienced by Service Consumers.

#### 5.4.5.4 EXPERIMENT 3.4 WORKLOAD, MANAGER AND CONFIGURATION DETAILS

This experiment uses the same custom Manager as used previously in experiment 3.3 which makes provisioning decisions based on a count of the number of invocation requests received by the target Web Service during each reporting period. Presented in Fig. 108 below, the workload for this experiment involves 32 Service Consumer machines invoking operations of the TargetExperimentEndpoint Web Service in a sin-wave pattern. This 200-second workload includes 7 half-periods (or 3.5 full wave periods) of 28 seconds each, during which the invocation rate cycles between troughs of 32CPS peaks of 1150CPS. The custom Manager makes provisioning decisions using identical criteria to the previous experiment and evaluates the reported usage data using the same five-second interval.

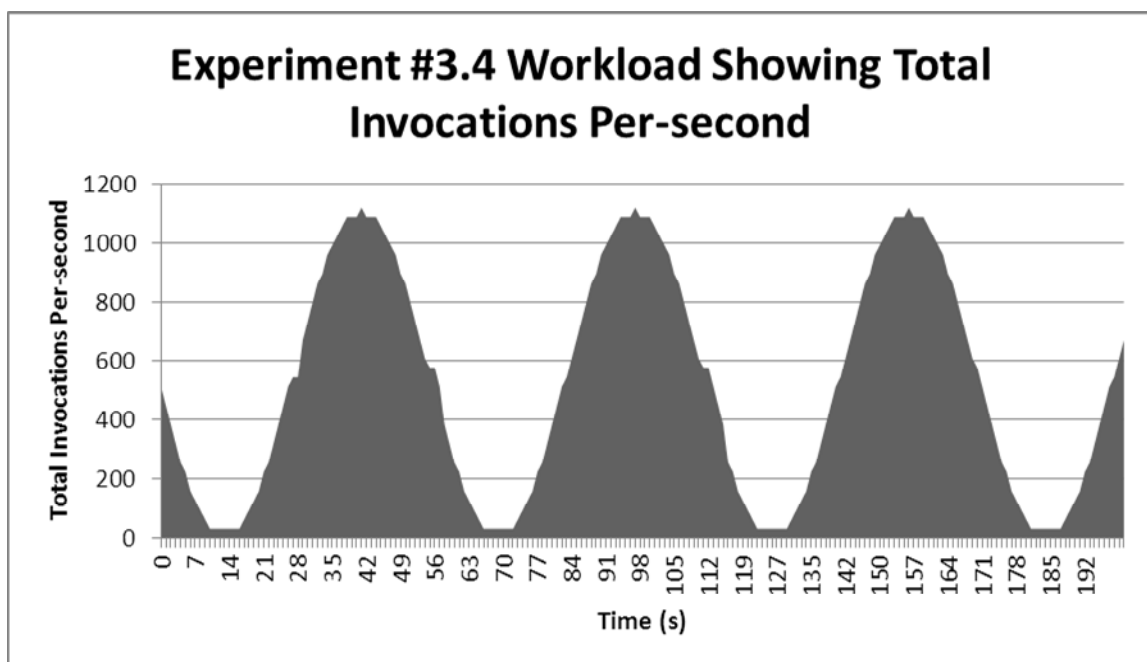


Fig. 108 Experiment #3.4 sin-wave workload to be executed from 32 Service Consumer machines invoking operations of the TargetExperimentEndpoint Web Service at the specified rate.

##### 5.4.5.4.1 Experiment 3.4 Results

The results of executing the described experiment are presented in Fig. 109 on page 176. As with the previous experiments, the light-blue wave indicates the number of invocations performed during each one-second period and is rendered on the secondary y-axis to the right side of the graph. The blue line indicates the average invocation response time experienced by Service Consumers executing the specified workload. Black vertical lines indicate provisioning events, with red square icons representing deployments and black 'X's representing undeployments. The number above each provisioning event icon indicates the number of endpoints available after the event.

As in experiment 3.3, the custom Manager in this experiment is shown to respond to changes in the measured level of demand for the TargetExperimentEndpoint Web Service by deploying and undeploying endpoints as the invocation rate changes. This behaviour was demonstrated in experiment 3.3 for a linear increasing workload and is demonstrated again here for a sin-wave workload.

The provisioning activities of the Manager are shown to correspond to the known workload, and the average invocation response-time experienced by Service Consumers is shown to change in response. The observed reductions follow significant response-time spikes experienced each time demand begins to increase (at the 25-, 80-, and 135-second marks). When compared to the results of the response-time Manager used in experiment 3.2, the response-time Manager was more sensitive to slight demand increases during the first steep increase (between seconds 15 and 40) and by increasing the number of deployed endpoints earlier it managed to keep the average invocation response-time experienced by Service Consumers to a lower level than the invocation-count Manager used in this experiment. Undeployments in experiment 3.2 were also performed based upon response-time and because of this fewer endpoints were undeployed than in this experiment where they were undeployed based on the measured invocation rate. We therefore observe two different behaviours: many early deployments followed by gradual undeployment as response-times stabilized (experiment 3.2) and deployment and undeployment events that directly follow the workload (experiment 3.4).

This thesis takes no stance on which management approach is ‘better’ because each Manager implements a different provisioning policy. What is important is that both Managers are shown to be capable of analyzing usage data reported by ServiceHosts and that the causes and results of their subsequent actions are representative of their individual policies. The policy approaches are each applicable to Web Services which experience different workload situations and, in particular, have different costs of deployment. With just 7 deployment events, the average endpoint remained deployed for a period of 120 seconds in experiment 3.2. In experiment 3.4 there were 19 deployment events and each endpoint remained deployed for an average of 66 seconds. For Web Services which experienced periods of extended use, the response-time Manager is able to take advantage of the ‘warming up’ effect observed previously, while the invocation-count Manager effectively provisions fixed capacity per-call. Deploying an endpoint of the TargetExperimentEndpoint is considered cost-free in this evaluation (the round-trip time for the Manager invoking the ‘requestDeployment’ operation of the ServiceHost Web Service takes an average of 9ms longer than a call which does not enact deployment). In practice,

however, this deployment cost may be much more expensive – Web Services which need to resolve external dependencies or startup databases, for example, could take significantly longer to deploy. In these cases the response-time Manager’s 7 deployment events in experiment 3.2 would result in significantly lower overall deployment costs than the 19 deployment events executed by the invocation-count Manager in experiment 3.4. Similarly, the length of time between the Manager executing a deployment event and the endpoint actually being ready for use may significantly impact the time taken to service the requests which initially prompted the deployment.

When compared with the results of experiment 3.2, the response-time spikes observed in this experiment can be attributed to an inadequate number of active endpoints to absorb a swiftly rising level of demand. With this in mind, two follow-up experiments are presented in the upcoming sections which implement minor policy changes in the invocation-count Manager. These Managers take two separate approaches. In an effort to respond more quickly to rising levels of demand, the first experiment evaluates an invocation-count Manager which can deploy multiple endpoints (i.e. more than one) during each evaluation period. The second experiment also uses the invocation-count Manager and aims to reduce the effects of sharp increases in demand by pre-provisioning a ‘sponge’ endpoint to soak up excess requests until the Manager can provision additional capacity. These experiments will be subjected to the same sin-wave workload as used in this experiment and their results presented in the same format. After these experiments the results of all four Managers subjected to the sin-wave workload will be discussed, and the chapter will conclude with an analysis of the various costs involved in managing a service.

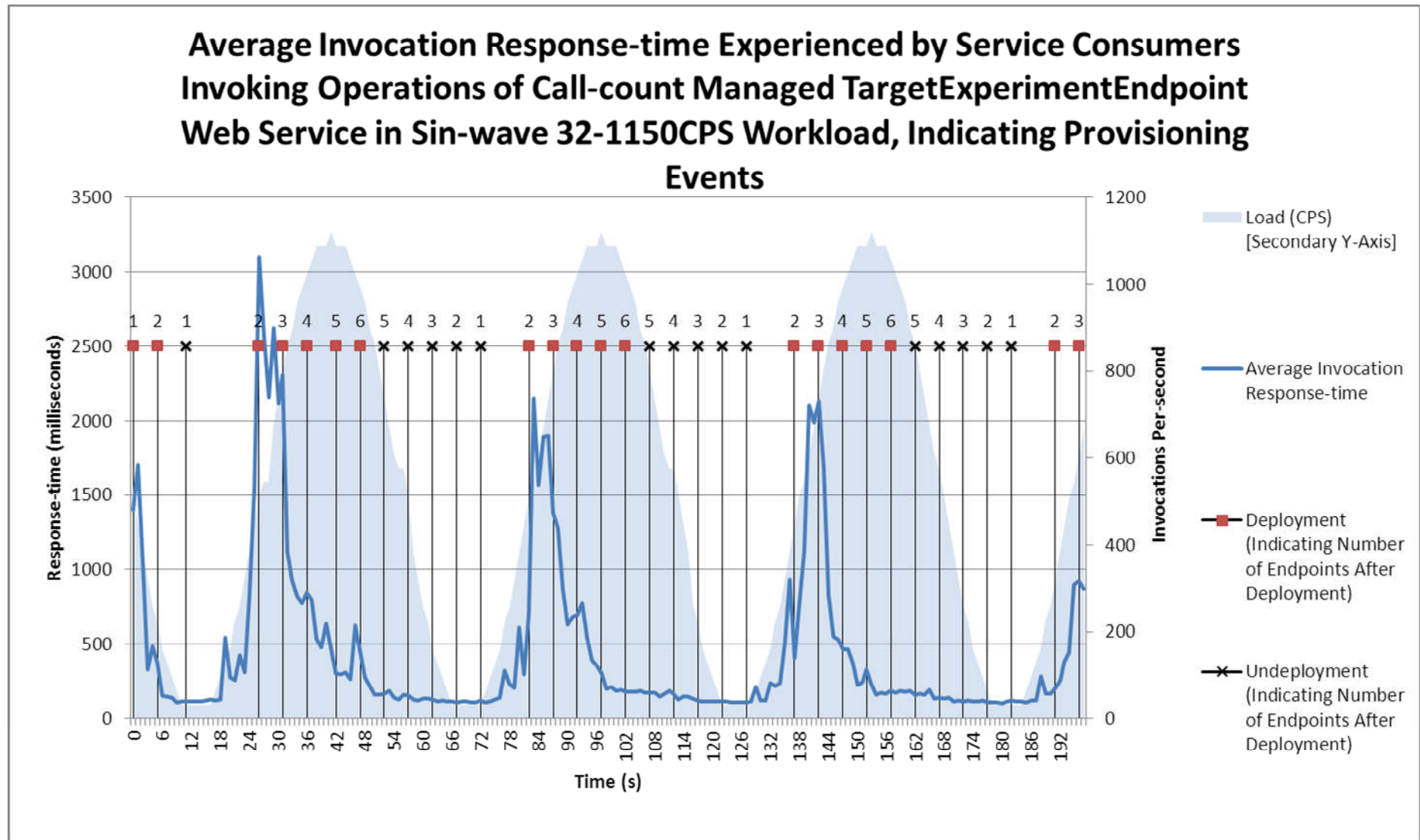


Fig. 109 The results of executing experiment 3.4, indicating the load imposed by Service Consumers (secondary y-axis), the provisioning events carried out by the Manager, and the response-time experienced by Service Consumers.

#### 5.4.5.5 ADDITIONAL FOLLOW-UP EXPERIMENTS

##### 5.4.5.5.1 Invocation-count Manager with Simultaneous Endpoint Provisioning

This experiment is identical to experiment 3.4 in all ways except that the Manager is designed to be able to deploy multiple endpoints at a time. If the excess demand is greater than 200 invocations per-second, per-endpoint, the Manager will deploy as many endpoints as necessary to address the reported level of demand (e.g. an extra 400CPS will yield two deployments, 600CPS three deployments, and so on). Undeployment will be performed more conservatively, with the Manager only undeploying a single endpoint per reporting period so as to not crush the remaining endpoints with demand. The experiment workload to be executed by the Service Consumers is shown in below in Fig. 110.

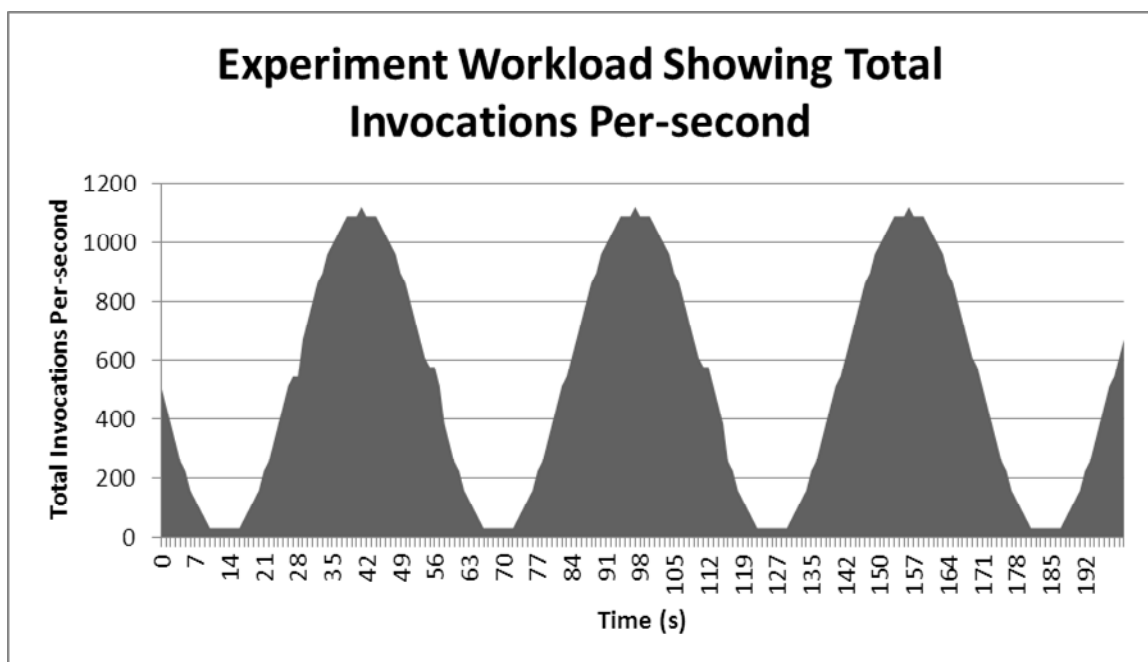


Fig. 110 Sin-wave workload executed from 32 Service Consumer machines invoking operations of the TargetExperimentEndpoint Web Service at the specified rate.

##### 5.4.5.5.2 Results of Invocation-count Manager with Simultaneous Endpoint Provisioning

The results of executing the described workload with the modified invocation-count Manager are shown in Fig. 111 on page 180. The results are presented in the same format as previous experiments, with the workload represented by the light blue wave and rendered on the secondary y-axis. Provisioning events are represented by black vertical lines, with red square icons indicating deployments and black 'X's indicating undeployments. The number of endpoints available after each provisioning event is indicated with a number above each icon. For ease of reading the relative height of the icons is increased slightly if more than one endpoint was deployed during the event. Average invocation response-time as experienced by Service Consumers is indicated by the blue line.

In this experiment, the ability of the Manager to deploy as many endpoints as necessary to meet the measured level of demand is shown to be somewhat effective in reducing the average invocation response-time experienced by Service Consumers. Compared to the results of experiment 3.4, the ability to deploy multiple endpoints enables the Manager to more effectively control average invocation response-time during periods of steeply rising demand. While the results of the current experiment still contain a response-time spike of over three-seconds around the 90-second mark, this spike is reduced twice as quickly as it was in experiment 3.4 when the Manager could only deploy one endpoint at a time.

Although the provisioning activities of the Manager are shown to roughly correspond to the known workload, closer inspection reveals a number of provisioning events that seem inappropriate: the double-deployment labeled '4' at second-mark 30, the gap in deployment at second-mark 90, and the undeployment marked '3' at second-mark 156. These provisioning patterns are symptomatic of some intricate behaviour related to the timing of usage data reports alluded to earlier: specifically, that the first usage report from a newly deployed endpoint can sometimes arrive at the Manager *after* the Manager's next evaluation period. In these cases the 'late' reports will not be considered in the Manager's provisioning decisions until the second evaluation period following deployment. In these situations the effects of the Manager's provisioning actions can be misrepresented in the results of its subsequent analyses and may lead to inappropriate provisioning decisions, as detailed below.

While the events that occur around the 100- and 150-second marks during the second and third peak of the workload are the most dramatic manifestations of this behaviour, the events labeled '2' and '4' during the first peak in fact present the most straight-forward example. As the experiment proceeds from second-mark 20 through second-mark 25, the invocation rate increases from 160CPS to nearly 400CPS. The ServiceHost hosting the single original endpoint (labeled '1' at second-mark 0) reports this usage data to the Manager who concludes correctly that the average invocation rate during the period was greater than 200CPS and that an additional endpoint should be deployed. The resultant deployment is represented by the provisioning event at the 25-second mark which is labeled with the number '2'. As the experiment proceeds from the 25-second mark to the 30-second mark the invocation rate continues to rise to nearly 800CPS. By this point the newly deployed endpoint has already been registered in the ActiveServiceDirectory and is beginning to receive invocation requests from ServiceConsumers as their POPs refresh the list of active endpoints of the TargetExperimentEndpoint Web Service.

Because of an overlap between the time when the new endpoint was deployed and the time when the Manager will next evaluate its received usage data, the Manager will not receive a report from the new deployed endpoint's ServiceHost during the reporting period immediately following its deployment. The Manager will therefore end up making its calculations based upon a single report from the ServiceHost hosting the single pre-existing endpoint (labeled '1' at second-mark 0). This report will indicate that the endpoint is supporting a large share of the demand. Under increasing levels of demand, as in the current example, the usage data report for the original endpoint may in fact indicate an even greater invocation rate than the previous report, even though an additional endpoint has just been deployed. This is what happens during the period between the 25- and 30-second marks which resulted in the Manager deploying two endpoints simultaneously, as indicated by the event labeled '4'. The demand rate reported by the original endpoint was above 600CPS and, as the Manager only received a single report during the period, the average invocation rate per-endpoint was calculated as over 600CPS per-endpoint. The Manager thus deployed enough additional capacity to absorb 400CPS of excess demand (the double-deployment labeled '4'). If the Manager had received the second report during this evaluation period it would have only deployed a single endpoint, as the invocation rate between the 25- and 30-second marks is above 600CPS but below 800CPS.

Depending on the quantity of load absorbed by newly deployed endpoints during their first reporting period, this overlapping of reporting and evaluating may have the opposite effect of the example just presented and may result in the *undeployment* of an endpoint instead. This behaviour can be observed during the third peak of the experiment, between seconds 150 and 165. The two endpoints deployed in the event labeled '4' at the 150-second mark draw enough un-reported load away from the two pre-existing endpoints that the reported average invocation rate drops below 200CPS and an undeployment is prompted (labeled '3' at the 155-second mark). When the reports for the two newly deployed endpoints arrive during the subsequent reporting period they indicate a very high call volume per-endpoint. Upon analyzing these reports the Manager concludes that additional capacity is required and deploys two more endpoints, as shown in the event labeled '5' at second-mark 160. This deployment occurs even though the invocation rate has begun to decline.

### Average Invocation Response-time Experienced by Service Consumers Invoking Operations of Call-count Managed TargetExperimentEndpoint Web Service in Sin-wave 32-1150CPS Workload, Indicating Provisioning Events

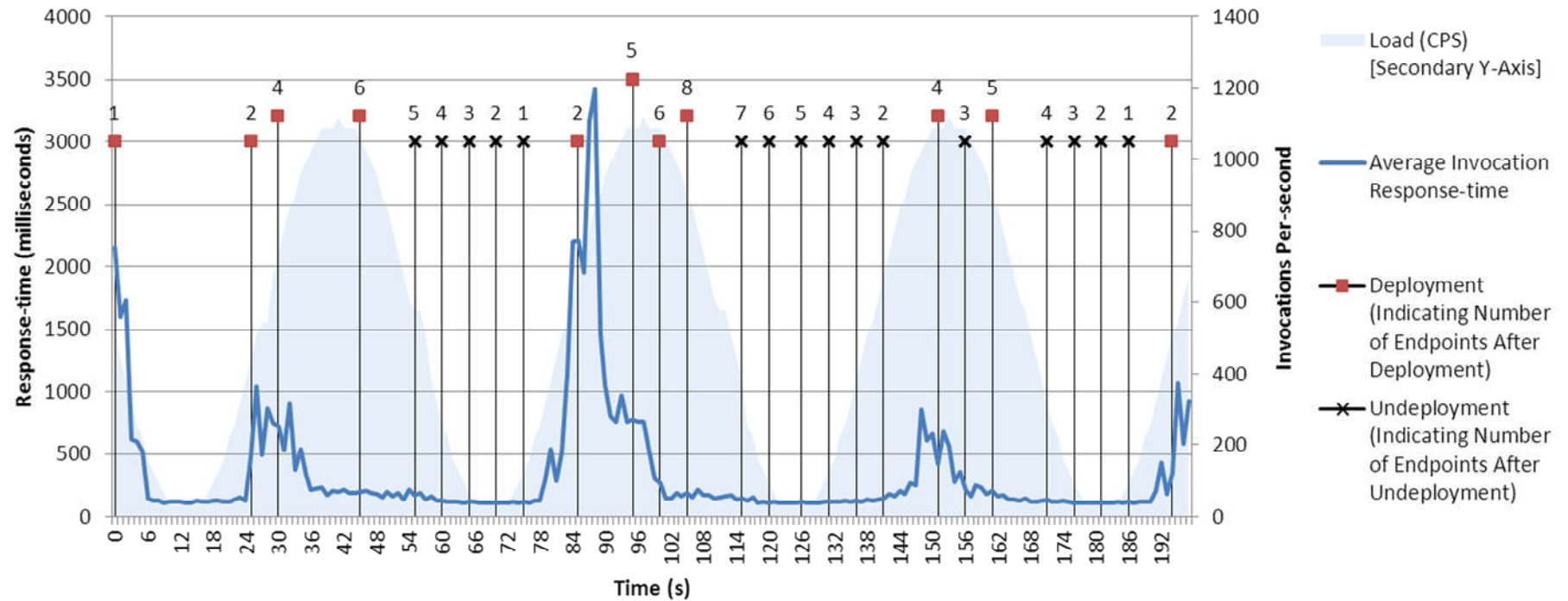


Fig. 111 The results of executing experiment 3.5, indicating the load imposed by Service Consumers (secondary y-axis), the provisioning events carried out by the Manager, and the response-time experienced by Service Consumers.

#### 5.4.5.5.3 Conclusions of Call-count Manager with Simultaneous Endpoint Provisioning

The previous experiment demonstrated that deploying multiple endpoints of a Web Service during a single reporting period can reduce the average invocation response-time experienced by Service Consumers more quickly than if only a single endpoint is deployed. It also highlighted the dangers of over-reacting to decisions made based on incomplete information. A number of approaches described below may be implemented at either the Manager or ServiceHost in order to reduce the likelihood of this problem. Any approach must be conscious of the design principles of the system: isolate domain-specific knowledge, eradicate redundant implementation artifacts such as error-handling routines, and avoid introducing case-specific branch logic into generic operations. An unacceptable solution, for example, would be to require every Manager to implement complex choreography techniques; a better solution would be to introduce an additional step into a linear task whose execution is unrelated to the reporting of usage data (such as, for example, deployment). A ideal approach would use the existing mechanisms of the architecture to induce the desired behaviour, rather than forcing any actor or actors to perform additional steps or provide additional choreography-related operations.

When a Manager requests the deployment of an endpoint by invoking the 'requestDeployment' operation of a ServiceHost's exposed IServiceHost Web Service, the request includes a long value indicating the number of milliseconds the ServiceHost should wait before first reporting endpoint usage data to the Manager. This value has the same effect as the value returned to the ServiceHost upon invoking the 'reportUsageData' operation of a Manager – in the case of these experiments, the value used was 5000ms. A straight-forward means of increasing the likelihood that the first usage data report for a Web Service arrives on-time could be for the ServiceHost to use half of the specified report-period value the first time it reported usage data. This would introduce case-specific branch logic into the ServiceHost implementation which is not ideal. As an alternative, a Manager could specify a value lower than its default reporting period value when requesting deployment. This would be an additional step in a linear task that does not branch and would provide the same result. If the ServiceHost reported on-time using the lower report period specified in the deployment request, the Manager could then return it the default report period value, which would result in the ServiceHost's reports lining up more precisely with the Manager's evaluations. Even so, this scenario only gets the Manager marginally closer to the 'real' truth of the state of the endpoints when they evaluate the reports, and the Manager may yet make technically accurate provisioning decisions based on technically inaccurate – or not wholly representative – usage data.

In the absence of monitor touch-points on every managed endpoint and hooks into every stage of every interaction, any entity attempting to manage a complex distributed system may be forced to make decisions based upon information that does not precisely reflect the current state of the system being managed. While inexact, Managers will have the opportunity to reassess the situation and correct any mistakes after another  $n$  seconds. In the specification of the presented architecture the reporting period returned to ServiceHosts invoking the `'reportUsageData'` operation of a Manager is only a request for when the ServiceHost should next report, not a requirement. Managers may decide to change where endpoints are deployed based upon misbehaving ServiceHosts, but it remains the case that Managers are inherently subjected to imperfect information from which to derive meaning about the state of the infrastructure. Any approach implemented by a Manager to influence the frequency of usage data reports by ServiceHosts is thus deemed to be policy-specific and is neither encouraged nor prohibited by the architecture.

One final management policy is presented and evaluated in the upcoming section which explores the utility of pre-provisioning spare endpoint capacity in order to cushion the ill effects that arise when the level of provisioning is insufficient to meet the current level of demand. The Manager is designed to both provide a buffer against the adverse effects of inappropriate provisioning decisions and absorb the impact of rapidly increasing demand. The aim was to demonstrate a straight-forward management policy that could provide some measure of relief from these adverse conditions without requiring any complex statistical analysis of historical usage data. The following invocation-count Manager pre-provisions a 'sponge' endpoint which soaks up the excess demand that causes the undesirable response-time spikes. Rather than waiting until a problem arises (like the average number of invocations per-endpoint rising above 200CPS) the Manager pre-provisions a cushion that both absorbs the impact of sharp increases in demand and reduces the repercussions of provisioning errors.

#### 5.4.5.5.4 Invocation-count Manager with 'Sponge' Endpoint

In terms of managing down the average invocation response time experienced by Service Consumers, the generic response-time Manager demonstrated in experiments 3.1 and 3.2 performed significantly better than the invocation-count Managers demonstrated thusfar. One cause of this poor performance is that the level of provisioning decided upon by invocation-count Managers was dictated by inadequate capacity – that is, a new endpoint was only deployed once the average load per-endpoint was above the decreed maximum. As a final experiment, the same workload (shown below in Fig. 112) will be executed using the same custom Manager with a slight

modification: when the Manager deploys the first endpoint of the TargetExperimentEndpoint Web Service, it also deploys a companion ‘sponge’ endpoint onto a separate ServiceHost – so called because its purpose is to soak up excess demand that will still lead to a deployment event but would, without the sponge, temporarily overload the other endpoint(s).

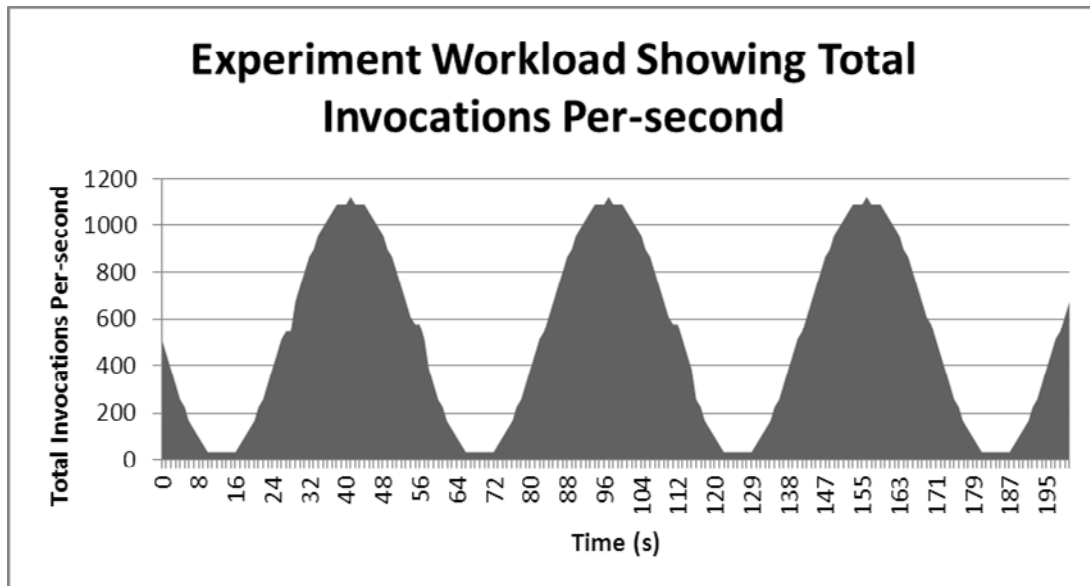


Fig. 112 Sin-wave workload executed from 32 Service Consumer machines invoking operations of the TargetExperimentEndpoint Web Service at the specified rate.

It doesn't matter which endpoint is considered the sponge – the effect is implemented in the Manager. When the Manager performs its evaluation it begins by adding up the invocation counts from all usage data reports to get a count of the total number of invocations performed during the period. The difference in this implementation is that in order to get the average number of invocations handled per-endpoint, the Manager divides the total invocation count by a number one less than the total number of reports received. This has the effect of making the average number of invocations received per endpoint during the past reporting period appear to be slightly higher, which prompts endpoints to be provisioned before they are needed, rather than after a problem has arisen.

#### 5.4.5.5 Results of Invocation-count Manager with ‘Sponge’ Endpoint

Of all the demonstrated Managers, the ‘sponge’ approach (Fig. 113 on page 186) is shown to be the best at controlling the average invocation response-time experienced by Service Consumers. The report-timing issue is still evident during the first wave of the workload but, because of the sponge, the repercussions of the erroneous undeployment labeled ‘2’ are much less severe. With the sponge deployed, the average invocation response time experienced by Service Consumers spikes to around 1-second after this undeployment; the identical workload

executed without the sponge in experiment 3.4 results in a spike in average invocation response time of over 2.5 seconds. Because of the ‘cushioning’ the sponge provides, steep increases in demand are not immediately borne out as steep increases in response-time, and response-time spikes are noticeably less-steep than in previous experiments. Lastly, the ‘sponge’ policy also resulted in Service Consumers experiencing the lowest average invocation response-time of all the management policies tested, with a reported average of 195ms versus 663ms for the multi-deployment Manager, 448ms for the non-sponge invocation-count manager (experiment 3.4), and 222ms for the response-time manager (experiment 3.2).

These results in no way imply that an invocation-count Manager is ‘better’ than a response-time Manager. Without having rigorously profiled the performance of the Axis implementation of the TargetExperimentEndpoint Web Service on ServiceHosts with known performance characteristics we would not have been able to select the deployment trigger of 200CPS per-endpoint. If operating in an infrastructure populated by ServiceHosts with varying performance characteristics this Manager would not work as well – either over- or under-provisioning services, and thus either wasting hosting resources or under-serving Service Consumers. As we have observed, the response-time characteristics of endpoints change over time, so response-time may indeed be a better long-term metric for making management decisions than simply counting the invocations. Not only is it more generally applicable to ServiceHosts with varying performance characteristics, but it is also better suited to Web Services which offer multiple operations that consume varying levels of system resources. Invocation counting has been useful for demonstrating the ability of a Manager to make quick decisions at short intervals based on an explicit measure of demand. It has demonstrated that the management framework is agile enough to make timely decisions based on fresh data and to provision the resources of the infrastructure accordingly. The different Manager implementations have served to demonstrate that different aspects of demand can be quickly communicated, compiled and used to make decisions on how and where resources are needed. They each represent different provisioning policies that may be appropriate for different Web Services in different situations, and thus one cannot be said to be ‘better’ than the other, only that the approaches should be applied where appropriate depending on the desired management behaviour.

While the ‘sponge’ approach does demonstrate a number of benefits for Service Consumers, it is not precisely aligned with the goals of an architecture that aims to directly allocate resources to meet a measured level of demand. It does, however, provide better response-times for Service Consumers, so evaluating the Manager depends on what the management policy considers as a

goal. Indeed, the 'sponge' and the multiple-deployment capability are just policy artifacts devised to deal with a manufactured situation which intentionally stresses the management framework by increasing demand faster than the Manager can keep up with. This is not to say that they are bad approaches to management and should be avoided by custom Manager implementations, but rather that they are just different ways of accomodating different observed behaviours under different workloads – different policies with different goals. All of the presented Managers have served as examples of how various management behaviour can be easily introduced into the architecture, and how the behaviour of a Manager directly affects the Service Consumer's experience of using a Web Service.

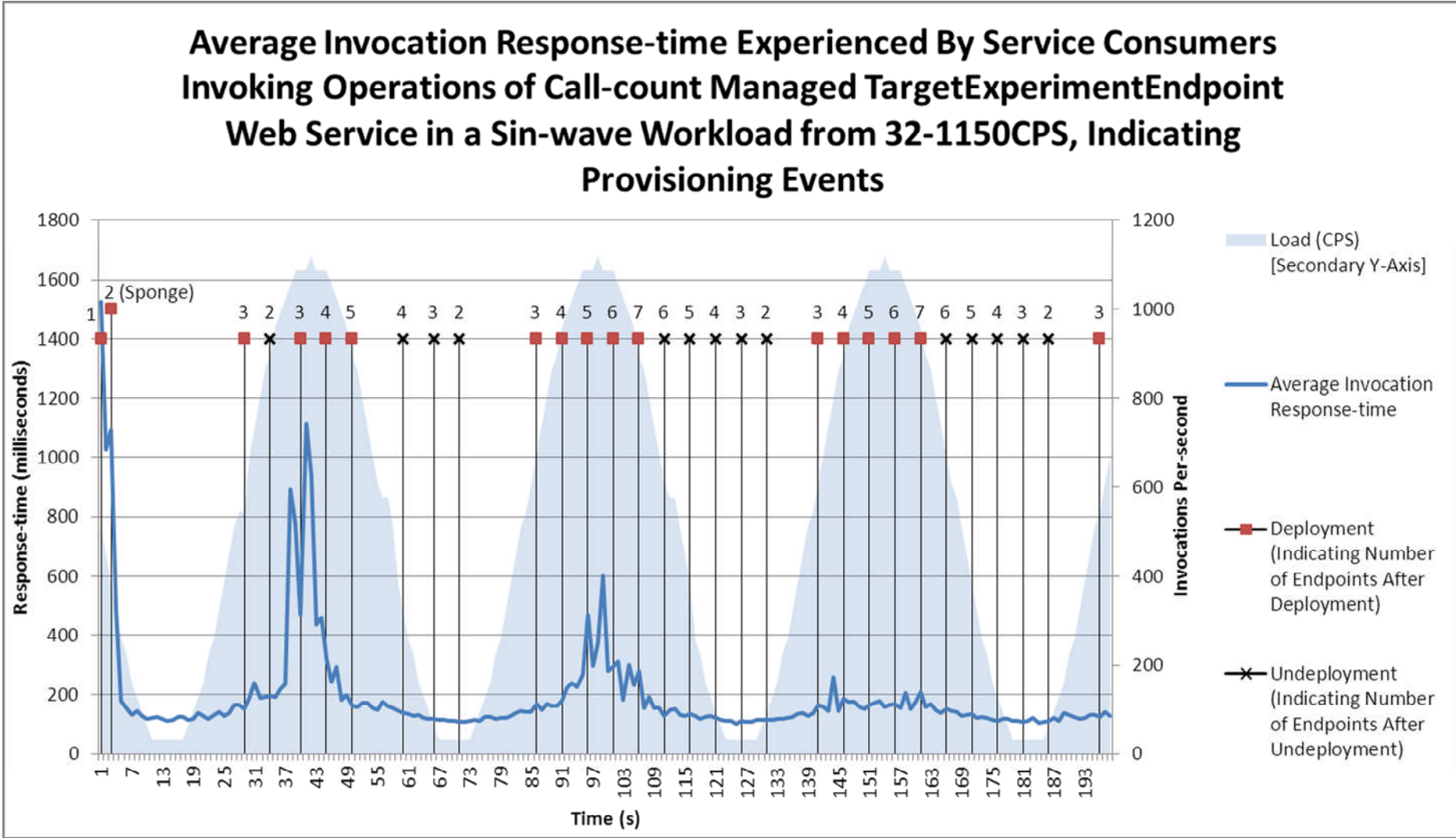


Fig. 113 The results of executing experiment 3.4 with an additional 'sponge' endpoint, indicating the load imposed by Service Consumers (secondary axis), the provisioning events executed by the Manager, and the response-time experienced by Service Consumers.

#### 5.4.6 EXPERIMENT 3 CONCLUSIONS

The results of experiment 3 provide insight into many aspects of the architecture that were either directly or indirectly tested. In terms of indirect features it was repeatedly demonstrated, for example, that ServiceHosts are capable of recording usage data for the endpoints deployed within their domain and reporting that data to the appropriate Manager. It was also demonstrated that Managers are capable of analyzing this usage data and making provisioning decisions based upon it. One point that was not constantly stressed but is nonetheless important is that while the published Manager implementation changed throughout the experiments, no other changes needed to be made to any infrastructure components or the generic mechanisms used to deploy and locate an endpoint or the Manager. Service Consumers executed the same consumer agent code, the TargetExperimentEndpoint implementation was not changed and, even when a different Manager implementation was published, ServiceHost reports arrived at the correct Manager without any changes to the ServiceHost. This is to say, everything worked as it was designed to. The only effort expended was by the only actor interested in introducing different behaviour into the management of a service, and this did not require any effort or knowledge on the part of any other actor or changes to any part of the infrastructure.

The results of the previous experiments show that ServiceHosts provide Managers with sufficient information to make informed decisions on the level of provisioning required to provide highly available services. Four Managers were demonstrated which made provisioning decisions based upon two aspects of demand, and their decisions have been shown to have a meaningful impact on the perceived availability of the service being managed. In experiments 3.1 and 3.2 the generic Manager was shown to be able to analyze the average time taken to service individual invocation requests and adjusting the level of provisioning accordingly. In experiments 3.3 and 3.4 the custom Manager made provisioning decisions based upon the reported invocation rate and was shown to deploy endpoints in direct response to the workload being executed during each reporting period.

In each of the two follow-up experiments the invocation-count custom Manager was altered slightly, firstly endowed with the ability to deploy multiple endpoints at a time, followed by a Manager that pre-provisioned a 'sponge' endpoint to soak up excess demand between evaluation periods. Both were subjected to the same sin-wave workload and each demonstrated different management behaviour in their results. While both were demonstrably capable of analyzing the reported usage data, the multi-endpoint Manager exhibited some complex behaviour related to the timing of the first usage data report from a newly deployed endpoint. This behaviour was

explained as the cause of ‘erroneous’ provisioning events during periods of increasing demand, which caused large spikes in the response-time experienced by Service Consumers. The ability to deploy multiple endpoints during a single evaluation period was shown to allow the Manager to over-react in the presence of incomplete usage data, but was also shown to be effective in reducing the average invocation response-time more swiftly than in experiment 3.4 when the Manager only deployed a single endpoint at a time.

In the final follow-up experiment a slight change to the invocation-count Manager was implemented which deployed two endpoints initially but counted the reported usage data as though there was only one endpoint. This Manager used the same provisioning policy as the other two invocation-count Managers (i.e. deploy when the invocation rate reached an average of 200CPS per-endpoint). The intent behind this Manager was that by pre-provisioning a buffer of capacity in the form of a ‘sponge’ endpoint, spikes in demand would be better absorbed without affecting the average invocation response-time experienced by Service Consumers, and that the endpoints would be deployed *before* they were needed rather than after a problem had been identified. The results of subjecting this Manager to the same sin-wave workload yielded some of the best results of any Manager and although it displayed the same ‘erroneous’ undeployment behaviour as the previous invocation-count Manager, the ‘sponge’ served to reduce the adverse consequences of this error, absorbing the excess demand until the Manager corrected its mistake.

In summary, the results of experiment 3 can be said to duly validate the hypothesis that “Managers are capable of identifying and dynamically responding to changes in the level of Service Consumer demand by increasing or decreasing the number of available endpoints of a Web Service.” The results satisfied the criteria for success on all fronts, namely that “each Manager implementation is shown to respond to measured changes in demand by altering the provisioning level of the Web Service being managed,” that “the generic Manager described previously is expected to be able to analyze the reported usage data and identify the rise in response-time, prompting a deployment which results in a drop in the response time,” and that “Managers making deployment decisions using a count of the invocation requests received during each reporting period, the provisioning decisions are expected to reflect a direct response to the reported workload being executed.”

Separate from the metric chosen to indicate changes in demand and implementation-specific evaluation processes employed by each presented Manager, the results of the previous experiments indicate that the newly described architecture, and the automated management of

the multi-endpoint framework in particular, is well-suited to the dynamic provision of software services in a distributed computing environment with varying levels of demand. Further, the results indicate that automating the service provisioning process by empowering a software Manager to make decisions on the level of provisioning required to service the measured demand is an effective means of controlling the average invocation response-time experienced by Service Consumers.

In the previous experiments the actions of Managers have been shown to materially resolve imbalances between resource-allocation and demand: increases in demand are met by the deployment of additional capacity, while the resources of over-provisioned services are reclaimed to that they may be applied elsewhere as necessary. While beneficial in terms of resource consumption and Service Consumers' experience with using a managed Web Service, this agile management does not come without a cost. The next and final section of this chapter details the overhead cost of providing and managing the infrastructure services (ActiveServiceDirectory, ServiceLibrary and HostDirectory) and the incremental cost of deploying and managing a service under increasing load and an increasing level of provisioning. The costs will be presented using the previous linearly-increasing workload experiment as a guide.

## 5.5 COST & SCALABILITY OF INFRASTRUCTURE SERVICES

### 5.5.1 INTRODUCTION

This section details the baseline overhead costs of providing the infrastructure services, the cost of deploying and managing a Web Service with a single active endpoint, and the incremental cost of deploying and managing each additional endpoint of a Web Service. The design of the management framework is analyzed in terms of the quantity of calls required to provide and manage the TargetExperimentEndpoint Web Service with a single endpoint, with an increasing number of endpoints, and under varying levels of Service Consumer demand. This section uses the linearly-increasing workload of experiment 3.3 as a guide since the demand rate and deployment patterns are consistent, easily understood, and the various costs can be easily demonstrated. A comparison of the long-term growth rates of demand are compared to the infrastructure-related costs associated with supporting that demand, and the qualitative benefits of the infrastructure will be weighted against the identified costs.

### 5.5.2 BASELINE INFRASTRUCTURE COSTS

A fully operational infrastructure includes one deployed endpoint of each of the three infrastructure services (ActiveServiceDirectory, ServiceLibrary and HostDirectory), one deployed Manager endpoint for each of these services, and two deployed MasterManager endpoints. Each of these endpoints is deployed on a ServiceHost whose responsibility it is to periodically report usage data to the endpoint's Manager. In the absence of ServiceConsumer demand, the sole contributor to the overhead communication cost of maintaining the infrastructure comes solely from these ServiceHosts reporting usage data.

ServiceHosts report usage data by binding to the POP and invoking the 'reportUsageData' operation of the Web Service's Manager. As the POP must locate an endpoint of the target Manager, reporting endpoint usage data requires two invocation requests: one from the POP to the ActiveServiceDirectory to locate the required Manager, and one from the POP to the Manager to deliver the report itself. The rate and quantity of intra-architecture communication during periods of zero demand is thus directly linked to the greater of either: the endpoint lease duration provided to the ServiceHost POPs, or the duration of the reporting period requested by Managers. The presented experiments used an endpoint lease duration of three seconds and ServiceHosts reported endpoint usage data to Managers every five seconds. While both of these

values are aggressive, they will be used in the forthcoming calculations in order to give a worst-case scenario for the baseline resource consumption figures.

When ServiceHosts send usage data reports for the ActiveServiceDirectory, HostDirectory and ServiceLibrary to each service's Manager, they invoke the 'reportUsageData' operation of the Manager Web Service via their local POP. If the ServiceHost's report period is greater than the endpoint lease duration (as it is in this example), the ServiceHost's POP will need to query the ActiveServiceDirectory for the location of the target Manager every time it reports usage data for an endpoint. Using a report period of five seconds, and a endpoint lease of three seconds, the list of Manager endpoints retrieved during the previous report period will be considered stale when the ServiceHost next reports, and thus each POP will need to look up a fresh list in the ActiveServiceDirectory. Thus each time a ServiceHost reports endpoint usage data, the direct communication cost is 2 intra-architecture invocation requests – one to the ActiveServiceDirectory and one to the target Manager – leading to a total direct communication overhead for the three infrastructure services of 6 invocations every five seconds.

Each of the three infrastructure services also requires a Manager, so there is an additional indirect cost associated with providing each service. Usage data for each Manager is reported to a MasterManager with the same frequency as reports for all other Web Services, adding a further 6 invocations every five seconds – again, three for the ActiveServiceDirectory lookups and three for the reports. Thus the total overhead communication costs for providing the infrastructure services is 12 invocations every five seconds, or 2.4 invocations per-second. (There is room for efficiency gains by sharing the results of POP queries between endpoints deployed on the same machine. Co-locating the Managers, for example, reduces the invocation rate to 1.8 invocations per-second. Also, because the overall demand on each Manager is very low, it is unlikely that it is necessary to report each Manager's usage data to the MasterManager as frequently as for other services.) Finally, the MasterManagers also communicate with one another and the ActiveServiceDirectory once every five seconds, contributing an additional overhead of 0.8 invocations per-second. In total, the infrastructure requires seven endpoints and 3.2 invocations per-second to sustain itself in periods of zero demand. Of this load the ActiveServiceDirectory supports a total of 1.6 invocations per-second. As detailed in the next sections, the baseline overhead costs of providing the infrastructure remain constant even under high levels of Service Consumer demand for high numbers of Web Services.

	Management of Infrastructure	Management of Managers of	Management of Master	Total
--	---------------------------------	------------------------------	-------------------------	-------

	Services	Infrastructure Services	Managers	
<b>Long-term Overhead Communication Cost</b>	1.2CPS	1.2CPS	0.8CPS	3.2CPS

**Proportion of Total Cost Borne by ActiveServiceDirectory**

1.6CPS
--------

### 5.5.3 COST OF DEPLOYING AND MANAGING A WEB SERVICE WITH ONE ENDPOINT

The following section details the worst-case cost of deploying the first endpoint of a Web Service for which there is no currently deployed Manager. The discussion uses the diagram in Fig. 114 on page 198 as a guide, and references to individual invocations performed by different actors in the diagram will be used throughout this section. The presented deployment process is sparked by the Service Consumer invoking the ‘doNothing’ operation of the ‘TargetExperimentEndpoint’ Web Service by binding to and sending the invocation request to its local POP. This action is labeled “SC1.1) TargetExperimentEndpoint.doNothing()” in the diagram. For the rest of the section each invocation will be referred to simply by its preceding identifier – in this case SC1.1, with the “SC” for “Service Consumer” and the 1.1 as the outgoing portion of the first invocation performed by SC. Because the name “TargetExperimentEndpoint” is quite long the abbreviation “TEX” is used.

When the Service Consumer POP receives invocation request SC1.1 from the Service Consumer, the POP begins its process of fulfilling the request by first retrieving the list of active endpoints of the TargetExperimentEndpoint Web Service from the ActiveServiceDirectory (SC2.1). Recall that the deployment of the first endpoint of any Web Service (including a Manager) is initiated when a POP sends a lookup request to the ActiveServiceDirectory for the list of all active endpoints of the requested Web Service and the ActiveServiceDirectory finds that none are listed. This circumstance is marked with a red ‘x’ in Fig. 114 and prompts the ActiveServiceDirectory to request the deployment of the first endpoint of the TargetExperimentEndpoint Web Service.

To request the deployment of the first endpoint of the TargetExperimentEndpoint Web Service the ActiveServiceDirectory binds to its local POP and sends an invocation request for the ‘requestFirstInstance’ operation of the TargetExperimentEndpoint\_MANAGER Web Service (ASD1.1). In order to fulfill this request the POP must first locate an endpoint of the TargetExperimentEndpoint’s Manager by looking it up in the ActiveServiceDirectory (ASD2.1). The

ActiveServiceDirectory again finds no deployed endpoints of the requested service, as indicated by the red 'x'. In this event the deployment request process recurses: the ActiveServiceDirectory binds to its POP and invokes the 'requestFirstInstance' operation of the Manager of the requested Web Service – in this case the requested Web Service is a Manager so the request is send to a Manager of Managers – a MasterManager – which is identified by the URI 'MANAGER\_MANAGER' (ASD3.1). In order to fulfill the request the POP looks up the list of all active endpoints of the 'MANAGER\_MANAGER' Web Service in the ActiveServiceDirectory (ASD4.1) and receives a set of ServiceInstanceDescriptors in response (ASD4.2). Now that the POP has an active endpoint to send the request to it binds to and invokes the 'requestFirstInstance' operation of the 'MANAGER\_MANAGER' Web Service endpoint described by one of the ServiceInstanceDescriptors returned from the ActiveServiceDirectory (ASD5.1). This invocation request prompts the MasterManager to undertake the deployment of the first 'TargetExperimentEndpoint\_MANAGER' Web Service endpoint.

The endpoint deployment process involves four main steps: locate all implementations of the Web Service to be deployed, locate all ServiceHosts willing to deploy Web Service implementations written by each implementation's Publisher, decide which implementation to deploy on which compatible ServiceHost, and invoke the 'requestDeployment' operation of the selected ServiceHost using the selected ServiceImplementationDescriptor as a parameter. Because the management of a Web Service may be performed by either a custom Manager or the generic Manager, the MasterManager begins the deployment process by checking the ServiceLibrary for any published custom Manager implementation by binding to its local POP and invoking the 'findAll' operation of the ServiceLibrary Web Service using the URI 'TargetExperimentEndpoint\_MANAGER' as a parameter (MM1.1). To fulfill this request the MasterManager's POP first locates an endpoint of the ServiceLibrary in the ActiveServiceDirectory (MM2.1), receives back a list of ServiceInstanceDescriptors describing the active endpoints of the ServiceLibrary Web Service (MM2.2), and performs the requested 'findAll' operation on a ServiceLibrary endpoint described in one of the returned ServiceInstanceDescriptors (MM3.1). Because there is no custom Manager published in this example the operation returns an empty set (MM3.2) which is returned as the result of the operation to the MasterManager (MM1.2). The MasterManager next locates the generic Manager implementation by binding to and invoking the same 'findAll' operation of the ServiceLibrary Web Service using the URI 'MANAGER' as a parameter (MM1.3). Rather than query the ActiveServiceDirectory for an endpoint, the POP can use the set of ServiceLibrary endpoints retrieved from the ActiveServiceDirectory during the previous ServiceLibrary invocation request. The POP binds to and invokes the requested

'findAll' operation of the ServiceLibrary on behalf of the MasterManager (MM4.1) and is returned a list of ServiceImplementationDescriptors describing all implementations of the generic Manager Web Service (MM4.2) which is then returned to the MasterManager (MM1.4). The next stage of deployment involves the MasterManager locating the list of ServiceHost willing to deploy Web Services published by each returned implementation's publisher.

This example assumes a single generic Manager implementation is published in the ServiceLibrary, and that at least one ServiceHost is registered in the HostDirectory as willing to deploy Web Service implementations published by the Publisher of the generic Manager implementation. To locate the set of HostDescriptors describing the willing ServiceHosts the MasterManager invokes the 'findAll' operation of the HostDirectory Web Service using the PublisherID element of the generic Manager implementation's ServiceImplementationDescriptor as a parameter (MM5.1). The POP receives the invocation request and proceeds to look up the list of active endpoints of the HostDirectory Web Service in the ActiveServiceDirectory (MM6.1), receives a set of ServiceInstanceDescriptors describing the active endpoints (MM6.2), selects an endpoint and performs the requested operation ('findAll') (MM7.1), receives a set of HostDescriptors in response (MM7.2) and returns these results to the MasterManager (MM5.2).

Having performed the first two steps of deployment – locating implementations of the requested Web Service and the ServiceHosts willing to deploy them – the third step involves the MasterManager applying local policy and selecting a ServiceImplementationDescriptor for deployment on a ServiceHost described by one of the HostDescriptors retrieved from the HostDirectory. There is one generic Manager implementation published in the current example and there are two ServiceHosts willing to deploy Web Service implementations published by its Publisher (called 'ServiceHost 1' and 'ServiceHost 2' respectively). The MasterManager selects a ServiceHost at random and ends up selecting 'Service Host 1'. To instigate deployment of an endpoint of the generic Manager Web Service the MasterManager binds to and invokes the 'requestDeployment' operation of the IServiceHost Web Service endpoint described in the HostDescriptor of 'Service Host 1'. If deployment is successful the MasterManager is returned the URL of the newly deployed endpoint (MM8.2), includes this URL in a ServiceInstanceDescriptor describing the newly deployed endpoint, and returns this ServiceInstanceDescriptor to the POP of the ActiveServiceDirectory that requested deployment of the TargetExperimentEndpoint\_MANAGER in the first place (ASD5.2).

The ActiveServiceDirectory POP returns the TargetExperimentEndpoint\_MANAGER ServiceInstanceDescriptor to the ActiveServiceDirectory (ASD3.2) which stores the

ServiceInstanceDescriptor under the URI 'TargetExperimentEndpoint\_MANAGER' (even though the endpoint uses the generic Manager implementation). Recall that the deployment of the first endpoint of the TargetExperimentEndpoint\_MANAGER Web Service was prompted by a lookup request for the service by the ActiveServiceDirectory's POP (ASD2.1). The ActiveServiceDirectory thus returns from the current level of recursion by one step when it returns the ServiceInstanceDescriptor of the newly deployed TargetExperimentEndpoint\_MANAGER endpoint to the ActiveServiceDirectory's POP (ASD4.2). The ActiveServiceDirectory's POP uses the endpoint described in this ServiceInstanceDescriptor to carry out the ActiveServiceDirectory's original request to deploy a new endpoint of the TargetExperimentEndpoint Web Service (ASD1.1).

The process of deploying an endpoint of the TargetExperimentEndpoint Web Service is identical to the process undertaken to deploy its Manager, with the exception that the Manager deployment process involves two lookups in the ServiceLibrary – one for a custom Manager and, failing that, one for the generic Manager. The ActiveServiceDirectory's POP initiates the deployment process by binding to the TargetExperimentEndpoint\_MANAGER endpoint described in the ServiceInstanceDescriptor returned from the previous operation and invoking the 'requestFirstInstance' operation with the URI 'TargetExperimentEndpoint' as a parameter. To fulfill this request the Manager begins by retrieving the list of ServiceImplementationDescriptors describing implementations of the TargetExperimentEndpoint Web Service from the ServiceLibrary (TXM1.1/TXM1.2) and the set of all HostDescriptors describing ServiceHosts willing to deploy Web Service implementations written by each implementation's Publisher (TXM4.1/TXM4.2). The Manager makes a decision to deploy the only published implementation on the only willing ServiceHost ('Service Host 2'), binds to the IServiceHost Web Service described in the 'Service Host 2's HostDescriptor, and invokes the 'requestDeployment' operation, using the selected ServiceImplementationDescriptor as a parameter (TXM7.1). Upon successful deployment the ServiceHost returns a URL to the TargetExperimentEndpoint\_MANAGER describing the location of the newly deployed endpoint. The TargetExperimentEndpoint\_Manager includes the URL in a ServiceInstanceDescriptor describing the newly deployed endpoint and returns the ServiceInstanceDescriptor to the POP of the ActiveServiceDirectory (ASD6.2) which returns it to the ActiveServiceDirectory (ASD1.2). The ActiveServiceDirectory first stores the ServiceInstanceDescriptor under the URI 'TargetExperimentEndpoint' before returning it to the POP of the Service Consumer that originally requested the service (SC2.2). To fulfill the original Service Consumer's request its POP binds to the TargetExperimentEndpoint Web Service described in the returned ServiceInstanceDescriptor, invokes the requested 'doNothing' operation (SC3.1), receives the empty return from the

operation (SC3.2) and returns this to the Service Consumer as the result of the requested operation (SC1.2).

#### 5.5.4 ADDING UP THE WORST-CASE DEPLOYMENT COSTS

In the absence of failure, the scenario described above and presented in Fig. 114 on page 198 is the worst-case scenario for deploying a Web Service and it involves 15 intra-architecture calls, 6 of which are to the ActiveServiceDirectory for lookups, 2 are from the ActiveServiceDirectory to a Manager, 2 are performed between a Manager and the HostDirectory, 3 from a Manager to the ServiceLibrary, and 2 are performed between a Manager and a ServiceHost. The total is reduced by one if a custom Manager is used.

Once a Web Service and its requisite Manager are deployed there are also long-term overhead costs associated with maintaining them. The ServiceHost hosting the single deployed TargetExperimentEndpoint endpoint must periodically report endpoint usage data to the newly deployed Manager. The ServiceHost hosting this Manager must also periodically report endpoint usage data to a MasterManager. Using a 5-second report rate, the long-term invocation rate necessary to manage a Web Service with a single deployed endpoint, including its Manager, is 4 invocations every five seconds, or 0.8 invocations per-second. Two of these invocations are to the ActiveServiceDirectory, which adds 0.4 invocations per-second to its total burden for each distinct Web Service being used by Service Consumers – that is, 0.4 invocations per-second for the ‘TargetExperimentEndpoint’ Web Service with a single active endpoint and 0.4 more for the ‘TimeService’ Web Service with a single active endpoint. The next section will detail the incremental costs of deploying and supporting additional endpoints of a Web Service.

	TEX Endpoint	TEX Manager	Total
<b>One-time Communication Cost</b>	7 calls	8 calls	15 calls
<b>Long-term Communication Cost</b>	0.4CPS	0.4CPS	0.8CPS

#### Proportion of Long-term Cost Borne by ActiveServiceDirectory

0.4CPS

## Worst-case Scenario for Deploying the First Endpoint of Web Service with no Deployed Manager

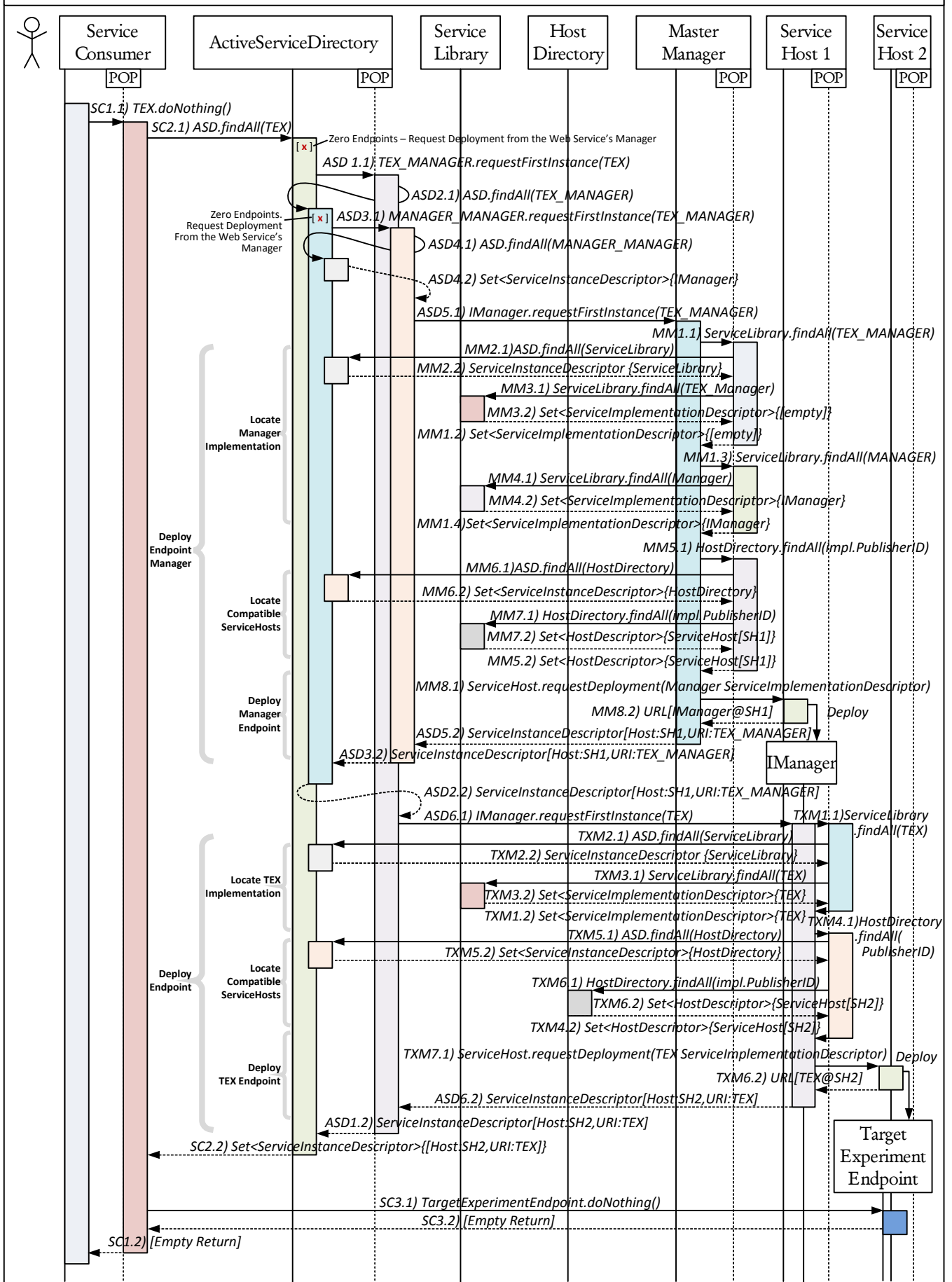


Fig. 114 Object interaction diagram detailing the intra-architecture communication for worst-case first-endpoint deployment scenario.

### 5.5.5 INCREMENTAL COSTS OF DEPLOYING AND MANAGING ADDITIONAL WEB SERVICE ENDPOINTS

Once the first endpoint of a Web Service is deployed its Manager is responsible for evaluating the reported usage data and provisioning additional endpoints as necessary. If a Manager decides to provision an additional endpoint it follows the same deployment procedure as described above, querying the HostDirectory and ServiceLibrary, finding a compatible pair of ServiceImplementationDescriptor and HostDescriptor, and invoking the 'requestDeployment' operation of the IServiceHost Web Service described in the HostDescriptor with the selected ServiceImplementationDescriptor as a parameter. The Manager describes the newly deployed endpoint with a ServiceInstanceDescriptor and invokes the 'addActiveService' operation of the ActiveServiceDirectory Web Service using the ServiceInstanceDescriptor as a parameter. The total number of invocations required for the deployment of each additional Web Service endpoints is 5: two to look up endpoints of the ServiceLibrary and HostDirectory in the ActiveServiceDirectory, one lookup request sent to each of the ServiceLibrary and HostDirectory, and a final request sent to the ActiveServiceDirectory to store the newly deployed endpoint's ServiceInstanceDescriptor. The incremental, one-time burden on the ActiveServiceDirectory for deploying additional Web Service endpoints is thus 3 invocations. The ServiceHost hosting the new endpoint adds an additional long-term burden on the ActiveServiceDirectory every time it reports usage data for the new endpoint, amounting to 1 invocation every five seconds or 0.2 invocations per-second over the lifetime of the endpoint. The burden of the Web Service's Manager also increases by the same rate – 0.2 invocations per-second over the lifetime of each additional endpoint.

Incremental Cost of Additional Endpoints	
One-time Deployment Cost	5 calls
Long-term Communication Cost	0.4CPS

Incremental Contribution to Long-term ActiveServiceDirectory Load
0.2CPS

### 5.5.6 INCREMENTAL BURDEN IMPOSED ON INFRASTRUCTURE SERVICES BY EACH ADDITIONAL SERVICE CONSUMER & POP

Upon startup the Service Consumer's POP is provided with the location of an ActiveServiceDirectory endpoint. Because the failure of this endpoint would result in the Service Consumer being unable to utilize the Web Services published in the infrastructure, the POP begins its initialization process by retrieving the list of all alternative ActiveServiceDirectory endpoints. The POP binds to and invokes the 'findAll' operation of the well-known ActiveServiceDirectory endpoint provided at startup, using the URI 'ActiveServiceDirectory' as a parameter. The POP stores the resultant set of ServiceInstanceDescriptors locally, and then proactively refreshes the list at a regular interval. As we have not established a rate of flux for ActiveServiceDirectory endpoints these calculations will assume that the POP refreshes the list of the ActiveServiceDirectory endpoints once a minute – contributing 1 invocation upon startup and 0.017 invocations per-second in total, per-POP. This is the total baseline cost of the POP staying in touch with the infrastructure and being constantly ready to fulfill Service Consumers requests. Using this refresh rate from an arbitrarily large example of 10,000 idle Service Consumer POPs, the total average load imposed upon the ActiveServiceDirectory would be 187CPS.

Incremental Burden of Each Additional ServiceConsumer POP on ActiveServiceDirectory	
POP Startup	1 Call
Long-term ASD Location Refresh	0.017CPS

Incremental Long-term Burden of Each Additional ServiceConsumer POP on Non-ActiveServiceDirectory Infrastructure Services
OCPS

### 5.5.7 INCREMENTAL BURDEN IMPOSED ON INFRASTRUCTURE SERVICES BY EACH DISTINCT WEB SERVICE USED BY A SERVICE CONSUMER

This section details the burden imposed on the infrastructure by each distinct Web Service used by a Service Consumer (such as the 'TargetExperimentEndpoint' or the 'TimeService' Web Service). As Service Consumers begin to invoke operations of a Web Service by binding and sending invocation requests to their local POPs, lookup requests are sent by the POP to the ActiveServiceDirectory in order to locate active endpoint of the Web Service. The frequency with which the POP looks up the list of active endpoints of the requested Web Service is dictated by the greater of two intervals: the length of time between the Service Consumer invoking operations of the Web Service, or the value of the 'lease duration' parameter provided to the POP

at startup. For example, if the Service Consumer invoked operations of the TargetExperimentEndpoint Web Service once every 10 seconds then the rate of ActiveServiceDirectory lookups would also be once every ten seconds, or 0.1CPS. If, on the other hand, the Service Consumer executed any of the workloads used in the previous experiments (which start at 1CPS, per-Service Consumer) the rate of requests sent to the ActiveServiceDirectory would be dictated by the endpoint lease duration – one call every three seconds, or 0.3CPS, per-Service Consumer, for the duration of the experiment.

For the purpose of this analysis we will assume a high invocation rate and will use the 3-second lease duration used during the experiments. Invoking operations of a Web Service at a rate higher than 1 invocation every three-seconds from a single Service Consumer machine results in the POP imposing an additional burden of 1 invocation request every 3 seconds on the ActiveServiceDirectory, or a long-term burden of 0.3 invocations per-second, per-Web Service used, per-POP.

---

**Incremental Load Imposed on ActiveServiceDirectory by a Service Consumer POP for Each Additional Web Service Being Actively Used by the Service Consumer**

---

0.3CPS

---

### 5.5.8 BURDEN OF INCREASING WEB SERVICE DEMAND RATE ON INFRASTRUCTURE SERVICES

In the presence of a rising rate of demand from a fixed number of Service Consumers, to a Web Service with a fixed number of endpoints, the burden on the infrastructure remains constant. As detailed above, if the Manager of the Web Service concludes that the provisioning level needs to be increased, the fixed cost for deploying an additional endpoint of the Web Service is 5 invocations, 3 of which are to the ActiveServiceDirectory. The long-term burden to the ActiveServiceDirectory increases by 0.2 invocations per-second for each additional active endpoint of a Web Service. In the absence of additional provisioning events, however, changes in the rate of Service Consumer demand for Web Services does not impose any additional costs on the infrastructure (see section 5.5.10 ‘Growth Rates & Scalability’ for example infrastructure costs in scenarios involving very high demand).

---

**Additional Cost to Maintain Infrastructure in the Presence of Rising Demand**

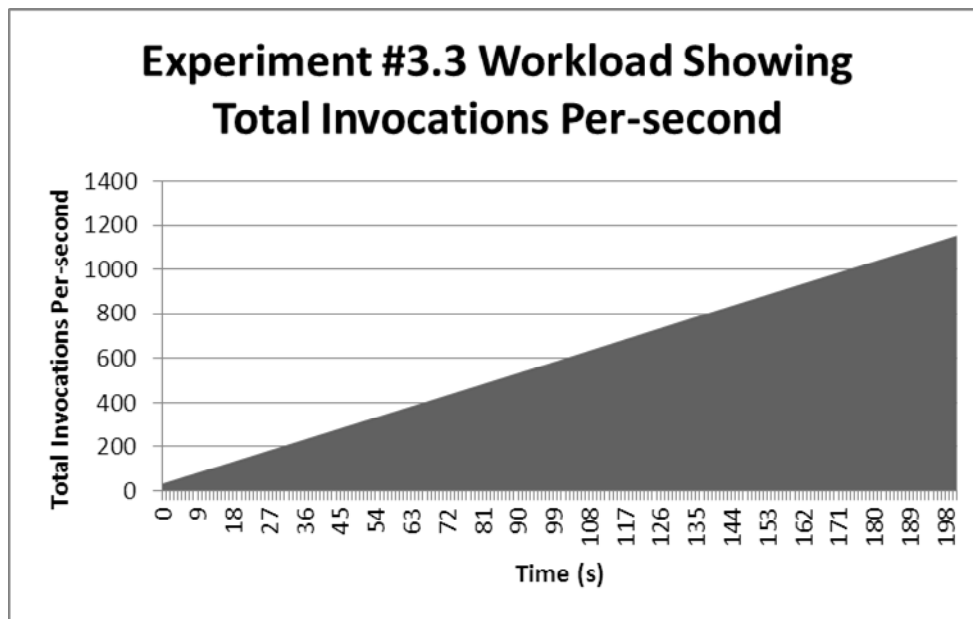
---

0CPS

---

### 5.5.9 USE-CASE: TOTAL COST OF PROVIDING INFRASTRUCTURE FOR EXPERIMENT 3.3

Using the various costs identified in the previous section, this section details the cost of providing the infrastructure services for the workload imposed in experiment 3.3. This experiment involved 32 Service Consumers invoking operations of the TargetExperimentEndpoint Web Service for 200 seconds, beginning at 35CPS and increasing linearly over the course of the experiment to 1150CPS. The TargetExperimentEndpoint Web Service was managed by a custom Manager that made deployment decisions based upon the reported invocation rate, resulting in 6 provisioning events over the course of the experiment. The workload executed in experiment 3.3 is provided below in Fig. 115 for ease of reference.



**Fig. 115 Workload executed by 32 Service Consumer nodes during experiment 3.3, indicating the total number of invocations per-second executed during each one-second interval of the 200-second experiment.**

The graph in Fig. 116 on page 204 details the total intra-infrastructure costs associated with supporting the described workload. The workload is represented by the diagonal black line and is rendered on the secondary Y-axis to the right as it rises at a rate of 5.6CPS, per-second. The remainder of the lines represent the intra-architecture communication required to service the specified load, first individually using dotted lines (such as 'MasterManager Overhead [CPS]') then together with solid lines (such as 'Total Infrastructure Maintenance Cost  $[(ASD/SL/HD + Managers) + [MasterManager Overhead)]$  [CPS]') as they approach the top solid purple line representing the combined total of all intra-infrastructure communication required to support the level of demand ('Load').

The dark purple dotted line at the very bottom of the chart represents the direct overhead cost of providing the MasterManager (a constant 0.4CPS). This is followed by the orange dotted line indicating the total load imposed on the ActiveServiceDirectory by all operational POPs refreshing their lists of ActiveServiceDirectory endpoints once every minute – for 52 POPs the demand rate is a constant 0.884CPS. The cost of managing the three infrastructure-provided services (ASD/SL/HD) and their Managers is represented by the green dotted line. The dark purple and green dotted lines are combined into a single solid salmon line which represents the baseline infrastructure overhead costs (excluding ASD lookups). The light purple solid line represents the costs associated with managing every endpoint of the TargetExperimentEndpoint Web Service (and its Manager endpoint), including the one-time costs each time a new endpoint is deployed. The ActiveServiceDirectory lookups associated with all of the infrastructure activities described so far are represented collectively with the light blue line labeled “ActiveServiceDirectory Load from All Infrastructure & TargetExperimentEndpoint Provisioning & Management Activities (CPS)”. These ActiveServiceDirectory costs are combined with the total infrastructure maintenance costs (salmon) and the cost of provisioning and managing the TargetExperimentEndpoint Web Service endpoints (and Manager) into a solid orange line representing the total infrastructure-related costs associated with supporting the current demand.

The red line in the center of the graph represents the total load imposed on the ActiveServiceDirectory by Service Consumers. While this is the single largest cost contributor, it is the result of a large number of Service Consumers invoking operations of a Web Service with dynamic endpoint locations and using a very aggressive endpoint lease duration of 3-seconds. The cost contribution is only 10.6CPS for 32 Service Consumers and, most importantly, it stays at 10.6CPS over the duration of the experiment even though the Service Consumers are invoking operations at an increasingly high rate. The combined total load imposed upon the ActiveServiceDirectory by the Service Consumers and all infrastructure-related activities is represented by the dark blue solid line labeled “Total ActiveServiceDirectory Load (CPS)”. Finally, the sum-total of all of the intra-infrastructure invocations is represented in the top-most solid purple line labeled “Total Cost of All Intra-infrastructure Communication (CPS)”. This value includes every invocation request sent between actors in the described architecture throughout the experiment.

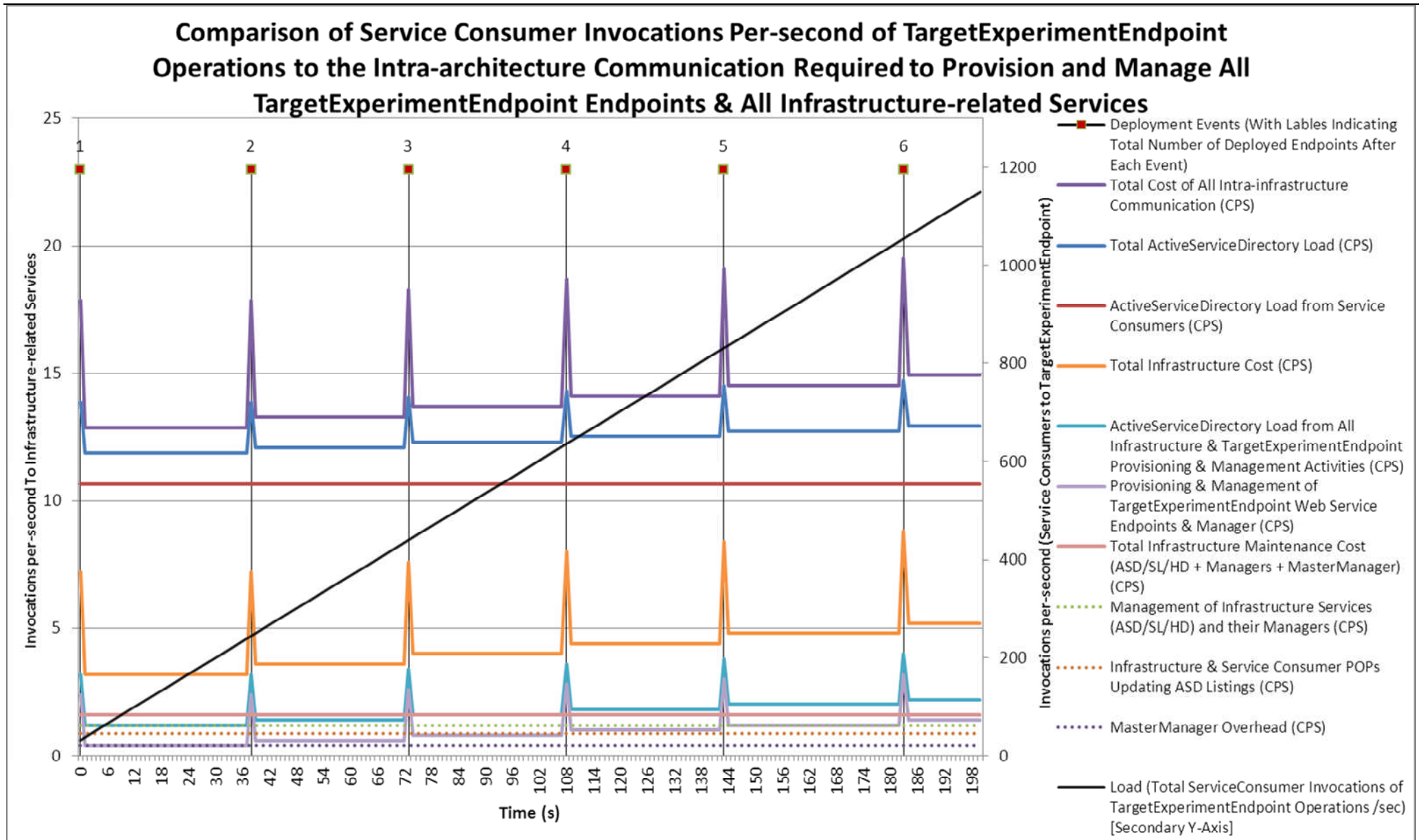


Fig. 116 Comparison of Service Consumer invocation rate (secondary y-axis) to the number of operations required by the infrastructure to support this behaviour, indicating deployment events.

### 5.5.10 GROWTH RATES & SCALABILITY

The growth rates of each of the cost groups represented by solid lines in the results of the previous experiment (Fig. 116) are detailed in Fig. 117 below together with the growth rate of demand ('Load'). These growth rates are used in the upcoming discussion in order to calculate the infrastructure-related costs of supporting workloads that grow at the same rate as the previous experiment, but for longer periods of time. The purpose of this comparison is to demonstrate the relationship between demand for Web Service and the cost of servicing that demand (i.e. providing and managing both the infrastructure and the required number of TargetExperimentEndpoint Web Service endpoints). From this analysis we should get a better idea of the long-term overhead costs of providing Web Services in the described architecture, and should also be able to identify the primary cost-driving activities. In the final example in this chapter we will identify the maximum Service Consumer load that the infrastructure can support before its overhead costs increase. The results will be presented graphically and will be used to draw conclusions about the scalability of the architecture.

	<b>Growth Rate (CPS per-second)</b>
<b>Load</b>	5.5900
<b>Total Intra-infrastructure Communication</b>	0.0106
<b>Total ActiveServiceDirectory Load</b>	0.0053
<b>ActiveServiceDirectory Load from All Provisioning &amp; Managing</b>	0.0053
<b>Provisioning &amp; Managing TargetExperimentEndpoint (+Manager)</b>	0.0053
<b>ActiveServiceDirectory Load from Service Consumers</b>	0
<b>Total Infrastructure Maintenance Cost</b>	0

Fig. 117 Table of invocation-rate growth rates for each major cost group shown in Fig. 116

The following graphs are generated from the growth rates specified above and are representative of the costs of supporting a workload that grows at the same rate as the previous experiment (5.59CPS, per-second) but for a period of 5,000 seconds. Because the costs associated with supporting this significantly greater workload are generated using the growth rates in Fig. 117 above, the results will be representative of the workload being executed from 32 Service Consumer machines. Recall from the analysis in section 5.5.7 that the additional long-term burden of each additional Service Consumer invoking operations of a single Web Service at a rate greater than or equal to the endpoint lease duration (i.e. once every three seconds) results in a fixed overhead cost of 0.3CPS per-Service Consumer and that this cost does not change as demand increases. This finding was demonstrated in the results of the previous experiment by the red line in Fig. 116 labeled "ActiveServiceDirectory Load from Service Consumers (CPS)". In the

presented architecture, the number of Service Consumers executing a workload only affects the fixed, long-term overhead burden imposed on the ActiveServiceDirectory.

The long-term infrastructure-related costs of supporting a steadily increasing rate of Service Consumer demand for the TargetExperimentEndpoint Web Service over a period of 5,000 seconds are presented in Fig. 118 on page 207 using a logarithmic scale. Represented by the solid blue line at the top of the graph, the total Service Consumer demand rate climbs at a rate of 5.59CPS per-second over the duration of the experiment to a total of 27,976.41CPS. As the demand rate rises so too does the total number of endpoints of the TargetExperimentEndpoint Web Service. As a new endpoint of the service is deployed when the demand rate reaches an average of 200CPS per-endpoint a total of 139 endpoints of the TargetExperimentEndpoint Web Service are required by the end of the 5,000 seconds. Each of these endpoints places an additional burden of 0.2CPS on the ActiveServiceDirectory, and a further 0.2CPS on the TargetExperimentEndpoint Manager.

The burden placed upon the ActiveServiceDirectory by all 139 endpoint at the end of the experiment comes to a combined total 26.8CPS and is represented by the purple line. This cost is combined with the burden imposed by the Service Consumers (a constant 10.6CPS) and the cost of managing the infrastructure services (1.6CPS of the 3.2CPS baseline total) to reach a combined total load imposed upon the ActiveServiceDirectory at the end of the experiment of 38.333CPS, as represented by the green line. Finally, the sum total of the intra-infrastructure communication is represented by the red line in Fig. 118. The intra-infrastructure communication required to support 32 Service Consumers invoking operations of the TargetExperimentEndpoint Web Service at a steadily increasing rate over a period of 5,000 seconds, with a maximum final invocation rate of 27,976.41CPS comes to a total of 69.046CPS, or 0.00246CPS per-Service Consumer request.

### Infrastructure-related Costs of Supporting an Increasing Rate of Service Consumer Demand for the TargetExperimentEndpoint Web Service, Rising from 32CPS to 27,976CPS for 5,000 Seconds, Rendered on a Logarithmic Scale

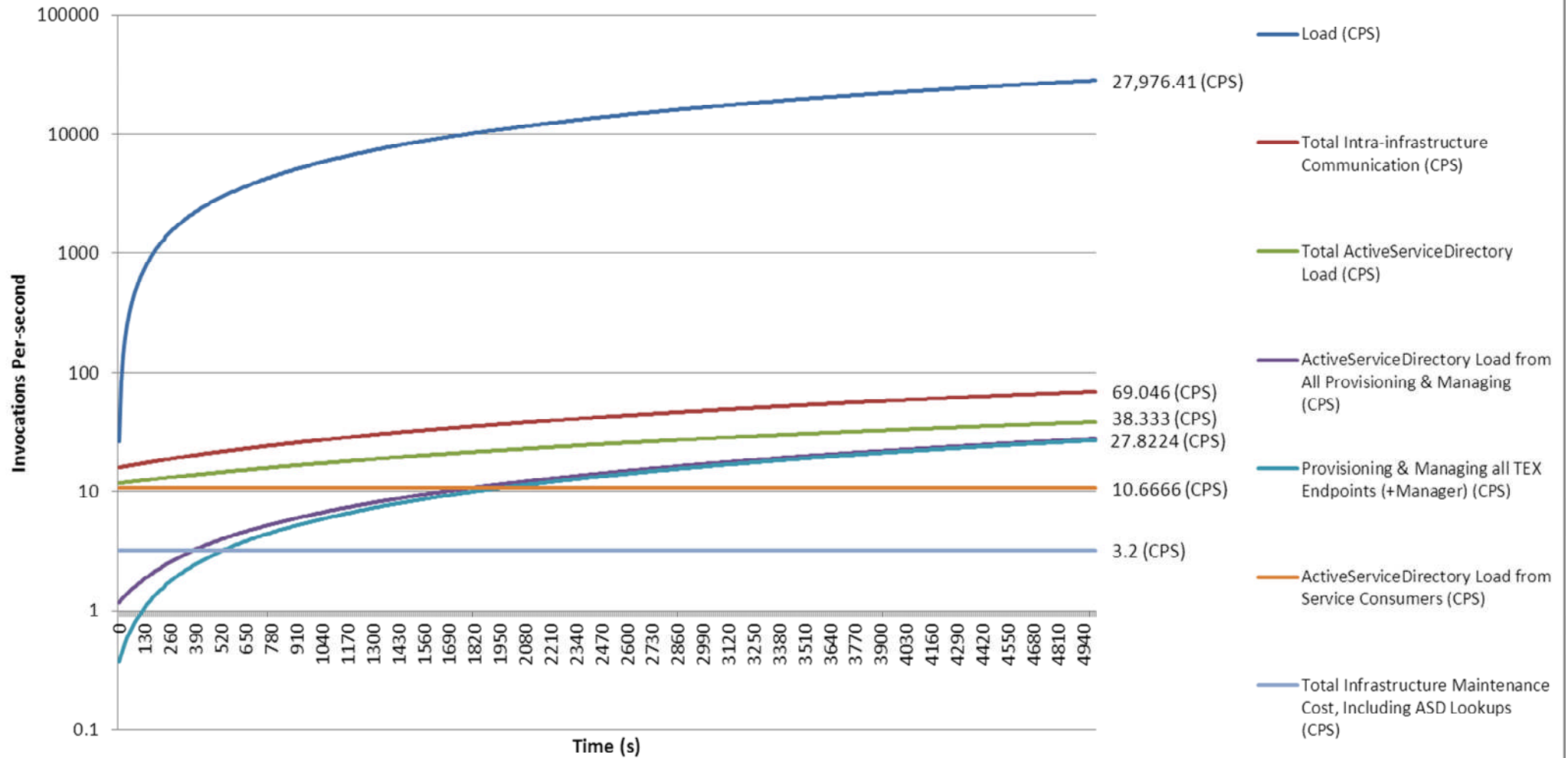


Fig. 118 Infrastructure-related Costs of Supporting and Increasing Level of Demand for the TargetExperimentEndpoint Web Service, Rising from 32CPS to 27.976CPS over 5,000 Seconds. Logarithmic Scale.

### 5.5.11 CONCLUSIONS ON THE COST & SCALABILITY OF THE INFRASTRUCTURE

In the preceding sections we detailed the costs associated with providing and using Web Services in the described architecture and explored how these costs grow (or don't) over time. In order to draw a conclusion on whether these costs provide value – whether they are 'worth it' – to the providers and consumers of Web Services we must compare these costs to the various benefits provided. In other words, we need to define the value that providers and consumers of Web Services get for their overhead resource expenditure of 7 deployed endpoints and 3.2 intra-architecture calls per-second.

When Service Consumers invoke operations of a Web Service their collective load is automatically distributed amongst a dynamic number of endpoints – if these endpoints fail the Service Consumers' POP automatically locates another. If there are no other endpoints available then, if resources permit, one will be deployed automatically. If the rate of demand rises too high for a single endpoint to support then an additional endpoint will be provided, automatically, and the burden shared amongst them. If a Web Service does not receive any invocation request for a period of time its active endpoints will be progressively undeployed so that the hosting resources they were consuming may be put to use hosting services that are actually being used. The Managers of these services are all just services themselves, so their usage data is also reported to a [Master]Manager. When the last endpoint of a consumer-facing Web Service has been undeployed, its Manager will no longer receive usage data reports from ServiceHosts – and thus the ServiceHost hosting the Manager will record no invocation requests. This ServiceHost reports the Manager's usage data to the MasterManager and the MasterManager will use it to make provisioning decisions just like any other Manager. Using its implemented provisioning policy the MasterManager can decide whether the Manager is no longer needed and can be undeployed. This means that while every Web Service published into the infrastructure can be considered 'available' as long as there are ServiceHosts capable of hosting their implementations, there are no direct overhead resource consumption costs associated with providing a Web Service when there is no demand for that Web Service. The long-term indirect resource consumption costs of providing a Web Service can be calculated by dividing the baseline overhead costs of providing the infrastructure services (7 endpoints, 3.2CPS) by the total number of distinct Web Services published in the Service Library. For one service, this is expensive; for 100 services it is less so – for 500 services the long-term indirect cost of providing a Web Service is 0.014 endpoints and 0.0064CPS, per-Web Service.

Using the worst-case scenario analysis, the long-term overhead cost associated with providing a single Web Service in the presented architecture is 7 endpoints and an intra-architecture communication rate of 3.2CPS. The cost of provisioning the first endpoint of that Web Service and its Manager results in two further endpoints and a fixed cost of either 14 or 15 intra-architecture calls. The long-term cost associated with providing an active endpoint of the Web Service adds an additional 0.4CPS to the total intra-architecture communication rate (assuming a 5-second report period). If a Service Consumer uses the Web Service at a rate equal to or greater than the endpoint lease duration (3 seconds) then the additional cost of providing the Web Service using the described architecture versus a manual approach is 8 endpoints and 4CPS. Using a 200CPS invocation-count Manager for both the consumer-facing Web Service and the infrastructure services, these overhead costs remain constant up to a total Service Consumer demand rate of 200,000CPS. Once the demand increases beyond this level, an additional endpoint of the ActiveServiceDirectory is required, which increases the long-term costs of providing the infrastructure by 0.2CPS and increases the incremental, one-time cost of deploying any Web Service endpoint by one call as Managers must register any newly deployed endpoint with both ActiveServiceDirectory endpoints.

If the provisioning level of a consumer-facing Web Service reaches 1,000 deployed endpoints, then an additional Manager must also be deployed for that service. The described Manager implementation does not accomodate this situation as it was not possible to create the circumstances that would trigger the deployment of tandem Managers using the available test infrastructure. However, any Manager implementation that could realistically be required to manage a Web Service with over 1,000 endpoints would need to coordinate with any additional Manager endpoint so as to not duplicate their provisioning decisions. This situation also applies to the MasterManagers once there are 2,000 distinct consumer-facing Web Services actively receiving demand.

Only in the situations described above do the overhead costs of providing Web Services in the described architecture increase marginally. With 1,999 distinct consumer-facing Web Services actively receiving demand, the total additional cost of providing each Web Service in the described architecture is 0.0035 endpoints and an intra-architecture communication rate of 0.0016CPS. With 2,001 distinct consumer facing Web Services these costs increase to 0.0039 endpoints and 0.0018CPS. For these expenses the Web Service is automatically deployed, when needed, on to as many ServiceHosts as necessary to meet an automatically measured and evaluated level of demand. Consumer demand for the Web Service is automatically distributed

amongst all available endpoints of the Web Service and invocation failures are automatically redirected to alternatives. Developers of consumer agent applications are relieved from the burden of detecting and recovering from the failure of endpoints, and of even locating them in the first place (or second place, etc.). And once a ServiceHost machine is configured, described and registered in the HostDirectory they have no further administration-related responsibilities, as all responsibility for deployment, monitoring, management and undeployment are all fully consumed by the architecture.

The long-term human capital costs associated with providing Web Services using the described architecture are notably absent from the previous discussions not because they are not unimportant, but because no such costs exist. ServiceHosts must be configured, described and registered, and Web Services must be developed, described and published. The only long-term costs associated with providing a Web Service after these responsibilities are fulfilled are hardware- and network-related: the overhead endpoint and intra-infrastructure communication costs discussed above. There is thus no long-term human capital expenditure associated with providing a Web Service in an implementation of the described architecture: no deploying, monitoring, redeploying, reprovisioning, deprovisioning, detecting or recovering from failure, locating migrated endpoints or recoding consumer applications. In light of the human capital saved by providers of Web Services, and the convenience and robustness provided to developers and consumers of applications which use Web Services, the additional qualitative benefits of the architecture are argued to heavily outweigh even the worst-case cost of using the architecture to provide a single endpoint of a single Web Service for a single Service Consumer. If the host- and network-resource costs in this scenario can be considered to provide high value in terms of the human capital saved, then the same costs can be judged to provide incrementally higher value each time the Service Consumer demand increases or an additional Web Service is published into the infrastructure. Before moving on to the final chapter 'Conclusions & Future Work', one last scenario is presented below as a demonstration of the infrastructure costs associated with supporting an extremely high level of demand.

The duration of the workload in Fig. 119 on page 212 was selected in order to identify the maximum possible rate of Service Consumer demand that the infrastructure could support before its costs increased due to the deployment of an additional endpoint of the ActiveServiceDirectory Web Service. The graph in Fig. 119 uses the same growth figures as that in Fig. 118 on page 207 and thus its cost calculations are based on the same 5-second report periods, 3-second leases, and a 200CPS call-count Manager. The labels to the right of the graph indicate the final value of

each identified cost group. As in the previous experiment the baseline overhead infrastructure maintenance cost remains constant, as does the demand imposed by Service Consumers on the ActiveServiceDirectory. The cost of reporting usage data and managing the deployed endpoints is shown as 189.6CPS, the load on the ActiveServiceDirectory just reached 201.45CPS, and the total rate of intra-architecture communication is 395.3CPS. Although the overhead costs of providing the infrastructure must now rise marginally, they remained constant for the preceeding 9.93 hours of increasing demand, with a final demand rate of 200,025.43CPS. In light of the human-capital savings and qualitative benefits provided in return, a cost of 0.000035 endpoints and 0.001991CPS per-Service Consumer invocation request is argued to provide a high return on investment to both the providers and consumers of Web Services in the presented architecture.

### Infrastructure-related Costs of Supporting an Increasing Rate of Service Consumer Demand for the TargetExperimentEndpoint Web Service, Rising from 32CPS to 200,000CPS for 35,778 Seconds, Rendered on a Logarithmic Scale

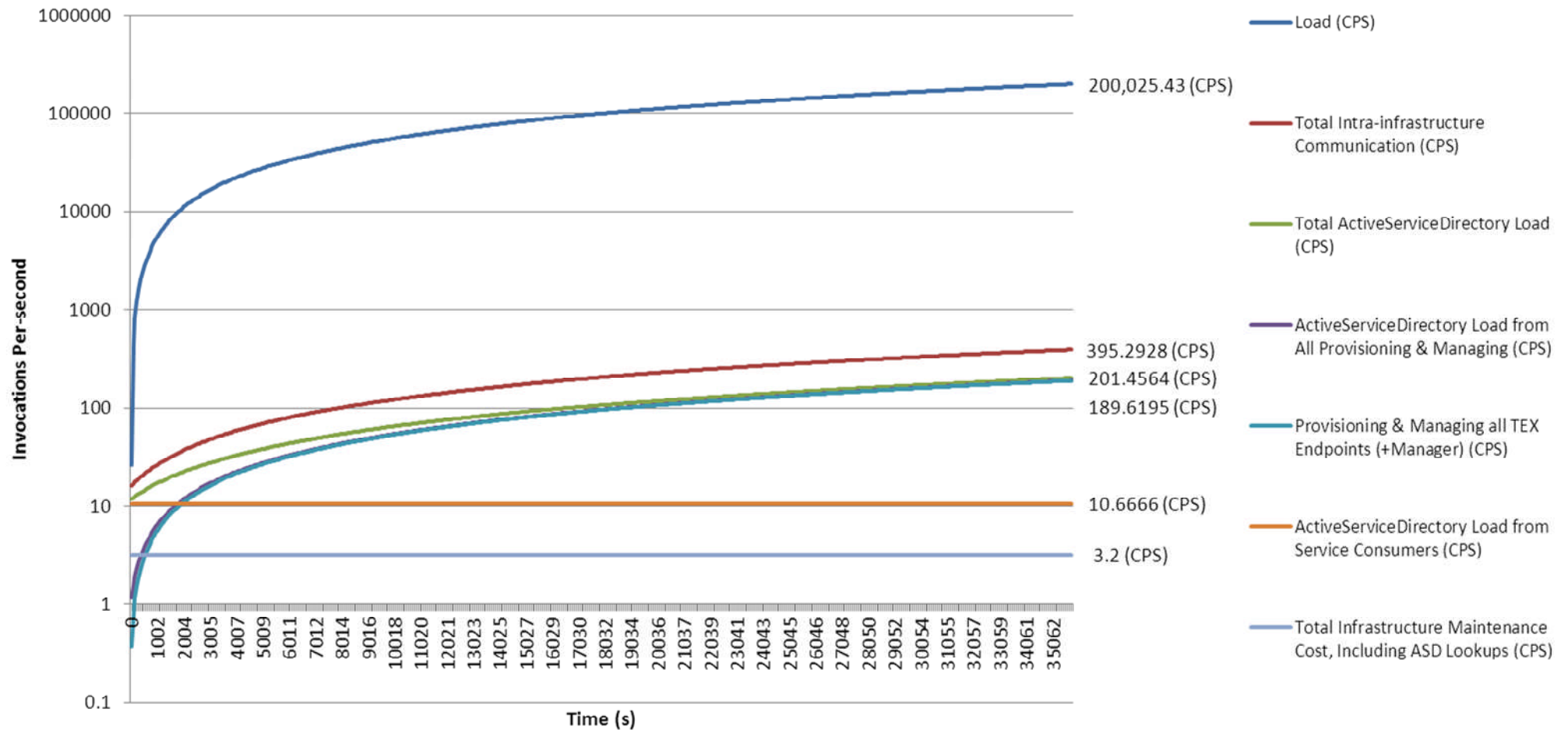


Fig. 119 Infrastructure-related costs associated with supporting the maximum rate of invocation for a consumer-facing Web Service from 32 Service Consumers before the baseline overhead infrastructure resource consumption rises for the first time. Logarithmic scale.

## 6 CONCLUSIONS & FUTURE WORK

### 6.1 INTRODUCTION

This thesis presents a scalable architectural model for the demand-driven deployment of location-neutral software services. This final chapter recalls the original motivation, for the work and the thesis hypothesis, before summarizing the conclusions and evaluating the contributions of the work. It concludes with a section dedicated to future work which identifies some of the many areas of the presented architecture that are ripe for further investigation and discusses how the architecture might be utilized in different real-world contexts.

### 6.2 THESIS MOTIVATION

#### 6.2.1 *CENTRAL LIMITATIONS OF THE WEB SERVICES MODEL*

The thesis was motivated by the notion that the Web Services model is well suited to the emergence of Internet-wide service-oriented computing, and that its promise is undermined by two central limitations: a rigid addressing model, and an over-burdened and under-specified Service Provider role. These limitations are shown to have various negative implications for the developers, providers, and users of Web Services.

The rigid addressing model does not take into account the intrinsic dynamism and fallibility of hosts on the Internet, and consumer agent applications which aim to be robust to the failure of hosts become littered with failure-recovery code. For developers of consumer agent applications, programming against fixed endpoints results in tightly-coupled applications which are fragile with respect to changes in the location and availability of Web Services and are susceptible to the failure of a single host or service. If a Web Service fails or is migrated, users must either re-write code to reflect an updated endpoint, or include failure recovery mechanisms in each implementation of a client which uses a Web Service. The resultant applications are permeated by exception-handling code and tightly-coupled to both the location and the mechanisms used to locate Web Services.

If providers of Web Services aim to provide a reliable or consistently available Web Service, the address of the endpoint must remain fixed, forcing an early decision on the location, amount of bandwidth, and amount of computational capacity required in the hosting infrastructure. Not only is it difficult to predict demand levels before a service is deployed, but fixed-resource deployments are not scalable to demand and waste useful resources by statically provisioning them before deployment – either over- or under-shooting the necessary capacity. Web Service

providers may host their services in dynamic deployment environments, which only further forces developers of consumer agent applications to incorporate run-time dynamic binding techniques which are specific to the host environment – cementing the already tight coupling in consumer agent applications by tying them to not only the location of the Web Service, but also the location and implementation of the endpoint directory.

In the traditional Web Services model, the Service Provider is responsible for all of the tasks of developing, deploying, hosting and managing a Web Service. The result of this broad set of responsibilities is the proliferation of proprietary deployment systems and hosting environments composed of services which cannot be deployed or managed by other systems and are only interoperable with other identical environments [7]. While the Web Services themselves are platform-independent, the techniques used to deploy and manage them typically rely on particular platform-specific attributes or tools. Platform-dependence forces developers to develop for specific target environments – a closed-world approach which results in islands of incompatible services and collections of service implementations which are not portable. The resultant technology lock-in is contrary to the central ethos of platform-independence which is otherwise a hallmark of the Web Services model.

### 6.2.2 *THESIS HYPOTHESIS*

The hypothesis tested in this thesis is predicated upon three core arguments: 1.) that the rigid addressing model forces the entanglement of application business logic with remote-invocation failure recovery techniques, is detrimental to the usability and reliability of applications, adds significant time and complexity to the creation of client applications, and is a serious drawback to the adoption of the existing Web Services model, 2.) that the over-burdened and under-specified Service Provider role has fostered the development of progressively insular, closed-world environments composed of Web Services and client applications for which the use of Web Services is only incidental, and 3.) that the goals of universal interoperability, loose-coupling and platform-independence espoused by Web Services and service-oriented computing are worthwhile goals, that they are currently undermined by the traditional Web Services model, and that a new, next-generation model can be developed which better facilitates the realization of these goals.

From these premises the following hypothesis was formulated (as quoted from Chapter 1):

*“The hypothesis of this thesis is that isolating the tasks and separating the concerns of participants in a distributed Web Services framework facilitates abstractions over location and technology which enable the provision of pervasive Web Services which can be reliably addressed, bound to, and utilized, at any time and from any location.*

*Further, it is proposed that mechanisms can be developed to provide reliability and reduce complexity for users of Web Services, and that these mechanisms can be equally applicable to static, homogeneous computing environments as to dynamic environments composed of heterogeneous Web Service implementations and Web Service-hosting technologies.”*

The following sections detail the process by which this hypothesis was tested, whether and how it was validated, the core contributions of the work, and finally areas for further research.

### 6.3 THESIS SUMMARY

This thesis began by outlining the motivation for the work, identifying the central limitations of the Web Services model, and proposing the hypothesis to be tested. It continued in Chapter 2 with an introduction to the software service lifecycle and a detailed overview of the actors and standards of the Web Services model. A comprehensive survey of work addressing individual shortcomings of the traditional Web Services model was presented, followed by a body of work which aims to address multiple shortcomings through broader, higher-level modifications to the model. These approaches were compared with a set of requirements for a next-generation Web Service architecture at the beginning of Chapter 3. Having identified thoughtful and promising approaches to addressing multiple shortcomings with well-designed architectural models, but finding none to meet the full set of requirements proposed at the beginning of Chapter 3, a new model was presented which does achieve the newly defined requirements. Next, the roles and responsibilities of each actor in the new architecture were defined, followed by their interaction and failure models. A reference implementation was presented in Chapter 4 which leveraged the new model in order to exemplify the beneficial properties it enables. Using this reference implementation the presented architectural model was evaluated in Chapter 5 through a series of experiments. The results of these experiments (described in the upcoming section 6.3.1 ‘Conclusions from Experimental Evaluation’) demonstrated the ability to simultaneously provide simplicity, flexibility, and robustness in a distributed Web Services framework composed of heterogeneous service implementations and service-hosting technologies. The mechanisms developed to dynamically provision Web Services according to demand were shown to enable

highly available Web Services. The model is shown to be resilient to the failure of both the nodes which host Web Services and the architectural entities which provision and manage Web Services. In the final experiment the model is shown to be scalable, and capable of effectively and efficiently managing a dynamic pool of latent hosting resources in order to meet changing levels of demand. Based on the identified qualitative benefits received as a result, the overhead resource consumption costs required to provide an active infrastructure were deemed to provide high value to both the providers and consumers of Web Services.

### *6.3.1 CONCLUSIONS FROM EXPERIMENTAL EVALUATION*

The experimental evaluation of the architecture verified a number of assumptions about its functionality and suitability to purpose and has provided insight into the relationships between the various actors. The first experiment tested whether, in the presence of an actively-managed endpoint directory (ActiveServiceDirectory) and client-side endpoint-selection mechanism (POP), increasing the number of available endpoints of a Web Service is an effective means of reducing the average response time for invocations of its operations. This hypothesis was validated and it was conclusively demonstrated that deploying an additional endpoint of a Web Service and listing it in the ActiveServiceDirectory is an effective means of reducing the average invocation response-time experienced by Service Consumers. Further, it shows that the mechanisms are effective independent of the technology used to host the endpoint. This experiment shows that by binding to and invoking operations via the local point of presence, the Service Consumer application did not need to be modified during testing, nor did the POP, even though the endpoints were deployed on multiple different machines and using multiple different technologies. Finally, the experiment verifies that the POP mechanism was indeed capable of transparently and dynamically distributing load amongst multiple available endpoints of a desired service.

Building on the success of the first experiment, its corollary was tested next: whether removing an endpoint of a Web Service is an effective means of increasing the average invocation response-time experienced by Service Consumers invoking its operations, and whether each Service Consumer POP can effectively re-direct its load to the remaining endpoints. This was an important experiment because undeploying an endpoint of a Web Service and removing its entry from the ActiveServiceDirectory is the sole means by which Managers reclaim hosting resources from over-provisioned (or under-demanded) services. By servicing identical periods of high consumer demand with progressively fewer endpoints it was shown that fewer deployed endpoints of a Web Service leads to an increase in the average invocation response-time

experienced by Service Consumers, and demonstrated that the mechanisms of the POP are effective at redirecting invocation requests to alternative endpoints of a Web Service in order to redistribute load.

Two important conclusions were drawn from the first two experiments. The first was that within the described architecture, changing the provisioning level of a Web Service is an effective tool for managing the average invocation response-time experienced by Service Consumers. This was important since it enabled the next experiments to verify whether endowing the Manager actor with the responsibility and capability of managing the provisioning level of a Web Service would enable the Manager to effectively control the average invocation response-time experienced by Service Consumers. The second conclusion drawn from the first two experiments was that the Service Consumer POPs, together with the infrastructure-provided services, are capable of automatically and transparently distributing invocation requests amongst a dynamic number of Web Service endpoints without requiring any action on the part of the Service Consumer or Service Provider. This verified a claim of the architecture and provided a solid foundation from which the testing of the capabilities and limitations of various Manager implementations could proceed.

The third set of experiments tested whether different Manager implementations were able to dynamically manage the availability of a Web Service under various workloads by analyzing the usage data returned from ServiceHosts and altering the provisioning level to meet the measured level of demand. Managers were tested which make provisioning decisions based upon two distinct metrics: response-time as measured at the ServiceHosts, and the load experienced by ServiceHosts as measured by the total rate of invocation requests received for a Web Service. By executing various workloads and recording the invocation events and the average invocation response-time experienced by the Service Consumers it was shown that, using either metric, Managers were capable of identifying and dynamically responding to changes in the level of Service Consumer demand by increasing or decreasing the number of available endpoints of the Web Service. Three different management policies were demonstrated for the invocation-count Manager and their differences and relative effectiveness discussed and compared to the response-time Manager. Through this investigation the hypothesis of the experiment was validated: that Managers are capable of responding to reported levels of demand, that they have the tools to act accordingly, can use those tools, and that their actions have a measurable effect on the average invocation response-time experienced by Service Consumers invoking operations of the Web Service being managed.

Independent of the implementation-specific evaluation process employed by the specific Managers, this third set of experiments demonstrated that both the architecture, and automated management of the multi-endpoint framework is well-suited to the dynamic provision of managed services in a distributing computing environment with varying levels of demand. Further, the appropriateness of the Manager's responses to measured changes in demand showed that the data returned by ServiceHosts is rich enough to provide Managers with insight into the state of the service under management. The Managers' actions were shown to materially resolve imbalances between resource-allocation and demand: excessive increases in response-time were met by the deployment of additional capacity, while the resources of over-provisioned services were reclaimed so that they could be applied elsewhere as necessary.

The evaluation section concluded with an analysis of the overhead costs associated with providing Web Services in the presented architecture. The total overhead cost of a running infrastructure with zero demand was evaluated first, followed by a scenario demonstrating the worst-case cost of deploying the first endpoint of a Web Service. The one-time and long-term costs associated with providing each additional endpoint of a Web Service were presented, followed by the additional burden imposed upon the infrastructure by each additional Service Consumer POP and each distinct Web Service used by a Service Consumer. The preceding experiment 3.3 was used in order to demonstrate a real-world scenario from which to evaluate how these costs grew over time. In the final analysis the growth rates from this evaluation were used to produce projections of the long-term costs associated with running the same workload as experiment 3.3 but for significantly longer periods of time. The final results indicated that the architecture was indeed highly scalable, and demonstrated that the baseline overhead resource provisioning level required to provide the infrastructure services does not increase until demand reaches a very high level.

## 6.4 THESIS CONTRIBUTIONS

This thesis makes a 5-fold contribution:

- 1.) The current state of Web Service authoring, deployment, management and use is described and factors contributing to redundant human effort, high cognitive complexity, and inefficient resource utilization are identified in each of these areas. From these, requirements are specified for an architectural model which mitigates the identified limitations.
- 2.) The design of a next-generation architectural model for Web Services is presented which meets the defined requirements. The model lowers the barriers to participation in a distributed Web Services framework by reducing the cognitive complexity, effort, and overlapping domain-specific knowledge required of developers, hosts, application programmers and users of Web Services.
- 3.) A reference implementation of the next-generation architecture is described which achieves high levels of fault-tolerance and scalability and can dynamically adapt to changes in the levels of demand and available resources. Mechanisms are developed for dynamically provisioning Web Service endpoints, and are shown to be equally applicable to static, homogeneous computing environments as to dynamic environments composed of heterogeneous Web Service implementations and Web Service-hosting technologies.
- 4.) An extensible framework for the scalable and resource-efficient management of Web Services is described. This framework enables the application of both generic and customisable management techniques to the maintenance of high availability and predictable performance.
- 5.) A mechanism for transparently abstracting over the location of Web Service endpoints and dynamically detecting and recovering from endpoint failure is described. The mechanism disentangles the orthogonal tasks of implementing application business logic and writing failure-recovery routines, factoring out redundant endpoint-recovery code from client applications and thus lowering the likelihood of application failure. It provides a reverse-compatible solution to the rigid Web Service addressing model which, crucially, requires no changes to any of the Web Service standards and maintains the existing actors' views on the architecture.

## 6.5 FUTURE WORK

The architecture presented in this thesis is implemented as a proof of concept. In evaluating the system, one point of performance degradation and two opportunities for resource efficiency were identified which would benefit from further investigation if one were to implement the presented architecture as a production-grade system. Descriptions of these optimizations are

presented below followed by a section describing other areas of the presented architecture which are ripe for further exploration.

### *6.5.1 OPTIMIZATIONS TO EXISTING WORK*

#### 6.5.1.1 Re-implement Local Point of Presence

The local point of presence implementation presented in Chapter 4 and used for the evaluation in Chapter 5 was implemented very conservatively and its performance suffered as a consequence. Approaches to proxying XML documents that are less computationally expensive than a full DOM parser with schema validation should be explored in order to improve the performance of this highly critical architectural entity. A 2007 XML parser benchmark study [104] ranks both native and Java XML parsers based on their maximum throughput of variously sized documents. The top-performing SAX-LIBXML2 parser [105] is implemented in C and is shown to offer nearly twice the throughput of WoodStox [106], the top-performing Java parser. Any performance gains realized in the POP through adoption of these or other technologies would not only benefit Service Consumers, but would also have a direct positive impact on the overall performance of the entire infrastructure.

#### 6.5.1.2 Reduce Infrastructure Footprint in Zero-demand State

While the number of endpoints and the quantity of intra-architecture communication required to sustain an infrastructure with no actively deployed consumer-facing Web Services is shown to be low, approaches could be implemented to reduce it further. One such optimization has been explored which requires only one endpoint of the ActiveServiceDirectory to remain deployed during periods of zero demand by bestowing upon it the authority (and capability) to deploy the ServiceLibrary, HostDirectory and MasterManager itself in the event that they are not currently deployed. This is merely a convenience as the remainder of the components need to be deployed before the infrastructure can function; it does, however, enable resource consumption in network of machines to be ‘wound down’ to nearly zero, with the infrastructure laying dormant but ready to ‘spin up’ into action the moment the ActiveServiceDirectory receives a lookup request.

#### 6.5.1.3 Multi-service Managers

The load imposed on Managers by ServiceHosts is directly proportional to the number of deployed endpoints of the Web Service being managed: with a five-second report period it takes 1,000 deployed endpoints of a single Web Service for its Manager to receive 200CPS of demand. As a further exercise in resource-conservation, a Manager may be implemented that is capable of managing multiple distinct Web Services from a single deployed Manager endpoint. Rather than

deploying dozens of generic Manager endpoints, a MasterManager could instead register a single existing multi-service Manager in the ActiveServiceDirectory under multiple Manager URIs. This scheme would not change the access or communication patterns between actors in the architecture and the only noticeable differences would be increased performance (due to quicker 'deployment' of the Manager) and reduced consumption of hosting resources.

### *6.5.2 INTEGRATION WITH THE 'CLOUD'*

The emergence of 'cloud' computing has provided consumers with a means of deploying applications that obviates the need for privately owned and operated IT infrastructures. Cloud computing infrastructures attempt to provide reliability through complete centralization, with massively redundant hardware and highly reliable, high-bandwidth connections to the Internet. They provide homogenous deployment environments with fixed target platforms and utilize proprietary deployment and load balancing techniques.

Opportunities exist to integrate infrastructures utilizing the presented architecture with one or multiple other 'cloud' infrastructures. This may be desirable for Service Consumers who wish to utilize services provided exclusively by a particular cloud infrastructure, or as a means of tying together a number of cloud infrastructures into a single system with a uniform addressing model. Further, there may be services that are offered in more than one 'cloud' which may be used in the event that one or the other is not available.

In order to introduce these services into the presented architecture, a person or system must take on the Manager role of managing the cloud service, registering and maintaining its entry or entries in the ActiveServiceDirectory. Because the provisioning and load-balancing activities are handled by the cloud infrastructure, the responsibility of providing high availability remains with the cloud provider. Although this does represent the old Service Provider role, the cloud services can still be included in the presented architecture because it does not require any changes to consumer-facing Web Service interfaces.

### *6.5.3 EXPLOITING LOCALITY IN ACTIVESERVICE DIRECTORY RESULTS*

In an operational infrastructure, the ActiveServiceDirectory receives a large quantity of requests for lists of active Web Service endpoints. An opportunity exists to reduce the number of ActiveServiceDirectory lookup requests by returning endpoints of the requested Web Service together with endpoints of those Web Services most frequently requested next. The ActiveServiceDirectory could record the lookup request patterns and group together Web Services which are most frequently requested together. This information may further be used to

pre-emptively deploy endpoints of Web Services which are frequently requested together, and possibly to guide the intelligent placement of the services, such as by co-locating them on a single machine or within the domain of a single Service Host.

#### *6.5.4 MANAGING THE GEOGRAPHICAL PROXIMITY OF DATA & COMPUTATION*

Managing the geographic proximity of data and computation has the potential to vastly improve service performance by reducing the quantity of data that must be transmitted across the Internet. Co-location of the services constituting a composite service could also improve response time by reducing the quantity of remote service calls. By publishing Web Services which act as accessors for particular datasets, information on the data sets most commonly accessed in close temporal proximity to a Web Service may be used to guide the intelligent placement of both the data and Web Service endpoints, reducing latency and improving the perceived service performance.

#### *6.5.5 UNIVERSAL STORESERVICE FOR STATE-MANAGEMENT*

Because of the challenges of managing data consistency between multiple instances of stateful Web Services, there may be benefit in devolving state-management responsibilities away from provider agent applications and into a logically centralized data-management service – a universal ‘StoreService’. The StoreService would assume responsibility for maintaining the consistency and durability of the information stored in it while allowing provider agent application developers to program against a simple Web Service interface. This ‘data plane’ would be complimentary to the described architecture because it would remove storage and state-management considerations from the process of developing Web Services, isolating functionality in an opaque component that can be progressively and transparently matured or updated without requiring applications to be re-engineered.

### **6.6 FINALLY**

In this thesis we have presented an architecture that takes an end-to-end or ‘holistic’ approach to addressing the identified shortcomings of the traditional Web Services model. The architecture presents a multi-endpoint Web Service environment which abstracts over Web Service location and technology and enables the dynamic provision of highly-available Web Services. The model describes mechanisms which provide a framework within which Web Services can be reliably addressed, bound to, and utilized, at any time and from any location.

The presented model eases the task of providing a Web Service by consuming deployment and management tasks. It eases the development of consumer agent applications by letting

developers program against what a service does, not where it is or whether it is currently deployed. It extends the platform-independent ethos of Web Services by providing deployment mechanisms which can be used independent of implementation and deployment technologies. Crucially, it maintains the Web Service goal of universal interoperability, preserving each actor's view upon the system so that existing Service Consumers and Service Providers can participate without any modifications to provider agent or consumer agent application code. Lastly, the model aims to enable the efficient consumption of hosting resources by providing mechanisms to dynamically apply and reclaim resources based upon measured consumer demand.

## REFERENCES

- [1] M. P. Papazoglou, P. Traverso, S. Dustdar and F. Leymann, "Service-Oriented Computing: A research roadmap," *International Journal of Cooperative Information Systems*, vol. 17, pp. 223-255, Jun, 2008.
- [2] M. N. Huhns and M. P. Singh, "Service-Oriented Computing: Key Concepts and Principles," *IEEE Internet Comput.*, vol. 9, pp. 75-81, 2005.
- [3] W3C. (2008). Web services @ W3C. [<http://www.w3.org/2002/ws/Activity>].
- [4] W3C. (2008). World Wide Web Consortium - Web Standards. [<http://www.w3.org/>].
- [5] W3C. (2007). (27 April 2007). SOAP version 1.2 part 0: Primer (second edition). W3C[<http://www.w3.org/TR/2007/REC-soap12-part0-20070427>].
- [6] W3C. (2007). Web services description language (WSDL). [<http://www.w3.org/2002/ws/desc>].
- [7] G. Alonso, F. Casati, H. Kuno and V. Machiraju, *Web Services: Concepts, Architectures and Applications*. Germany: Springer-Verlag Berlin Heidelberg, 2004.
- [8] Sun Microsystems. (2003). Web services life cycle: Managing enterprise web services. [[http://www.sun.com/software/whitepapers/webservices/wp\\_mngwebsvcs.pdf](http://www.sun.com/software/whitepapers/webservices/wp_mngwebsvcs.pdf)]2008).
- [9] Object Management Group. [<http://www.omg.org/>].
- [10] Object Management Group, "Deployment and configuration of component-based distributed applications specification - version 4.0 - OMG documents formal/06-04-02," April 2006. 2006.
- [11] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. v. D. Hoek and A. L. Wolf, "A characterization framework for software deployment technologies," Department of Computer Science, University of Colorado, Boulder, Colorado, April 1998.
- [12] A. Dearle, "Software deployment, past, present and future," in *International Conference on Software Engineering*, 2007, pp. 269-284.
- [13] T. Vanish, M. Dejan, W. Qinyi, P. Calton, Y. Wenchang and J. Gueyoung, "Approaches for Service Deployment," *IEEE Internet Computing*, vol. 9, pp. 70-80, 2005.
- [14] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray and P. Toft, "SmartFrog: Configuration and automatic ignition of distributed applications," in *HP OpenView University Association (OVUA)*, 2003.
- [15] F. Casati, S. Ilnicki, L. J. Jin, V. Krishnamoorthy and M. C. Shan, "Adaptive and dynamic service composition in eFlow," *Advanced Information Systems Engineering*, vol. 1789, pp. 13-31, 2000.
- [16] Object Management Group. (2002). Model driven architecture. [<http://www.omg.org/mda>].
- [17] T. Eilam, M. Kalantar, A. Konstantinou and G. Pacifici, "Model-based automation of service deployment in a constrained environment," IBM, Tech. Rep. RC23382, 2004.

- [18] NovaDigm. Radia. [<http://www.novadigm.com/>].
- [19] Hewlett Packard (2008). HP OpenView application manager using radia. [[http://www.openview.hp.com/products/radia\\_appm/ds/radia\\_appm\\_ds.pdf](http://www.openview.hp.com/products/radia_appm/ds/radia_appm_ds.pdf)].
- [20] T. Vanish, W. Qinyi, P. Calton, Y. Wenchang, J. Gueyoung and M. Dejan, "Comparison of approaches to service deployment," in *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, 2005, .
- [21] W3C. (2002). Web services management concern. [<http://www.w3c.org/>].
- [22] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, vol. 36, pp. 41-50, 2003.
- [23] IBM, "Autonomic computing: IBM's perspective on the state of information technology," IBM, 2002.
- [24] IBM. (2008). Autonomic computing @ developerWorks: Self-managing autonomic technology. [<http://www.ibm.com/developerworks/autonomic/>]2008).
- [25] IBM. (2006, June 2006). An architectural blueprint for autonomic computing. IBM. [<http://www-01.ibm.com/software/tivoli/autonomic/>].
- [26] J. Martin-Flatin, "Push vs. pull in web-based network management," Eidgenössische Technische Hochschule Lausanne, Lausanne, Switzerland, Tech. Rep. SSC/1998/022, 1998.
- [27] W3C. (2004). Web services architecture. [<http://www.w3.org/TR/ws-arch/>].
- [28] W3C, *Extensible Markup Language (XML) 1.0 W3C Recommendation*. 1998, [<http://w3c.org/>].
- [29] Apache Software Foundation. (2008). Apache CXF 2.0 user's guide - WSDL to java. [<http://cwiki.apache.org/CXF20DOC/wsdl-to-java.html>].
- [30] Microsoft. (2008). SPROXY: XML web service proxy generator. [[http://msdn.microsoft.com/en-us/library/ztta389h\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/ztta389h(VS.80).aspx)].
- [31] Eclipse Foundation. (2008). Eclipse project. [<http://www.eclipse.org/projects/>].
- [32] OASIS. (2008). Universal description discovery and integration (UDDI). [<http://uddi.xml.org/>].
- [33] OASIS. (2008). Organization for the advancement of structured information standards. [<http://www.oasis.org>]
- [34] IBM, H. Adams, D. Gisolfi, J. Snell and R. Varadan. (2004). Best practices for web services. 2008).
- [35] G. Saez, A. L. Sliva and M. B. Blake, "Web services-based data management: Evaluating the performance of UDDI registries," in *Proceedings of the IEEE Internal Conference on Web Services*, 2004, pp. 830.
- [36] J. Metso, "Suitability of UDDI registry for the web: Performance measurements," University of Helsinki, Finland, Tech. Rep. C-2003-72, 2003.

- [37] S. Miles, J. Papay, V. Dialani, M. Luck, K. Decker, T. Payne and L. Moreau, "Personalized grid service discovery," in *Proceedings of the Nineteenth Annual UK Performance Engineering Workshop (UKPEW'03)*, 2003, .
- [38] M. B. Blake, A. L. Sliva, M. Muehlen and J. V. Nickerson, "Binding now or binding later: The performance of UDDI registries," in *Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, 2007, pp. 171.
- [39] L. Andrade and J. Fiadeiro, "Composition Contracts for Service Interaction," *Journal of Universal Computer Science*.
- [40] L. Andrade, J. Fiadeiro, J. Gouveia, G. Koutsouko and M. Wermelinger, "Coordination for orchestration," in *Proceedings of the 5th International Conference on Coordination Languages and Models*, 2002 .
- [41] V. Kunal, S. Kaarthik, S. Amit, P. Abhijit, O. Swapna and M. John, "METEOR-S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services," *Inf. Technol. and Management*, vol. 6, pp. 17-39, 2005.
- [42] L. Quanhao, R. Ruonan and L. Minglu, "DWSDM: A web services discovery mechanism based on a distributed hash table," in *Proceedings of the Fifth International Conference on Grid and Cooperative Computing Workshops*, 2006, .
- [43] F. Banaei-Kashani, C. -. Chen and C. Shahbi, "WSPDS: Web services peer-to-peer discovery service," in *Intl. Symposium on Web Services and Applications*, 2004, .
- [44] S. Yu, J. Liu and J. Le, "DHT facilitated web service discovery incorporating semantic annotation," in *Lecture Notes in Computer Science* Anonymous Springer Berlin / Heidelberg, 2004, .
- [45] J. Lukasz, L. Jaroslaw and D. Schahram, "Web service discovery, replication, and synchronization in ad-hoc networks," in *Proceedings of the First International Conference on Availability, Reliability and Security*, 2006, .
- [46] D. Schahram and T. Martin, "Integration of transient Web services into a virtual peer to peer Web service registry," *Distrib. Parallel Databases*, vol. 20, pp. 91-115, 2006.
- [47] OASIS. Electronic business using eXtensible markup language. [<http://www.ebXML.org/>].
- [48] S. Hagemann, C. Letz and G. Vossen, "Web Service Discovery - Reality Check 2.0," *International Journal of Web Service Practices*, vol. 3, pp. 41-46, 2008.
- [49] Web Services Interoperability Organization. (2008). About WS-I - overview. [<http://www.ws-i.org/about/>].
- [50] Ballinger,K., D. Ehnebuske, M. Gudgin, M. Nottingham and P. Yendluri. (2004). (April 16 2004). WS-I basic profile version 1.0. Web Services Interoperability Organization [<http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html>].
- [51] UDDI.org, *UDDI Version 2.04 API, Published Specification*. 2002. [<http://www.uddi.org>].
- [52] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte and D. Winer, "Simple object access protocol (SOAP) 1.1," W3C, 8 May 2000. 2000.

- [53] International Organization for Standardization, "ISO/IEC 29361:2008 Information technology - Web Services Interoperability - WS-I Basic Profile Version 1.1," June 30th, 2008, 2008.
- [54] A. Thomas, "Enterprise JavaBeans technology; server component model for the java platform," Sun Microsystems Inc., 1998.
- [55] Apache Software Foundation. (2004). Apache axis. [<http://ws.apache.org/axis/>].
- [56] Sun Microsystems. (1998). *Technical Report* (May 30, 1998). JDBC overview. [<http://java.sun.com/products/jdbc/>].
- [57] Microsoft Corporation, Microsoft Corporation. (2008, Microsoft open database connectivity (ODBC). [<http://msdn.microsoft.com/en-us/library/ms710252.aspx>].
- [58] G. Alonso, F. Casati, H. Kuno and V. Machiraju, "Application servers," in *Web Services: Concepts, Architectures and Applications* Anonymous Heidelberg, Germany: Springer-Verlag Berlin Heidelberg, 2004, pp. 102.
- [59] Sun Microsystems, (2002). Sun one application framework. 2008.
- [60] IBM. (2008). IBM - WebSphere application server.
- [61] Microsoft Corporation. (2008). Microsoft .NET framework.
- [62] Oracle Corporation. (2008). BEA WebLogic server. [<http://www.bea.com/content/products/weblogic/server/>].
- [63] D. Florescu, A. Grunhagen and D. Kossmann, "XL: An XML programming language for web service specification and composition," in *Proceedings of the 11th International World Wide Web Computer Conference (WWW02)*, Honolulu, Hawaii, USA, 2002.
- [64] F. Leymann, "Web services flow language. version 1.0." International Business Machines Corporation (IBM), May 2001. 2001.
- [65] A. Arkin, "Business process modeling language 1.0. technical report," BPMI Consortium, June, 2002. 2002.
- [66] IBM, BEA Systems, Microsoft, SAP AG and Siebel Systems. (2003). (2007). BPEL4WS specification. [<http://www.ibm.com/developerworks/library/specification/ws-bpel/>].
- [67] Januszewski,K. and Microsoft Corporation. (2007). Using UDDI at run time, part I. [<http://msdn.microsoft.com/en-us/library/ms953944.aspx>].
- [68] Januszewski,K. and Microsoft Corporation. (2007). Using UDDI at run time, part II. [<http://msdn.microsoft.com/en-us/library/ms953948.aspx>].
- [69] C. Pautasso and G. Alonso, "Flexible binding for reusable composition of web services," in *Software Composition* Anonymous Springer Berlin / Heidelberg, 2005, pp. 151-166.
- [70] T. Friesse, J. Müller and B. Freisleben, "Self-healing execution of business processes based on a peer-to-peer service architecture," in *18th Int. Conference on Architecture of Computing Systems*, Innsbruck, Austria, 2005, .

- [71] A. Erradi and P. Maheshwari, "Dynamic binding framework for adaptive web services," in *Proceedings of the 2008 Third International Conference on Internet and Web Applications and Services*, 2008, pp. 162-167.
- [72] M. Stefano, M. Enrico and B. Pernici, "SH-BPEL: A self-healing plug-in for ws-BPEL engines," in *Proceedings of the 1st Workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*, Melbourne, Australia, 2006, .
- [73] L. Steffen, A. Anupriya, S. Rudi and G. Stephan, "Preference-based selection of highly configurable web services," in *Proceedings of the 16th International Conference on World Wide Web*, Banff, Alberta, Canada, 2007, .
- [74] M. D. Penta, R. Esposito, M. L. Villani, R. Codato, M. Colombo and E. D. Nitto, "WS binder: A framework to enable dynamic binding of composite web services," in *SOSE '06: Proceedings of the 2006 International Workshop on Service-Oriented Software Engineering*, Shanghai, China, 2006, pp. 74-80.
- [75] A. Michlmayr, F. Rosenberg, C. Platzer, M. Treiber and S. Dustdar, "Towards recovering the broken SOA triangle: A software engineering perspective," in *IW-SOSWE '07: 2nd International Workshop on Service Oriented Software Engineering*, Dubrovnik, Croatia, 2007, pp. 22-28.
- [76] M. Amoretti, F. Zanichelli and G. Conte, "SP2A: A service-oriented framework for P2P-based grids," in *Proceedings of the 3rd International Workshop on Middleware for Grid Computing*, Grenoble, France, 2005, .
- [77] S. Dustdar and M. Treiber, "A View Based Analysis on Web Service Registries," *Distrib.Parallel Databases*, vol. 18, pp. 147-171, 2005.
- [78] S. Dustdar and M. Treiber, "WiZNet - Integration of different Web service Registries," .
- [79] B. Benatallah, Q. Z. Sheng and M. Dumas, "The Self-Serv environment for Web services composition," *IEEE Internet Computing*, vol. 7, pp. 40-48, Jan-Feb, 2003.
- [80] M. P. Papazoglou, B. J. Krämer and J. Yang, "Leveraging web-services and peer-to-peer networks," in *Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAiSE 2003)*, 2003, pp. 485-501.
- [81] L. Gong, "Project JXTA: A technology overview," Sun Microsystems Inc., 2002.
- [82] G. Kan, "Chapter 8: Gnutella," in *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, A. Oram, Ed. O'Reilly, 2001.
- [83] I. Clarke, O. Sandberg, B. Wiley and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *Designing Privacy Enhancing Technologies: Lecture Notes in Computer Science 2009*, H. Federrath, Ed. Springer, 2000, pp. 46-66.
- [84] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *ACM SIGCOMM 2001*, San Diego, CA, USA, 2001, pp. 149-160.
- [85] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker, "A scalable content-addressable network," in *SIGCOMM'01*, San Diego, CA, USA, 2001, pp. 161-172.

- [86] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, 2001, pp. 329-350.
- [87] B. Y. Zhao, J. Kubiatowicz and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," University of California at Berkeley, 2001.
- [88] D. Talia, P. Trunfio, R. Orlando and C. Silvestri, "A DHT-based peer-to-peer framework for resource discovery in grids," in *Achievements in European Research on Grid Systems* Anonymous Springer US, 2008, pp. 123-137.
- [89] S. J. Norcross, A. Dearle, G. N. C. Kirby and S. M. Walker, "A peer-to-peer infrastructure for resilient web services," in *IEEE International Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications (AAA-IDEA 2005)*, Orlando, Florida, USA, 2005, pp. 65-72.
- [90] D. Schahram and S. Wolfgang, "A survey on web services composition," *Int. J. Web Grid Serv.*, vol. 1, pp. 1-30, 2005.
- [91] P. Watson, C. Fowler, C. Nubicek, A. Mukherjee, J. Colquhoun, M. Hewitt and S. Parastatidis, "Dynamically deploying web services on a grid using dynasoar," in *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distribution Computing (ISORC 2006)*, Gyeongju, Korea, 2006, pp. 8.
- [92] P. Watson and C. Fowler, "Dynasoar: An architecture for the dynamic deployment of web services on a grid or the internet," in *UK e-Science all Hands Meeting 2005*, 2005.
- [93] S. Cavalieri, F. Scibilia, C. Fowler, S. Parastatidis and P. Watson, "Web services usage monitoring for dynasoar," in *Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services*, 2006, .
- [94] H. Andrew and J. T. Ian, "WSPeer - an interface to web service hosting and invocation," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 4 - Volume 05*, 2005, .
- [95] A. Harrison and I. Taylor, "Dynamic web service deployment using WSPeer," in *13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies*, 2005, .
- [96] M. Keidl, S. Seltzsam, K. Stocker and A. Kemper, "ServiceGlobe: Distributing E-services across the internet," in *28th International Conference on very Large Databases (VLDB 2002)*, Hong Kong, China, 2002, pp. 1047-1050.
- [97] M. Keidl, S. Seltzsam and A. Kemper, "Reliable web service execution and deployment in dynamic environments," *Technologies for E-Services, Proceedings*, vol. 2819, pp. 104-118, 2003.
- [98] Apache XML Project. (2000). Xerces-J. [<http://xml.apache.org/xerces-j/>].
- [99] Apache Software Foundation. (2009). Apache tomcat. 2009.

- [100] A. Dearle, S. Walker, S. J. Norcross, G. N. C. Kirby and A. J. McCarthy, "RAFDA: Middleware supporting the separation of application logic from distribution policy," University of St Andrews, 2005.
- [101] S. M. Walker, A. Dearle, S. J. Norcross, G. N. C. Kirby and A. J. McCarthy, "RAFDA: A policy-aware middleware supporting the flexible separation of application logic from distribution," University of St Andrews, 2006.
- [102] Apache Software Foundation. (2009). Apache Axis2. [<http://ws.apache.org/axis2/>].
- [103] Fielding, J., J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. J. Leach and T. J. Berners-Lee. (June 1999). HTTP/1.1: Status code definitions *as section 10 of* Hypertext transfer protocol -- HTTP/1.1.
- [104] Farwick, M. and M. Hafner. (2007). XML parser benchmarks: Part 1. O'Reilly Media, Inc. [<http://www.xml.com/pub/a/2007/05/09/xml-parser-benchmarks-part-1.html>].
- [105] Saloranta, T., D. Srinivas, S. Pericas-Geertsen, B. Margulies and P. Brown. Codehaus: Woodstox XML processor. [<http://xircles.codehaus.org/projects/woodstox>].
- [106] Veillard, D. (2010). LIBXML2: The XML C parser and toolkit of gnome. [<http://xmlsoft.org/>].
- [107] OASIS. (2006). OASIS Web Services Security (WSS) TC, 2006. [[www.oasis-open.org/committees/wss/](http://www.oasis-open.org/committees/wss/)].
- [108] The Internet Society. (2000). RFC 2818 – HTTP over TLS, 2000. [<http://tools.ietf.org/html/rfc2818>].
- [109] Freier, A., Karlton, P., Kocher, P., "The SSL Protocol Version 3.0, 1996". [<http://tools.ietf.org/html/draft-ietf-tls-ssl-version3-00>].
- [110] Wesley Chou, "Inside SSL: The Secure Sockets Layer Protocol," IT Professional, pp. 47-52, July/August, 2002.
- [111] Modeling Stateful Resources with Web Services v. 1.1. I. Foster (ed), J. Frey (ed), S. Graham (ed), S. Tuecke (ed), K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, I. Sedukhin, D. Snelling, T. Storey, W. Vambenepe, S. Weerawarana, March 5, 2004. IBM Technical Report. [<http://www-106.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf>].