Biomimetic and Autonomic Server Ensemble Orchestration

Sunil Nakrani
Keble College

Trinity Term 2005



Programming Research Group
Oxford University Computing Laboratory

Thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy at the University of Oxford

To my family,
for inspiration, belief and boundless support.

Biomimetic and Autonomic Server Ensemble Orchestration

Sunil Nakrani
Keble College, Oxford

Doctor of Philosophy
Trinity Term 2005

Abstract

This thesis addresses orchestration of servers amongst multiple co-hosted internet services such as *e*-Banking, *e*-Auction and *e*-Retail in hosting centres. The hosting paradigm entails levying fees for hosting third party internet services on servers at guaranteed levels of service performance. The orchestration of server ensemble in hosting centres is considered in the context of maximising the hosting centre's revenue over a lengthy time horizon.

The inspiration for the server orchestration approach proposed in this thesis is drawn from nature and generally classed as swarm intelligence, specifically, sophisticated collective behaviour of social insects borne out of primitive interactions amongst members of the group to solve problems beyond the capability of individual members. Consequently, the approach is self-organising, adaptive and robust.

A new scheme for server ensemble orchestration is introduced in this thesis. This scheme exploits the many similarities between server orchestration in an internet hosting centre and forager allocation in a honeybee (*Apis mellifera*) colony. The scheme mimics the way a honeybee colony distributes foragers amongst flower patches to maximise nectar influx, to orchestrate servers amongst hosted internet services to maximise revenue. The scheme is extended by further exploiting inherent feedback loops within the colony to introduce self-tuning and energy-aware server ensemble orchestration.

In order to evaluate the new server ensemble orchestration scheme, a collection of server ensemble orchestration methods is developed, including a classical technique that relies on past history to make time varying orchestration decisions and two theoretical techniques that omnisciently make optimal time varying orchestration decisions or an optimal static orchestration decision based on complete knowledge of the future.

The efficacy of the new biomimetic scheme is assessed in terms of adaptiveness and versatility. The performance study uses representative classes of internet traffic stream behaviour, service user's behaviour, demand intensity, multiple services co-hosting as well as differentiated hosting fee schedule. The biomimetic orchestration scheme is compared with the classical and the theoretical optimal orchestration techniques in terms of revenue stream. This study reveals that the new server ensemble orchestration approach is adaptive in a widely varying external internet environments. The study also highlights the versatility of the biomimetic approach over the classical technique. The self-tuning scheme improves on the original performance. The energy-aware scheme is able to conserve significant energy with minimal revenue performance degradation. The simulation results also indicate that the new scheme is competitive or better than classical and static methods.

Declaration

The work presented by this dissertation has not been submitted, in whatever form, to support an application for another degree or qualification at the University of Oxford or any other university or other institution of learning.

Acknowledgements

I am deeply indebted to my supervisors, Professor William F. McColl (Computing Laboratory, University of Oxford) and Professor Craig A. Tovey (School of Industrial and Systems Engineering, Georgia Institute of Technology, USA) for providing invaluable advice and encouragement throughout the development of this thesis. I would also like to thank Malcolm Low and Shashi Merugu for their comments on my work.

Special thanks are due to following: Professor Richard Brent and Oxford Supercomputing Centre for the use of parallel and cluster computing systems; Professor Sudhakar Yalamanchili for making it possible to be a visiting scholar at the School of Electrical and Computer Engineering, Georgia Institute of Technology; and Professor Tom Seeley at Cornell University for providing invaluable insights into the workings of honeybee colonies.

I would also like to express my gratitude to following: Julie Sheppard for handling administrative affairs on my behalf while I was away from Oxford; Dr. Joe Pitt-Francis for advice on parallelising; Dr. Ian Collier for providing Linux support; Billy Kutulas for help in obtaining request traffic logs; Trey Palmer at the School of Industrial and Systems Engineering, Georgia Institute of Technology for the use of crown cluster computing system; Joel Armstrong at Roosevelt Institute for providing Linux system and support; Management and staff at Squires Building Contractors Limited for providing a highly conducive working environment in London and finally, Frances Page, Mike Shadix, Randy Sherwood and Samantha Massengale for proofreading the thesis.

I am deeply thankful to my wife, Sonal and rest of my family— Dhanji, Laxmi, Gordhan, Vilas, Nastassia, Joakim, Aashvi and Jahnvi for their love, support, understanding and sacrifice. I am very grateful to Sonal's parents— Raojibhai and Madhukanta for their invaluable support and patience in caring for our children while I was busy working on the thesis. Finally, I would also like to thank many friends in the UK and the USA for their help and encouragement.

Table of Contents

Abstract iii

Declaration iv

Acknowledgements v

List of Figures xi

List of Tables xv

Glossary of Abbreviations and Notation xvi

1 Introduction 1

1.1 Motivation 1

1.2 Proposed Server Ensemble Orchestration Approach 3

1.3 Thesis Organisation 4

2 Managing Internet Services 6

2.1 Introduction 6

2.2 Internet Services 6

2.3 Single Server 7

2.4 Server Ensemble 8

2.5 Web Server 10

2.6 Geographical Server Replication 10

2.7 Caching 11

2.8 Summary 13

3 Server Ensemble 14

3.1 Introduction 14

3.2 Internet Hosting Centres 15

3.3 Virtual Server 16

3.4 Service Level Agreements (SLAs) 19

3.5 Internet Services: Performance Perception 20

3.6 Server Orchestration Problem 22

3.7 Summary 25

4 Biomimetic Server Ensemble Orchestration **27**
4.1 Introduction . . . 27
4.2 Related Work . . . 28
4.3 Social Insect Societies . . . 29
4.4 Honeybee (*Apis mellifera*) Colony . . . 30
4.4.1 Forager Orchestration Problem . . . 31
4.4.2 Self-Organising Forager Orchestration . . . 32
4.4.3 Homology: Internet Hosting Centre and Honeybee Colony . . . 38
4.5 Self-Organising Server Orchestration Algorithm . . . 43
4.6 Summary . . . 49

5 Efficacy Assessment: Server Ensemble Orchestrations **51**
5.1 Introduction . . . 51
5.2 Related Work . . . 53
5.3 Greedy Server Orchestration Algorithm . . . 54
5.4 Optimal Omniscient Server Orchestration Algorithm . . . 58
5.4.1 Residual Queue Buckets and Interpolation . . . 62
5.4.2 Optimal Local Neighbour Search . . . 65
5.4.3 An Approximate Optimal Omniscient Orchestration with Queue Buckets . . . 66
5.5 Optimal-Static Omniscient Server Orchestration Algorithm . . . 66
5.6 Summary . . . 69

6 Efficacy of Self-Organising Server Ensemble Orchestration **71**
6.1 Introduction . . . 71
6.2 Simulation Model: Server Orchestration Algorithms . . . 72
6.2.1 Self-Organising Server Orchestration . . . 73
6.2.2 Optimal Omniscient Server Orchestration . . . 77
6.2.3 Optimal-Static Omniscient Server Orchestration . . . 79
6.2.4 Greedy Server Orchestration . . . 80
6.3 Simulation Model: Internet Request Traffic . . . 80
6.3.1 Traffic Streams . . . 82
6.3.2 Hosting Centre: Demand Intensity . . . 86
6.3.3 Baulk Manager . . . 89
6.4 Comparative Assessment of Self-Organising Server Orchestration Algorithm . . . 90
6.4.1 Adaptiveness . . . 91
6.4.1.1 Real Stream . . . 91
6.4.1.2 Step and Heavy Streams . . . 92
6.4.1.3 Step Stream Variability . . . 94
6.4.1.4 Variability of Traffic Streams and Demand Intensity with Differentiated SLA Fees . . . 96
6.4.2 Versatility . . . 110

6.4.2.1 Improved and Degraded Self-Organising Algorithm 111
6.4.2.2 Tuning Self-Organising and Greedy Algorithms 113
6.4.2.3 Quantitative Analysis of Self-Organising Versus Greedy 132
6.5 Summary 136

7 Self-Tuning Server Ensemble Orchestration **139**
7.1 Introduction 139
7.2 Self-Tuning Server Orchestration Algorithm 140
7.3 Efficacy of Self-Tuning Server Ensemble Orchestration 142
7.3.1 Simulation Model 142
7.3.2 Comparative Assessment of Self-Tuning Server Orchestration Algorithm 144
7.4 Summary 149

8 Energy-Aware Server Ensemble Orchestration **151**
8.1 Introduction 151
8.2 Related Work 153
8.3 Energy-Aware Server Orchestration Algorithm 154
8.4 Efficacy of Energy-Aware Server Ensemble Orchestration 156
8.4.1 Simulation Model 156
8.4.2 Comparative Assessment of Energy-Aware Server Orchestration Algorithm 158
8.5 Summary 163

9 Conclusions **165**
9.1 Contribution Summary and Concluding Remarks 165
9.2 Future Research Directions 172

Bibliography **174**

List of Figures

3.1 An internet hosting centre. . . . 15
3.2 Architectural model of a virtual server. . . . 17
3.3 Server ensemble orchestration at two distinct time instants. . . . 23

4.1 Forager orchestration at two distinct time instants. . . . 33
4.2 The information flow and decision processes in a honeybee colony (adapted from Seeley [See95]). . . . 34
4.3 Server and forager orchestration problems. . . . 39
4.4 The information flow and decision processes in an internet hosting centre. . . . 44
4.5 Self-Organising server orchestration. . . . 48
4.6 Self-Organising server orchestration algorithm. . . . 50

5.1 Greedy orchestration algorithm. . . . 55
5.2 Local Neighbour Search algorithm for optimising Greedy orchestration. . . . 59
5.3 Optimal Omniscient orchestration algorithm. . . . 60
5.4 An Approximate Optimal Omniscient orchestration algorithm. . . . 63
5.5 Interpolation algorithm. . . . 66
5.6 Local Neighbour Search algorithm for optimising approximate Optimal Omniscient orchestration. . . . 67
5.7 An optimised Approximate Optimal Omniscient orchestration algorithm with Residual Queue Buckets and Local Neighbour Search. . . . 68
5.8 An Optimal-Static Omniscient orchestration algorithm. . . . 70

6.1 Self-Organising server orchestration simulation model. . . . 74
6.2 An advertisement board manager for the Self-Organising server orchestration algorithm. . . . 75
6.3 Optimal Omniscient server orchestration simulation model. . . . 78
6.4 Optimal-Static Omniscient server orchestration simulation model. . . . 79
6.5 Greedy server orchestration simulation model. . . . 80
6.6 Synthetic request stream generator. . . . 81
6.7 Trace request stream generator. . . . 82
6.8 Hybrid request stream generator. . . . 82
6.9 Inhomogeneous Poisson request arrival pattern. . . . 83
6.10 Step request arrival pattern. . . . 84
6.11 Heavy tail request arrival pattern. . . . 85

6.12 Request arrival pattern derived from traces of real internet services. 85
6.13 Hybrid request arrival pattern. 86
6.14 Baulk manager. 89
6.15 Real revenue stream normalised with respect to the Optimal Omniscient algorithm. 92
6.16 Step (4:1) revenue stream normalised with respect to the Optimal Omniscient algorithm. 94
6.17 Step (10:1) revenue stream normalised with respect to Optimal Omniscient algorithm. 95
6.18 Heavy revenue stream normalised with respect to Optimal Omniscient algorithm. 95
6.19 Adapting to variability in peak-to-trough ratio of Step request stream for two services co-hosted. 97
6.20 Adapting to variability in peak-to-trough ratio of Step request stream for three services co-hosted. 98
6.21 Adapting to variability in peak-to-trough ratio of Step request stream for four services co-hosted. 98
6.22 A process model for design of experiments tuning methodology. 116
6.23 An excerpt of a fractional factorial screening experimental design for Self-Organising algorithm. 118
6.24 An excerpt of a typical fractional factorial interaction experimental design for Self-Organising algorithm. 119
6.25 Linear model fit for the Self-Organising screening experiment design. 121
6.26 Scaled estimates of the model parameters for the Self-Organising screening experiment design. 122
6.27 Linear model fit for the Self-Organising interaction effects experiment design. . 123
6.28 Prediction profile for the Self-Organising screening experiment design. 124
6.29 Scaled estimates of the model parameters for the Self-Organising interaction effects experiment design. 126
6.30 Interaction plots for the Self-Organising interaction effects experiment design where superimposed labels in rows are as follows: 1) TrafficStream(Heavy, Hybrid, IP, Real and Step); 2) Rho(low, med, high and shigh); 3) Hosted(2, 3 and 4) whilst the remainder have the labels high (+1) and low (-1). 127
6.31 An excerpt of a fractional factorial interaction experimental design for Greedy algorithm. 128
6.32 Linear model fit for Greedy process interaction experiment design. 129
6.33 Scaled estimates of the model parameters for the Greedy process interaction experiment design. 130
6.34 Interaction plots for the Greedy process interaction experiment design where superimposed labels in rows are as follows: 1) TrafficStream(Heavy, Hybrid, IP, Real and Step); 2) Rho(low, med, high and shigh); 3) Hosted(2, 3 and 4) whilst the remainder have the labels high (+1) and low (-1). 131
6.35 Schematic for individually-tuned and best-single-tuned quantitative analysis. . 134

7.1 Self-Tuning server orchestration algorithm. 143
7.2 Demand intensity manager. 145

8.1 Energy-Aware server orchestration algorithm. 155

List of Tables

3.1 An example of a Service Level Agreement. 20
3.2 Behaviour patterns of internet service users to response delay. 22

4.1 Parallels between Server and Forager orchestration problems. 40
4.2 Bio-mimicking mechanisms of a honeybee colony. 45
4.3 Adjusting probability (threshold) of reading the advertisement board 47

6.1 Simulation model parameters for an internet hosting centre. 73
6.2 Simulation model parameters for Self-Organising server orchestration algorithm. 77
6.3 Simulation model parameters for dynamic update of advertisement reading threshold. 77
6.4 Simulation model parameters for Optimal Omniscient server orchestration algorithm. 78
6.5 Simulation model parameters for Optimal-Static Omniscient server orchestration algorithm. 79
6.6 Simulation model parameters for Greedy server orchestration algorithm. 81
6.7 Simulation model parameters for various traffic streams. 87
6.8 Mean arrival and service rates for different traffic streams. 88
6.9 Simulation model parameters for Baulk manager. 90
6.10 Revenue from Real traffic streams. 92
6.11 Revenue from Step and Heavy traffic streams. 94
6.12 Paired observation test revenue difference and standard deviation for Step and Heavy traffic streams. 96
6.13 Interarrival times for Step traffic stream variability. 97
6.14 Modified Simulation model parameters from its default value for respective server orchestration algorithms. 101
6.15 Normalised revenue stream performance of the Self-Organising algorithm aggregated across traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 20$ and for overall, $n = 60$. 101

6.16 Normalised revenue stream performance of the Self-Organising algorithm aggregated across traffic streams (Step, Heavy, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 160$ and for overall, $n = 480$. 103
6.17 Wilcoxon matched-pairs signed ranks test for Self-Organising against Greedy and Optimal-Static Omniscient aggregated across traffic streams (Step, Heavy, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 160$ and for overall, $n = 480$. 103
6.18 Normalised revenue stream performance of the Self-Organising algorithm aggregated across traffic streams (Step, Heavy, Inhomogeneous Poisson and Hybrid) where for each co-hosting instance (two, three and four services), $n = 40$ and for overall, $n = 120$. 105
6.19 Wilcoxon matched-pairs signed ranks test for Self-Organising against Greedy and Optimal-Static Omniscient aggregated across traffic streams (Step, Heavy, Inhomogeneous Poisson and Hybrid) where for each co-hosting instance (two, three and four services), $n = 40$ and for overall, $n = 120$. 106
6.20 Normalised revenue stream performance of the Self-Organising algorithm aggregated across demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 40$ and for overall, $n = 160$. 107
6.21 Wilcoxon matched-pairs signed ranks test for Self-Organising against Greedy and Optimal-Static Omniscient aggregated across demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 40$ and for overall, $n = 160$. 108
6.22 Normalised revenue stream performance of the Self-Organising algorithm aggregated across short timescale traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 40$ and for overall, $n = 120$. 109
6.23 Wilcoxon matched-pairs signed ranks test for Self-Organising against Greedy and Optimal-Static Omniscient aggregated across short timescale traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 40$ and for overall, $n = 120$. 109
6.24 Normalised revenue stream performance of the Self-Organising algorithm aggregated across long timescale traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 40$ and for overall, $n = 120$. 110

6.25 Wilcoxon matched-pairs signed ranks test for Self-Organising against Greedy and Optimal-Static Omniscient aggregated across long timescale traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 40$ and for overall, $n = 120$. 110
6.26 High and low simulation model parameter values for 2^{8-3} fractional factorial experiment. 111
6.27 High and low simulation model parameter values for 2^{6-3} fractional factorial experiment. 113
6.28 Revenue streams for three versions of the Self-Organising algorithm indicating its versatile performance. 114
6.29 Internal and external process model parameters for Self-Organising orchestration algorithm and an internet hosting centre. 120
6.30 Greedy algorithm and internet hosting centre process parameters. 125
6.31 Reduced Self-Organising algorithm and internet hosting centre process parameters. 133
6.32 Greedy algorithm and reduced internet hosting centre process parameters. . . . 133
6.33 Normalised tuned revenue streams for Self-Organising and Greedy algorithm aggregated across traffic streams (Heavy, Real, Inhomogeneous Poisson), demand intensities (medium and high) and co-hosting instance (two and three services). 135
6.34 Normalised tuned revenue streams with sample standard deviation for Self-Organising and Greedy algorithms aggregated across traffic streams (Heavy, Real, Inhomogeneous Poisson), demand intensities (medium and high) and co-hosting instance (two and three services). 136

7.1 Modified and new simulation model parameters for Self-Tuning and Self-Organising server orchestration algorithms. 146
7.2 Normalised revenue stream performance and Wilcoxon matched-pairs signed ranks test for Self-Tuning against Self-Organising algorithms aggregated across traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), n and number of matched pairs is 200 whilst for overall, n and number of matched pairs is 600. 147
7.3 Normalised revenue stream performance and Wilcoxon matched-pairs signed ranks test for Self-Tuning algorithm against Self-Organising algorithm aggregated across traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) where for each co-hosting instance (two, three and four services), n and number of matched pairs is 50 whilst for overall, n and number of matched pairs is 150. 148

8.1 Modified and new simulation model parameters for Energy-Aware and Self-Tuning server orchestration algorithms. 159

8.2 Normalised power-on-time performance and Wilcoxon matched-pairs signed ranks test for Energy-Aware algorithm against Self-Tuning algorithm aggregated across traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), n and number of matched pairs is 200 and for overall, n and number of matched pairs is 600. . . 160

8.3 Normalised revenue stream and mean waiting time for service performance of the Energy-Aware algorithm against Self-Tuning algorithm aggregated across traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 200$ and for overall, $n = 600$. 161

8.4 Wilcoxon matched-pairs signed ranks test for Energy-Aware algorithm against Self-Tuning aggregated across traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), number of matched pairs is 200 and for overall, number of matched pairs is 600. 162

8.5 Normalised Power-On time performance for Energy-Aware algorithm against Self-Tuning algorithm aggregated across traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) where for each co-hosting instance (two, three and four services), $n = 50$ whilst, for overall, $n = 150$. 163

Glossary of Abbreviations and Notation

Honey Bee Colony

M	Total Number of Flower Patches
N	Total Number of Nectar Foragers
$\mathcal{N}_i$	Nectar Replenishment Rate Per Second of i^{th} Flower Patch
$f_i()$	i^{th} Flower Patch Value Function
x_i	Number of Nectar Foragers at i^{th} Flower Patch

Hosting Centre

A_t	t^{th} State of Request Arrivals for all M services
$\tilde{A}_t$	t^{th} State Request Arrivals Forecast for all M services
a_i^t	t^{th} State of Request Arrivals for i^{th} Service
D	Advertisement Display Duration
E_i	Revenue Earning Rate of i^{th} server
$E_{hosting\text{-}centre}$	Revenue Earning Rate of the Hosting Centre
$f()$	Next System State Function
I_j	j^{th} hosted service
IntArr	Interarrival Times
IP	Inhomogeneous Poisson
L	Advertisement Display Duration Scaling Factor
$\bar{\lambda}_i$	Mean Request Arrival Rate of i^{th} Service
M	Total Number of Services Co-Hosted
M-VS	M Virtual Servers Co-Hosting M Services
$\bar{\mu}_i$	Mean Service Rate of i^{th} Service
n_i	Total Number of Servers Composing Virtual Server V_i
N	Total Number of Server in the Server Ensemble
Opt-Omni	Optimal Omniscient Server Ensemble orchestration Algorithm
Opt-Stat-Omni	Optimal-Static Omniscient Server Ensemble orchestration Algorithm
ρ	Demand Intensity
p	Advertisement Posting Threshold
p_i	Hosting fee schedule for pay-per-request-served SLA at i^{th} Service
π_G^t	t^{th} Greedy Orchestration Policy

π^t	t^{th} Orchestration Policy
π_i^t	t^{th} Orchestration Policy for i^{th} Service
π_{Opt}^t	t^{th} Optimal Omniscient Orchestration Policy
$\pi_{opt-stat}$	Optimal-Static Omniscient Orchestration Policy
π_*^t	t^{th} Optimal Orchestration Policy
Q_i	i^{th} Service Queue
q	Total Number of Residual Request Queue Buckets
QoS	Quality of Service
r	Advertisement Reading Threshold
$R()$	Revenue Function
R_i	Total Number of Request Served by i^{th} server in a given interval
$\mathcal{R}_i()$	i^{th} Service Queue Revenue Function
S	Advertisement Scaling Factor
S_t	t^{th} System (hosting centre) State
s_i	i^{th} Server
Self-Org	Self-Organising Server Ensemble Orchestration Algorithm
Service-App	An application(implementation) of an internet service
SLAs	Service Level Agreements
V_i	i^{th} Virtual Server
$\mathcal{V}_i$	Revenue Replenishment Rate of i^{th} service queue
v^t	t^{th} Value Function
$\hat{v}^t$	Interim Optimal Omniscient Revenue Table for t^{th} interval
t	Time Step Index in $1, \ldots, T$ Intervals of the Time Horizon

Internet

DNS	Domain Name Server
FTP	File Transfer Protocol
HTTP	HyperText Transfer Protocol
IP	Internet Protocol
IMAP	Internet Message Access Protocol
POP	Post Office Protocol
URL	Universal Resource Locater
WWW	World Wide Web

Network

LAN	Local Area Network
SAN	System Area Network
WAN	Wide Area Network

Statistics

W^+	Sum of Positive Ranks

W^-	Sum of Negative Ranks
N	Total Number of Matched Pairs with Ties Discarded
p-value	Probability Value for Accepting or Rejecting Hypotheses
$\frac{\bar{\sigma}}{\sqrt{n}}$	Standard Deviation or Standard Error of the Sample Mean

System

CPU	Central Processing Unit
FCFS	First Come First Served
FIFO	First In First Out
NOW	Network of Workstations
PC	Personal Computer
PDA	Personal Digital Assistant
RAID	Redundant Array of Inexpensive Disks
SMP	Symmetric Multi-Processor
I/O	Input and Output

Chapter 1

Introduction

If you don't know where you are going, any road will take you there.

Lewis Carroll *(1832-1898) - Alice's Adventures in Wonderland*

1.1 Motivation

The Internet landscape is richly populated with entities such as banks, stock brokerages, retailers, news media, communities, and governments. The myriad internet services offered by these entities include searching, instant messaging, e-mail, directories, maps, online purchase, stock trading, banking, ticket reservation, tax filing and auctions. The ubiquity of the Internet, the plethora of internet services and their ease of use have led a large fraction of human civilisation to become dependent upon the internet computing infrastructure for business-critical and day-to-day activities in a manner not too dissimilar from dependence on electricity, telecommunication, water and gas infrastructures.

Since the global user audience now depends on the Internet for many activities, the computing infrastructure must possess similar performance characteristics as the other utility infrastructures. These include high availability (twenty-four hours a day), fault tolerance, scalability to support incremental service growth and absolute scalability to support evolving user populations, high utilisation and performance guarantees (response time, throughput) [BM00, Bre00, GWBC99, Wan99, ZFJ97]. Meeting these performance criteria, however, has proven difficult. The inherently dynamic Internet is much more complex than the other utility infrastructures. Service request traffic often exhibits bursts above the mean for lengthy periods of time over a wide range of timescale and varies in terms of peak-to-trough by factors

of more than 100 in less than an hour [AJ99, BBC00b, BBC01, BBC03, CB97, Yan01].

Despite the promise of access to global audiences and dependence that is predicted to burgeon well into the future, internet computing, in general, has been unsuccessful at guaranteeing the quality of performance that is essential for such a critical infrastructure [Alt00a, Alt00b, BBC00a, BBC99, KEY01, Tow03]. The two main causes of failure is the *ad hoc* manner in which the internet computing infrastructure has evolved and service request traffic that is unpredictable due to the open and dynamic Internet environment. The failure often results in users of the service experiencing flash-crowd or hot-site-of-the-day type of phenomena [KEY01, Tow03, Wig01].

In an attempt to address utility-like performance characteristics [Bre00, GBHC00] for the internet computing infrastructure, a large number of approaches have been proposed recently in the literature: 1) classical control theory to manage server performance [ASB, LAS01, PGH+01]; 2) server overload protection by admission control [AL00, BF99, CM99, JR00]; 3) reducing content quality when the server approaches maximum capacity condition [AB99]; 4) differentiated performance level by priority scheduling [ADMC, CM99]; 5) scalable system architecture based on server ensemble [ACP95, Bre00, CADAD+97, CB98, Che99]; 6) load balancing [ALT03, APB96, BCLM98, CIS03, DKMT96, HGKM98, JK00, PAB+98]; 7) server ensemble management for multiple co-hosted services [AFF+01, CATV01, FKSR02, JKK+01, WY01]; 8) scalable web server [Apa03, HPS97, HPS98, KBM94, KMR95, PDZ99, squ00, WGBC00, Zeu03]; 9) replicating services geographically [APB96, BLFN95, HGKM98, KKO98, STA01, YCE+97] and 10) caching [BC96, CDN+96, CID99, CKR98, CY97, CZB98, FCAB98, FCLJ99, IC97, KLL+97, KLM97, MC98, MLB95, PH97, PM96, RCG98, TDVK99, VR98, Wan99, WC97a, WC97b, ZFJ97].

Whilst each of the above research areas tackles a specific performance issue, there exists an inescapable fundamental relationship between performance and provisioning of server capacity. However, provisioning server capacity for highly variable and unpredictable service request traffic is an extremely challenging proposition. Most approaches to server capacity provisioning are threshold and rule-based where many *ad hoc* rules have to be defined, continually monitored and updated by the operations staff. Examples of approaches include system resource monitors, load balancers, static provisioning and manual tracking of load metrics. These approaches do not account for multi-faceted complexities across the whole service infrastructure, the rapidly changing dynamic environment or the need to adapt to real-time

operation to meet performance requirements.

Recent emerging market trends offer an internet service hosting model where multiple internet services are co-hosted for a fee in a centralised hosting centre comprising an ensemble of commodity servers. The model has gained a significant foothold in the commercial world. From 1997 to 2002, a thirty-fold increase in revenue was predicted with a total revenue of approximately £6.5 billion for the year end 2002 [Che99]. In 2003, International Business Machines Corporation reported £3.9 billion revenue earned from its on-demand services whilst saving an additional £3.9 billion by applying the model to its own operations [Mar03]. Combined with other major players (American Telegraph and Telephone Company, Hewlett Packard, Microsoft and Sun), it is estimated that the revenue for 2005 will be approximately £22 billion. In the centralised hosting model, servers are shared amongst co-hosted services thereby benefiting from an opportunity to dynamically provision server capacity to meet unpredictable demands whilst offering the economy of scale benefits to service content owners [JKK+01].

In this thesis, we build on the new hosting model to provide a solution to dynamic server capacity provisioning. In order to achieve this solution, the thesis addresses the server ensemble orchestration problem in the internet hosting centres. Specifically, a biomimetic orchestration method based on the swarm intelligence of honeybee colonies is used to meet challenges of the highly variable Internet environment.

1.2 Proposed Server Ensemble Orchestration Approach

Whilst solutions for server ensemble orchestration in the internet hosting model have been developed in the commercial domain, they remain proprietary and, therefore, largely unknown. The solutions developed in the academic and research domain are more well known and fall into two broad categories. The first category includes solutions that address server ensemble orchestration assuming no knowledge about future service request traffic loads [AFF+01, CATV01]. Here, vital performance metrics are monitored and any deviation is reported to a central resource director whose function is to choose an appropriate orchestration policy. These techniques formulate orchestration policies with ease but suffer from their centralised reactionary nature.

The second category of solutions tackles the server orchestration problem by predicting

future service request traffic loads [ASB, FKSR02, JKK+01, PM03, WY01]. Here, orchestration policy is generated from a prediction of near future demand. Although these techniques offer the advantage of being proactive, the prediction of future demand can be unreliable given the highly variable internet environment.

In this thesis, we address the server ensemble orchestration in internet hosting centres using an approach that falls in the first category. That is, no *a priori* knowledge of service request traffic load. The thesis proposes an entirely new approach to the server ensemble orchestration problem. This novel approach is inspired from the swarm intelligence of social insects to solve problems in a distributed manner without centralised coordinations whereby the solution to a problem is derived by sophisticated collective behaviour borne out of primitive interactions amongst members of the group that are characterised as self-organising, adaptive and robust [BDT99].

We apply the self-organising behaviour of a honeybee (*Apis mellifera*) colony, which dispatches forager bees to collect nectar from flower patches that vary in availability and quality, to the internet hosting centre which must orchestrate its server ensemble amongst co-hosted services to collect hosting fees from serving unpredictable service requests load. We map key characteristics of forager orchestration to server orchestration under a pay-per-use hosting agreement and show that the problems map extremely well.

The novel server ensemble orchestration technique presented in this thesis is implemented using the discrete event simulation library C++SIM [Lit94]. In addition, we have devised other server orchestration techniques in order to assess the efficacy of the innovative approach. For the sake of readability, we have described each technique using pseudo-code. The simulation can be run using synthetically generated demand patterns or trace driven using demand patterns from a real internet service provider. The entire simulation framework provides models for an internet hosting centre being managed by server ensemble orchestration algorithms and an internet environment comprised of traffic streams and customer behaviour.

1.3 Thesis Organisation

The remainder of the thesis is organised as follows: Chapter 2 offers an overview of selected techniques employed to manage performance of internet services. Chapter 3 examines the server ensemble framework for co-hosting multiple internet services in a centralised internet

hosting centre. We also outline the *de facto* virtual server architecture that enables sharing of servers, an outsourcing model and the complexity of internet traffic. Finally, we give a mathematical formulation of the server orchestration problem.

In Chapter 4, we study a honeybee (*Apis mellifera*) colony and its self-organising solution to the forager orchestration problem. We exploit the homology between a honeybee colony and an internet hosting centre to propose a new biomimetic server ensemble orchestration algorithm that inherits the autonomic property of the honeybee solution. Next, in Chapter 5, in order to evaluate the biomimetic scheme, we devise a collection of server ensemble orchestration algorithms based on techniques different from our biomimetic approach. In Chapter 6, we introduce a simulation framework and models for an internet hosting centre and external environment that drive the co-hosted services. The effectiveness of our biomimetic approach is assessed in terms of adaptiveness and versatility against the collection of algorithms devised in Chapter 5.

In Chapter 7, we extend the new Self-Organising server ensemble orchestration algorithm by proposing an additional feedback loop to enable the algorithm to tune orchestration behaviour when certain external conditions arise. The efficacy of new tuning behaviour is comparatively assessed with the original algorithm. The new Self-Tuning algorithm is used as a base onto which an extension is proposed with a dual objective of optimising revenue whilst minimising energy consumption. In Chapter 8, the new Energy-Aware server orchestration algorithm is compared against the base Self-Tuning algorithm to assess its effectiveness. Finally, in Chapter 9, we conclude the thesis with a summary of contributions and a discussion of future research directions.

Chapter 2

Managing Internet Services

The Internet is so big, so powerful and pointless that for some people it is a complete substitute for life.

Andrew Brown *(1938 -)*

2.1 Introduction

This chapter provides an overview of selected research work focusing on the management of internet services. The research space can be divided into five broad categories: Server, Server Ensemble, Web Server, Geographical Server Replication and Caching. We begin by reviewing performance management schemes related to a single server. The next section reviews server ensembles employing virtual server system architecture and techniques for load balancing and server orchestration amongst co-hosted services. Next, we review schemes to improve concurrency to meet the unpredictable demand placed on a web server. We then outline geographical replication used to alleviate network congestion at a single site and overloading. We also review how geographical replication is used to improve availability of services. We continue this section with a review of schemes for selecting geographically dispersed servers. Finally, we review caching schemes to reduce load on the server and improve response latency.

2.2 Internet Services

At the dawn of the twenty-first century, an information revolution continues to gather momentum aided by the ubiquitous Internet and widespread adoption of the World Wide Web

(WWW) as the medium of choice for transactions. Currently, the Internet is richly populated with entities such as banks, stock brokerages, retail, communities and governments all offering their services to the global audience, not to mention the myriad internet services such as searching, instant messaging, directories, maps, e-mail, news and weather amongst others being offered by the likes of Yahoo, Microsoft, America Online and Google.

Potentially, users are distributed across the globe and interact with internet services by submitting requests over the Internet. The burgeoning user population is heavily reliant on the vast array of internet services for day-to-day activities akin to power or telephony infrastructures. Consequently, internet services must possess utility-like characteristics including high availability, scalability to support growing user population and performance guarantees [Bre00, GBHC00].

An entity with a presence on the Internet is identified by a unique service name, for example `news.bbc.co.uk`. Users on the Internet submit requests for service using this name whilst the entity in question provides access in order for requests to reach its service infrastructure and respond accordingly. The service infrastructure entails hosting content data on a suitable server platform connected to the Internet. The server platform comprises a web server, an application server (service), database and system hardware (CPU, memory, network cards and disks). The service infrastructure is the workhorse of an internet service which is responsible for providing content in response to incoming requests. The Internet infrastructure, on the other hand, is the pipeline which is responsible for delivering requests to the server and response (content) to users. It comprises systems, networks and software that form the fabric between servers and clients (PCs, PDAs, mobile phones). Given the strong requirements for internet service performance guarantees, both the service and Internet infrastructures have been the focus of major attention for performance improvement by the research community.

2.3 Single Server

Managing server performance to guarantee service levels such as response time or throughput is essential for maintaining a favourable user perception of the service. Classical control theory has been used to propose techniques to manage server performance and overload protection [ASB, LAS01, PGH+01]. These techniques are largely based on developing software controllers using closed loop or feedback control systems to tune parameters that affect

service level performance. Server overload protection is another technique where requests passthrough admission control before reaching the server in order to guarantee performance to admitted requests [AL00, BF99, CM99, JR00]. Reducing content quality (e.g. providing low resolution images) when a server approaches an overload condition allows more requests to be served [AB99]. Providing differentiated levels of service by prioritising requests scheduling [ADMC, CM99] for a variety of users is desirable in cases where certain types of users are paying more than others to use the service.

2.4 Server Ensemble

Dependence upon internet services by a growing number of users is akin to dependence on power or telephony infrastructures in that services must guarantee twenty-four hours a day availability together with incremental and absolute scalability to support evolving and growing user populations. To this end, the ubiquitous Network of Workstations (NOW) is a highly scalable system architecture composed of commodity servers interconnected with a high speed and bandwidth network. The NOW cluster architecture has proven to be a natural platform for constructing server infrastructure given that each server in the cluster represents an independent failure boundary which allows replication techniques to provide fault tolerance. The NOW permits incremental and absolute scalability as servers can be augmented to increase capacity [ACP95, Bre00, CADAD+97, CB98, Che99]. The architecture allows deployment of internet services in a single geographical location by hosting each service on clusters comprising an ensemble of servers. Each service application and its data are replicated on each server of a respective cluster on which it is hosted [Bre00, DGLS99]. The requirements for high utilisation, good average response time and throughput, for example, raise important questions about cluster resource management in the face of unpredictable future demand particularly in circumstances where aggregate demand amongst co-hosted services exceeds available resources.

At the granularity level of a single cluster, an incoming stream of requests is distributed across the servers using load balancing techniques that usually entail mapping IP addresses to servers in the cluster. The centralised approach has a front-end connection router which maps a unique service name to multiple IP addresses of servers in the cluster. This is achieved by modifying the destination IP address based on a load balancing policy such as round

robin [ALT03, APB96, CIS03, DKMT96, HGKM98, JK00]. In order to improve response time performance, a locality-aware load balancing policy proposed by Pai *et al.* [PAB+98] uses content-based distribution of incoming requests to appropriate servers in the cluster to achieve high locality in the main memory cache. The distributed approach proposed by Bestavros *et al.* [BCLM98] entails m servers in a cluster acting as a connection router to n servers with the properties of having no single point of failure and scalability of routing capacity with any number of servers in the cluster.

For the case of more than one cluster at a single geographical location, co-hosting multiple services and sharing a pool of servers entail orchestrating servers amongst competing services to meet performance requirements. To date, a number of researchers have framed the problem and proposed orchestration techniques of various types. Wolf and Yu [WY01] have framed the problem where a single server can be shared across multiple clusters based on whether it is serving private or public requests and propose a resource allocation algorithm that determines hosting of services to servers on a static basis (i.e. once a week) and real-time routing of incoming requests such that the load is balanced across the servers. De Farias *et al.* [FKSR02] have framed the problem in the context of multiple co-hosted services with different SLA performance guarantees for each class of request. They attack the problem with dynamic programming but the exact solution of the formulation is computationally infeasible and, therefore, they propose an approximate orchestration solution based on linear programming. In a similar context, Jayram *et al.* [JKK+01] confront the problem with a general benefit task system with system states defined by a composition of clusters from a server ensemble for each time step t and a benefit function. The system can be in any single state and the benefit derived by transitioning from state s_t to s_{t+1} is a function of the states at t and $(t+1)$. The goal is to determine a sequence of states that maximises the total benefit derived. Deterministic and randomised algorithms which rely on known future benefit (lookahead) are given. In the context of SLA, Appleby *et al.* [AFF+01] propose an approach where the centralised resource director receives feedback from performance monitoring agents and makes orchestration decisions based on violations of performance metrics. Similarly, an approach proposed by Chase *et al.* [CATV01] entails servers monitoring load metrics and providing feedback to a central resource manager which uses an economic framework to orchestrate the server ensemble for meeting SLA performance guarantees and a reduction in energy consumption.

2.5 Web Server

Given the unpredictability of demand for service [Alt00a, Alt00b, BBC99, BBC00a, BBC00b], the web server must cope with concurrency on scales hitherto unknown. Various architectures for building web servers with a desirable property of high concurrency have been proposed [KBM94, KMR95]. The Single Process Event Driven (SPED) architecture entails structuring a program thread that loops continuously, processing events of different types from the queue. Examples include the Zeus web server [Zeu03] and the squid proxy cache web server [squ00]. The Multi-Threaded (MT) architecture uses a single dedicated thread to service the network which farms-off incoming tasks to individual task-handling threads, Multi-Process (MP) architecture, on the other hand, uses a single process to perform all the steps necessary to satisfy a request. However, since multiple processes are employed, multiple requests can be satisfied concurrently. An example of MT and MP together is the Apache web server [Apa03]. The asymmetric multi-process event-driven architecture combines principle ideas of SPED and MT such that it uses SPED to handle all requests, When request content is not in the cache it uses MT in order to preserve the efficiency advantage of SPED [PDZ99]. To aid the design of highly concurrent web servers, design frameworks have been proposed to implement concurrency, protocol and I/O strategies [HPS97, HPS98, WGBC00].

2.6 Geographical Server Replication

In order to guarantee service availability in case of failure, overloading or network congestion at a single geographical location, mirroring is used to replicate services at dispersed geographical locations. The distributed architecture requires selecting an appropriate replicated server to increase availability, improve response time and balance request load, for example. The most rudimentary server selection technique, popular with software repositories, is to let the user choose a server that is close in geographical proximity. The simple redirect technique uses a primary server responding to clients by instructing them to send the request to a replicated server at another geographical location instead of responding to the request itself [BLFN95]. The Domain Name Server (DNS) aliasing technique entails mapping a known service name to multiple IP addresses at replicated servers. Thus, when a client resolves a service name, alternate IP addresses are provided using a policy such as round-robin [HGKM98, STA01]. The IP layer technique uses a router to examine and, possibly, modify IP packets before

forwarding them to their destination. In effect, it maps a known service name to multiple replicated servers [APB96]. The above techniques can successfully perform load balancing but they suffer from several weaknesses: single points of failure, server selection made without taking geographical load balancing into account, or current load on the server and router or server becoming a bottleneck itself. An approach proposed by Karaul et al. [KKO98] uses individual clients to select replicated servers. Each client and associated server in a pool of replicated servers has an associated agent. The server agent communicates with the client agents to provide addressing information while client agents monitor network conditions and server load by measuring end-to-end delay and make server selection decisions based on the average response times of each server. Clients also make the server selection decision in an approach proposed by Yoshikawa *et al.* [YCE+97]. Each client has a service specific interface applet which accepts user input and presents these to a director applet which is responsible for selecting a suitable replicated server. The director applet maintains knowledge of replicated servers and service related meta-information by receiving updates from individual servers.

2.7 Caching

Internet services resemble a large distributed system which provides access to shared content data. Caching popular content data at locations close to clients has been recognised as an effective technique for alleviating server overload, reducing network traffic and providing service scalability [Wan99]. In general, content data can be cached on clients, cache proxies and servers under different caching architectures. In the Hierarchical Caching Architecture caches are placed at various levels: bottom (client or browser), institutional (proxy), regional (proxy) and national (proxy). Under this scheme, if a request is not completed by the bottom level cache then it is forwarded to the institutional, regional and national level caches provided each preceding level is unable to satisfy the request. If the national level cache cannot service the request then it contacts the original server to do so. Whenever content data is found in a hierarchical cache or in the original server, it traverses down the hierarchy, leaving a copy at each of the intermediate caches along the path so that a future request for the same content can be satisfied by one of the caches in the hierarchy [CDN+96, ZFJ97]. In the Distributed Caching Architecture caches only exist at one level in the network, usually at the bottom. Under this scheme, all caches keep meta-information about the content of every participating

cache so that this information can be used to determine from which cache to obtain the content data in an event of a cache miss. A hierarchical mechanism is employed to distribute directory information about the location but not the content data [FCAB98, PH97, TDVK99]. In the Hybrid Caching Architecture caches cooperate with other caches at the peer level or at a higher level using distributed caching. The content data is retrieved from either a parent or neighbour with a minimum round trip time [RCG98].

The key challenge for hierarchical, distributed and hybrid cache architectures is to locate the sought content data with minimum latency, low bandwidth consumption and high cache hit ratio. The inter-cache messaging uses Internet Cache Protocol (ICP) [WC97b] to send queries, exchange and gather information about the location of content data. Caching trees rooted at popular servers are constructed towards demand hot-spots with resolution being performed using a cache routing table or hash function techniques. A number of techniques use routing table resolution but attempt to optimise the method by using IP multicast [MLB95]. Examples include querying siblings before propagating the request towards the root [CDN+96], querying overlapped multicast cache groups [ZFJ97] and co-operating caches exchanging routing information periodically and pre-computing cache routing tables [WC97a]. Similarly, resolution techniques based on the hash function optimise by having each proxy keep a summary in the cache directory of the other participating proxies. The proxy consults its summary before propagating any request [FCAB98] or dividing content data (URL) space amongst loosely coupled proxy cache servers, such that requests can be intelligently routed [VR98] and minimal changes are required to maintain consistent hashing when the function range increases [KLL+97].

A server provides static data from files stored at the server and dynamic data by executing a program when a request is received. Proxy caching stores data away from the originating server and reduces the load on the server by responding to requests from a local proxy thereby reducing network traffic and response time. The DynamicWeb cache [IC97] aims to reduce CPU workload on a server generating dynamic pages. It allows an application program to explicitly store, invalidate and update content thus ensuring that the cache is up to date. The Data Update Propagation [CID99] algorithm addresses the issue of precisely identifying which cached pages have become obsolete as a result of new information received by the system. Caching static data at proxies offers a significant performance advantage as data do not change frequently and, therefore, most requests can be served by proxies. However,

dynamic data tends to be computed by the server for each request and caching at proxies, therefore, seems meaningless. The Active Cache [CZB98] allows dynamic data to be cached at proxies that have server supplied cache applets attached to data. The proxies are required to execute the cache applet upon a hit to furnish the necessary computation without contacting the originating server.

The foregoing caching techniques assume a passive approach. For example, the caches are populated with content data in response to the historical request pattern. In an active approach, caches are populated with content data by predicting future request patterns based on content access history. Generally, the active approach has been applied to prefetching between clients-servers, clients-proxies and proxies-servers. The prefetching techniques for clients-servers require servers to predict which content is likely to be accessed in the near future based on access patterns of clients, and initiating prefetching based on this prediction [BC96, PM96]. In the case of clients-proxies prefetching, proxies predict which content data is likely to be required next and coordinates with clients to push or pull the content data [FCLJ99]. The prefetching technique for proxies-server entails proxies using local information to make a decision on which content to prefetch from the server or servers whereas servers use information on content specific request patterns from global clients to decide which content data should be pushed to proxies [CKR98, CY97, KLM97, MC98]. Finally, a hybrid prefetching technique uses information at the proxy as well as at the server to coordinate content data prefetching [CZ01].

2.8 Summary

In this chapter, we surveyed the research focusing on performance management of internet services, specifically single server, server ensemble, web server, geographical server replication and caching. An emerging trend of internet hosting centres co-hosting multiple internet services on the shared server ensemble offers utility-like performance characteristics. This thesis is concerned with the problem of server ensemble orchestration in the hosting centre. An internet hosting centre must orchestrate its servers amongst co-hosted services on an ongoing basis to satisfy performance guarantees and optimise its revenue stream. In the subsequent chapters, we will elaborate further on the challenges of server ensemble orchestration and propose a novel biomimetic technique to tackle such challenges.

Chapter 3

Server Ensemble

Computers are useless. They can only give you answers.

Pablo Picasso *(1881-1973)*

3.1 Introduction

The unrelenting growth of myriad internet services coupled with the ubiquitous Internet characterised by its open and dynamic environment presents key challenges for designing and managing internet computing infrastructure on which such services are hosted. This chapter provides the background to an emerging trend toward the building of centralised service hosting facilities from an ensemble of commodity servers and the virtual server system architecture that supports performance guarantees necessary for services on which users rely upon for day-to-day as well as business-critical activities.

The centralised internet hosting facility offers an outsourcing paradigm that entails providing infrastructure for a fee to host services and deliver its content over the Internet to its users whilst service owners provide content. The hosting arrangement is formalised by a contract that outlines performance guarantees on the part of the hosting centre and fee obligations on the part of content owners. Given the unpredictable nature of Internet request traffic, the hosting centre is motivated to maximise fees earned from co-hosted services. In this context, issues related to orchestrating a server ensemble are described with a formal description of the orchestration problem, concluding with a summary. The remainder of this chapter is organised as follows. Firstly, we describe details of the internet hosting centre, its commodity server appliances and the outsourcing model that is driving the shift towards co-hosting

services in a centralised facility. Thence, we describe the *de facto* virtual server architectural model that enables sharing of servers amongst co-hosted services in Section 3.3. Following this, in Section 3.4, component details of Service Level Agreements (SLAs) are described and outlines of the user's perception of service performance presented. Next, in Section 3.5, we describe the role of service performance in baulking behaviour of users as well as aggregated user behaviour in relation to using internet services. To conclude, in Section 3.6, we develop a mathematical formulation of the non-deterministic server ensemble orchestration problem.

3.2 Internet Hosting Centres

As discussed in Chapter 1, internet infrastructure has become central to human civilisation given the ubiquity of the Internet and wide range of service offerings. As a consequence, we are witnessing a paradigm shift in the way we think about computing infrastructure and its architecture. Traditionally, individual entities have offered services by building and operating the service infrastructure themselves. Historically, this is analogous to a trend observed during early years of electricity production where each business built and operated its own generator. Of course, we now know, in hindsight, that development of centralised power stations and the transmission grid enabled reliable and low cost access to the service [FK99]. We are witnessing a similar metamorphosis in the Internet computing infrastructure with the ubiquitous Internet providing global access and a recent emerging market trend indicating a shift towards the building of a centralised infrastructure facility, as depicted in Figure 3.1, to co-host diverse and wide ranging services [AT03, BT03, IBM03d].

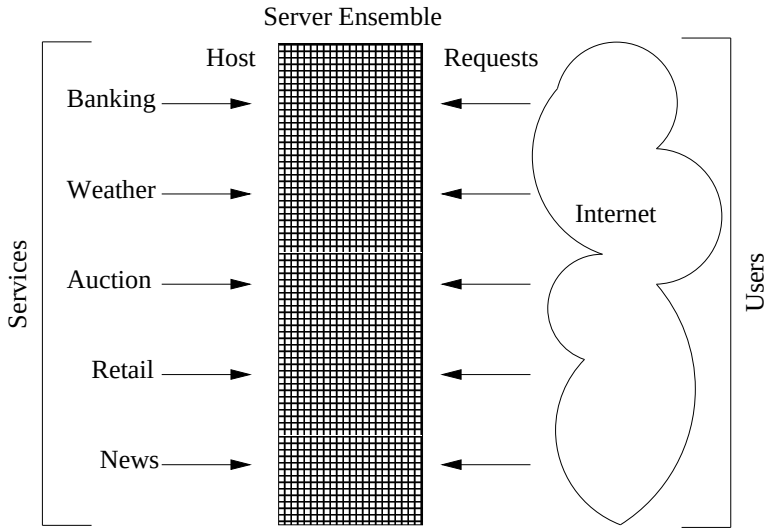


Figure 3.1: An internet hosting centre.

The prevailing architecture of choice for internet hosting centres is an ensemble of commodity servers such as rack-mounted or blade-server technology [App03, HP03a, IBM03a, IBM03c, SUN03a] on which multiple services are co-hosted. The hosting centre provides network access so that requests from global users can be routed to appropriate services. Each hosted service is accessed by users with a uniquely identifiable service name (for example `http://news.bbc.co.uk`).

The outsourcing paradigm entails hosting centres providing infrastructure for a fee to host services and delivering content to global users. The service owners provide content to hosting centres and update content on some regular or "as needed" basis. The driving shift behind this paradigm is the fact that the Internet is central to the core business strategy of many organisations that do not have the necessary capital to invest in such expensive infrastructure outlay, nor do they have the in-house Information Technology expertise to build and maintain such service infrastructure. Nevertheless, they wish to utilise complex and expensive software and hardware to provide a quality online experience to users (customers) of their services. Furthermore, technology advancement does not entail any investment on the organisation's part since it is the hosting centre's undertaking to offer advanced technology in order to maintain a competitive edge in the market place. Given the economy of scale due to resource sharing, the hosting centre benefits from opportunity to dynamically provision resources to meet the unpredictable demand for services which, in turn, shields content owners from over-provisioning costs [CATV01, Che99, JKK+01].

3.3 Virtual Server

An internet hosting centre co-hosts services by partitioning its server ensemble into clusters known as virtual servers. Each virtual server hosts a specific internet service and is composed of five key components: servers, switch, backplane network, service application and content data. A *de facto* architectural model, as depicted in Figure 3.2, for virtual server hosting Internet services has emerged [Bre00, Che99, DKMT96, JK00, KBM94, KMR95] to address challenges of building service infrastructure that possesses utility-like characteristics. Fundamentally, the functionality of each component in the architectural model is as follows:

Servers: A server in a virtual server is a "processing-engine" unit. It comprises CPU, memory and disks such that replication is possible. It is the basic unit of scaling (augmentation

or reduction) and replacement if failure occurs.

Switch: Maps external name of an internet service to physical names (Internet Protocol (IP) addresses) of each server in the virtual server.

Backplane: This is used for System Area Networking (SAN) amongst servers in the virtual server to provide for system management and content data update traffic.

Service-Application: This is the software that implements a specific functionality of a service e.g. stock trading and it is installed on each server of the virtual server.

Persistent Content Data: This is a replicated "database" across disks of servers. It may also include a RAID subsystem or network attached disks.

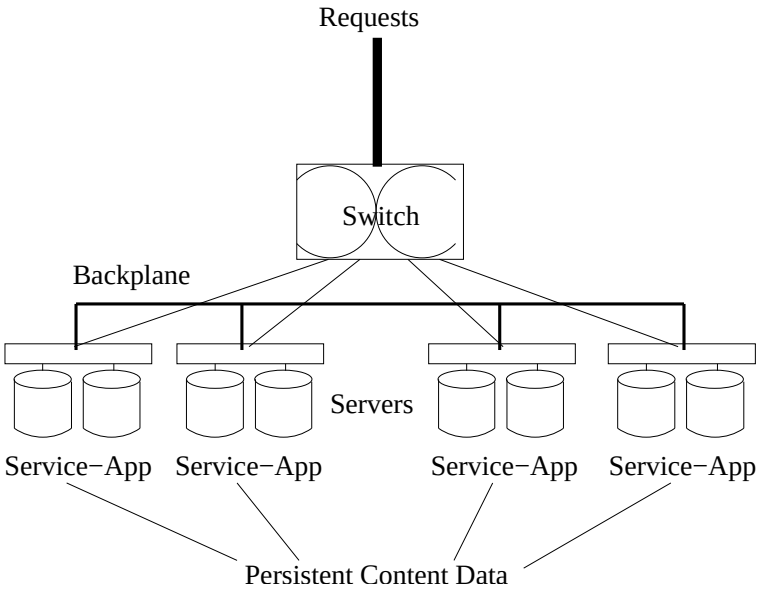


Figure 3.2: Architectural model of a virtual server.

The model in Figure 3.2 represents a single virtual server. A stream of requests submitted by users via the Internet drives the internet service. The notion of "request" is intrinsic to most Internet protocols such as HTTP, FTP POP and IMAP. Each request is regarded as independent though there can be a concept of session whereby state is shared amongst a group of requests. For example, HTTP does not have session as it is stateless; however, FTP requires authentication exchange which is remembered for the duration of the session in question.

The request stream is routed to a unique service name (virtual IP address) which is mapped to a switch that performs two functions:

i) Balances load by distributing incoming requests amongst servers of the virtual server.

ii) Operates the mapping facility to map the name of the internet service to physical names (IP addresses) of the server in the virtual server such that the external name of the internet service is preserved in the event of server failure.

These are essential for meeting performance guarantee and high availability requirements. The switch is reconfigurable such that the mapping facility recognises scaling (augmentation or reduction) of servers in the virtual server [ALT03, CIS03, HGKM98]. Given that the switch is in the path of the incoming request stream, it must be fault tolerant. As shown in Figure. 3.2, there are two switches in the path with an automated facility to divert requests from one to each other in case of a failure.

A server is the unit of scaling. It is an integrated system comprised of one or more CPU, memory and internal and/or external disks. A server ensemble is also known as a server farm and represents the technology of choice for building virtual server infrastructure for internet service deployment [IBM03b, HP03b, SUN03b]. In this context, the cluster technology offers numerous advantages over a single monolithic system such as Symmetric Multi-Processor systems (SMPs). It has the property to incrementally scale over time which is extremely beneficial since capacity provisioning is an art given the many unknown variables. Hence, capacity provisioning is supplanted by reciprocal seamless scaling. Moreover, clusters have built-in redundancy due to independent servers which results in high availability property. In an event of failure(s), a faulty server can be masked by software and replaced or repaired offline. Finally, given that the fundamental building block of a cluster is a commodity server, it is able to take advantage of the cost to performance ratio owing to tracking of leading edge technology for commodity components such as memory, disk and processor.

Each server in the virtual server is installed with system and webserver software [Apa03, Lin03, Zeu03] along with service application that provides service functionality to respond to requests. The persistent content data which is needed by the service application to fulfil service requests is replicated across all servers of the virtual server. In effect, each server has access to complete content data and, therefore, can fulfil any request from any user. The content data, depending on its size, may be partitioned as well as replicated across the server in the virtual server. In such cases, some requests may need to be directed to a specific server. This can be achieved by the switch having request-specific knowledge or by re-routing requests via the backplane network to an appropriate server. In addition, the backplane network allows the capability to manage nodes and update content data. Any server belonging to a

virtual server can be reallocated to another virtual server by reconfiguration, thus providing scalability and fault tolerance [Bre00, CATV01]. When a server is reallocated from one virtual server to another, it becomes unavailable by as much as 5–7 minutes due to the time involved in scrubbing the existing service application and data followed by installation of the new service application and data [AFF+01]. Given that servers are shared amongst virtual servers, security and privacy are vital issues for many content owners wishing to protect themselves against online fraud [Blo05, Ker04]. Consequently, servers must be scrubbed when being reallocated to avoid any leakage of sensitive information amongst virtual servers in the hosting centre. As far as the user audience of any service is concerned, the hosting centre appears as an individual virtual server.

3.4 Service Level Agreements (SLAs)

As previously described in Section 3.2, an emerging trend in internet service deployment is the outsourcing paradigm where an internet hosting centre manages the content delivery of multiple co-hosted services to global users. Under this paradigm, the content owners are clients who enter into an SLA with the hosting centre to purchase infrastructure resources for a fee to guarantee certain service performance. The hosting centre enjoys the economy of scale benefit by sharing its resources amongst the co-hosted services whilst individual content owners are shielded from maintenance and over-provisioning costs.

An SLA is a contractual obligation placed upon the hosting centre and the content providers. Typically, it is defined by the following key components [KKL+02]:

Parties: The primary parties involved in the agreement. This is the content deliverer (hosting centre) and the content provider (service owner).

Service: This defines the name and operational characteristics of the service.

SLA Parameters: These defines Quality of Service (QoS) parameters for the service.

Metrics: These defines how SLA parameters are measured.

Evaluation: This defines how to evaluate the SLA parameters in terms of sampling rate and measurement duration.

Guarantee: This defines when, in terms of time frame, the SLA parameters are guaranteed.

Violation: This defines how to detect a violation of the SLA parameters.

Actions: These defines corrective actions to be taken by both parties in case of a violation.

Fee: This defines the payment structure for hosting the service and providing QoS guarantees.

One of the common ways in which an SLA is negotiated between the hosting centre and the content provider involves two components: (i) servers to guarantee minimum QoS parameters and (ii) additional servers on a pay-as-you-go basis to handle request load peaks. In Table 3.1, we illustrate a typical SLA between the hosting centre and a content provider. Pay-per-request-served is a common way in which a pay-as-you-go SLA is negotiated and such an SLA implicitly acknowledges that limited servers, reallocation costs and uncertainty may lead to overloading and lost requests [JKK+01]. Consequently, the prime incentive of the hosting centre is to maximise the total value of requests served.

SLA Component	**Definition**
Parties	Hosting Centre: BT WebWorld Content Provider: MapQuest
Service	`www.mapquest.co.uk` `DriveDirService:GetDriveDir()`
SLA Parameter	Average request throughput
Metric	Number of requests served
Evaluation	Every 30 minutes over twenty-four hour period
Guarantee	Mon-Thu: 7am–7pm Fri-Sun: 7am–10pm
Violation	Average throughput $>$ 200,000 requests/hour
Action	Pay premium
Fee	Ave. throughput $\leq$ 200,000 req/hour: Fixed charge Ave. throughput $>$ 200,000 req/hour: Pay-per-request-served

Table 3.1: An example of a Service Level Agreement.

3.5 Internet Services: Performance Perception

It is anticipated that more than 750 million people will have access to Internet infrastructure in some form or shape in 2005 [BM00, Bre00, GWBC99, Wan99, ZFJ97]. Potentially, this implies a set of global users whose aggregate behaviour is very difficult to predict. An ever increasing fraction of human civilisation dependent on Internet computing infrastructure and the ubiquity of the Internet means that the Internet is an open and dynamic environment. It

is subject to dramatic changes in load as unconstrained demand for service can create sudden surges. It is not uncommon to experience daily and seasonal changes in request loads. In extreme cases, one may even experience a tsunami of requests which leads to the flash-crowd or hot-site-of-the-day phenomenon as experienced by news, entertainment and search engine services during a heavily advertised event or some catastrophe [BBC01, BBC03, KEY01, Tow03, Wig01].

There is some research reported on user perception in relation to usability of internet services [BKB00, BSD01, CF01, Nie00, RE01]. Specifically, a pertinent aspect of this research states that response delay, defined to be the elapsed time between mouse-click and delivery of content to the user, has a psychological implication on behaviour. It has been found to be highly influential in affecting user behaviour in terms of length of interaction and retention to a specific service. It is not unusual for users to render a service defunct owing to excessive delay and baulk [Bic97, KW01]. Table 3.2 illustrates insight on perception and behaviour of users according to Nielsen [Nie00]. Slow response times have a negative implication on user interaction with the service, attracting new users to the service and loyalty to service brand-name.

To date, numerous studies have been performed on trace logs of request arrivals from real internet services to gain an insight into behaviour of their users. The internet service for the sixteenth football World Cup held in France during twelfth June to twelfth July 1998 reveals a peak-to-trough request load fluctuation to be approximately 65:1. The service had been in operation from the month of May during which it experienced very light load. Starting from tenth June, request load increased significantly and exhibited bursts throughout the event and diminished toward the end of the event [AJ99]. This is a prime example of an internet service becoming very popular swiftly and just as swiftly becoming unpopular. Another prominent example is the effect of eleventh September 2001 World Trade Centre event on search and news services [Wig01]. A study of heavily accessed internet services during the 1998 Winter Olympics games in Japan revealed that it served 56.8 million requests per day at peak and 110,414 requests per minute. Similarly, the 1998 Wimbledon Tennis internet service served 145,478 requests per minute at peak. Analyses of trace data from the internet service of the 1998 Winter Olympics illustrate both light-tail and heavy-tail access patterns [SYZ00]. Internet traffic on Local Area Network (LAN) as well as Wide Area Network (WAN) was traditionally modelled as a Poisson process. However, studies by Crovella [CB97]

and Yang [Yan01] find that the traffic is self-similar (i.e. traffic that exhibits bursts above the mean for lengthy periods at a wide range of timescale). Similarly, a statistical analysis of data from a real call centre where user initiate a call to receive service concludes an inhomogeneous Poisson call arrival process that varies depending on date, time of the day and type of call among other factors [BGM+02].

Response Delay	**Perception/behaviour**
< 0.1 second	Users observe instantaneous response and exhibit total devotion
< 1.0 second	Users observe small response delay but remain devoted
< 10 seconds	Users become, somewhat, distracted but may still exhibit some devotion
> 10 seconds	Users become distracted, exhibit apathy and are likely to click-away

Table 3.2: Behaviour patterns of internet service users to response delay.

3.6 Server Orchestration Problem

The outsourcing paradigm, as discussed in Section 3.2, entails service owners providing content to an internet hosting centre who co-hosts multiple services on its server ensemble and delivers the content to users globally in response to requests submitted over the Internet. The hosting of a service is formalised by an SLA contract between the hosting centre and the service owner defining such components as fee and SLA QoS parameters. One of the common SLAs involves guaranteeing minimum QoS performance for a fixed fee and charging pay-per-request-served for handling requests in excess of the minimum. Given multiple SLAs associated with co-hosted services, uncertainty of future request arrivals and sharing of its finite server ensemble amongst hosted services, the hosting centre's prime incentive is to maximise the total value of request served. The maximisation of total revenue is a challenging operational problem because of the uncertainty and significant downtime associated with scrubbing, as discussed in Section 3.3, when a server is switched to serve requests for another hosted service.

The server ensemble orchestration, as depicted in Figure 3.3, involves orchestrating a finite number of servers in the ensemble to be a compositional part of any virtual server in the hosting centre at any time on a continual basis over a lengthy time horizon such that it leads to maximisation of revenue collection from negotiated SLA fees associated with each

co-hosted service. For the entire time horizon, orchestration decisions have to be made as to which of its servers in the ensemble are part of which virtual servers. At any moment, any virtual server may be augmented, reduced or left unchanged by orchestrating a server to leave a given virtual server and join another or orchestrate a server to continue to be part of a given virtual server. If a server is orchestrated to leave a given virtual server and join another then the server in question suffers a downtime during which it earns no revenue. All these orchestration decisions have to be made in the face of unknown and highly variable future request arrivals for each of its co-hosted services. The average time to serve a request and fee paid per request as per negotiated SLA may differ for each co-hosted service along with request queueing behaviour in terms of how long a customer is willing to wait for a response. A revenue opportunity is lost if a user baulks.

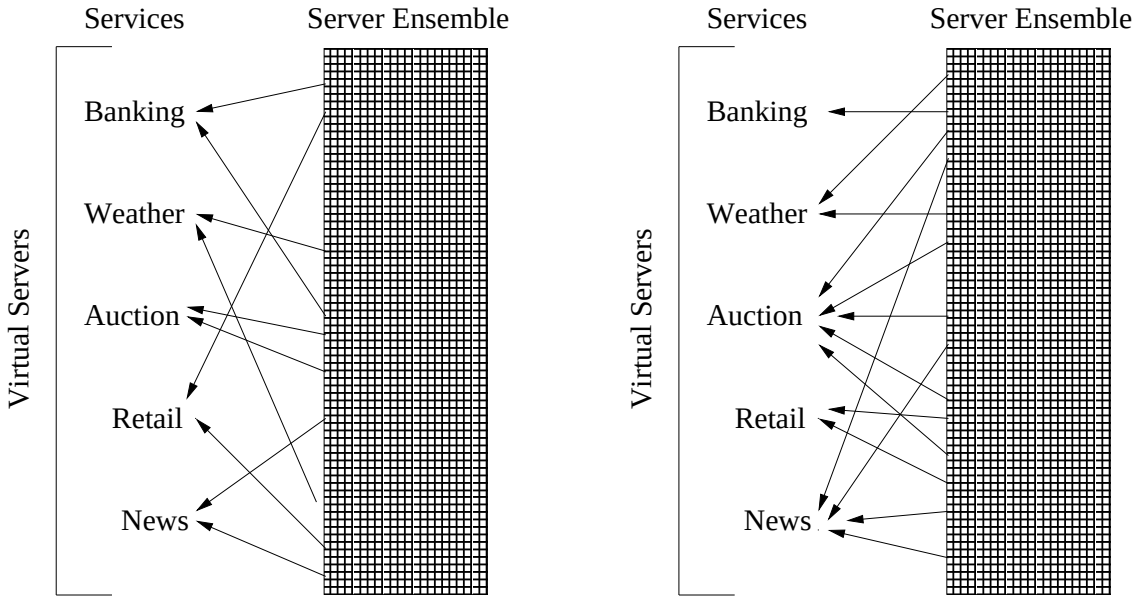


Figure 3.3: Server ensemble orchestration at two distinct time instants.

A mathematical model for the server ensemble orchestration is as follows. Assume an internet hosting centre co-hosting M internet services $I_1, \ldots, I_M$ on the server ensemble with a total of N servers. It follows that there are M virtual servers $V_1, \ldots, V_M$, hosting internet services $I_1, \ldots, I_M$, with respective service request queues $Q_1, \ldots, Q_M$ buffering streams of arriving requests. Each virtual server V_i comprises n_i servers, $n_i \geq 0$, such that

$$\sum_{i=1}^{M} n_i = N. \tag{3.1}$$

The unit of orchestration is a single server and the cost of orchestration is some fixed number

of time units (i.e. the server becomes unavailable for this fixed time interval as it undergoes re-purposing process). Assume pay-per-request-served SLAs for each co-hosted service and that server $s_j \in V_k$ serving requests from queue Q_k is paid v_k, value-per-request-served. Let the time horizon be divided into T time steps of equal duration indexed by $t = 1, \ldots, T$ and S_t denote the state of the system at the start of time step t, including the composition of virtual servers $V_1, \ldots, V_M$ with available servers from the ensemble and residual requests in the queues from time step $(t-1)$. Let $A_t = \{a_1^t, \ldots, a_M^t\}$ denote the request arrivals during time step t and $\pi = \{\pi_1^t, \ldots, \pi_M^t\}$ denote the server ensemble orchestration policy for time step t. We let $R(\pi, S, A)$ denote the revenue earned by an orchestration policy π during a time step with initial state S and arrivals A. Similarly, let $f(\pi, S, A)$ denote the state of the system which would eventuate at the start of the next time step, if the current time step starts in state S and orchestration policy π is used in the current step with arrivals A. The objective of the hosting centre is to find an orchestration policy π^t for each time step t over the entire time horizon split into $1, \ldots, T$ time steps such that the total revenue is maximised.

Maximise:

$\sum_{t=1}^{T} R(\pi_t, S_t, A_t)$ (revenue function)

Subject to:

$S_{t+1} = f(\pi_t, S_t, A_t)$ (Orchestration, System State, Request Traffic).

Since the A_t are not known in advance, the problem is not deterministic. Equivalently, define the server ensemble value function for time step t as follows

$$v^t(S_t, S_{t+1}) = \max_{\pi: S_{t+1} = f(\pi, S_t, A_t)} \{R(\pi, S_t, A_t)\} \tag{3.2}$$

with the corresponding optimal orchestration policy given by

$$\pi^*(S_t, S_{t+1}) = arg\max_{\pi} \{R(\pi, S_t, A_t)\}. \tag{3.3}$$

As before, the objective of the hosting centre is to find a sequence of orchestration policies

over the time horizon with $1, \ldots, T$ time steps such that the total value is maximised.

Maximise:

$$\sum_{t=1}^{T} v^t(S_t, S_{t+1}) \qquad \text{(value function)}$$

Subject to:

$$S_{t+1} = f(\pi_t, S_t, A_t) \quad \text{(Orchestration, System State, Request Traffic).}$$

3.7 Summary

This chapter introduced the server ensemble framework for co-hosting multiple internet services in hosting centres and discussed the key advantages driving the shift towards centralised internet hosting centres. We also discussed the open and dynamic nature of the Internet environment which can lead to unconstrained demand for service.

The key component of the framework is the concept of a virtual server architected from a switch and servers from the ensemble. Each virtual server in the hosting centre hosts an internet service which is replicated on its servers in order to provide fault tolerance and availability. Any server belonging to a given virtual server can be reallocated to another virtual server by reconfiguration thus providing scalability but the server in question suffers significant downtime due to time involved in scrubbing of existing service and installation of new service. When a server is reallocated, scrubbing is necessitated by the security and privacy concerns of content owners to protect themselves and their customers against online fraud and pernicious attack.

The outsourcing paradigm involves hosting centres delivering content of multiple co-hosted services to a global audience whilst content owners enter into an SLA contract with the hosting centre in order to guarantee service performance for a fee. The paradigm is beneficial to both parties as it enables the hosting centre to share its servers amongst co-hosted services and shield content owners from over-provisioning costs.

In the context of the server ensemble framework, outsourcing paradigm and the dynamic Internet environment, we described the server ensemble orchestration problem faced by internet hosting centres in order to maximise its revenue collection from the negotiated SLA contract fees and gave a mathematical formulation of the server orchestration problem. In the next chapter, we review self-organising inner workings of honeybee colonies to tackle

their forager orchestration problem to collect nectar from the surrounding countryside. We then propose a biomimetic technique inspired by honeybee colonies for the server ensemble orchestration problem.

Chapter 4

Biomimetic Server Ensemble Orchestration

It has yet to be proven that intelligence has any survival value.

Arthur C. Clarke *(1917 -)*

4.1 Introduction

This chapter addresses the server ensemble orchestration problem using a biomimetic approach that has inherent autonomic properties. Our inspiration for the technique is the self-organising characteristic of honeybee colonies to solve the difficult problem of orchestrating foragers amongst flower patches in the surrounding countryside to collect nectar required to ensure its survival. The internet hosting centres face an equally difficult problem of orchestrating servers amongst co-hosted services to collect revenue fees for serving respective requests. Both honeybee colonies and internet hosting centres must orchestrate their respective resources in the face of an uncertain future.

The chapter is organised as follows. First, we review other biomimetic techniques that have been inspired from organisation of social insect colonies. Next, in Section 4.3, we review the salient characteristics of social insect colonies and highlight their sophisticated collective behaviour that lends itself to solving difficult problems. In Section 4.4, we describe the honeybee colonies, the forager orchestration problem faced by the colonies, the self-organising inner workings of the colonies that solves the problem and a remarkable homology between the server and forager orchestration problems. In Section 4.5, we propose a biomimetic server or-

chestration algorithm inspired by the self-organising forager orchestration in honeybee colonies that autonomically responds to the dynamic environment and we describe key details of the mimicked mechanisms for the new algorithm.

4.2 Related Work

The terms biomimetic, biomimesis or biomimicry all refer to the fundamental idea of studying nature's models and processes that have evolved over millenniums and taking inspiration from such models to solve problems in the human domain. Sketches of human powered flying machines by Leonardo da Vinci are an early example of the biomimetic approach. The Velcro fastening system modelled on tiny hooks on weed seeds is, perhaps, the best example of a very successful biomimetic approach. Recent examples abound in material design, engineering, and medicine [Bio05].

Amongst the natural models that have been studied, social insect societies have attracted considerable interest from engineers faced with myriad complex engineering problems of optimisation, network routing and scheduling (see Bonabeau *et al.* [BDT99] and references therein). The social insect societies are capable of performing complex tasks through sophisticated collective behaviour that emerges from pedestrian members of society following simple rules without guidance from centralised supervisor. These characteristics lend themselves well to solving difficult engineering problems.

A class of algorithms often referred to as Ant Colony Optimisation (ACO) imitate ant colony models for solution to problems in the human domain. Proposed applications have enjoyed mixed success and include Travelling Salesman Problem [CDM91, DG97], Graph Colouring Problem [CH97], Quadratic Assignment Problem [GTD99, MC99] and Combinatorial Optimisation Problem [CDM$^+$]. Another class of algorithms often referred to as Ant Colony Routing (ACR) use the ant model for application to routing problems in communication networks. Examples include routing in telephone networks [SHBR96] and packet based network [CD98]. There are many examples of ant-inspired solutions to problems in scheduling and resource allocation in an excellent survey by Merloti [Mer04].

Our work is also biomimetically inspired but we use honeybee (*Apis mellifera*) colonies as the natural model to draw motivation for a solution to the problem of managing the server ensemble in an internet hosting centre. Specifically, we exploit an excellent homology

between the problem faced by honeybee colonies and internet hosting centres to propose a new algorithm. Our algorithm derives autonomic properties from the collective self-organising behaviour of honeybees.

4.3 Social Insect Societies

The history of the world, as classified by some zoologists involved in morphography, is divided into periods with each being named after the dominant type of animal. There have been periods dominated by single cell organisms, reptiles, mammals and so on. The period of the past few thousand years is distinguished by dominance of man. Even though insects have existed in abundance over a massive period of time stretching back to before the age of reptiles, no single period is attributed to the age of insects. Nevertheless, they are the only animal group that challenges human dominance given that their population dwarfs mammals in terms of species, individuals and total biomass. Insects such as bees, ants, wasps and termites living in colonies are known as social insects whereby a specialised caste carries out reproduction amongst non-reproductive members. In terms of organisation, insect societies are certainly worthy of being compared with certain aspects of human society. Their societies are characterised by kin altruism where individual behaviour for the benefit of the society and only the basic minimum for that individual.

Social insect societies (bees, ants, wasps and termites) have been studied extensively as a collective entity to understand general principles which govern their organisation [BDT99, Dum81, Lin67, Mor54, Ric70, See85, See95]. A colony performs various tasks including food collection, temperature regulation, defence, nest building, and reproduction. Examining honeybees collecting nectar from flower patches in the surrounding countryside, hunting expeditions involving up to 200,000 army ants, weaver ants building bridges to allow them to cross gaps by forming a chain with their own bodies and wasps and termites building complex nest structures raises an important question with respect to the collective organisation of insect societies given the limited ability of an individual insect.

The collective behaviour of a colony is the result of interaction amongst colony members and the external environment. An insect is equipped with numerous sensory receptors such that it can receive and process sensory inputs. It can receive large amounts of information and make a decision to adapt its behaviour accordingly. Fundamentally, insect societies use

direct and indirect communication mechanisms known as Signal and Cue respectively [BDT99, See95]. An interaction via Signal mechanism involves visual or chemical contact or food or liquid exchange, whilst a Cue mechanism involves an individual changing the environment and others responding to the new environment. Examples of Signal mechanisms include pheromone trails laid by ants or waggle dances performed by forager bees; an example of a Cue mechanism is the difficulty in unloading nectar collected from flower patches [TS94]. These simple communication mechanisms enable the colony to organise its members to engage in various tasks and respond to changes in the environment in a highly coordinated fashion– a sophisticated collective behaviour is borne out of interactions amongst individuals that are only capable of primitive behaviour [BDT99].

A salient characteristic of social insect colonies is the accomplishment of sophisticated organising behaviours through decentralised control. That is, no single entity is aware of the capability of the colony or overall plan. In this context, social insect societies are said to be self-organised. The idea that self-organisation is inherent in social insects offers a new paradigm for highly complex systems design and the opportunity to adopt insect-like behaviour to solve difficult problem in such systems. Below, we consider the honeybee colony for our biomimetic approach to the self-organising server ensemble orchestration in internet hosting centres.

4.4 Honeybee (*Apis mellifera*) Colony

The honeybee (*Apis mellifera*) indigenous to Europe, Africa and the Middle-East has been introduced to other part of the world by beekeepers. Colonies of honeybees have been studied extensively for many years and specifically, a significant volume of literature is devoted to understanding their social organisation (survey Seeley [See95] and references therein). The social organisation is adapted to local geographical climate conditions in order for the colony to survive and propagate. Particularly, in colder regions (Northern Europe and North America), the colony must accrue adequate honey to sustain itself through the winter. It must, therefore, coordinate its foraging activities in order to efficiently collect food during summer months when the surrounding countryside is replete with flora whilst facing competition from other foraging insects.

Typically, a honeybee colony comprises approximately 20–50 thousand bees with one queen, a few drones and the rest workers. Depending on the age of a worker bee, a worker bee

specialises in performing different tasks that range amongst cleaning, maintenance, guarding, temperature regulation and foraging for food (nectar, pollen, water). During infancy, worker bees serve the queen and nurse the young whilst adulthood is spent in temperature regulation, food storing and foraging endeavours. Approximately 25% of worker bees are engaged in foraging to satisfy an annual requirement of up to 120kg of nectar, 20 kg of pollen and 20 litres of water. It is evident that nectar collection is a principal foraging activity given that it needs approximately 50 kg of nectar as a reserve for winter survival without which the queen freezes and the colony dies [See95]. During summer, the colony consumes approximately 70 kg of nectar for brood plus workers. A significant portion of this nectar is required to meet the energy demands for foragers that explore at least 100 km^2 of the surrounding countryside in search and collection of food.

4.4.1 Forager Orchestration Problem

In colder climate zones such as Northern Europe, the blossoming of flora that yields nectar is regulated by seasonal and daily variations. At the seasonal level, winter months are devoid of any flora whilst summer months are marked by flora repleteness. On a daily level, the availability and quality of nectar vary directly with micro-climatic conditions, blooming and withering cycle and exploitation. Consequently, within the space of a few days, conditions may give rise to a dearth or an abundance in nectar supply. Effectively, nectar is available during summer for about ten weeks each year. During this period, the availability of nectar is unpredictable and the scale of fluctuation from dearth to abundance is unknown. It is not uncommon for a colony to experience abundance-to-dearth nectar in-take rate fluctuation by a factor of more than 100:1 within a day or so [See95].

During summer months when flowers are in bloom, a honeybee colony must orchestrate its foragers amongst an array of flower patches in the surrounding countryside to collect nectar in accordance with requirements of the colony. Given the volatility of nectar supply, the colony must adapt to the dynamic environment and make time varying orchestration decisions, as depicted in Figure 4.1, on the number of foragers to be deployed to each known flower patch in order that nectar can be collected quickly and efficiently. Of course, a flower patch that offers good quality nectar in abundance as well as with an ease of collection should have a higher number of foragers deployed whilst patches that are small and offer sparse opportunity to collect nectar should have very few foragers deployed. The colony can

decide to orchestrate a forager from one flower patch to another but the forager in question requires time to learn the location of the new flower patch and during this time it makes no contribution to colony's nectar collection endeavours. Therefore, the honeybee colony must solve the forager orchestration problem which entails making ongoing orchestration decisions on deployment of its finite foragers to flower patches in the surrounding countryside in face of flower volatility and uncertainty of nectar supply.

Consider a honeybee colony with N nectar forager and M flower patches. Let function $f_i(x_i)$ denote the value returned from the ith flower patch with x_i foragers collecting nectar at the patch. Provided the value functions $f_i()$ do not change over time then the resulting static forager orchestration problem is

Maximise:

$$\sum_{i=1}^{M} f_i(x_i) \quad \text{(Value function)}$$

Subject to:

$$x_i \geq 0 \quad \text{(Orchestration)}$$

$$\sum_{i=1}^{M} x_i \leq N \quad \text{(Foragers).}$$

Applying the Karush-Kuhn-Tucker condition [Wak99] yields the optimal static solution

$$f_i^{'}(x_i) = \lambda \ \forall x_i > 0; f_i^{'}(0) \leq \lambda \ \forall x_i = 0. \tag{4.1}$$

That is, the marginal contributions are equal at each flower patch. This is intuitive since if the marginal contribution $f_i^{'}(x_i)$ at flower patch i exceeded the marginal contribution $f_j^{'}(x_j)$ at another flower patch j then more nectar could be obtained by orchestrating a forager from flower patch j to i.

4.4.2 Self-Organising Forager Orchestration

Colonies of social insects possess what has been classed as Swarm Intelligence [BDT99]. A broad definition of the term implies a sophisticated collective behaviour borne out of primitive interactions amongst members of the group to solve problems beyond the capability of individual members. Such colonies are characterised by:

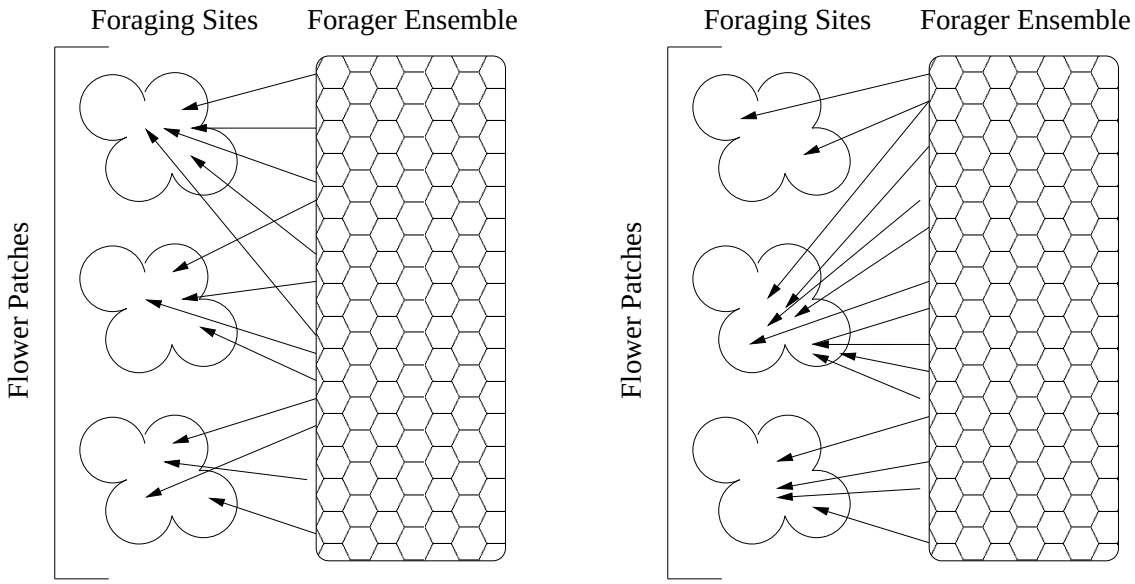


Figure 4.1: Forager orchestration at two distinct time instants.

i) Self-Organisation: decentralised and unsupervised coordination of activities.

ii) Adaptiveness: response to dynamically varying environments.

iii) Robustness: accomplishing the group's objective even if some members of the group are unsuccessful.

A model of self-organising behaviour that takes place within a colony of honeybees has been presented by Seeley [See95]. The model describes interactions between members of the colony and the environment that leads to dynamic distribution of foragers to efficiently collect nectar from an array of flower patches that are capricious in terms of their value to the colony. The honeybee colony, as depicted in Figure 4.2, acquires information with respect to foraging prospects in the surrounding countryside continuously and acts upon this information in concert with nectar requirements of the colony to orchestrate foragers amongst prospective flower patches. The self-organising behaviour model comprises the following key elements:

Information Procurement: A honeybee colony procures two different types of information. It must not only continually refresh information with respect to flower patches currently being capitalised by the colony but also procure information with respect to any new enticing flower patches.

1. Forager: A forager bee engaged in nectar collection at a flower patch returns to the beehive not only with nectar but also with information about the patch. Since

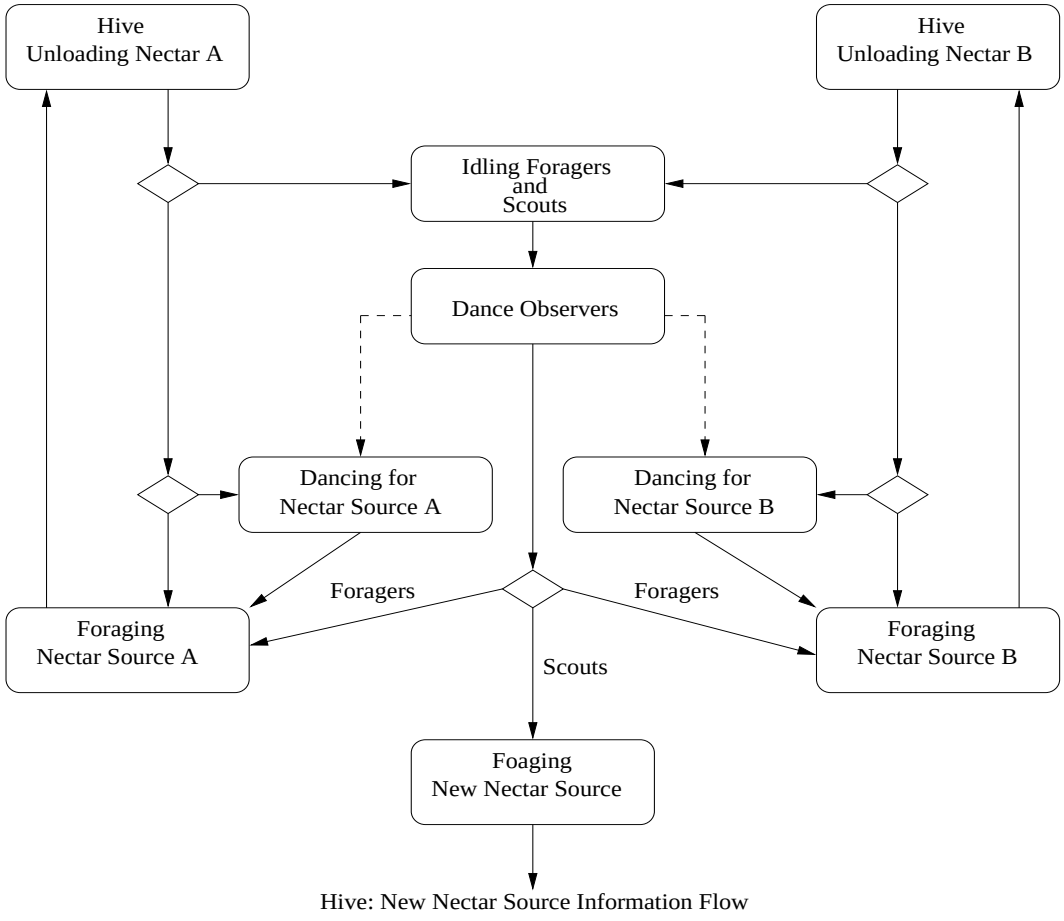


Figure 4.2: The information flow and decision processes in a honeybee colony (adapted from Seeley [See95]).

a typical colony has several thousand bees foraging, the colony can procure and continually refresh information about current foraging assets.

2. Scout: Amongst several thousand foraging bees, a small percentage($\approx$10%) of bees are scouts. Scout bees behave differently from foragers in that scouts are engaged in unearthing new flower patches and, therefore, return to the beehive with information about possible new foraging assets that may prove to be profitable to the colony.

Information Type: On each visit to a flower patch, a forager bee returns to the hive with two types of information along with nectar:

1. Value: During the process of nectar collection at a flower patch, a forager bee utilises her local wisdom about the patch defined by variables such as nectar sugar concentration, nectar bounty and distance from the hive and an internal scale of

quality to determine patch value rating.

2. Location: In addition to value rating, a forager also has information about the geographical location of the flower patch in relation to the position of the sun and the beehive.

Information Filtering: Given the continual influx of a large volume of information to the beehive about newly discovered and known flower patches, the colony ensures that information pertaining to high value rated patches is broadcast whilst discarding information about relative low value rated patches. This has an implication that the information broadcast is restricted to the most valued patches available to the colony:

1. Foraging Status: Each foraging bee returning to the hive must find a receiver bee to offload collected nectar. It is the job of receiver bees to store nectar in the beehive and the time it takes to find a receiver bee provides a cue as to the nectar influx rate of the colony.
2. Filtering Threshold: Based on nectar influx rate cue as well as variables such as current climatic conditions and time of the day, forager bees adapt their sensory-threshold which decides whether information pertaining to her patch is to be broadcast or suppressed. Generally, the threshold is higher when the nectar influx rate is high and vice-versa.

The filtering mechanism enables a colony to capitalise on the most valued patches when the supply of nectar is abundant and be selective as to where foraging effort should be concentrated whilst capitalising on a wide range of low valued patches during periods of nectar scarcity.

Information Broadcast: Each forager, having delivered the nectar payload on her return to the beehive and, in the process, set its filtering threshold, will decide whether to broadcast information pertaining to her patch or not. If the threshold is crossed then she will broadcast the information using the communication signal and medium at her disposal:

1. Signal Mechanism: To disseminate information about her patch, foragers use a signal mechanism known as waggle dance.

2. Signal Medium: Forager bees use an area located inside the beehive near the entrance known as the dance floor to convey her patch information.

Information Emphasis: If the criteria for dancing are satisfied, forager bees combine the decision to waggle dance with a value rating to present the information that conveys the degree of goodness and location of her patch:

1. Dance Potency: A forager bee expresses the goodness of her patch via a series of waggle runs that constitute the waggle dance. If a patch is deemed to be highly valuable then it will elicit a waggle dance comprising of greater number of waggle runs. Effectively, the value of a flower patch is represented by dance duration.
2. Patch Location: A forager bee expresses her flower patch location by the direction of waggle runs on the dance floor. If the patch is located directly in line with sun then this is expressed by waggle run in straight vertical direction. Any angular deviation from this vertical represents an angle between the sun (azimuth) and a flight direction from the beehive to the patch.

Information Audience: Given the continual bombardment of information on a colony pertaining to known and newly discovered flower patches, there are some foragers that are consumers whilst others are oblivious to this information.

1. Oblivious: There are two groups of foragers:
   (a) This is a group of active foragers that are currently busy collecting nectar from flower patches. Each member of the group has a local knowledge of their own patch and does not use the global information available.
   (b) This is a group of scout foragers that do not use the global information available. Instead, they tend to explore independently for new flower patches.
2. Consumer: This is a group of idling foragers that use the available global information and are potential candidates for deployment to flower patches.

Information Utilisation: The behaviour of foragers known as consumers is driven by the process of sampling and deciphering the global information being presented on the dance floor:

1. Dance Sampling: Even though an array of waggle dances are being performed on the dance floor, a forager does not take the opportunity to review and select the dance that represents the most valuable patch but, instead, randomly select a dance.

2. Forager Response: Having selected a dance at random, it follows the dance carefully for several waggle runs to learn the location of the patch and leave the hive to collect nectar. There may be several attempts before it successfully finds a patch with each failure resulting in return to the hive and following a randomly selected dance.

Fundamentally, the self-organising forager orchestration in a honeybee colony emerges due to information flow into the colony and its response to this information. The information and its behaviour is a consequence of primitive interactions amongst members of the colony and the surrounding country side using signals and cues. Basically, foraging bees visiting flower patches return to the hive with nectar and with a value rating of respective flower patches. At the hive, forager bees interact with receiver bees to offload collected nectar. This interaction provides feedback on the current status of nectar flow into the hive. This feedback mechanism sets a response threshold for an enlisting signal. An amalgamation of response threshold and value rating (function of nectar quality, nectar bounty and distance from the hive) influences the length of the enlisting signal known as the waggle dance. The waggle dance is performed on the dance floor where inactive foragers can observe and follow. Effectively, each active forager bee provides feedback on her local flower patch while observing bees have access to the set of attractive food sources being capitalised upon by the colony. However, individual foragers do not acquire the full set of global knowledge but rather randomly select a dance to observe from which they can learn the location of the flower patch and leave the hive to forage.

The self-organising proportionate orchestration pattern, derived from multiple and proportionate feedback on goodness of food sources, is described by Seeley *et al.* [SCS91] and validated by experimental study on a natural honeybee colony. The model given by Bartholdi *et al.* [BSTV93] predicts a steady state pattern of forager orchestration where rate of value accumulation equalises amongst forage sites being exploited. For instance, self-organising orchestration results in equal average nectar return $\frac{f_i(x_i)}{x_i} = \mu \ \forall i$ where $f_i(x_i)$ denotes the value returned from x_i foragers collecting nectar at the ith forage site.

Intriguingly, this allocation pattern is not optimal (unlike the famed hexagonal comb structure), although Bartholdi *et al.* [BSTV93] proved that no allocation is more than twice as efficient. Why would a sub-optimal foraging orchestration mechanism evolve in honeybee colonies? We suggest that the static sub-optimality is related to the dynamic nature of the foraging environment. Flower patch availability and quality may wax and wane rapidly even during a single day. We believe that a honeybee colony is quite adaptive to changes in flower patch availability and quality and that its orchestration method is considerably less sub-optimal when evaluated in a realistic dynamic environment rather than a static one. This performance quality was a key motivation for our biomimetic algorithm. One possible explanation for a performance difference is that, as noted previously, static optimisation requires equalisation of derivatives (marginal rates). That is, static optimisation requires $f_i'(x_i) = \lambda$ for all active patches i. This would seem to require a marginal rate bee for each patch, a bee who acquires nectar value at rate $f_i(x_i) - f_i(x_i - 1)$, thus limiting migration rates to one bee at a time. It also would seem to require less decentralised control. If this explanation is correct, the honeybee colony trades-off static optimality for adaptiveness. It has no marginal rate bee, but it has the ability to migrate many bees at once, increasing its responsiveness to changed circumstances.

4.4.3 Homology: Internet Hosting Centre and Honeybee Colony

A cursory glance at the server and forager orchestration problems, as depicted in Figure 4.3, would immediately suggest an obvious similarity between the two problems, at least on a superficial level. The orchestration of servers amongst co-hosted internet services to collect SLA revenue in an internet hosting centre parallels the orchestration of foragers to collect nectar from flower patches in the surrounding country side in a honeybee colony. The finite server ensemble in an internet hosting centre is analogous to forager bees in a honeybee colony whilst the service request queues pertaining to co-hosted internet services are analogous to sources of nectar to be exploited profitably. However, an indepth examination of server and forager orchestration problem at finer granularity revealed a remarkably close mapping between the two problems as depicted in Table 4.1. The strong homology between the server and the forager orchestration problems motivated our biomimetic approach.

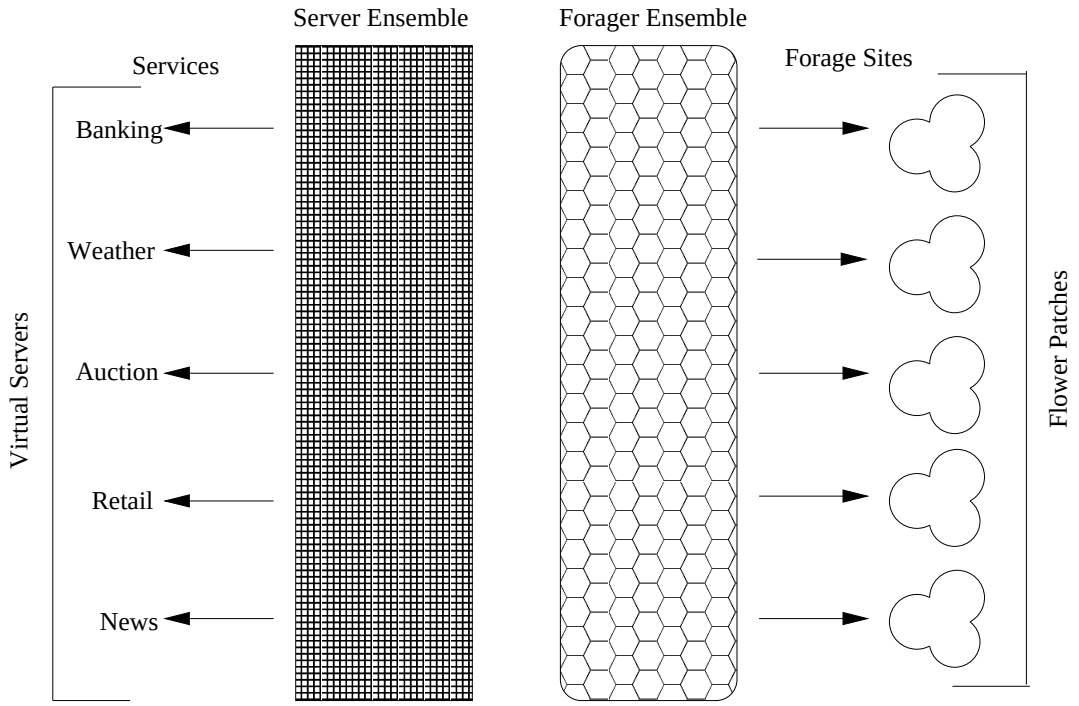


Figure 4.3: Server and forager orchestration problems.

An internet hosting centre co-hosting multiple internet services on its finite server ensemble is analogous to a honeybee colony with a certain number of bees foraging at multiple forage sites in the surrounding countryside. The amount of time required by a server to earn revenue from a hosted internet service will depend on the service characteristics as well as the total number of servers servicing the said request queue. If more servers than necessary are orchestrated to a service, the revenue rate will degrade due to each server having to wait longer to find a request to serve. Similarly, the amount of time needed by a forager to return with a nectar payload depends on characteristics of the forage site as well as the number of foragers collecting nectar at the said flower patch. The rate of nectar payload being returned to the colony will degrade if more than necessary foragers are orchestrated to forage the patch since each forager will take much longer to find flowers whose nectar has not been harvested yet.

Formally, the above homology can be described as follows. In the case of a honeybee colony, let function $f_i(x_i)$ denote nectar returned per second by x_i bees foraging at ith flower patch whose nectar replenishment rate per second is denoted by $\mathcal{N}_i$. The rate of nectar return from patch i with one bee collecting nectar will thus be given by $f_i(1)$. Now, if two bees are collecting nectar from patch i then there is a possibility of interference due to the activity of each bee. That is, consider a situation where the first bee collects nectar from a specific flower

Homology	
Internet Hosting Centre	**Honeybee Colony**
An ensemble of N servers	An ensemble of N nectar foragers
Unit of orchestration: Single server	Unit of orchestration: Single nectar forager
Co-host M internet services to be accessed by requests	M distinct flower patches in the surrounding countryside.
The group of servers (virtual server) with the same internet service application serving requests from a specific service queue	The group of nectar foragers collecting nectar at a specific flower patch
Time to service a request will be dependent on the internet service application	Time to travel will be dependent on the location of the flower patch
(*)Time taken by a server to find a request to serve at an internet service application increases if number of servers in the group increases	(+)Time taken by a forager to collect nectar at a flower patch increases if number of foragers in the group increases
(*)A server orchestrated to another internet service incurs downtime due to purging of current and installation of new internet service application and data	(+)A forager orchestrated to another flower patch incurs downtime due to time spent learning the location and successful discovery of the patch
(*)An internet service application and its requests have a value-per-request	(+)A flower patch has quality of nectar (sugar concentration)
Variable rates of requests arrival and baulking behaviour for each service	Variable rates of quality, density and replenishment for each flower patch
The flagged(*) items change unpredictably over time for each service	The flagged(+) items change unpredictably over time for each flower patch

Table 4.1: Parallels between Server and Forager orchestration problems.

whose nectar will be replenished with some delay. Meanwhile, the second bee arrives at the patch and while foraging chooses to collect nectar from the said flower with an outcome that it will fail to find any nectar and will lose time. Therefore, the nectar return rate with two bees collecting nectar at patch i will be $f_i(2) = \frac{2 \cdot f_i(1)}{2+\epsilon}$ where ϵ represents the small interference effect which increases the round trip time or, approximately $f_i(2) \approx 2 \cdot f_i(1)$. Similarly, $f_i(3) \approx 3 \cdot f_i(1)$, $f_i(4) \approx 4 \cdot f_i(1)$ and so on. The interference effect will slightly reduce the true value of nectar return rate. Notice that the interference effect increases with x_i because more bees will deplete more flowers at a given patch. Suppose we have x_i bees collecting nectar at patch i such that $x_i \cdot f_i(1) << \mathcal{N}_i$. Then nectar return rate will be roughly

$$f_i(x_i) \approx x_i \cdot f_i(1) \tag{4.2}$$

with slight interference effects. If x_i bees are collecting nectar at patch i such that $x_i \cdot f_i(1) = \mathcal{N}_i$ then there will be interference effect so that $f_i(x_i) < \mathcal{N}_i$ and nectar return rate will be approximately

$$f_i(x_i) \approx \mathcal{N}_i. \tag{4.3}$$

Finally, suppose there are x_i bees collecting nectar at patch i such that $x_i \cdot f_i(1) > \mathcal{N}_i$ then nectar return rate will also be

$$f_i(x_i) \approx \mathcal{N}_i \tag{4.4}$$

and $f_i(x_i) \to \mathcal{N}_i$ as $x_i \to \infty$. Similarly, mean nectar return rate $\frac{f_i(x_i)}{x_i} \to \frac{\mathcal{N}_i}{x_i}$ as $x_i \to \infty$. The plot of $\frac{f_i(x_i)}{x_i}$ against x_i will exhibit a slight negative slope due to the interference effect and

$$\lim_{x_i \to \infty} \frac{f_i(x_i)}{x_i} = \frac{\mathcal{N}_i}{x_i} = 0. \tag{4.5}$$

Now, consider an internet hosting centre and let function $\mathcal{R}_i(n_i)$ denote the revenue rate per second of n_i servers in the virtual server V_i serving requests from Q_i service queue. The hosted service on virtual server V_i has an SLA with pay-per-request-served fee schedule such that request arrivals provide the revenue replenishment. We denote revenue replenishment rate per second of service queue Q_i as $\mathcal{V}_i$. Reasoning in the same manner as we did for the honeybee colony, the revenue rate with one server serving requests from service queue Q_i will be given by $\mathcal{R}_i(1)$. Whilst two servers serving requests from Q_i give revenue rate $\mathcal{R}_i(2) \approx 2 \cdot \mathcal{R}_i(1)$ since each server simply takes an available request from the queue and services it with only a slight interference effect $\delta > 0$. For $n_i = 2$, each server simply takes an available request from the service queue and services it. There is only a slight interference effect which occurs when only exactly one request is present in the service queue. The actual resulting revenue rate $\mathcal{R}_i(2) = 2 \cdot \mathcal{R}_i(1) - \delta \approx 2 \cdot \mathcal{R}_i(1)$. The revenue rate from Q_i as servers are added to virtual server V_i will be $\mathcal{R}_i(3) \approx 3 \cdot \mathcal{R}_i(1)$, $\mathcal{R}_i(4) \approx 4 \cdot \mathcal{R}_i(1)$ and so on. Let n_i servers serve requests from service queue Q_i such that $n_i \cdot \mathcal{R}_i(1) << \mathcal{V}_i$. Then the revenue rate will be approximately

$$\mathcal{R}_i(n_i) \approx n_i \cdot \mathcal{R}_i(1). \tag{4.6}$$

The interference effect will still be small since it occurs only when there are at least one but fewer than n_i requests are present in the service queue. Note that the interference effect increases in n_i, as it does with the bees. If n_i server are serving requests from service queue

Q_i such that $n_i \cdot \mathcal{R}_i(1) = \mathcal{V}_i$ then revenue rate per second will be given by

$$\mathcal{R}_i(n_i) \approx \mathcal{V}_i. \tag{4.7}$$

Finally, suppose there are n_i servers serving requests from service queue Q_i such that $n_i \cdot \mathcal{R}_i(1) > \mathcal{V}_i$, then revenue rate will also be

$$\mathcal{R}_i(n_i) \approx \mathcal{V}_i \tag{4.8}$$

and $\mathcal{R}_i(n_i) \to \mathcal{V}_i$ as $n_i \to \infty$. Similarly, mean revenue rate $\frac{\mathcal{R}_i(n_i)}{n_i} \to \frac{\mathcal{V}_i}{n_i}$ as $n_i \to \infty$. The graph of $\frac{\mathcal{R}_i(n_i)}{n_i}$ against n_i exhibits a negative slope which is due to the interference effect and

$$\lim_{n_i \to \infty} \frac{\mathcal{R}_i(n_i)}{n_i} = \frac{\mathcal{V}_i}{n_i} = 0. \tag{4.9}$$

A stream of requests arriving over the Internet at a hosting centre for co-hosted internet service is buffered by respective service queues. The equivalent in a honeybee colony are available flower patches (forage sites) in the surrounding countryside. This is intuitive since servers collect SLA revenue for serving requests from the service queues whilst foragers collect nectar from flowers at the forage site. The aptness of the analogy is further demonstrated by similarities in variability of request stream behaviour and flower patch volatility. Request streams are variable due to fluctuating request arrival rates and baulking behaviour of users waiting for service whilst flower patches are variable due to fluctuations in micro-climate, quality, density and nectar replenishment rates.

In an internet hosting centre, a virtual server is composed of one or more servers collecting revenue from its service queue. The unit of augmentation or reduction is a single server. In a honeybee colony this is equivalent to the set of forager bees collecting nectar from a specific flower patch. The unit of augmentation or reduction is a single forager bee. When a server is orchestrated to leave a given virtual server and join another virtual server, it incurs a downtime penalty since it undergoes a re-purposing process whereby current internet service application and associated data are purged and new internet service application and associated data are installed. During the re-purposing process, the said server makes no contribution to the revenue earning endeavour of the hosting centre. Similarly, when a forager is enticed to forage at a particular flower patch, it must follow a waggle dance to learn the location of

the flower patch. This learning process usually requires multiple attempts [See03]. During the time spent in learning the location and subsequent discovery of the flower patch, the said forager makes no contribution to the nectar collection endeavour of the colony [SCS91].

Each virtual server hosting an internet service in a hosting centre has an associated SLA contract with the service owner which defines value-per-request-served amongst other components whilst each flower patch has an associated nectar quality (mol./L sugar concentration). Thus, a server in a hosting centre collects some value-per-request-served whilst a forager bee in a honeybee colony collects some mol./L sugar concentration from the nectar.

Finally, the time to service a request in a hosting centre depends on the application of the internet service and the time to find a request to serve will be a function of the number of servers serving the said request queue as well as the request arrival pattern. In a honeybee colony, the travel time to a specific forage site depends on its location in the countryside and the time needed to collect nectar will be a function of the number of foragers foraging at the said forage site as well as the rate of nectar replenishment of the flowers at that site. In a hosting centre, different internet services will have different values of service-time-per-request and value-per-request-served as per negotiated SLA contract. In a honeybee colony, different forage sites in the countryside have different values of forager-round-trip-time due to distance travelled by foragers to location of forage sites from the beehive and different nectar quality.

4.5 Self-Organising Server Orchestration Algorithm

In this section, we present an autonomic server ensemble orchestration algorithm for internet hosting centres based on the biomimetic approach. Principally, we were motivated by the remarkable homology between an internet hosting centre and a honeybee colony. The homology exists not only at the superficial level but at the finest granularity level as outlined in Section 4.4.3.

Let an internet hosting centre co-host M internet services on respective virtual servers composed from a server ensemble with N servers as previously described in Section 3.6. There are M virtual servers $V_1, \ldots, V_M$ with respective service queues $Q_1, \ldots, Q_M$ that buffer stream of arriving requests. Each virtual server V_i is composed of $n_i \geq 0$ servers with $\sum_{i=1}^{M} n_i = N$. The unit of orchestration is a single server (i.e. a server can be orchestrated such that it ceases to be part of V_i and becomes part of V_j $i \neq j$). However, the cost of orchestration is

unavailability due to some fixed downtime whilst the server in question is re-purposed to serve requests from Q_j. Each hosted service has an associated pay-per-request-served SLA such that a server $s_i \in V_k$ serving requests from service queue Q_k is paid p_k pence-per-request-served.

As in the case of the forager orchestration in a honeybee colony, the self-organising server ensemble orchestration algorithm must acquire information on an ongoing basis in respect of prospective co-hosted internet services and request load on their service queues. It must act upon this information to orchestrate the server ensemble amongst hosted services as depicted in Figure 4.4. In order to gather relevant information, we mimic, as depicted in Table 4.2, signals, cues as well as other mechanisms of a honeybee colony for an internet hosting centre.

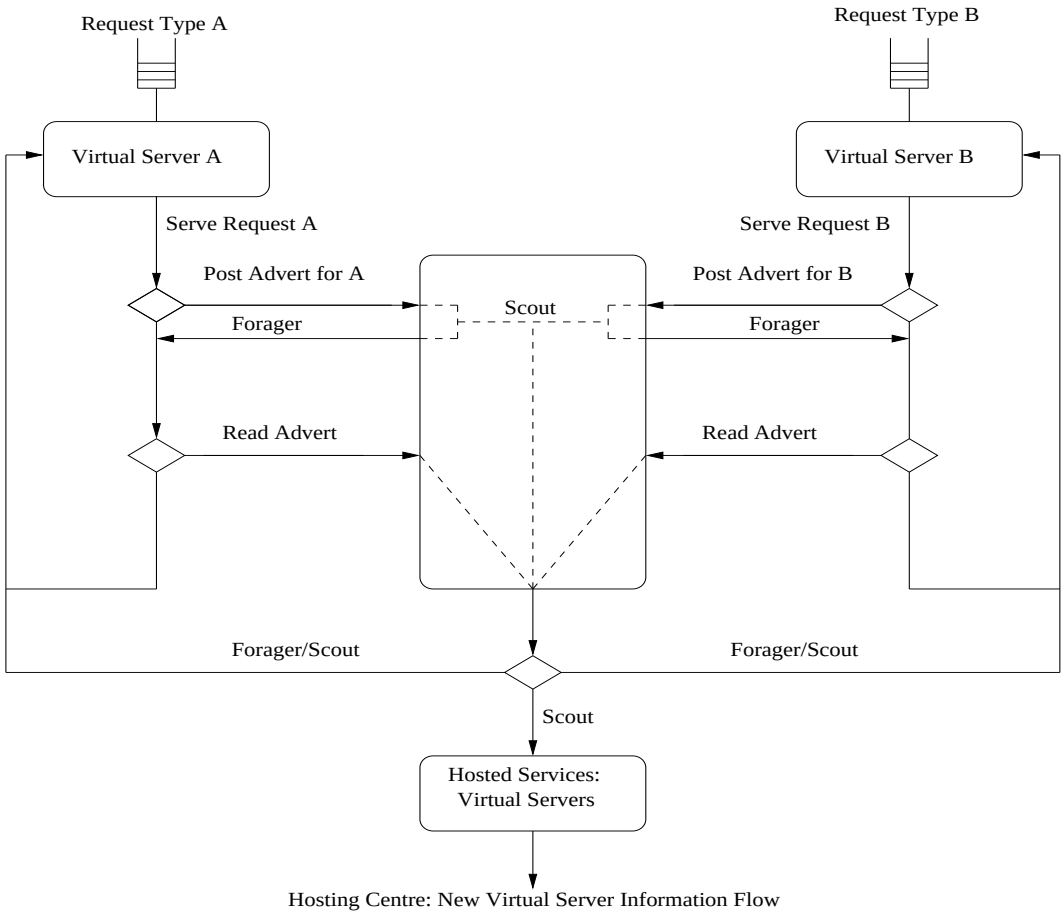


Figure 4.4: The information flow and decision processes in an internet hosting centre.

Below, we propose a biomimetic model for the server ensemble orchestration algorithm comprising key elements:

Information Procurement: A server in an internet hosting centre is defined as either a

Bio-Mimicry	
Honeybee Colony	**Internet Hosting Centre**
Waggle dance	Advertisement
Dance floor	Advertisement board
Dance duration	Advertisement duration
Flower patch location	Hosted service identifier
Following a waggle dance	Reading an advertisement
Performing a waggle dance	Posting an advertisement

Table 4.2: Bio-mimicking mechanisms of a honeybee colony.

forager or a scout server. The designation governs its unique behaviour as in a honeybee colony. However, a server $s_i \in V_j$ defined as either a forager or a scout will generate update information on the internet service hosted on virtual server V_j.

Information Type: A server $s_i \in V_j$ defined as either a forager or a scout generates two types of information:

1. Earning Rate: A server $s_i \in V_j$ generates Earning Rate E_i for a total number of requests, R_i, it served over a period T given p_j pence-per-request-served SLA

$$E_i = \frac{p_j R_i}{T}. \tag{4.10}$$

2. Service Identifier: A server $s_i \in V_j$ is aware that it belongs to virtual server V_j and, therefore, can generate service identifier.

Information Filtering: A server $s_i \in V_j$ generates information for each request it serves from Q_j. However, it randomly decides based on a threshold which information to advertise or suppress akin to dance threshold in a honeybee colony.

Information Broadcast: A server $s_i \in V_j$ has a communication signal and the medium at its disposal to broadcast information about internet services hosted on V_j and the service queue Q_j:

1. Advertisement: To disseminate information, an advertisement object is generated.
2. Advertisement Board: An advertisement object is posted on the advertisement board where it can be seen by servers in an internet hosting centre.

Information Emphasis: A server $s_i \in V_j$ serving request from service queue Q_j having reached a decision to advertise will do so in the following manner:

1. Advertisement Duration: This determines the length of time, D, an advertisement will be displayed on the advertisement board. The duration of the advertisement is given by:

$$D = p_j S \tag{4.11}$$

where p_j is pence-per-request-served SLA at V_j and S is the advertisement scaling given by

$$S = L\left(\frac{\text{Wait Time}_{\text{hosting centre}}}{\max(p_1, \ldots, p_M)}\right) \tag{4.12}$$

with L denoting a duration factor and Wait $\text{Time}_{hostingcentre}$ is the mean waiting time experienced by served requests across all hosted internet services over a given time interval.

2. Advertisement Identifier: A server $s_i \in V_j$ serving Q_j uses V_j as the advertisement identifier.

Information Audience: A server $s_i \in V_j$ serving Q_j and defined as a forager is a potential audience for advertisement posted on the advertisement board.

Information Utilisation: A server $s_i \in V_j$ serving Q_j use available hosting-centre-wide information being presented on the advertisement board in the following manner:

1. Advertisement Board Reading: A server $s_i \in V_j$ defined as a forager will attempt to view the advertisement board at some regular time interval. However, whether the attempt is successful or not is randomly decided based on a threshold. The threshold varies as function of server's own earning rate in relation to the hosting centre's earning rate over a given time interval. The hosting centre's earning rate is given by

$$E_{hosting\text{-}centre} = \frac{1}{T_{host}} \sum_{j=1}^{M} p_j R_j \tag{4.13}$$

where p_j is pence-paid-per-request served and R_j is the total number of request served by V_j during time interval T_{host}. Each server s_i compares its own earning rate, E_i, with overall earning rate, $E_{hosting\text{-}centre}$, to adjust its threshold. The threshold is represented by the probability of reading the advertisement board on each attempt as depicted in Table 4.3. In essence, a server is more likely to read the advertisement board if it is serving requests from a service queue that has a

low earning rate.

Reading Advertisement	
Earning Rate	**P[read adboard]**
$E_i \leq 0.5E_{hosting\text{-}centre}$	0.60
$0.5E_{hosting\text{-}centre} < E_i \leq 0.85E_{hosting\text{-}centre}$	0.20
$0.85E_{hosting\text{-}centre} < E_i \leq 1.15E_{hosting\text{-}centre}$	0.02
$1.15E_{hosting\text{-}centre} < E_i$	0.00

Table 4.3: Adjusting probability (threshold) of reading the advertisement board

2. Hosted Service Reading: A server $s_i \in V_j$ defined as a scout will attempt to view an array of internet services co-hosted on the server ensemble at some regular time interval with success being governed by the same criteria as above for a server defined as a forager.

3. Migrate: If a server $s_i \in V_j$ defined as a forager serving requests from the service queue Q_j is successful at an attempt to read the advertisement board then it will randomly select an advertisement from an array of advertisements on display. From the selected advertisement, it determines the service identifier V_k, $j \neq k$ and migrates to virtual server V_k. Similarly, if a server $s_i \in V_j$ defined as a scout succeeds in its attempt to view an array of hosted services then it will randomly select an internet service hosted on V_k, $j \neq k$ and migrate to virtual server V_k. In both case, the migrating server will be unavailable for some fixed time interval owing to time involved in re-purposing the server for V_k.

The self-organising server orchestration in an internet hosting centre relies on each server generating information about its own service queue and making it globally available whilst all servers in the ensemble have the opportunity to view this information and act upon it as depicted in Figure 4.5. Basically, a server defined as either a forager or a scout will serve requests from its service queue and post advertisements to inform other servers in the ensemble of current earning potential at its service queue. At regular interval, it will compute its earning rate and compare it against the hosting centre's earning rate to set a threshold for randomly deciding whether to view the advertisement board if it is a forager or view other hosted services if it is a scout. A forager will randomly select an advertisement and migrate

to a virtual server that had placed the said advertisement whilst a scout will randomly select a virtual server from all virtual servers in the hosting centre and migrate to it. Essentially, a scout server ensures that the hosting centre does not permanently forget about a virtual server which is currently a poor source of earning.

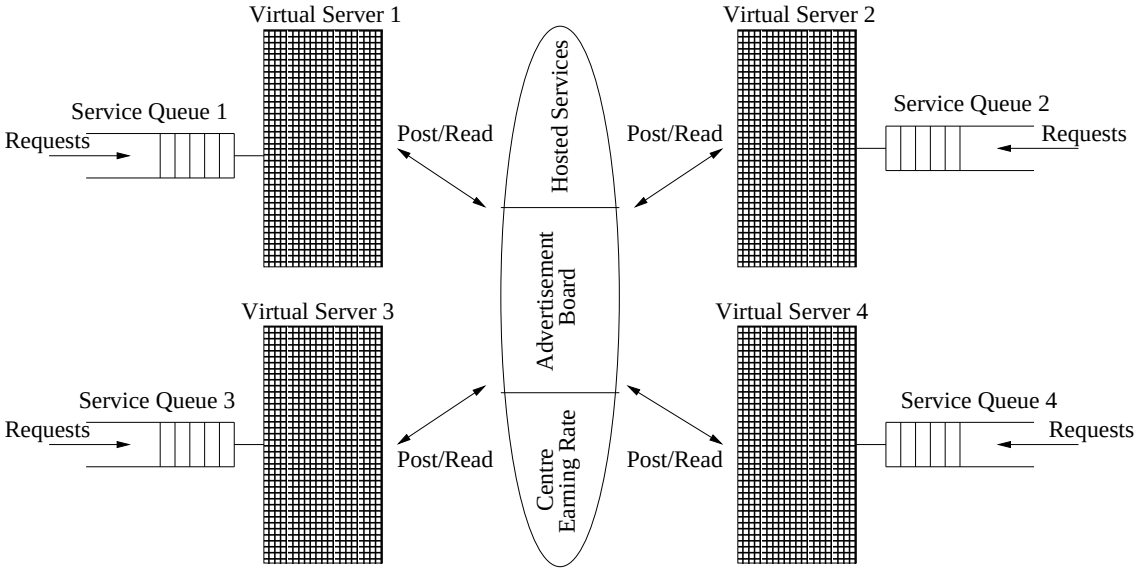


Figure 4.5: Self-Organising server orchestration.

The self-organising server orchestration algorithm is depicted in Figure 4.6. If the service queue has requests to be served (line 4), then the server takes a request from the queue, responds to it and collects earning (lines 5–6). In lines 6–9, it computes its own earning rate over a given time interval and updates the reading threshold by comparing its own earning rate to the overall earning rate of the hosting centre. In lines 11–13, the server looks to attract other servers to come and serve requests from its own queue. It uses the advertisement posting threshold p to randomly decide whether to advertise or not. An advertisement is placed on the advertisement board with an identifier of the virtual server to which the server belongs. The server looks for an earning opportunity elsewhere in lines 15–26 at regular intervals and migrates if the random decision based on threshold r is affirmative. If the server is a forager then it randomly selects an advertisement from the advertisement board whilst if it is a scout then it randomly selects a virtual server amongst the range in the hosting centre. In both cases, it migrates to another virtual server and begins to serve requests from the new queue after incurring a downtime penalty for re-purposing. If the service queue becomes empty then the server idles but in lines 28–43, it continues to update its reading threshold r and look for earning opportunities elsewhere. Intuitively, since its earning rate will degrade

significantly if it is idling, the reading threshold will be set low such that an attempt to read an advertisement will have a higher chance of succeeding thus making the server more active in search for earning opportunities at other services in the hosting centre.

4.6 Summary

In this chapter, we devised a biomimetic server ensemble orchestration algorithm inspired by self-organising behaviour of honeybee colonies to solve the forager orchestration problem. Principally, the algorithm is devised through bio-mimicking of key forager behaviour and interactions amongst members of the colony that lead to successful forager orchestration. The self-organising forager orchestration allows the colony to efficiently collect nectar from flower patches in the surrounding countryside and ensure its survival.

The biomimetic approach relies on the artificial domain problem's sharing key characteristics with its counterpart in the natural domain. We described the details of the remarkable homology that exists between internet hosting centres and honeybee colonies at various levels for respective orchestration problems. The new Self-Organising server ensemble orchestration algorithm exploits the homology and mimics the honeybee colony. An internet hosting centre co-hosting multiple services for a fee with its servers partitioned amongst hosted services to serve requests is equivalent to a honeybee colony with its foragers distributed amongst available flower patches in the surrounding countryside to collect nectar. In honeybee colonies, there is no centralised coordinator controlling forager orchestration amongst flower patches but foragers have an autonomic response to feedback with respect to the nectar sources and collectively self-organise to optimise nectar collection. Our algorithm is also decentralised and individual servers have an autonomic response to feedback on revenue opportunity at co-hosted services and collectively self-organise to optimise revenue collection.

In the next chapter, we devise three additional server orchestration algorithms in order to assess efficacy of the Self-Organising biomimetic algorithm against them. These algorithms differ from the new algorithm in that they are all based on centralised control and make orchestration decisions based on knowledge of either recent history or the future.

```
SELF_ORGANISING_ORCHESTRATE(id, V_j, Q_j, AdBoard, E_centre, S)
 1  Initialise: Earning rate interval T_E, Adboard reading interval T_r
 2  Initialise: Reading threshold r, Advert posting threshold p, Advertisement A
 3  forever
 4    while Q_j ≠ Empty do
 5      serve request
 6      collect earning // value-per-request-served
 7      if T_E Elapsed then
 8        compute earning rate e
 9        r ← LookUp(e, E_centre)
10      endif
11      if Post(p) then
12        A ← V_j // virtual server identifier
13        Advertise(A)
14      endif
15      if Read(r) && T_r elapsed then
16        if id == forager then
17          A ← Random_Advert_Select(AdBoard)
18          V_k ← A
19        else // scout
20          V_k ← Random_VirtualServer_Select(S)
21        endif
22        if V_j ≠ V_k then // migrate to virtual server k
23          V_j ← V_k; offline // re-purpose;downtime
24          Q_j ← Q_k // serve requests from queue k
25        endif
26      endif
27    endwhile
    //request queue empty -- idling
28    if T_E Elapsed then
29      compute earning rate e
30      r ← LookUp(e, E_centre)
31    endif
32    if Read(r) && T_r elapsed then
33      if id == forager then
34        A ← Random_Advert_Select(AdBoard)
35        V_k ← A
36      else // scout
37        V_k ← Random_VirtualServer_Select(S)
38      endif
39      if V_j ≠ V_k then // migrate to virtual server k
40      V_j ← V_k; offline // re-purpose;downtime
41      Q_j ← Q_k // serve requests from queue k
42      endif
43    endif
44    sleep // idling-- goto sleep for a bit
45  endforever
```

Figure 4.6: Self-Organising server orchestration algorithm.

Chapter 5

Efficacy Assessment: Server Ensemble Orchestrations

Prediction is very difficult, especially about the future.

Niels Bohr *(1885 - 1962)*

5.1 Introduction

In this chapter, we introduce a collection of algorithms for the server ensemble orchestration problem. These algorithms differ from the Self-Organising orchestration algorithm proposed in the previous chapter in that a centralised controller makes all orchestration decisions based on recent history or complete knowledge of the future. An algorithm that uses recent history as a guide to orchestrate servers for the near uncertain future is a well known classical technique whilst an algorithm that uses knowledge of the future to orchestrate servers is classed as omniscient. We devise three algorithms for the server orchestration problem, one classical, the other two omniscient and therefore optimal but operating at opposite ends of a functional spectrum defined by the number of orchestration decisions made over the time horizon. The first algorithm makes only one orchestration decision for the entire time horizon whilst the second makes a sequence of orchestration decisions throughout the time horizon. As discussed in Section 5.2, there are number of server orchestration methods proposed in the literature. These approaches provide an appropriate collection against which the Self-Organising algorithm would benefit from an efficacy assessment. However, obtaining executable code or even

algorithm specifications for comparison has proven to be difficult and, therefore, necessitated designing our own efficacy assessment algorithms.

As discussed in Section 3.6, an internet hosting centre co-hosts multiple internet services on its server ensemble and delivers its content to users. The content owners and the hosting centre enter into an SLA contract which defines, amongst other components, the hosting fee and performance guarantees. Given the multiple internet services with associated SLA contracts, unpredictability of future request arrivals and sharing of servers amongst the co-hosted services the hosting centre's goal is to maximise the total value of requests served. Thus, the internet hosting centre faces the following revenue collection problem: how to utilise its limited number of servers to collect as much revenue in SLA fees as possible over a lengthy time horizon. For the entire time horizon, it must make orchestration decisions as to which of its servers in the ensemble are part of which co-hosted internet service. At any instant, it may persist with the current orchestration decision or change to a new decision involving orchestrating servers away from one internet service to be part of another internet service. In such cases, an orchestrated server suffers a downtime during which it does not earn any revenue. The orchestration decisions are difficult given highly variable future request arrivals for each internet service as well as differential SLA fees paid per request, times to serve a request and request queueing behaviours amongst co-hosted services.

The chapter begins with an overview of previous work on orchestration techniques. In Section 5.3, we present a classical orchestration technique known as Greedy which represents a conventional heuristic approach to the orchestration problem. It uses the recent past as a guide to the immediate uncertain future in order to make time varying orchestration decisions at regular time intervals for the entire time horizon. Next, we present two theoretical orchestration techniques which we class as omniscient since both techniques perform optimal orchestration based on *a priori* knowledge of future request arrivals. The Optimal Omniscient technique, described in Section 5.4, makes time varying orchestration decisions at regular time intervals for the entire time horizon. Finally, The Optimal-Static, described in Section 5.5, makes a single orchestration decision at the beginning which remains unchanged for the entire time horizon.

5.2 Related Work

The problem of dynamically orchestrating resources as unpredictable requests arrive has been studied in a generally abstract context as the K-server problem [MMS88]. In this formulation, we are given a metric space S and K mobile servers that reside at points in S. When a request arrives at point $x \in S$ it must be processed by moving one of the K servers to x, at a cost equal to the distance moved in S. The objective is to minimise cost in serving a sequence of requests.

Dynamic resource orchestration has been well studied in the context of effectively exploiting available resources to realise certain performance goals. Several papers address the problem of sharing processor resources between competing applications to improve throughput and/or realtime response [BDM99, DAC96, FS96, JLDB95, JRR97, WE95, WW94]. Numerous papers focus on sharing processors in distributed systems of networked machine for throughput [ADC97, BAG00, BBL00, BDK95, BIKZ98, CFK+98, DKR95, HDL98, LTBL97]. However, these research efforts focus on sharing CPU cycles amongst competing applications as opposed to a whole server being shared in a serial manner and, moreover, they do not consider orchestration downtime cost.

A smaller number of papers have considered the problem in a context similar to the one described in this thesis. Wolf and Yu [WY01] propose a server orchestration algorithm for the case where a single server can handle requests for multiple services and the performance objective is to minimise response time by load balancing. Farias *et al.* [FKSR02] formulate the server orchestration problem in a dynamic programming framework but propose an approximate linear programming solution given that the dynamic programming approach is computationally infeasible. It computes an orchestration policy based on a specific arrival rate model. In contrast, we do not make any *a priori* assumptions regarding request arrivals to make orchestration decisions. Palmer and Mitrani [PM03] examine the server orchestration problem and propose non-optimal heuristics that are easily implementable. However, only a simple case of hosting two services with a total of nine servers is considered. Chase *et al.* [CATV01] propose an economic approach to server allocation for cases where energy conservation is the primary goal. Appleby *et al.* [AFF+01] propose a SLA based approach where monitoring agents provide load and performance feedback to a central resource director which makes orchestration decisions. In our approach, feedback is provided by all servers in

the hosting centre but individual servers are responsible for their own orchestration decisions.

Finally, Jayram *et al.* at IBM [JKK+01] prove that no finite competitive ratio guarantee is possible for any orchestration algorithm which has no knowledge of the future. The authors give a theoretical algorithm which, like our Optimal Omniscient algorithm, requires knowledge of the future and hence cannot be used in practise. They also describe a related heuristic which does not require such knowledge but apparently was not implemented or tested at the time. Their heuristic turns out to be of precisely the same type as our Greedy orchestration algorithm since it employs a forecasting module, details of which are not described. This confirms that our Greedy algorithm provides a good comparative base for classical optimisation methods.

5.3 Greedy Server Orchestration Algorithm

The Greedy orchestration technique represents a conventional heuristic approach to the server ensemble orchestration problem. Generally, this type of technique is optimistic in that it always chooses an orchestration policy that is locally optimal at the moment in expectation that locally optimal policies will lead to globally optimal orchestration policy.

As previously described in Section 3.6, consider an internet hosting centre hosting M internet services on its server ensemble of N servers. Each internet service is hosted on a respective virtual server comprises servers such that the sum of servers distributed across M virtual servers is N. Assume pay-per-request-served SLA contract where a differential value-per-request-served is paid for serving requests for respective hosted service and the time horizon split into $1, \ldots, T$ time steps of equal duration. Let the state S_t represent the state of the server ensemble at the beginning of time step t in terms of distribution of servers amongst virtual servers and unserviced requests in the queues from time step $(t-1)$. The request arrivals across all virtual servers at the server ensemble is A_t and the respective orchestration policy is π^t during time step t. The revenue earned for time step t with initial server ensemble state S_t, arrivals A_t and orchestration policy π^t is given by function $R(\pi^t, S_t, A_t)$ and the state of the server ensemble at the start of time step $(t+1)$ is given by function $f(\pi^t, S_t, A_t)$. Define the value function of Greedy orchestration policy for time step t as

$$v^t(S_t) = \max_{\pi}\{R(\pi, S_t, A_t)\} \tag{5.1}$$

GREEDY_ORCHESTRATE($S_t, \tilde{A}_t, T, M, N$)

1 $t = 1 \rightarrow \tilde{A}_1 = 0; E_{Greedy} = 0.0; \pi_G^1 = \left\{ \left\lfloor \frac{N}{M} \right\rfloor_1, \ldots, \left[\left\lfloor \frac{N}{M} \right\rfloor + N \bmod M \right]_M \right\}$
2 $E_{Greedy} = R(\pi_G^1, S_1, A_1)$
3 $S_{t+1} = f(\pi_G^1, S_1, A_1)$
4 **for** $t = 2$ **upto** $t = T$ **do**
5 $\quad \tilde{A}_t = A_{t-1}$
6 $\quad \pi_G^t = \arg\max_\pi \{R(\pi, S_t, \tilde{A}_t)\}$
7 $\quad E_{Greedy} = E_{Greedy} + R(\pi_G^t, S_t, A_t)$
8 $\quad S_{t+1} = f(\pi_G^t, S_t, A_t)$
9 **endfor**
10 **return** (E_{Greedy})

Figure 5.1: Greedy orchestration algorithm.

since it is myopic about the future and makes an optimal decision for current time step. To maximise $v^t(S_t)$ exactly require knowledge of future arrivals A_t and as this is infeasible, it requires a forecast $\tilde{A}_t$ of A_t. Using the forecast, it chooses an orchestration policy

$$\pi_G^t = \arg\max_\pi \{R(\pi, S_t, \tilde{A}_t)\}. \tag{5.2}$$

The state of the server ensemble at time step (t+1) is then updated according to

$$S_{t+1} = f(\pi_G^t, S_t, A_t). \tag{5.3}$$

In order to determine the arrival forecast, we use the idea that the immediate past is best available guide to the uncertain future and let

$$\tilde{A}_t = A_{t-1}. \tag{5.4}$$

Thus, the Greedy orchestration chooses policy for time step t

$$\pi_G^t(S_t) = \arg\max_\pi \{R(\pi, S_t, A_{t-1})\}. \tag{5.5}$$

The Greedy orchestration algorithm that computes total earning over a time horizon of T time steps is depicted in Figure 5.1. Since the forecast $\tilde{A}_1$ is not available for the first time step, the algorithm computes orchestration policy π_G^1 in line 1 by distributing $\left\lfloor \frac{N}{M} \right\rfloor$ servers to virtual servers $V_1, \ldots, V_{M-1}$ and $\left\lfloor \frac{N}{M} \right\rfloor + N \bmod M$ to V_M. The earning during the first

time step is given by $R(\pi_G^1, S_1, A_1)$ and the state of the server ensemble is updated according to $f(\pi_G^1, S_1, A_1)$ in lines 2–3. For the $(T-1)$th loop iteration in lines 4–9, optimal π_G^t is computed based on request arrival forecast $\tilde{A}_t$ and server ensemble state S_t. Using π_G^t and S_t, earning is computed for actual arrivals A_t and the state S_{t+1} is updated. The total earning over the time horizon T is returned in line 10.

The next state, S_{t+1}, of the server ensemble is a function of current state s_t, current orchestration policy π_G^t and the request arrivals A_t during time step t. The change in current orchestration policy from π_G^t to next policy π_G^{t+1} results in orchestrated servers suffering downtime attributed to purging of current and installation of new service application. Therefore, for the duration of downtime, only available servers given by $\min(\pi_G^t, \pi_G^{t+1})$ will be serving requests whilst π_G^{t+1} will be serving requests for the remainder of the time step $(t+1)$.

The critical part of the Greedy orchestration algorithm is computing the locally optimal orchestration policy, π_G^t, for time steps $2, \ldots, T$. We can compute the locally optimal orchestration policy, $\pi_G^t = \arg\max_\pi \{R(\pi, S_t, \tilde{A}_t)\}$, which maximises earning using exhaustive search as illustrated in Example 5.1. The permutations of orchestration policy for N servers and M virtual server is akin to number of different ways to distribute N indistinguishable balls into M distinguishable urns and it is given by

$$\binom{N+M-1}{M-1} = \frac{(N+M-1)!}{(M-1)!((N+M-1)-(M-1))!}. \tag{5.6}$$

The numerator of the Equation 5.6 will grow much faster than the denominator. For example, $N = 50$ and $M = 5$ will require exploration of $316,251$ orchestration policies whilst $M = 10$ results in searching 12.56×10^9 orchestration policies. Clearly, the running time would be prohibitive except for small N and M. In order to make locally optimal orchestration policy tractable, we propose an alternative algorithm called Local Neighbour Search assuming the revenue function $R(\pi, S_t, \tilde{A}_t)$ is concave. This assumption was tested during computation and, indeed, the revenue function was found to be concave. The basic strategy of the algorithm for time step t is as follows:

Step 0 Let $\pi_{BestPolicy}$ be equal to the orchestration policy π_G^{t-1} used during time step $(t-1)$.

Step 1 Let $\pi_{Neigh} = \{\pi_1, \ldots, \pi_n\}$ be a set of all feasible neighbours of $\pi_{BestPolicy}$.

Step 2 Find optimal $\pi_i^{opt} \in \pi_{Neigh}$. If π_i^{opt} is better than $\pi_{BestPolicy}$ then replace $\pi_{BestPolicy}$

with π_i^{opt}.

Step 3 Repeat steps 1 and 2 until π_i^{opt} does not improve $\pi_{BestPolicy}$.

Step 4 $\pi_{BestPolicy}$ is the orchestration policy for time step t.

Example 5.1 *Consider, for instance, an internet hosting centre with N servers and M virtual servers, the exhaustive policy search algorithm will be:*

for $i_1 = 0, i_1 \leq N$ **do** `// Virtual server 1`
  for $i_2 = 0, i_2 \leq (N - i_1)$ **do** `// Virtual server 2`
    for $i_3 = 0, i_3 \leq (N - (i_1 + i_2))$ **do** `// Virtual server 3`
      $\vdots$
      for $i_{M-1} = 0, i_{M-1} \leq (N - (i_1 + i_2 + \cdots + i_{M-2}))$ **do**
        $\pi_G^t = \{i_1, i_2, \ldots, i_{M-1}, N - (i_{M-1})\}$
        $E_{Tmp} = R(\pi_G^t, S_t, \tilde{A}_t)$
        if $E_{Tmp} > E_{G-Opt}$ **then**
          $E_{G-Opt} = E_{Tmp}$
          $\pi_{G-Opt}^t = \pi_G^t$
        endif
      endfor
      $\vdots$
    endfor
  endfor
endfor
return π_{G-Opt}^t

For an internet hosting centre with N server in the ensemble and M virtual servers, we define neighbours of an orchestration policy $\pi =< \pi^1, \ldots, \pi^M >$ to be set of all feasible orchestration policies $\pi_{Neigh} = \{\pi_1, \ldots, \pi_n\}$ such that each neighbour $\pi_i \in \pi_{Neigh}$ is an instance of an orchestration policy with N servers distributed across M virtual servers i.e. $\pi_i =< \pi_i^1, \ldots, \pi_i^M >$ subject to constraints $\sum_{j=1}^{j=M} \pi_i^j = N$, $\pi_i^j \geq 0$ and $\forall_{i:1 \leq i \leq n, j:1 \leq j \leq M}\ 0 \leq |\pi^j - \pi_i^j| \leq 1$. In general, the number of neighbours will be a function of M since it will decide how many ways we can change an orchestration policy satisfying the above constraints as illustrated in Example 5.2. In Figure 5.2, we give the Local Neighbour Search algorithm which is used to compute $\pi_G^t = \arg\max_\pi \{R(\pi, S_t, \tilde{A}_t)\}$ in line 6 of the Greedy orchestration algorithm depicted in Figure 5.1.

Example 5.2 *Consider, for instance, an internet hosting centre with $M = 4$ virtual servers and N servers in the ensemble. We can generate neighbours of an orchestration policy by*

finding permutations of patterns `(+1)(-1)00` *and* `(+1)(+1)(-1)(-1)` *and adding each permutation to the orchestration policy. There are*

$$\frac{M!}{2!} \tag{5.7}$$

permutations of `(+1)(-1)00`*. This is akin to augmenting a single server at a virtual server and simultaneously reducing by a single server at another virtual server. This gives 12 neighbours and, similarly, there are*

$$\frac{M!}{2!2!} \tag{5.8}$$

permutations of `(+1)(+1)(-1)(-1)`*. This is akin to augmenting a single server at two distinct virtual servers respectively and simultaneously reducing by a single server at two other distinct virtual servers respectively. This gives another six neighbours resulting in total of eighteen neighbour orchestration policies. We point out that not all permutations of the pattern will generate a feasible neighbour due to other constraints For example, orchestration policy* `000N` *and permutation pattern* `00(-1)(+1)` *will result in* `00(-1)(N+1)` *as a neighbour orchestration policy which violates* $\pi_i^j \geq 0$ *constraint. In case* $M = 6$*, not only is there an opportunity to augment or reduce at 2(*`(+1)(-1)0000`*) and 4(*`(+1)(+1)(-1)(-1)00`*) distinct virtual servers simultaneously but also at 6(*`(+1)(+1)(+1)(-1)(-1)(-1)`*) distinct virtual servers as well resulting in* $\frac{6!}{4!} = 30$*,* $\frac{6!}{2!2!2!} = 90$ *and* $\frac{6!}{3!3!} = 20$ *neighbours respectively.*

5.4 Optimal Omniscient Server Orchestration Algorithm

The optimal orchestration technique is a theoretical approach to the server ensemble orchestration problem. We class this technique as omniscient since it is aware of future request arrivals. Consequently, it continually makes time-local decisions that will result in a globally optimal orchestration policy and provides an upper bound on possible revenue which can be earned by a server ensemble. We are taking some liberty here in using the term optimal since the proposed technique is an approximation as it considers time to be discrete instead of continuous and request queues to be ranges (queue buckets) instead of aggregate queue lengths.

Whether approximate or exact, as described here, the technique would be impossible to implement since we cannot know the future. However, in practise, there are classes of services

```
LOCAL_NEIGHBOUR(π_G^{t-1}, S_t, Ã_t, M, N)
 1  BestEarn = R(π_G^{t-1}, S_t, Ã_t)
 2  Neigh(π) = {π_1, ..., π_n} // M and N needed to generate neighbours
 3  π_BestNeigh = NULL
 4  forever
 5    for i = 1, i = n do
 6      EarnNeigh = R(π_i, S_t, Ã_t)
 7      if EarnNeigh > BestEarn then
 8        BestEarn = EarnNeigh
 9        π_BestNeigh = π_i
10    endif
11    endfor
12    if π_BestNeigh ≠ NULL then // better neighbour found
13      Neigh(π_BestNeigh) = {π_1, ..., π_n} // go back and search new neighbours
14      π_BestNeigh = NULL
15    else
16      break // no better neighbour found -- exit forever
17    endif
18  endforever
19  return (π_BestNeigh)
```

Figure 5.2: Local Neighbour Search algorithm for optimising Greedy orchestration.

where it may be possible to predict with reasonable degree of accuracy the access pattern of a service based on historical data. In Figure 6.12, we give access patterns derived from traces of real internet services. When comparing the access patterns from traces recorded on different days for the same service, we observed strong similarities. Hence, for service classes such as entertainment, it is possible to implement Omniscient algorithm based on knowledge of yesterday's access pattern. Obviously, the technique would not be optimal but offers a viable solution to the problem. As previously described in Section 5.3, consider an internet hosting centre with N servers in the ensemble and M virtual servers co-hosting M internet services respectively. Let the time horizon be split into $1, \ldots, T$ time steps of equal duration. As before, S_t denotes the state of the server ensemble at the beginning of time step t, A_t denotes the request arrivals across all virtual servers during time step t and π^t denotes the orchestration policy for M virtual servers during time step t. The revenue earned during time step t is given by function $R(\pi^t, S_t, A_t)$ and the state S_{t+1} of the ensemble at beginning of time step $(t+1)$ is given by function $f(\pi^t, S_t, A_t)$. We define the value function of Optimal

OPTIMAL_ORCHESTRATE(S_t, A_t, t, T)
1 $E^t_{Opt} = 0.0$
2 **if** $(t == T)$
3 $E^T_{Opt} = R(\pi^T_{Opt}, S_T, A_T) + 0.0$ // `no salvage value at T+1`
4 **return** (E^T_{Opt})
5 **else**
6 $E^t_{Opt} = R(\pi^t_{Opt}, S_t, A_t)$ +
OPTIMAL_ORCHESTRATE$(f(\pi^t_{Opt}, S_t, A_t), A_{t+1}, t+1, T)$
7 **return** (E^t_{Opt})
8 **endif**

Figure 5.3: Optimal Omniscient orchestration algorithm.

orchestration policy for time step t as

$$v^t(S_t) = \max_{\pi}\{R(\pi, S_t, A_t) + v^{t+1}(f(\pi, S_t, A_t))\} \tag{5.9}$$

and the value at the end of the time horizon to be

$$v^{T+1}(S_{T+1}) = 0 \tag{5.10}$$

since there is no state-dependent salvage value at the end of the time horizon. The optimal orchestration policy for time step t will be given by

$$\pi^t_{Opt}(S_t) = \arg\max_{\pi}\{R(\pi, S_t, A_t) + v^{t+1}(f(\pi, S_t, A_t))\}. \tag{5.11}$$

The Optimal orchestration algorithm that computes optimal total revenue earned over a time horizon of T time steps is illustrated in Figure 5.3. The recursive nature of the value function given in Equation 5.9 calls for dynamic programming to solve for the optimal orchestration policy decision over the complete time horizon. The algorithm assumes that request arrivals A_t for time steps $1, \ldots, T$ are known in advance. In line 6, the algorithm recursively calls itself to compute optimal earning for each successive time steps $1, \ldots, (T-1)$. In line 3, the recursion stops when $t = T$ as there is no salvage value for $(T+1)$ time step. In essence, the optimal revenue for the entire time horizon is computed by starting with $E^T_{Opt} = R(\pi^T_{Opt}, S_T, A_T) + 0.0$ where $\pi^T_{Opt}(S_T) = \arg\max_{\pi}\{R(\pi, S_T, A_T) + 0.0\}$ and going backward using $E^t_{Opt} = R(\pi^t_{Opt}, S_t, A_t) + E^{t+1}_{Opt}$ for time steps $1, \ldots, (T-1)$ where

$\pi^t_{Opt}(S_t) = \arg\max_\pi\{R(\pi, S_t, A_t) + E^{t+1}_{Opt}\}$. Consequently, the revenue earning generated at time step 1 and returned in line 7 is equal to the optimal total revenue over the entire time horizon with the set of policies $\pi^* = \{\pi^1_{Opt}, \ldots, \pi^T_{Opt}\}$ optimal for respective time steps.

As in the case of Greedy orchestration algorithm, the next state S_{t+1} is a function of current state S_t, request arrivals A_t and orchestration policy π^t_{Opt} used during time step t. The change in orchestration policy from π^t_{Opt} to π^{t+1}_{Opt} entails orchestrated servers will be unavailable for downtime. Hence, during downtime, only $\min(\pi^t_{Opt}, \pi^{t+1}_{Opt})$ servers will be available to service requests and π^{t+1}_{Opt} will be available to service requests for the remainder of the time step $(t+1)$. The unserviced requests in the queues at the beginning of time step $(t+1)$ will be a function of orchestration policy π^t_{Opt} and behaviour of users in response to waiting time between initiating a request and receiving a response. The baulking factor represents the rate of requests loss due to user behaviour over the time step. Hence, the residual requests in the queues at the beginning of time step $(t+1)$ will be unserviced requests from time step t scaled by the baulking factor.

The dynamic programming characteristic of the problem means that the algorithm is time and space intensive as a function of the problem size. An exhaustive state-space search would be akin to exploring a complete k-ary tree in which all leaves have the same depth and all internal nodes have degree k. An ensemble with N servers compose M virtual servers such that there are

$$k = \frac{(N+M-1)!}{(M-1)!N!} \tag{5.12}$$

permutations of the orchestration policy. The root of the tree will have k children at depth 1 and each child will have k children at depth 2 and so on. The height of the tree will be T hence, the number of leaves at depth T will be k^T and the number of internal nodes will be $\frac{k^T-1}{k-1}$. Therefore, the total number of states which must be searched to determine optimal orchestration policy for the entire time horizon will be

$$\left(\frac{(N+M-1)!}{(M-1)!N!}\right)^T + \frac{\left(\frac{(N+M-1)!}{(M-1)!N!}\right)^T - 1}{\left(\frac{(N+M-1)!}{(M-1)!N!}\right) - 1}. \tag{5.13}$$

Clearly, except for very small M, N and T, the search space will be computationally prohibitive.

We propose an approximate technique with time and space trade-off for solution to the

dynamic programming optimal orchestration problem. The basic strategy of the approximate technique is as follows:

Step 0 Let π be a set of all feasible orchestration policies for internet hosting centre with M virtual servers and N servers in the ensemble. Assume each feasible policy is a candidate for time step $(t-1)$.

Step 1 For each feasible candidate orchestration policy that could be used in the time step $(T-1)$, compute $\pi^T_{Opt}(S_T) = \arg\max_\pi\{R(\pi, S_T, A_T) + v^{T+1}(S_{T+1})\}$ and $v^T(S_T) = \{R(\pi^T_{Opt}, S_T, A_T) + v^{T+1}(S_{T+1})\}$ assuming no salvage value for state S_{T+1} hence, $v^{T+1}(S_{T+1}) = 0.0$. Store result table $\hat{v}_T$ in memory.

Step 2 Traverse backward from time steps $(T-1), \ldots, 1$. For each feasible candidate orchestration policy that could be used in $(t-1)$, compute and store in $\hat{v}_t$ for each time step t, $\pi^T_{Opt}(S_t) = \arg\max_\pi\{R(\pi, S_t, A_t) + v^{t+1}(S_{t+1})\}$, $v^t(S_t) = \{R(\pi^t_{Opt}, S_t, A_t) + v^{t+1}(S_{t+1})\}$ using $\hat{v}_{t+1}$ in memory.

Step 3 Finish. $v^1(S_1)$ and $\pi^* = \{\pi^1_{Opt}, \ldots, \pi^T_{Opt}\}$ are optimal for the entire time horizon.

Based on this strategy, an initial version of the approximate optimal orchestration algorithm that computes total revenue and orchestration policy over a time horizon of T time steps is depicted in Figure 5.4.

An initial version of the approximate optimal orchestration algorithm has a number of limitations. Below, we describe these limitations and propose solutions to overcome them. We conclude with an approximate Optimal Omniscient orchestration algorithm that encompasses these solutions.

5.4.1 Residual Queue Buckets and Interpolation

The computation is being performed by traversing backward in time however, unserviced requests from one time step to another can only be known if traversing forward in time. Therefore, the magnitude of unserviced requests scaled by the baulking factor from time step $(t-1)$ is unknown. To overcome this limitation, we propose computing a result table for each time step with an entry for each permutation of orchestration policy and q residual queue buckets with size of each bucket given by b^j where $1 \leq j \leq q$. Therefore, the number of

APPROX_OPTIMAL_ORCHESTRATE_INITIAL(A, M, N, T)

for $(t = T, t = 1)$ **do** // `Traverse backward`
  for $(i1 = 0, i1 \leq N)$ **do** // `Virtual servers`
    for $(i2 = 0, i2 \leq (N - i1))$ **do**
      $\vdots$
      for $(i_{M-1} = 0, i_{M-1} \leq (N - (i1 + \ldots + i_{M-2}))$ **do**
        if $(t == T)$ **then**
          $\pi^t_{Opt}(S_t) = \arg\max_\pi \{R(\pi, S_t, A_t) + 0.0\}$
          $v^t(S_t) = \{R(\pi^t_{Opt}, S_t, A_t) + 0.0)\}$
          $\hat{v}_t \leftarrow \pi^t_{Opt}(S_t),\ \hat{v}_t \leftarrow v^t(S_t)$
        else
          $\pi^t_{Opt}(S_t) = \arg\max_\pi \{R(\pi, S_t, A_t) + \hat{v}_{t+1}\}$
          $v^t(S_t) = \{R(\pi^t_{Opt}, S_t, A_t) + \hat{v}_{t+1})\}$
          $\hat{v}_t \leftarrow \pi^t_{Opt}(S_t),\ \hat{v}_t \leftarrow v^t(S_t)$
        endif
      endfor
      $\vdots$
    endfor
  endfor
  $\hat{v}_{t+1} \leftarrow \hat{v}_t$
endfor
return $(v^1(S_1), \pi^*)$

Figure 5.4: An Approximate Optimal Omniscient orchestration algorithm.

entries in the result table will be a function of M, N, and q

$$q^M \cdot \left(\frac{(N + M - 1)!}{(M - 1)!N!} \right) \tag{5.14}$$

i.e. there are M queues for each respective virtual server with q buckets and for each feasible orchestration policy, each bucket will add an entry into the result table. Clearly, the memory and CPU requirements of the approximate optimal omniscient algorithm will depend on the values of M, N and q as illustrated in Example 5.3. We notice that even for small M, the growth in size of result table and computation severely restricts the value of M, number of discrete time steps and the length of the time horizon that we can consider for the algorithm.

Example 5.3 *Consider an internet hosting centre with $M = 2$ virtual servers, $N = 50$ servers in the ensemble with $q = 7$ residual queue buckets. The number of entries in the result table, given by Equation 5.14, will be $\approx 2.50 \times 10^6$. If we consider $M = 5$, then the number of entries in the result table will be $\approx 5.3 \times 10^9$ and increases to $\approx 3.57 \times 10^{18}$ for $M = 10$.*

Defining residual queue buckets with a given magnitude for time step t creates another limitation when computing optimal orchestration policy during time step $(t-1)$. There is no guarantee that when computing $v^{t-1}(S_{t-1})$, the number of unserviced requests scaled by the baulking factor will map exactly to one of the defined residual queue buckets and, hence, allow us to retrieve $v^t(S_t)$ from the result table $\hat{v}_t$ computed during time step t and stored in the memory.

In order to overcome the mapping limitation, we propose using interpolation to compute $v^t(S_t)$. Let $v^t(S_t)$ computed for each permutation of feasible orchestration policy and queue bucket sizes during time step t be stored in a direct addressable table $\hat{v}_t$ in the memory and accessed by a key drawn from a set $\mathcal{U}$ where each element is a permutation of orchestration policy and queue bucket sizes. The cardinality of the set $\mathcal{U}$ is given by Equation 5.14 in terms of M, N and q. Let key $\hat{k}$ be defined by $(2M-1)$-tuple

$$\hat{k} = (\pi_1, \ldots, \pi_{M-1}, \hat{q}_1, \ldots, \hat{q}_M) \tag{5.15}$$

where $\pi_i : 1 \leq i \leq (M-1)$ denotes feasible server orchestration policy and $\hat{q}_j : 1 \leq j \leq M$ denotes feasible queue bucket size. Also, let $\hat{v}_t(\hat{k})$ denote the value $v_t(S_t)$ such that the orchestration policy is $\pi_1, \ldots, \pi_{M-1}$ and unserviced requests scaled by the baulking factor are $\hat{q}_1, \ldots, \hat{q}_M$ at the end of time step $(t-1)$. To determine $v_t(S_t)$ for unserviced requests $\tilde{q}_1, \ldots, \tilde{q}_M$ that may or may not map to queue buckets exactly and orchestration policy $\pi_1, \ldots, \pi_{M-1}$, we use the interpolation technique with the following strategy:

Step 0 For each $\tilde{q}_i : 1 \leq i \leq M$ unserviced requests, compute $\tilde{\jmath}_i$ such that $b^{\tilde{\jmath}_i} \leq \tilde{q}_i \leq b^{(\tilde{\jmath}_i+1)}$.

Step 1 For each $\tilde{q}_i : 1 \leq i \leq M$, compute $l_i = \tilde{q}_i - b^{\tilde{\jmath}_i}$ and $u_i = b^{(\tilde{\jmath}_{i+1})} - \tilde{q}_i$.

Step 2 Generate $2M$ keys such that each key is a combination of $b^{\tilde{\jmath}_1}, \ldots, b^{\tilde{\jmath}_M}$ and $b^{(\tilde{\jmath}_1+1)}, \ldots, b^{(\tilde{\jmath}_M+1)}$. For example,

$\hat{k}_1 = (\pi_1, \ldots, \pi_{M-1}, b^{\tilde{\jmath}_1}, \ldots, b^{\tilde{\jmath}_M})$

$\hat{k}_2 = (\pi_1, \ldots, \pi_{M-1}, b^{\tilde{\jmath}_1}, \ldots, b^{(\tilde{\jmath}_{M-1})}, b^{(\tilde{\jmath}_M+1)})$

$\hat{k}_3 = (\pi_1, \ldots, \pi_{M-1}, b^{\tilde{\jmath}_1}, \ldots, b^{(\tilde{\jmath}_{M-2})}, b^{(\tilde{\jmath}_{M-1}+1)}, b^{(\tilde{\jmath}_M+1)})$

$\vdots$

$\hat{k}_{2M-2} = (\pi_1, \ldots, \pi_{M-1}, b^{(\tilde{\jmath}_1+1)}, \ldots, b^{(\tilde{\jmath}_{M-2}+1)}, b^{(\tilde{\jmath}_{M-1})}, b^{(\tilde{\jmath}_M)})$.

$\hat{k}_{2M-1} = (\pi_1, \ldots, \pi_{M-1}, b^{(\tilde{\jmath}_1+1)}, \ldots, b^{(\tilde{\jmath}_{M-1}+1)}, b^{(\tilde{\jmath}_M)})$.

$\hat{k}_{2M} = (\pi_1, \ldots, \pi_{M-1}, b^{(\tilde{\jmath}_1+1)}, \ldots, b^{(\tilde{\jmath}_M+1)})$.

Step 3 Compute $2M$ products $\hat{p}$ such that each product is a combination of $l_1, \ldots, l_M$ and $u_1, \ldots, u_M$. For example,

$\hat{p}_1 = \prod_{i=1}^{M} l_i$

$\hat{p}_2 = \left(\prod_{i=1}^{M-1} l_i\right) u_M$

$\hat{p}_2 = \left(\prod_{i=1}^{M-2} l_i\right) \left(\prod_{i=M-1}^{M} u_i\right)$

$\vdots$

$\hat{p}_{2M-2} = \left(\prod_{i=1}^{M-2} u_i\right) \left(\prod_{i=M-1}^{M} l_i\right)$

$\hat{p}_{2M-1} = \left(\prod_{i=1}^{M-1} u_i\right) l_M$

$\hat{p}_{2M} = \prod_{i=1}^{M} u_i$.

Step 4 Compute $v^t(S_t) = \left(\frac{\sum_{n=1}^{2M} \hat{p}_n \hat{v}_t(\hat{k}_n)}{\sum_{n=1}^{2M} \hat{p}_n}\right)$

Step 5 Finish. The interpolated value is $v^t(S_t)$ for orchestration policy $\pi_1, \ldots, \pi_{M-1}$ used during time step $(t-1)$ and unserviced requests $\hat{q}_1, \ldots, \hat{q}_M$ at the beginning of time step t.

We illustrate the interpolation technique in Example 5.4 and give the algorithm in Figure 5.5.

Example 5.4 *Consider an internet hosting centre with* $M = 2$ *virtual servers,* $N = 50$ *servers in the ensemble and residual queue buckets with* $b = 2$ *and* $q = 7$. *Suppose at the beginning of time step* t, *unserviced requests from time step* $t-1$ *are* $\hat{q}_1 = 20$, $\hat{q}_2 = 70$ *for orchestration policy* $\pi_1 = 15$, $\pi_2 = 35$. *Therefore, it follows that* $\tilde{\jmath}_1 = 4$, $\tilde{\jmath}_2 = 6$ *such that* $2^4 \leq 20 \leq 2^{4+1}$ *and* $2^6 \leq 70 \leq 2^{6+1}$ *and* $l_1 = 4$, $u_1 = 12$, $l_2 = 6$, $u_1 = 58$. *The four keys will be* $\hat{k}_1 = (15, 25, 16, 64)$, $\hat{k}_2 = (15, 35, 16, 128)$, $\hat{k}_3 = (15, 35, 32, 64)$ *and* $\hat{k}_4 = (15, 35, 32, 128)$. *Similarly, products will be* $\hat{p}_1 = (12)(58)$, $\hat{p}_2 = (12)(6)$, $\hat{p}_3 = (4)(58)$ *and* $\hat{p}_4 = (4)(6)$. *Assuming table* $\hat{v}_t$ *in memory has the result and* $\hat{v}_t(\hat{k}_1) = 320$, $\hat{v}_t(\hat{k}_2) = 325$, $\hat{v}_t(\hat{k}_3) = 355$, $\hat{v}_t(\hat{k}_4) = 375$ *then the interpolated value* $v_t(S_t) = \left(\frac{\sum_{n=1}^{2M} \hat{p}_n \hat{v}_t(\hat{k}_n)}{\sum_{n=1}^{2M} \hat{p}_n}\right)$ *will be* 329.00.

5.4.2 Optimal Local Neighbour Search

It is evident from the Equation 5.14 that computing the result table $\hat{v}_t$ for each time step will be computationally challenging given that for each entry we must compute $\pi^t_{Opt}(S_t) = \arg\max_\pi \{R(\pi, S_t, A_t) + v^{t+1}(S_{t+1})\}$ and hence $v^t(S_t) = \{R(\pi^t_{Opt}, S_t, A_t) + v^{t+1}(S_{t+1})\}$. To reduce the search time for π^t_{Opt} we propose employing the local neighbour search technique

INTERPOLATE_QUEUE_BUCKETS($M, \hat{v}_t, b, \pi_1, \ldots, \pi_{M-1}, \hat{q}_1, \ldots, \hat{q}_M$)

1 **if** $\hat{v}_t == NULL$ **then**
2 **return** (0.0)
3 **else**
4 **for** $i = 1, M$ **do**
5 $\tilde{j}_i = \lfloor \log_b \hat{q}_i \rfloor$
6 $l_i = \hat{q}_i - b^{\tilde{j}_i}$
7 $u_i = b^{(\tilde{j}_i+1)} - \hat{q}_i$
8 **endfor**
9 **for** $i = 1, 2M$ **do**
10 $\hat{k}_i \leftarrow \pi_1, \ldots, \pi_{M-1}, b^{\tilde{j}_1}, \ldots, b^{\tilde{j}_M}, b^{(\tilde{j}_1+1)}, \ldots, b^{(\tilde{j}_M+1)}$
11 $\hat{p}_i \leftarrow l_1, \ldots, l_M, u_1, \ldots, u_M$
12 **endfor**
13 $v^t(S_t) = \left(\frac{\sum_{i=1}^{2M} \hat{p}_i \hat{v}_t(\hat{k}_i)}{\sum_{i=1}^{2M} \hat{p}_i} \right)$
14 **return** ($v_t(S_t)$)
5 **endif**

Figure 5.5: Interpolation algorithm.

described in Section 5.3 and given in Figure 5.2. In order to encompass residual queue buckets and interpolation technique for Optimal omniscient orchestration, we modify the algorithm of Figure 5.2 and give the Local-Optimal neighbour search algorithm in Figure 5.6.

5.4.3 An Approximate Optimal Omniscient Orchestration with Queue Buckets

We are now in position to present an approximate optimal omniscient orchestration algorithm with residual queue buckets which uses interpolation. The algorithm is depicted in Figure 5.7.

5.5 Optimal-Static Omniscient Server Orchestration Algorithm

The optimal-static orchestration is also a theoretical approach to server ensemble orchestration problem. As previously described for the optimal orchestration in Section 5.4, this technique is omniscient as it is aware of future request arrivals. However, as opposed to making multiple orchestration decisions over the time horizon, it makes an optimal orchestration decision at the beginning which remains static over the entire time horizon. Again, the technique would be impossible in practise as future request arrivals are unknown. Nevertheless, it provides

```
LOC_OPT_NEIGH(π, S_t, A_t, M, N, v̂_{t+1}, b)
    // Orchestration policy π = π_1, ..., π_M
 1  BestEarn = R(π, S_t, A_t) // In state S_{t+1} q̂ = q̂_1, ..., q̂_M unserviced requests
 2  Neigh(π) = {π^1, ..., π^n} // M and N needed to generate neighbours
 3  π_{BestNeigh} ← NULL
 4  forever
 5    for i = 1, i = n do
 6      EarnNeigh = R(π^i, S_t, A_t)+
              INTERPOLATE_QUEUE_BUCKETS(M, v̂_{t+1}, b, π^i_1, ..., π^i_{M-1}, q̂_1, ..., q̂_M)
 7      if EarnNeigh > BestEarn then
 8        BestEarn = EarnNeigh
 9        π_{BestNeigh} = π^i
10      endif
11    endfor
12    if π_{BestNeigh} ≠ NULL then // better neighbour found
13      Neigh(π_{BestNeigh}) = {π^1, ..., π^n} // go back and search new neighbours
14      π_{BestNeigh} = NULL
15    else
16      break // no better neighbour found -- exit forever
17    endif
18  endforever
19  return (BestEarn, π_{BestNeigh})
```

Figure 5.6: Local Neighbour Search algorithm for optimising approximate Optimal Omniscient orchestration.

an upperbound on possible revenue earned by a server ensemble for internet hosting centres currently employing *ad hoc* static techniques as previously described in Section 1.1.

Consider an internet hosting centre with M virtual servers co-hosting internet services on N servers in the ensemble. The time horizon is divided into $1, \ldots, T$ equal duration time steps. The state S_t, arrivals A_t, orchestration policy π^t, the revenue function $R(\pi^t, S_t, A_t)$ and the next state function $f(\pi^t, S_t, A_t)$ have the usual meaning as previously described in Section 5.3. The optimal-static omniscient orchestration policy for the entire time horizon will be

$$\pi_{\text{opt-stat}} = \arg\max_{\pi} \sum_{t=1}^{T} R(\pi, S_t, A_t) \tag{5.16}$$

subject to

$$S_{t+1} = f(\pi, S_t, A_t). \tag{5.17}$$

APPROX_OPTIMAL_ORCHESTRATE$(A, M, N, T, b.q)$

for $(t = T, t = 2)$ **do** // `Traverse backward`
  for $(i1 = 0, i1 \leq N)$ **do** // `Virtual servers`
    for $(i2 = 0, i2 \leq (N - i1))$ **do**
      ⋮
      for $(i_{M-1} = 0, i_{M-1} \leq (N - (i1 + \ldots + i_{M-2}))$ **do**
        for $(j_1 = 0, j_1 \leq q)$ **do** // `Queue buckets`
          for $(j_2 = 0, j_2 \leq q)$ **do**
            ⋮
            for $(j_M = 0, j_M \leq q)$ **do**
              $\pi = i_1, \ldots, i_{M-1}, \tilde{q} = b^{j_1}, \ldots, b^{j_M}$
              if $(t == T)$ **then**
                $\hat{v}_{t+1} = NULL$
                $(\pi^t_{Opt}(S_t), v^t(S_t)) \leftarrow$ LOC_OPT_NEIGH$(\pi, S_t, A_t, M, N, \hat{v}_{t+1}, b)$
                $\hat{v}_t(\pi, \tilde{q}) \leftarrow (\pi^t_{Opt}(S_t), v^t(S_t))$
              else
                $(\pi^t_{Opt}(S_t), v^t(S_t)) \leftarrow$ LOC_OPT_NEIGH$(\pi, S_t, A_t, M, N, \hat{v}_{t+1}, b)$
                $\hat{v}_t(\pi, \tilde{q}) \leftarrow (\pi^t_{Opt}(S_t), v^t(S_t))$
              endif
            endfor
            ⋮
          endfor
        endfor
      endfor
      ⋮
    endfor
  endfor
  $\hat{v}_{t+1} \leftarrow \hat{v}_t$
endfor
$\hat{\pi} = \hat{\pi}_1, \ldots, \hat{\pi}_{M-1}$ // `server ensemble state at the beginning`
$(v^1(S_1), \pi^1_{Opt}(S_1)) \leftarrow$ LOC_OPT_NEIGH$(\hat{\pi}, S_t, A_t, M, N, \hat{v}_2, b)$
return $(v^1(S_1), \pi^1_{Opt}(S_1))$

Figure 5.7: An optimised Approximate Optimal Omniscient orchestration algorithm with Residual Queue Buckets and Local Neighbour Search.

The optimal-static value function for any time step t is

$$v^t(S_t) = R(\pi_{\text{opt-stat}}, S_t, A_t) \tag{5.18}$$

since the same orchestration policy is used in every time step. The state of the server ensemble is updated according to

$$S_{t+1} = f(\pi_{\text{opt-stat}}, S_t, A_t) \tag{5.19}$$

and the total revenue over the entire time horizon will be given by

$$\sum_{t=1}^{T} v^t(S_t). \tag{5.20}$$

The optimal-static orchestration algorithm is depicted in Figure 5.8. It returns total revenue earned over the entire time horizon using the optimal-static orchestration policy under the assumption that the request arrivals A_t for each time step $1, \ldots, T$ is known in advance. A feasible orchestration policy is generated by lines 2–9 and used in the loop of lines 11–14 to compute total earning over time steps $1, \ldots, T$. In lines 15–18, it determines whether the current orchestration policy and hence total revenue is best so far whilst returning the optimal-static orchestration policy and total revenue in line 28. As in the case of Greedy and Optimal omniscient orchestration algorithms, the next state S_{t+1} is a function of current state S_t, request arrivals A_t and the orchestration policy used during time step t. However, since there is no change in orchestration policy over the entire time horizon all servers will be available throughout. The residual requests which have not been serviced in the time step t will be scaled by the baulking factor and available for processing in time step $(t + 1)$ along with new requests arrivals A_{t+1}. For N servers and M virtual servers, the algorithm will implicitly enumerate orchestration policy permutations given by Equation 5.6 to determine the optimal-static orchestration policy.

5.6 Summary

This chapter introduced three new server ensemble orchestration algorithms for assessing efficacy of the Self-Organising orchestration algorithm introduced in Chapter 4. All three algorithms use the strategy of centralised controller orchestrating servers amongst co-hosted services based either on knowledge of recent history or the future. The algorithm that relies on recent history is a classical technique whilst others are theoretical techniques since it is impossible to know the future and thus implementation in practise is infeasible.

The classical Greedy orchestration algorithm makes orchestration decisions at regular intervals based on request arrival information from the previous interval. In fact, it makes the orchestration decision that would be optimal if the request arrival information for the next interval matched the information from the previous interval. The Optimal Omniscient orchestration algorithm, as the name suggests, makes optimal orchestration decisions at regular

```
OPTIMAL-STATIC_ORCHESTRATE(A, M, N, T)
 1  E = E_opt-stat = 0.0 // Earning
 2  for (i1 = 0, i1 ≤ N) do // Virtual servers
 3    for (i2 = 0, i2 ≤ (N − i1)) do
        ⋮
 9      for (i_M-1 = 0, i_M-1 ≤ (N − (i1 + ... + i_M-2)) do
10        π = i_1, ..., i_M-1, N − (i_1 + ... + i_M-1)
11        for (t = 1, t = T) do // Traverse forward
12          E = E + R(π, S_t, A_t)
13          S_t+1 = f(π, S_t, A_t)
14        endfor
15        if E > E_opt-stat then
16          E_opt-stat = E
17          π_opt-stat = π
18        endif
19        E = 0.0
20      endfor
        ⋮
26    endfor
27  endfor
28  return (E_opt-stat, π_opt-stat)
```

Figure 5.8: An Optimal-Static Omniscient orchestration algorithm.

intervals as it possesses complete knowledge of the future request arrivals whilst Optimal-Static Omniscient assumes complete knowledge of the future request arrivals but it makes an optimal orchestration decision at the beginning which remains unchanged for the remainder of the time horizon. We described all three algorithms and gave details of their implementation. An additional algorithm called Local Neighbour Search was introduced to support Greedy and Optimal Omniscient given frequent and prohibitive requirement on their part to compute optimal orchestration policy for a given set of circumstances.

In the next chapter, the three new server orchestration algorithm are used to assess efficacy of the Self-Organising algorithm in terms of revenue stream performance. The Greedy algorithm provides performance comparison for a standard orchestration heuristic whilst Optimal Omniscient algorithm provides an upperbound revenue stream comparison for any orchestration technique. Finally, the Optimal-Static Omniscient provides an upperbound revenue stream performance comparison for *ad hoc* static techniques applied to manage servers in an internet hosting centre.

Chapter 6

Efficacy of Self-Organising Server Ensemble Orchestration

Better indeed is knowledge than mechanical practise. Better than knowledge is meditation. But better still is surrender of attachment to results, because there follows immediate peace.

Bhagavad Gita

6.1 Introduction

In the preceding chapters, we introduced a series of server ensemble orchestration algorithms based on biomimetic, classical and theoretical approaches. In order to assess both the efficacy and limitations of the biomimetic Self-Organising server orchestration algorithm, this chapter studies its comparative revenue stream performance against classical and theoretical algorithms in a simulated Internet environment characterised by highly dynamic traffic streams behaviour as well as varying demand intensity levels.

This chapter begins by introducing simulation models for Self-Organising, Greedy, Optimal Omniscient and Optimal-Static Omniscient server orchestration algorithms. In Section 6.3, we present a simulation model of the dynamic Internet environment which comprises five types of traffic stream behaviour and a baulking behaviour of Internet service users. Section 6.4 is entirely dedicated to a comparative assessment study of the Self-Organising algorithm using the simulation framework of an internet hosting centre. The assessment study is divided into two categories that define algorithmic characteristics. In an adaptiveness category, we

study an algorithm's ability to change in response to dynamic external circumstances so as to remain effective in its objective of maximising revenue stream performance. Finally, in the versatility category, we study the algorithm's ability to remain immune to changes in its own internal circumstances so as to remain effective towards the goal of maximising revenue stream performance under widely varying external circumstances.

6.2 Simulation Model: Server Orchestration Algorithms

In this section, we describe the simulation model of the dynamic server orchestration problem. We developed a Discrete Event Simulation model of an internet hosting centre along with the four server orchestration algorithms: Self-Organising; Optimal Omniscient; Greedy; and Optimal-Static Omniscient. The simulation model is implemented in C++SIM [Lit94] simulation library on an Intel processor platform with Linux operating system. The overall simulation workload related to experimental work is significantly intensive. Therefore, we used system resources at the Oxford Supercomputing Centre as well as at the School of Industrial and Systems Engineering, Georgia Institute of Technology. The following assumptions are common to all orchestration algorithms: All servers are homogeneous in terms of processing capacity and employ First-Come-First-Served scheduling policy. The time to serve a request is exponentially distributed with a mean service time depending on request type. Each server is paid a fixed revenue per request served, the amount depending on request type. An orchestrated server becomes unavailable for a fixed downtime time [AFF+01]. An internet service is hosted on a group of servers that constitute a virtual server where each server is installed with the service application and serves requests for the said internet service. A server processes waiting requests by dequeuing a request from the service queue and simulates response processing as an exponentially distributed request service time. Each service has an independent request stream that generates requests for the service. The arriving requests at the hosting centre for a specific internet service are held in that service's own queue. Each request has a waiting threshold to receive service. On crossing this threshold, a request randomly chooses to wait further or baulk.

The simulation model parameters for the hosting centre are illustrated in Table 6.1. Generally, we have chosen environment parameter values based on data from studies or observations. The number of servers in the hosting centre is chosen to be 50 in order to meet

time and space constraints when computing optimal omniscient orchestration polices offline as well as limiting the memory footprint when running simulations. A typical server platform of choice for a hosting centre is blade technology with a homogeneous processing capacity [HP03a, IBM03a, SUN03a]. The queueing model of a virtual server in the hosting centre is based on an M|M|m queueing system with Markovian interarrival and service times, First-Come-First-Served scheduling discipline and m servers. The orchestration downtime results from scrubbing the existing service application and associated data and installing a new service application and data. Appleby *et al.* [AFF+01] give 270–300 seconds downtime to re-purpose a server under a given network used in their hosting centre setup. Hosting a service entails the service owner and the hosting centre entering into an agreement. One of the common types of Service Level Agreements is a fixed fee plus pay-per-request-served [JKK+01, KKL+02]. Finally, we limited the co-hosting of services (two, three and four), discrete time step (1800 seconds) and the length of the time horizon (twenty-four hours) due to memory and CPU requirements of the optimal omniscient algorithm outlined in Section 5.4.1 since we deem this algorithm to be an important comparative benchmark to determine the performance quality of the biomimetic approach.

Internet Hosting Centre	
Parameter	**Value**
Total number of servers	50
Server Type	Homogeneous
Server scheduling policy	FCFS (First-Come-First-Served)
Service request queue	FIFO (First-In-First-Out)
Request service time distribution	Exponential
Server idle sleep time distribution	Exponential
Server orchestration downtime	300 seconds
Service Level Agreement (SLA)	pay-per-request-served
Number of Services hosted (virtual servers)	2, 3 and 4
Request stream	Independent per service
Simulation time horizon	24 hours

Table 6.1: Simulation model parameters for an internet hosting centre.

6.2.1 Self-Organising Server Orchestration

The self-organising server ensemble orchestration simulation model depicted in Figure 6.1 represents a case of an internet hosting centre co-hosting internet services A and B. The group of servers serving a specific request type form a virtual server. Each server in the

virtual server is installed with service applications capable of responding to the request type in question. In the self-organising model, a server from a given virtual server can be orchestrated to join another virtual server at any time. An orchestrated server suffers downtime due to purging of the currently installed Service-App and installation of the new Service-App in order that it can serve the request type of the new virtual server. As previously discussed in Section 3.3, purging or scrubbing of the currently installed application and data is necessitated by security and privacy concerns of content owners to protect themselves against fraud. A server is designated either as a scout or a forager. In natural honeybee colonies, a small proportion of bees are scouts [See95] and, accordingly, we designate a small number of servers as scouts and the rest as foragers. A scout server randomly orchestrates itself to any virtual server in the hosting centre at any time whilst forager servers randomly orchestrate themselves in response to an advertisement read from the advertisement board. Each server (scout or forager) may randomly choose to advertise its own virtual server by placing an advertisement on the advertisement board. An advertisement consists of a virtual server identifier and display expiry time which defines the length of time it will appear on the advertisement board. As depicted in Figure 6.2, an advertisement board manager monitors advertisements on the board and expunges any advertisement with expired time, thereby keeping the advertisement board updated.

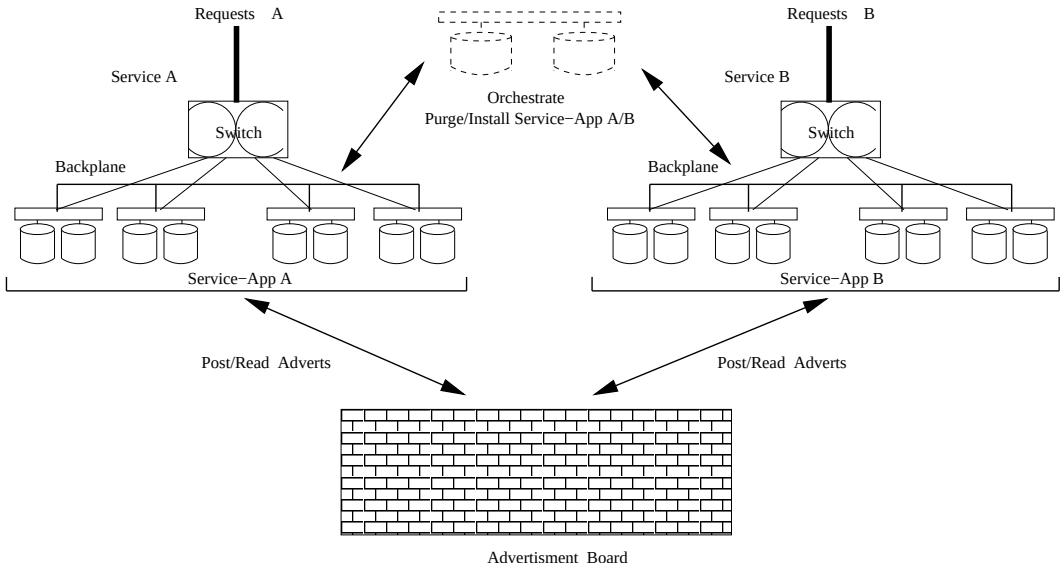


Figure 6.1: Self-Organising server orchestration simulation model.

The simulation model parameters and their values are depicted in Tables 6.2 and 6.3. We selected parameter values based on data from studies of honeybee colonies and common-sense

```
ADVERTISEMENT_BOARD_MANAGER(AdBoard)
 1  Initialisation– Current Time T_cur, Display Expiry Time T_exp
 2  forever // monitor advertisement board
 3    while AdBoard ≠ Empty do
 4      Scan adverts;
 5      if advert.T_exp ≤ T_cur then
 6      Expunge advert;
 7      T_cur = Time(); // update current time
 8    endwhile
 9  endforever
```

Figure 6.2: An advertisement board manager for the Self-Organising server orchestration algorithm.

scale reasoning. The advertisement posting threshold was chosen to be 0.1 since approximately 10% of the bees returning after collecting nectar from the surrounding countryside perform waggle dances in a natural honeybee colony [See95]. Thus, servers that are busy will post advertisements more frequently on the advertisement board. Each server decides to read the advertisement board upon expiration of the exponentially distributed advertisement reading interval and on each occasion has a chance of succeeding equal to the advertisement reading threshold. If a server succeeds then it will orchestrate itself to another virtual server but only become available to serve requests at the new virtual server after orchestration downtime has elapsed. In order to limit orchestration thrashing i.e. a server orchestrates itself often and pays excessive downtime penalty, we think that an orchestrated server should not be a candidate for re-orchestration at the new virtual server until some multiple of downtime interval has elapsed. We express this idea with a rule of thumb

$$\frac{\text{Advertisement reading interval}}{\text{Advertisement reading threshold}} >> \text{Server orchestration downtime}$$

and approximately estimate the ratio of advertisement reading interval to advertisement reading threshold to be greater than server orchestration downtime by a factor of 10. Therefore, we chose the advertisement reading scale value to be 10 and a low initial reading threshold value of 0.1. Given server orchestration downtime of 300 seconds, the mean exponentially distributed advertisement reading interval is given by

$$\text{Advertisement reading interval} = \text{AdReadScale} \times \text{AdReadThreshold} \times \text{Downtime}$$

where AdReadScale denotes parameter advertisement reading scale, AdReadThreshold denotes parameter advertisement reading threshold and Downtime denotes parameter server orchestration downtime. The advertisement reading threshold is dynamically updated according to rules and values given in Table 6.3. The rules quantitatively model bee behaviour in natural honeybee colonies where waggle dancing bees never get recruited to another patch [See03] and foragers use global information (hive's nectar intake rate cued by time to find a receiver bee to offload collected nectar) to decide whether to dance or not. Thus, bees at a profitable patch have a decreased chance of following another waggle dance. We designated one server as a scout per virtual server and the rest as forager servers in the hosting centre. In a natural honeybee colony, a returning forager provides feedback on the goodness of the nectar she has collected, whilst waiting time experienced in offloading nectar to a receiver bee indicates the colony's overall influx rate [TS94]. In the hosting centre, in order to gauge how well an individual server is performing at a given virtual server and how well the hosting centre is performing in terms of revenue, we compute the earning rate over a defined time interval. We have chosen the server and hosting centre earning rate computation intervals to be 5 seconds so that the update frequency is fairly high for the earning potential whilst filtering out the effect of very short bursts. The exponentially distributed mean service and sleep times are arbitrarily chosen to represent a type of request and typical mean processing time required to respond to such requests. Moreover, value for pay-per-request-served is arbitrarily chosen to be 0.5 pence as lower or higher values will simply scale the revenue stream and advertisement display duration. Finally, each advertisement posted by a server has a display duration time that is a function of an advertisement scale computed whenever a given interval elapses. We have chosen this interval to be 5 seconds to get representative feedback whilst filtering out short bursts. The advertisement scale is given by the ratio of mean waiting time experienced by requests across the hosting centre over a given interval to the maximum of the pay-per-request-served across the hosting centre and scaled by the parameter advertisement display duration scale. The advertisement scale is updated whenever an interval defined by advertisement scale computation interval elapses. Again, for the reason stated above, the interval duration and the interval over which mean waiting time is computed are chosen to be 5 seconds.

Self-Organising Server Orchestration	
Parameter	**Value**
Advertisement posting threshold	0.1
Advertisement reading threshold (Initial)	0.1
Advertisement reading scale	10.0
Mean exp. dist. advertisement reading interval	300 seconds
Scouts	1 per hosted service
Server earning rate computation interval	5.0 seconds
Hosting centre earning rate computation interval	5.0 seconds
Exp. dist. mean service time	15.0 milliseconds
Exp. dist. mean sleep time	15.0 milliseconds
Pay-Per-Request-Served	0.5 pence
Advertisement scale computation interval	5.0 seconds
Advertisement display duration scale	0.5
Hosting centre waiting time averaging interval	5.0 seconds

Table 6.2: Simulation model parameters for Self-Organising server orchestration algorithm.

Self-Organising Server Orchestration	
Dynamic Update Rules	**Parameter/Value**
Earning Rate (E)	**Read Threshold**
$E_{server} \leq 0.5E_{hosting\text{-}centre}$	0.60
$0.5E_{hosting\text{-}centre} < E_{server} \leq 0.85E_{hosting\text{-}centre}$	0.20
$0.85E_{hosting\text{-}centre} < E_{server} \leq 1.15E_{hosting\text{-}centre}$	0.02
$1.15E_{hosting\text{-}centre} < E_{server}$	0.00

Table 6.3: Simulation model parameters for dynamic update of advertisement reading threshold.

6.2.2 Optimal Omniscient Server Orchestration

The Optimal Omniscient server orchestration simulation model for an internet hosting centre co-hosting two internet services is illustrated in Figure 6.3. The simulation time horizon is split into n discrete time steps such that n orchestration settings which optimise revenue are computed offline using complete knowledge of future request arrival information from traffic stream traces. These traces are produced beforehand by recording request arrival information when running simulation of the hosting centre managed by the Self-Organising algorithm. At the beginning of each time step, servers are orchestrated according to the associated precomputed orchestration settings and do not change during the remainder of the time step.

A re-orchestrated server suffers fixed downtime due to time involved in purging old Service-App and installation of new Service-App. The offline computation of optimal omniscient orchestration policies from traffic stream traces is very time and space intensive as the number of services co-hosted in the hosting centre increases. We employ the Message Passing Interface (MPI) parallelised program to compute these policies on the Oxford Supercomputing Centre's 84 processor SunFire 6800 Shared Memory System with a maximum of 48 GB of memory. The simulation model parameters and their values used in simulation as well as offline computation are depicted in Table 6.4. We have chosen a time step duration of 1800 seconds to manage the significant offline computation necessary to compute orchestration policies. The request stream traces used in offline computation are recorded with total request arrivals for each time step duration over the complete simulation time horizon. The value for service time, idle sleep time and pay-per-request-served are as per parameter values of the Self-Organising model.

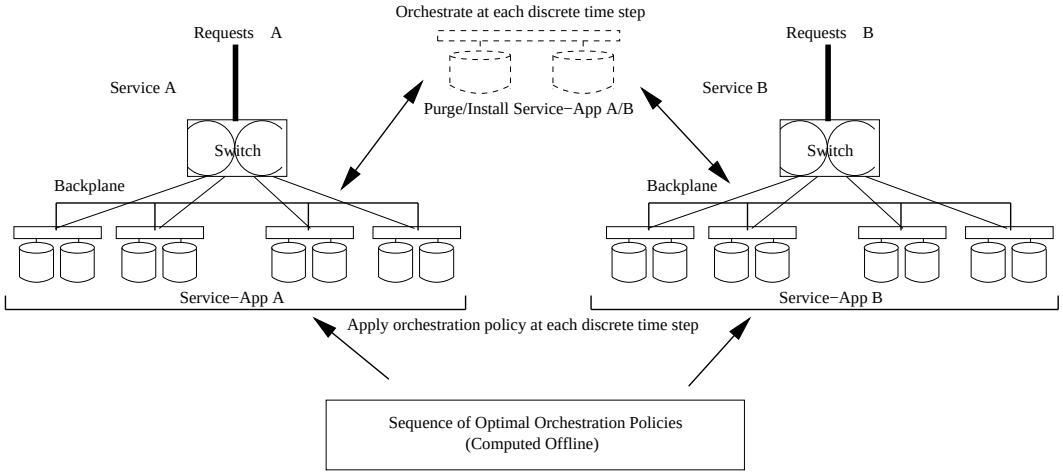


Figure 6.3: Optimal Omniscient server orchestration simulation model.

Optimal Omniscient Server Orchestration	
Parameter	**Value**
Exp. dist. mean service time	15.0 milliseconds
Exp. dist. mean sleep time	15.0 milliseconds
pay-per-request-served	0.5 pence
Time step duration	1800 seconds
Stream trace data time step duration	1800 seconds

Table 6.4: Simulation model parameters for Optimal Omniscient server orchestration algorithm.

6.2.3 Optimal-Static Omniscient Server Orchestration

The simulation model for the Optimal-Static Omniscient server orchestration is depicted in Figure 6.4 for an internet hosting centre with two services hosted. The simulation time horizon is treated as a single time step such that a single orchestration setting or no re-orchestration at all that optimises revenue over the entire time horizon is computed offline using complete knowledge of the future request arrival pattern from the pre-recorded traffic stream traces. At the beginning, the optimal-static orchestration policy determines how servers in the hosting centre are orchestrated amongst co-hosted services and subsequently remain fixed for the entire simulation time horizon. As a result, servers do not suffer any downtime. The simulation model parameters and their values used in simulation as well as offline computation are depicted in Table 6.5. Again, the traces are pre-recorded when running simulation of the hosting centre managed by the Self-Organising algorithm. The values for service time, sleep time and pay-per-request-served are as per parameter values of the Self-Organising simulation model.

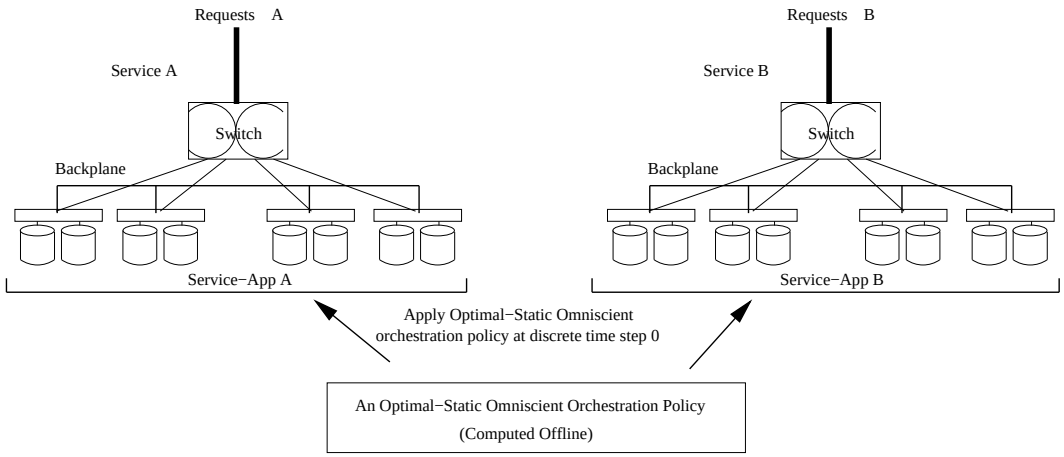


Figure 6.4: Optimal-Static Omniscient server orchestration simulation model.

Optimal-Static Omniscient Server Orchestration	
Parameter	**Value**
Exp. dist. mean service time	15.0 milliseconds
Exp. dist. mean sleep time	15.0 milliseconds
pay-per-request-served	0.5 pence
Stream trace data time step duration	1800 seconds

Table 6.5: Simulation model parameters for Optimal-Static Omniscient server orchestration algorithm.

6.2.4 Greedy Server Orchestration

The Greedy server orchestration simulation model for an internet hosting centre co-hosting two services is given in Figure 6.5. The simulation time horizon is divided into n discrete time steps such that a Greedy orchestration policy is computed online for the next time step using the traffic stream arrival data from the current time step. At the beginning of each time step, servers are orchestrated amongst the co-hosted services and remain unchanged for the rest of the time step. As before, an orchestrated server suffers downtime owing to time involved in re-purposing the server for new internet service. The online computation of Greedy orchestration policy takes into account the current state of the hosting centre, downtime and the request arrival pattern forecast for the next time step based on the arrival data from the current time step. A server simulates servicing a request as an exponentially distributed service time. The simulation model parameters and their values are given in Table 6.6. We have set the request arrivals forecast interval to be 1800 seconds in order to obtain representative forecast data on which to base the orchestration decision. The time step duration is chosen to be 1800 seconds as a trade-off for minimising the downtime penalty for orchestrating servers whilst being responsive to dynamic changes in request arrivals.

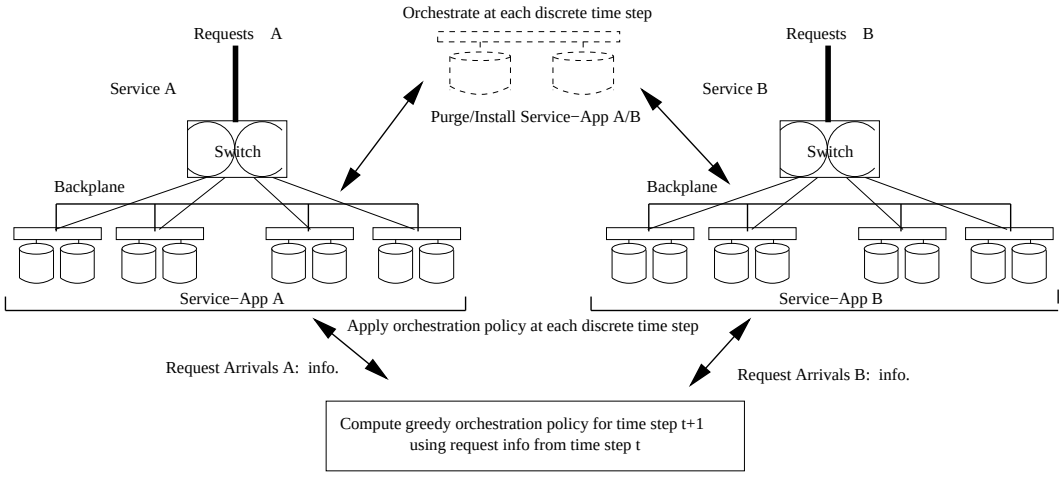


Figure 6.5: Greedy server orchestration simulation model.

6.3 Simulation Model: Internet Request Traffic

In this section, we describe the simulation model of the internet request traffic. The model comprises generating requests either synthetically from a specific traffic distribution or using

Greedy Server Orchestration	
Parameter	**Value**
Exp. dist. mean service time	15.0 milliseconds
Exp. dist. mean sleep time	15.0 milliseconds
pay-per-request-served	0.5 pence
Time step duration	1800 seconds
Forecast interval	1800 seconds

Table 6.6: Simulation model parameters for Greedy server orchestration algorithm.

```
SYNTHETIC_TRAFFIC_GENERATOR(TrafficDist, ServiceQueue)
 1  Initialisation– Interarrival Time T_intarr, Service Request R
 2  forever
 3    T_intarr = TrafficDist.Draw()
 4    Wait(T_intarr)
 5    Enqueue(R, ServiceQueue)
 9  endforever
```

Figure 6.6: Synthetic request stream generator.

trace data from logs of request activity recorded by a commercial service provider and baulking behaviour of service users. We developed a Discrete Event Simulation model of traffic generation and baulking behaviour in a C++SIM simulation model library. The traffic generation model has three synthetic streams(Inhomogeneous Poisson, Step and Heavy Tail), Real stream based on trace data from commercial service provider and, as the name suggests, a Hybrid stream based on both synthetic and trace data.

As depicted in Figure 6.6, the synthetic request streams are generated by drawing an interarrival time from a relevant traffic distribution and en-queueing a request on a service queue upon its expiry. The request logs used for generating the Real traffic stream record activity at a granularity of one second. Therefore, we know how many requests arrived in each one second interval but we cannot determine the request arrival pattern within the interval in question. We make the assumption that the requests are equally spaced within one second interval and use the interarrival time accordingly as illustrated in Figure 6.7. The Hybrid traffic stream is generated by randomly selecting one of the three synthetic streams or trace data streams to draw interarrival times. The selected stream is used to generate request traffic for a random time interval and, at the end of the interval, another stream is selected. Below, we discuss details of these request traffic streams.

```
TRACE_TRAFFIC_GENERATOR(TraceData, ServiceQueue)
 1  Initialisation– Interarrival Time T_intarr
 2  Initialisation– Total number of request per second n
 3  while
 3    n = TraceData.GetTotRequests // Total request arrivals in one second
 3    if n = NULL then // end of trace data
 3      break // exit
 3    else
 4      T_intarr = 1/(n+1) //equally spaced requests in one second interval
 4      for i = 1, n do
 4        Wait(T_intarr)
 5        Enqueue(R, ServiceQueue)
 4      endfor
 3    endif
 9  endwhile
```

Figure 6.7: Trace request stream generator.

```
HYBRID_TRAFFIC_GENERATOR(TrafficDist, TraceData,ServiceQueue)
 1  Initialisation– Stream Duration Interval T_si, Interval Distribution IntvDist
 2  Initialisation– Stream Identifier StreamID, Current Time T_cur
 3  Initialisation– Synthetic streams S = {StreamID}
 4  forever
 5    T_si = IntvDist.Draw()
 6    StreamID = RandomSelectStream()
 7    for t = T_cur, t = T_cur + T_si do // use stream for stream duration interval
 8      if StreamID ∈ S then
 9        SYNTHETIC_TRAFFIC_GENERATOR(TrafficDist.StreamID, ServiceQueue)
10      else
11        TRACE_TRAFFIC_GENERATOR(TraceData, ServiceQueue)
12      endif
13    endfor
14  endforever
```

Figure 6.8: Hybrid request stream generator.

6.3.1 Traffic Streams

The Inhomogeneous Poisson request stream pattern is depicted in Figure 6.9 for mean requests per second over a twenty-four hour simulation time horizon. We have chosen this stream pattern because statistical analysis of a real call centre concluded an Inhomogeneous Poisson call arrival process that is a function of date, time of the day and type of call amongst other factors [BGM+02]. The stream simulates the Inhomogeneous Poisson process by generating

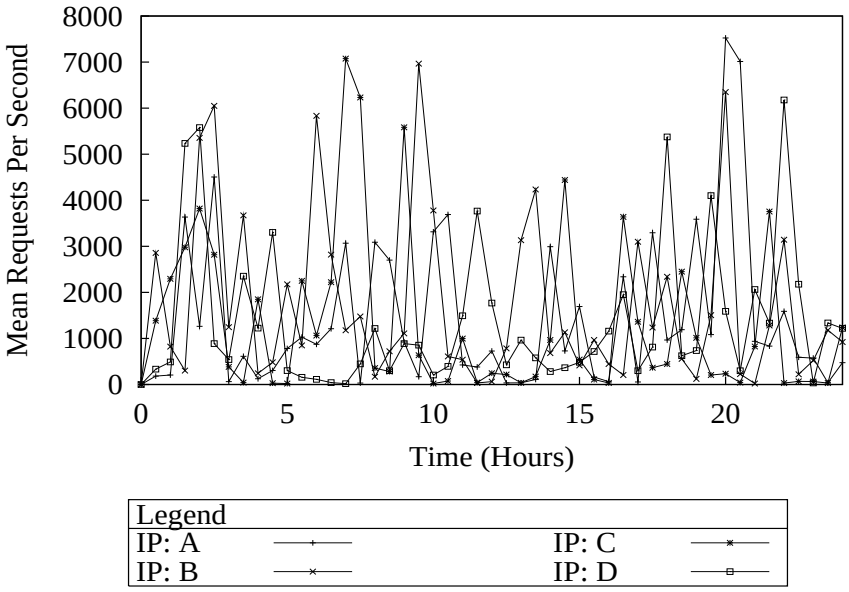


Figure 6.9: Inhomogeneous Poisson request arrival pattern.

exponentially distributed interarrival times and using two parameters to control variability in terms of intensity and rate of change of intensity. The intensity parameter sets the intensity to be in the range from a minimum of 10 to a maximum of 10,000 requests per second whilst the rate of change parameter controls the duration of a specific request intensity. The sojourn time of the inhomogeneous process at a specific intensity is chosen randomly within range from a minimum of 450 seconds to a maximum of 1800 seconds.

The Step request stream is a reduced form of Inhomogeneous Poisson process in which the process is either at a maximum or minimum level of intensity controlled by a ratio parameter which determines the peak-to-trough variation. The interarrival times are drawn from an exponential distribution with one of two levels of intensity (i.e peak and trough intensity of mean requests per second). The parameter that controls the sojourn time of the process at either peak or trough request intensity ranges from minimum of 1800 seconds to a maximum of 7200 seconds with exact time chosen randomly within this range. The Step request stream is illustrated in Figure 6.10 for mean requests per second of 3000 (peak) and 200 (trough) over a twenty-four hour simulation time horizon.

The Heavy Tail request stream is modelled by drawing interarrival times from an exponential distribution whose mean is drawn from the Cauchy distribution which has an undefined mean and standard deviation. We chose this stream distribution because numerous statistical studies have revealed heavy tail request arrival behaviour [AJ99, CB97, SYZ00, Wig01]. The

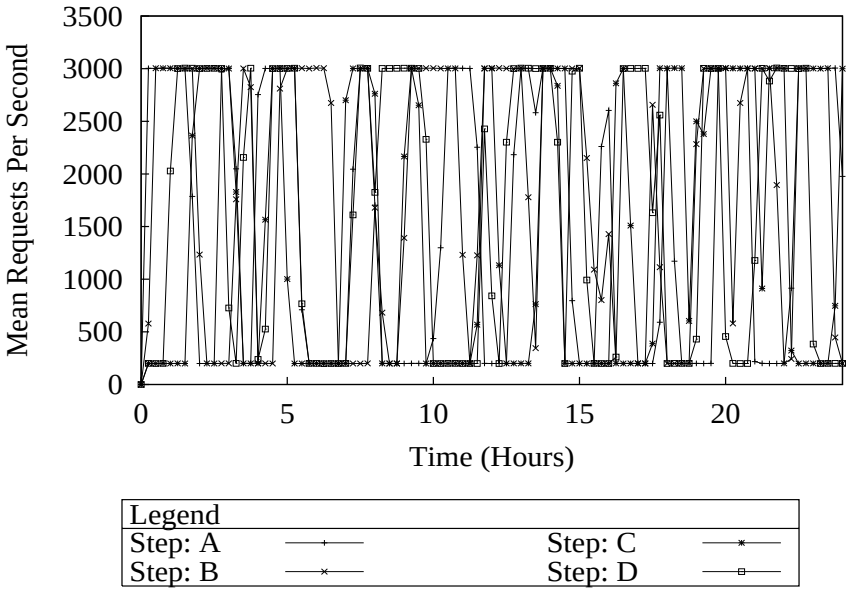


Figure 6.10: Step request arrival pattern.

mean interarrival time used by the exponential distribution is defined by a slightly truncated Cauchy distribution as $\frac{1}{tan(\theta)}$, $0 \leq \theta \leq \frac{\pi}{2} - \epsilon$. The request intensity is scaled by a parameter to be within 0 and 45,000 requests per second. The rate of change of the mean interarrival times is governed by another parameter which allows the stream to use a specific mean request intensity from a minimum of 1800 seconds to a maximum of 7200 seconds with exact time chosen randomly within the range. The resulting request pattern is depicted in Figure 6.11 for a twenty-four hour simulation time horizon.

The Real request stream is derived from request activity logs provided by an international service owner (a confidentiality agreement prohibits us from disclosing any details here) who creates content and delivers it by managing its own server ensemble infrastructure. The request activity logs record service access at one second granularity such that we can create a trace file with an entry for each second and a number of request arrivals during the second in question. As illustrated in Figure 6.7, the trace data file is used to generate equally spaced request arrivals for each second. The resulting mean requests per second pattern is depicted in Figure 6.12.

Finally, the Hybrid stream is derived from a combination of the Inhomogeneous Poisson, Step, Heavy Tail and Real streams detailed above. The variation in the stream is governed by a random choice of stream (synthetic or trace) to be the source of the request arrival characteristic and rate of change parameter that controls how long a chosen stream will be

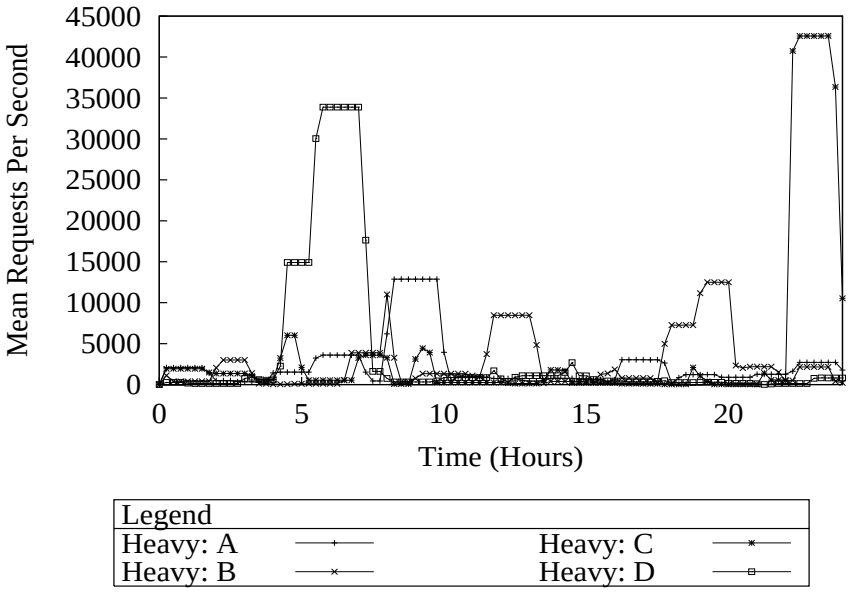


Figure 6.11: Heavy tail request arrival pattern.

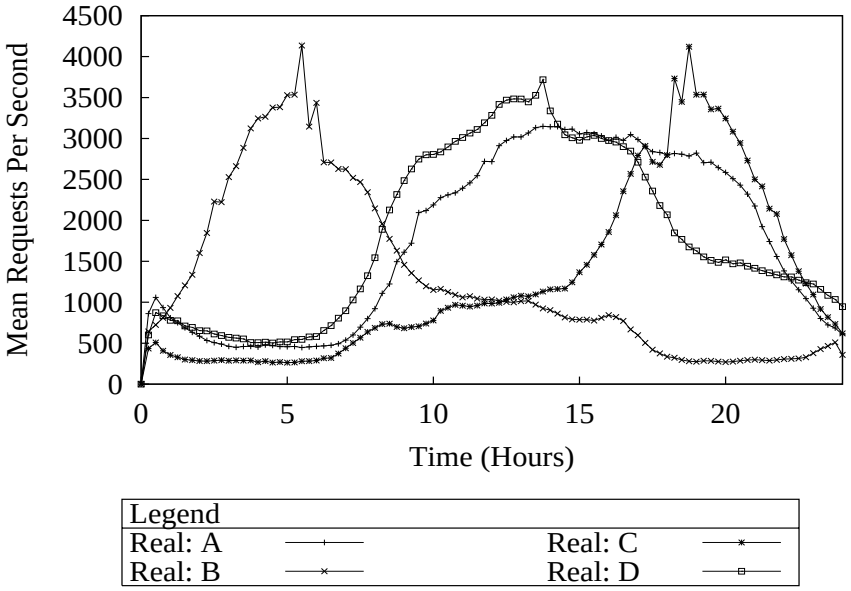


Figure 6.12: Request arrival pattern derived from traces of real internet services.

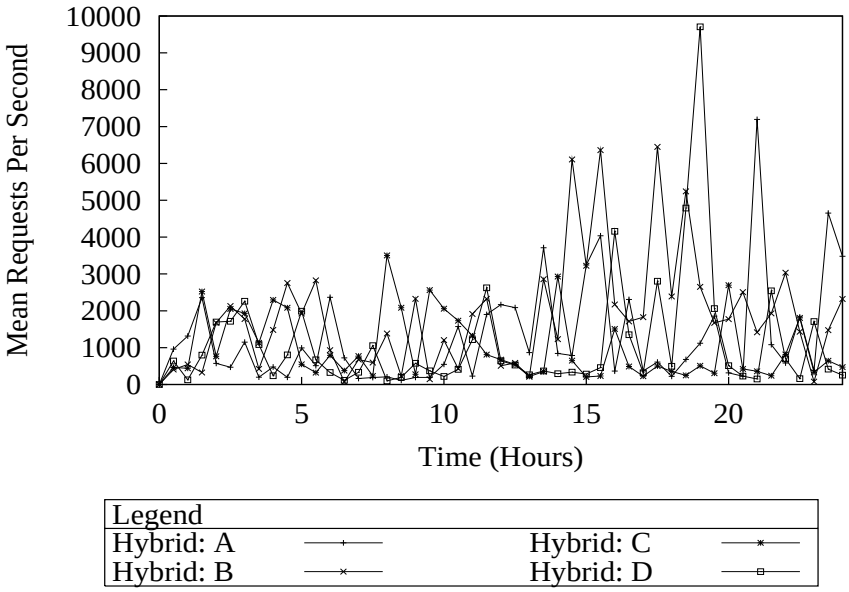


Figure 6.13: Hybrid request arrival pattern.

used. A specific stream chosen randomly is used for a time duration randomly drawn from the range 5760 seconds to 7200 seconds. The resulting request pattern is illustrated in Figure 6.13 for a twenty-four hour simulation time horizon.

The simulation model parameters and their values are depicted in Table 6.7 for the above described traffic streams. The behaviour of each traffic stream, with the exception of Real stream, which is derived from trace data, is governed by the distribution of the Poisson arrival intensity and its rate of change, which is defined by the parameter Change of Mean IntArr. For the Hybrid stream, mean interarrival times are governed by individually chosen streams but the random choice of stream and random length of time a given stream is used is given by the interval defined by parameter Change of Stream. We have chosen various values for each stream with respect to behaviour parameters so the orchestration algorithms can be stress tested in various ways to reveal circumstances where performance may or may not be favourable.

6.3.2 Hosting Centre: Demand Intensity

Each of the five traffic request streams detailed above has predefined demand intensity. Since each service hosted on the server ensemble of the hosting centre has an independent traffic stream, the intensity of the hosting centre overall varies as a function of request stream type and the number of services co-hosted. However, to compare different circumstances we need

Traffic Streams				
Traffic Stream	**IntArr Distribution**	**Mean IntArr Range** (seconds)	**Change of Mean IntArr** (seconds)	**Change of Stream** (seconds)
Inh. Poisson	Exponential	0.1 - 0.0001	450 - 1800	Not Applicable
Step	Exponential	0.005 and 0.00033	1800 - 7200	Not Applicable
Heavy Tail	Exponential (Mean←Cauchy)	∞ - 0.000022	1800 - 7200	Not Applicable
Real	Not Applicable (Request Trace)	Not Applicable (Request Trace)	Not Applicable (Request Trace)	Not Applicable
Hybrid	As for each stream	As for each stream	As for each stream	450 - 1800

Table 6.7: Simulation model parameters for various traffic streams.

a uniform measure of demand intensity for the hosting centre independent of traffic stream type and the number of services co-hosted. We define the demand intensity of the hosting centre as the ratio of the total number of servers required to service all request arrivals to the total number of servers available for servicing requests in the hosting centre. Assume we have M services co-hosted on n servers in the hosting centre with M independent traffic streams arrivals for respective services. Then the demand intensity, ρ, of the hosting centre will be given by

$$\rho = \frac{1}{n} \sum_{i=1}^{M} \frac{\bar{\lambda}_i}{\bar{\mu}_i} \tag{6.1}$$

where $\bar{\lambda}_i$ is the mean arrival rate and $\bar{\mu}_i$ is the mean service rate of the ith hosted service. Thus ρ denotes the fraction of time each server would be busy if the load were evenly distributed among all servers with no downtime incurred. A value of $\rho > 1$ corresponds to a workload arrival rate in excess of the hosting centre's capacity. In order for the hosting centre to operate at a specific ρ independent of the request streams and number of services either the mean arrival rate or mean service rate of each service needs to be controlled.

We wanted the hosting centre to operate at four levels of ρ for each instance of a traffic stream and service hosting. Given the highly variable traffic stream behaviour, it is far easier to control $\bar{\mu}$ than $\bar{\lambda}$ to meet a specific ρ value. For each traffic stream, we can observe $\bar{\lambda}$ for a given service hosting by running a simulation over a fixed time horizon. Subsequently, we can use the $\bar{\lambda}$ to compute the $\bar{\mu}$ of the hosting centre as

$$\bar{\mu} = \frac{\bar{\lambda}}{n\rho}. \tag{6.2}$$

We fixed the time horizon to twenty-four hours and ran simulations to observe a mean arrival rate $\bar{\lambda}$ of the hosting centre for each service hosting and traffic stream instance. From the observation, we calculated mean service rates $\bar{\mu}$ of the hosting centre with server ensembles of 50 servers for ρ value of 0.25 (low), 0.7 (medium), 1.0 (high) and 1.6 (super high) using Equation 6.2 as given in Table 6.8. We used these service rates to set the demand intensity of the hosting centre at four levels for experimentation. Of course, if any traffic stream control parameters are varied then service rates have to be recalculated to obtain the appropriate levels of demand intensity.

Demand Intensity						
	2-VS		**3-VS**		**4-VS**	
Traffic Stream	$\bar{\lambda}$	$\bar{\mu}$	$\bar{\lambda}$	$\bar{\mu}$	$\bar{\lambda}$	$\bar{\mu}$
Step	3225.94		4623.60		6280.55	
ρ = Low		258.08		369.89		502.44
ρ = Medium		92.17		132.10		179.44
ρ = High		64.52		92.47		125.61
ρ = Super High		40.32		57.80		78.51
Heavy Tail	2210.88		2824.66		4855.12	
ρ = Low		176.87		225.97		388.41
ρ = Medium		63.17		80.70		138.72
ρ = High		44.21		56.49		97.10
ρ = Super High		27.64		35.31		60.69
Real	3059.9		4331.58		6135.81	
ρ = Low		244.79		346.53		490.86
ρ = Medium		87.43		123.76		175.31
ρ = High		61.20		86.63		122.72
ρ = Super High		38.25		54.14		76.70
Inhomogeneous Poisson	3234.88		4588.55		5984.39	
ρ = Low		258.79		367.08		478.75
ρ = Medium		92.43		131.10		170.98
ρ = High		64.70		91.77		119.69
ρ = Super High		40.44		57.36		74.80
Hybrid	3772.92		4415.62		5335.66	
ρ = Low		301.83		353.25		426.85
ρ = Medium		107.80		126.16		152.45
ρ = High		75.46		88.31		106.71
ρ = Super High		47.16		55.19		66.70

Table 6.8: Mean arrival and service rates for different traffic streams.

```
BAULK_MANAGER(ServiceQueueID, WaitThresh, BaulkProb, ScanIntv)
 1  Initialisation– Current Time T_cur, Request Birth Time T_bt
 2  forever // monitor service queue
 3    while ServiceQueueID ≠ Empty do
 4      T_cur = Time(); // update current time
 5      Wait(ScanIntv);
 6      Scan Requests;
 7      if (T_cur - Request.T_bt) ≥ WaitThresh && Baulk(BaulkProb) then
 8      Expunge Request;
 9    endwhile
10  endforever
```

Figure 6.14: Baulk manager.

6.3.3 Baulk Manager

The response delay experienced by internet service users in receiving service plays a fundamental role in their behaviour. It is not uncommon for a user to baulk or click-away as a result of excessive response delay. The baulk manager simulates user behaviour for each co-hosted service in the hosting centre by continuously scanning the respective service queue at regular intervals and randomly deciding with some probability to expunge requests if waiting time in the queue has equalled or exceeded some waiting threshold as illustrated in Figure 6.14, where the function Baulk(p) returns true with probability p and the function Time() returns current simulation clock time. The waiting threshold represents user patience for receiving a response to a submitted request whilst the request scanning interval represents how often service users think about baulking, having waited for at least the waiting threshold. The baulking probability represents the chance that a user may then baulk by cancelling the request. The simulation model parameters and their values are given in Table 6.9. The waiting threshold is determined by the behaviour of users when experiencing a delay between a mouse-click and receiving a response. Nielsen [Nie00] suggests that a delay greater than 10 seconds causes users to become distracted and makes them more likely to click-away. The parameter Baulk scan interval represents how often users, having waited more than 10 seconds for service, think about baulking and the parameter Baulk probability represents the likelihood of baulking on each occasion. Given that we have chosen a baulk scan interval of 1.01 seconds for each waiting user, a low baulk chance of 4% is chosen based on appropriate scaling.

Baulk Manager	
Parameter	**Value**
Wait Threshold	10.0 seconds
Baulk probability	0.04
Baulk scan interval	1.01 seconds

Table 6.9: Simulation model parameters for Baulk manager.

6.4 Comparative Assessment of Self-Organising Server Orchestration Algorithm

The principal aim of this section is to demonstrate the adaptiveness and versatility properties of the Self-Organising server orchestration algorithm. Adaptiveness generally denotes attributes which change depending on external circumstances in order to remain effective in attaining an objective. In the context of an internet hosting centre, adaptiveness denotes the selection of orchestrations over a lengthy time horizon so as to achieve high revenue earning in response to demands from a wide variety of traffic stream behaviours. Versatility, on the other hand, generally denotes attributes which are immune to internal circumstances so as to remain effective in attaining an objective under a wide variety of external circumstances. In the context of an internet hosting centre, versatility denotes effective selection of orchestrations in spite of a wide range of internal parameter settings of the algorithm over a lengthy time horizon to achieve revenue earning that exhibits low variability (i.e. sensitivity in response to demand from a wide variety of traffic stream behaviours).

In order to demonstrate these properties, we can vary the external or internal circumstances of an orchestration algorithm and observe the resulting revenue stream over a lengthy time horizon. We assess the Self-Organising algorithm by comparing its revenue streams to theoretical and classical (conventional) orchestration algorithms' revenue streams in response to demands from a wide variety of traffic stream behaviours, intensity, co-hosting, Service Level Agreements (SLAs). We class the Optimal Omniscient and Optimal-Static Omniscient orchestration algorithms as theoretical since they require knowledge of the future, whilst the Greedy orchestration algorithm is classed as conventional since it requires knowledge only of past demands to select appropriate orchestrations. The methodology entails designing and conducting a series of experiments on the hosting centre's simulation framework followed by reasoning about its characteristics through comparative and statistical analyses. Earlier in

this chapter, we described a rich simulation model environment for an internet hosting centre, orchestration algorithms, traffic streams, varying intensity and user baulking behaviour. However, we employ experimental methodology that begins with a subset of the environment and increase complexity in stages. First, we focus on the adaptiveness of the Self-Organising algorithm in Section 6.4.1. We then turn our attention to the versatility of the algorithm in Section 6.4.2.

6.4.1 Adaptiveness

In this section, we design, conduct and analyse results of experiments to demonstrate the adaptiveness of the Self-Organising server orchestration algorithm. First, we consider experiments with a subset of the external environment and evaluate performance. We then consider experiments that test performance under a complete external environment and a differentiated SLA hosting fee schedule.

6.4.1.1 Real Stream

The first experiment considers the case of co-hosting two, three and four services on an internet hosting centre with demand for service driven by Real traffic streams without a notion of demand intensity. The real traffic streams, depicted in Figure 6.12, labelled Real:B/C were used to drive two-hosted services; Real:A/B/C were used to drive three-hosted services; whilst all were used to drive four-hosted services. The simulation model parameters and their values are given in Table 6.1 for the hosting centre, Tables 6.2, 6.3, 6.4, 6.5 and 6.6 for respective algorithms. Finally, the baulk manager simulation model parameters and their values are depicted in Table 6.9.

The revenue earned by the hosting centre over a twenty-four hour time horizon using the Self-Organising, Optimal Omniscient, Greedy and Optimal-Static Omniscient orchestration algorithms for each service hosting configuration is depicted in Table 6.10 whilst normalised performance is depicted in Figure 6.15. As expected, the Optimal Omniscient algorithm outperforms the other three algorithms. The Self-Organising algorithm performs the best of the other three algorithms, earning revenue within 1.9% (two services), 8.9% (three services) and 10.0% (four services) of the Optimal Omniscient algorithm. The Self-Organising algorithm outperforms the Greedy algorithm by 0.96% (two services), 51.2% (three services) and 10.3% (four services); it outperforms Optimal-Static Omniscient by 25.0% (two services), 11.7%

Revenue($\pounds \times 10^6$)			
Algorithm	**2-Services**	**3-Services**	**4-Services**
Optimal Omniscient	1.07	1.35	1.42
Self-Organising	1.05	1.24	1.29
Greedy	1.04	0.82	1.17
Optimal-Static Omniscient	0.84	1.11	1.24

Table 6.10: Revenue from Real traffic streams.

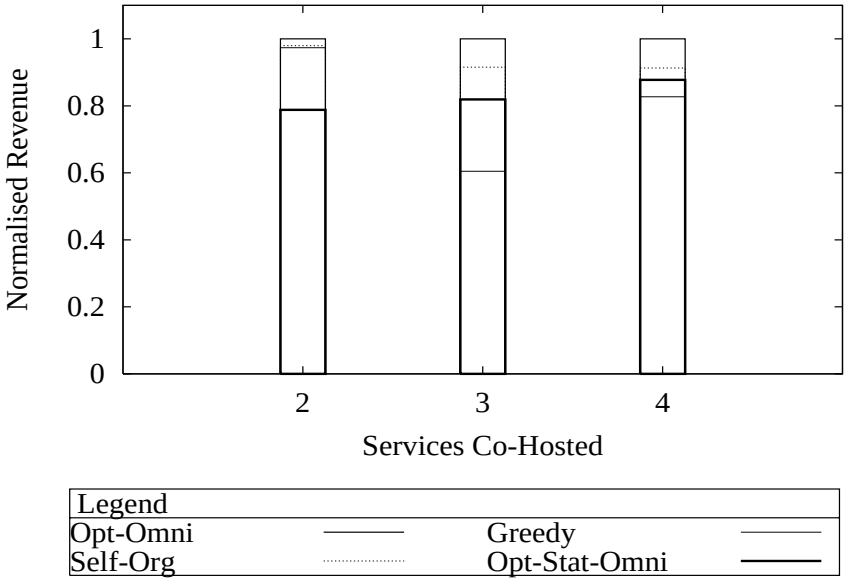


Figure 6.15: Real revenue stream normalised with respect to the Optimal Omniscient algorithm.

(three services) and 4.0% (four services). It is evident from the experimental results that the Self-Organising algorithm adapts well to the highly varying request stream patterns generated by real customers of internet services.

6.4.1.2 Step and Heavy Streams

In the second experiment, we consider the case of co-hosting two and three services with demand for services driven by Step and Heavy traffic streams. We use the Step request stream in the experiment to control service demand at ratios of 4:1 and 10:1 peak-to-trough mean requests per second. Again, the simulations are run without any notion of hosting centre demand intensity. Examples of Step and Heavy request streams were depicted in Figure 6.10 and Figure 6.11. The simulation model parameters for Heavy and Step stream are as given in Table 6.7 with the modification of the Mean interarrival time Range parameter to 0.00125:0.005 second (4:1) and 0.0005:0.005 second (10:1) for Step traffic streams. The

hosting centre, four algorithms and baulk manager simulation model parameter values are as used in the first experiment. We run 20 simulation runs for each instance of hosting, traffic stream and orchestration algorithm employed to manage the server ensemble of the hosting centre. Each simulation run uses a different initial seed value for distribution function utilised to generate respective traffic streams. Therefore, this experiment entailed running $2 \times 3 \times 4 \times 20 = 480$ simulation runs.

The mean revenue earned by the hosting centre over a twenty-four hour time horizon using all four algorithms is depicted in Table 6.11 and the normalised performance is depicted in Figures 6.16, 6.17 and 6.18. The revenues reported are the mean revenue based on the 20 simulation runs for each combination of hosting, traffic stream type and orchestration algorithms. We define variability in Step traffic stream along the dimension of peak-to-trough ratio (i.e. higher ratio implies higher stream variability). In the case of low variability Step request streams with a peak-to-trough ratio of 4:1, the Self-Organising algorithm is outperformed by the Greedy, Optimal-Static Omniscient and, as expected, Optimal Omniscient algorithms for both two and four services hosted. The Greedy algorithm outperforms the Self-Organising algorithm by 6.8% (two services) and 5.1% (four services). The Optimal-Static Omniscient and Optimal Omniscient algorithms outperform the Self-Organising algorithm by approximately 8.2% (two services) and 8.9% (four services). In the case of higher variability Step request streams with a 10:1 peak-to-trough ratio, the Self-Organising algorithm is outperformed by Optimal Omniscient algorithm but it performs better than the Greedy or Optimal-Static Omniscient algorithms. It performs within 24.2% (two services) and 31.0% (four services) of the Optimal Omniscient algorithm. It outperforms the Greedy algorithm by 11.8% (two services) and 8.7% (four services); it outperforms Optimal-Static Omniscient by 13.1% (two services) and 11.1% (four services). In the case of Heavy request streams, the performance of the Self-Organising algorithm is similarly good. It performs within 8.2% (two services) and 24.0% (four services) of the Optimal Omniscient algorithm. It outperforms the Greedy algorithm by 2.1% (two services) and 7.5% (four services) whilst, against Optimal-Static Omniscient, it outperforms by 5.4% (two services) and 3.1% (four services). To test for statistical significance of the results, we used the standard method for a paired observations test. This method is appropriate because within each single simulation run, the same random seed was used for each orchestration algorithm. The sample means and standard errors of the differences are given in Table 6.12. Every comparison supports the qualitative conclusions above with a

Mean Revenue($\pounds \times 10^6$)					
Services	**Stream**	**Self-Org**	**Greedy**	**Opt-Stat-Omni**	**Opt-Omni**
2	Step (4:1)	0.73	0.78	0.79	0.79
4	Step (4:1)	0.79	0.83	0.86	0.86
2	Step (10:1)	0.95	0.85	0.84	1.18
4	Step (10:1)	1.00	0.92	0.90	1.31
2	Heavy	0.97	0.95	0.92	1.05
4	Heavy	1.00	0.93	0.97	1.24

Table 6.11: Revenue from Step and Heavy traffic streams.

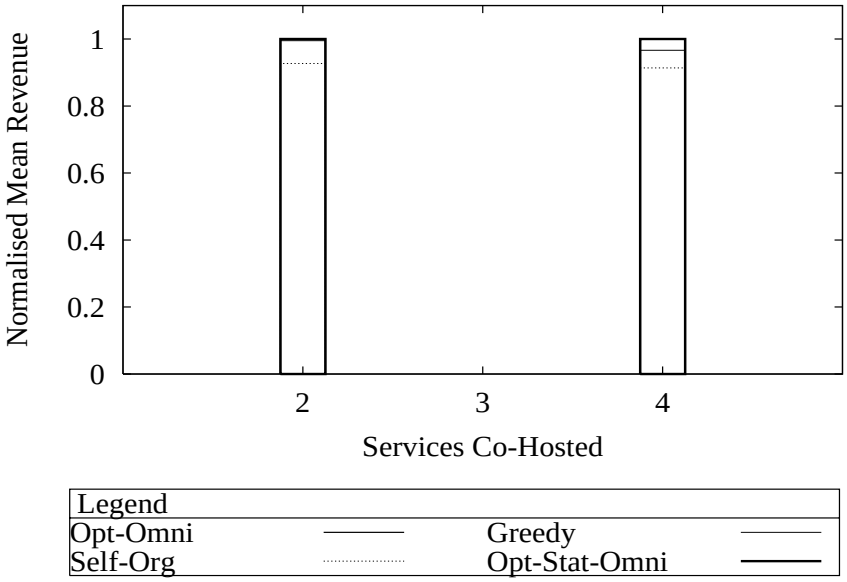


Figure 6.16: Step (4:1) revenue stream normalised with respect to the Optimal Omniscient algorithm.

confidence level better than 99%.

6.4.1.3 Step Stream Variability

To assess the adaptiveness of the Self-Organising algorithm to variability of the Step traffic stream, we extend the second experiment to consider the case of co-hosting two, three and four services on an internet hosting centre with demand for service driven by Step request streams with varying degrees of peak-to-trough ratio of mean requests per second. The degree of peak-to-trough ratio is denoted by a numerical scale where $1 = 1:1, \ldots, 30 = 30:1$ and the mean interarrival times for each peak-to-trough ratio are set as depicted in Table 6.13. The simulation model parameter values for the hosting centre, each algorithm and baulk manager are as given previously. The revenue performance of each algorithm normalised to revenue

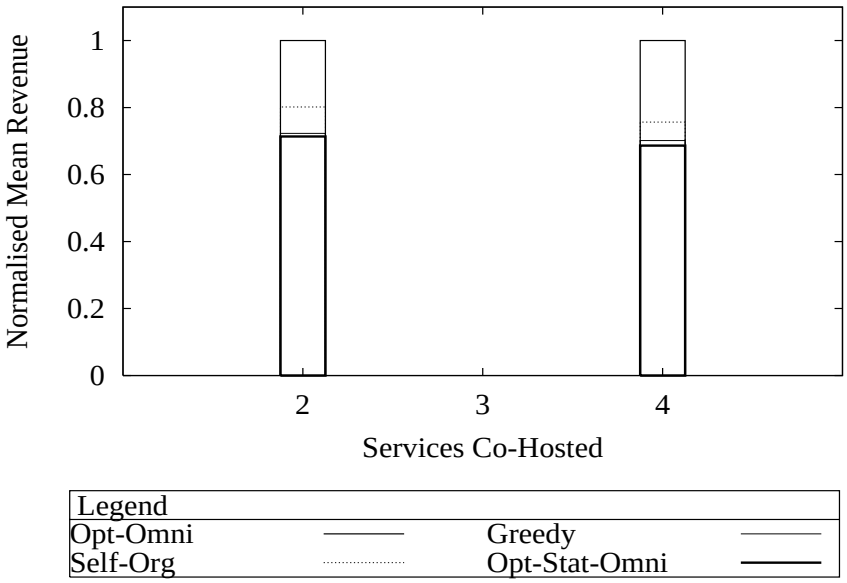


Figure 6.17: Step (10:1) revenue stream normalised with respect to Optimal Omniscient algorithm.

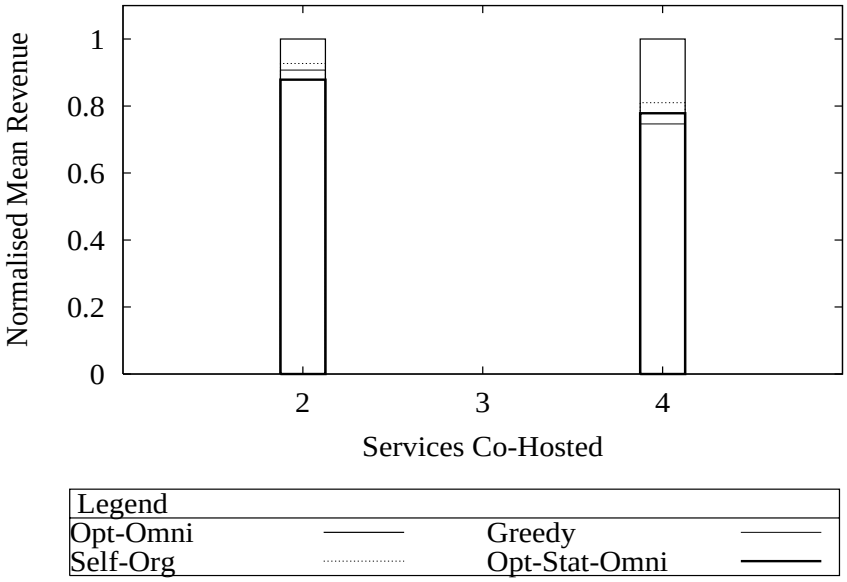


Figure 6.18: Heavy revenue stream normalised with respect to Optimal Omniscient algorithm.

Revenue ($\pounds \times 10^6$)							
Service	**Stream**	**SO-G**		**SO-OSO**		**SO-OO**	
		Mean Diff.	$\frac{\overline{\sigma}}{\sqrt{n}}$	Mean diff.	$\frac{\overline{\sigma}}{\sqrt{n}}$	Mean Diff.	$\frac{\overline{\sigma}}{\sqrt{n}}$
2	Step(4:1)	-0.054	0.0018	-0.057	0.0019	-0.057	0.0019
4	Step(4:1)	-0.045	0.0019	-0.074	0.0016	-0.074	0.0016
2	Step(10:1)	0.093	0.0091	0.104	0.0088	-0.234	0.0067
4	Step(10:1)	0.072	0.0100	0.092	0.0086	-0.319	0.0072
2	Heavy	0.021	0.0071	0.051	0.0099	-0.077	0.046
4	Heavy	0.078	0.0600	0.039	0.0174	-0.236	0.013
Legend: SO:Self-Organising, G: Greedy, OSO: Optimal-Static Omniscient, OO: Optimal Omniscient							

Table 6.12: Paired observation test revenue difference and standard deviation for Step and Heavy traffic streams.

earned by the Optimal Omniscient algorithm is depicted in Figures 6.19, 6.20 and 6.21 for two, three and four services co-hosted respectively.

The main qualitative observation we draw from the first and second experiment results is that the Self-Organising algorithm performs very well, except in situations of low variability. We emphasise that all these results are for an untuned Self-Organising algorithm. That is, we chose the parameter values for the algorithm based on biological data or common-sense scaling reasoning, as described in previous sections, and froze those values before we ran any test cases. Despite the stringency imposed by not tuning, our algorithm adapts well to a wide variety of circumstances as manifested by the different types of traffic streams and the range of arrival behaviours.

6.4.1.4 Variability of Traffic Streams and Demand Intensity with Differentiated SLA Fees

In the three experiments described above, we demonstrated the adaptiveness of the Self-Organising algorithm with respect to a subset of the external environmental factors. These experiments considered only three traffic streams (Step, Heavy and Real) with behaviour defined by parameters depicted in Tables 6.7 and 6.13 whilst disregarding any notion of the hosting centre's demand intensity, which varies as a function of the number of services co-hosted given that each service has an independent traffic stream. Furthermore, the SLA

Peak-To-Trough	**Mean Interarrival Time**	
Ratio	**Peak** (second)	**Trough** (second)
1:1	0.00500	0.005
2:1	0.00250	0.005
3:1	0.00170	0.005
4:1	0.00125	0.005
5:1	0.00100	0.005
6:1	0.00083	0.005
7:1	0.00071	0.005
8:1	0.00063	0.005
9:1	0.00056	0.005
10:1	0.00050	0.005
15:1	0.00033	0.005
20:1	0.00025	0.005
25:1	0.00020	0.005
30:1	0.00017	0.005

Table 6.13: Interarrival times for Step traffic stream variability.

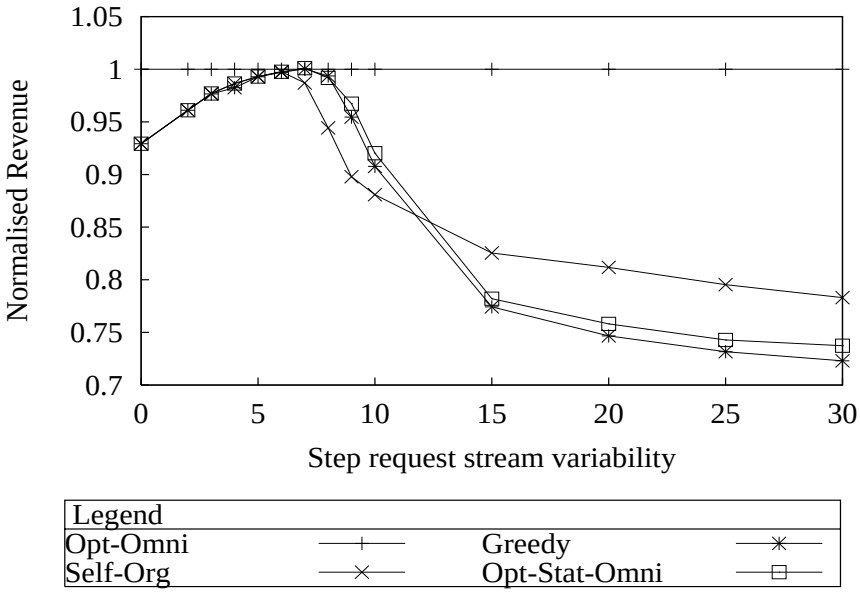


Figure 6.19: Adapting to variability in peak-to-trough ratio of Step request stream for two services co-hosted.

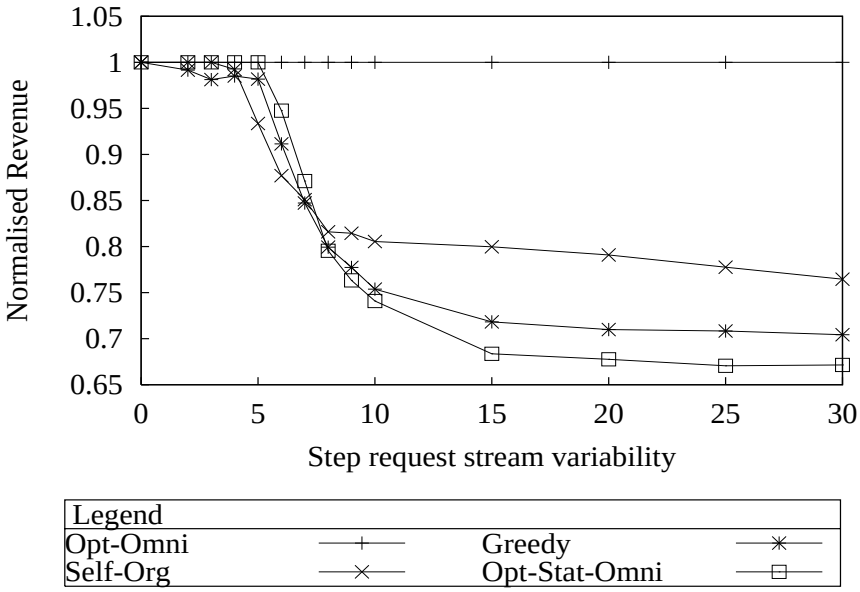


Figure 6.20: Adapting to variability in peak-to-trough ratio of Step request stream for three services co-hosted.

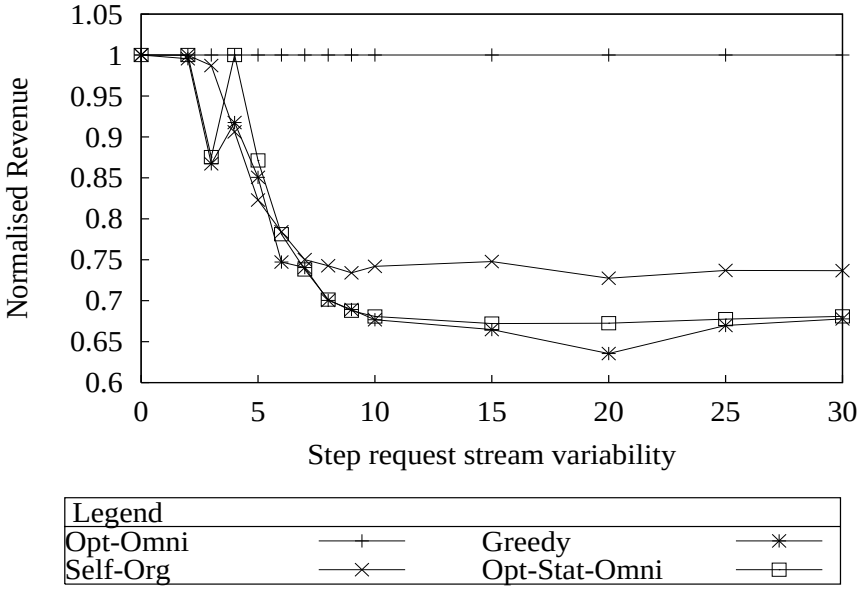


Figure 6.21: Adapting to variability in peak-to-trough ratio of Step request stream for four services co-hosted.

pay-per-request-served fees amongst hosted services were not differentiated. That is, equal fees were paid to serve requests for any service. In this section, we design experiments to assess the adaptiveness of the Self-Organising algorithm under the expanded external environment that includes additional types of traffic streams (Inhomogeneous Poisson and Hybrid) and four levels of the hosting centre's demand intensity (low, medium, high and super high) irrespective of the number of services co-hosted. Moreover, we expand the internal environment of the hosting centre to encompass differentiated SLA fees amongst co-hosted services. Earlier, we limited the scope of variability to Step traffic stream in terms of peak-to-trough ratio along the demand intensity axis. Here, the experiment also considers different timescale which determine the rate at which a new mean interarrival time is chosen for relevant traffic streams. The timescale used has an effect on behaviour of traffic streams by altering its degree of variability along the time axis.

The experimental methodology employs a multi-pronged approach to evaluating the adaptiveness property of the Self-Organising algorithm. It considers simulation runs with single unique seed values as well as multiple random seed values across different orchestration algorithms and external environments. The results are analysed and a comparative revenue stream performance is given in terms of other assessment algorithms. The statistical significance of adaptiveness is analysed using a nonparametric technique known as the Wilcoxon matched-pairs signed-ranks test [Con80]. The technique takes into account the size and direction of difference between matched pairs and tests the null hypothesis that the difference between each pair has a median value of zero. Briefly, to perform the test, we calculate differences of each matched pair and rank order the difference by their absolute value i.e. giving one to smallest difference, two for the next smallest and so on with zero differences ignored and all tied differences getting equal rank order. Next, we sum the ranks of the positive difference (W^+) and sum the ranks of the negative differences (W^-). The test statistics is the lesser of the two sums and if the null hypothesis were true and there was no difference then we would expect the rank sums for positive and negative ranks to be identically distributed. All experiments consider the case of co-hosting two, three and four services with demand for service being driven by combination of five traffic stream types (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) at three different timescale of variability and four levels of demand intensity (low, medium, high and super high).

Medium Timescale

First, we consider a medium timescale for all traffic streams except Real since it is derived from traces of arrival logs. The simulation model parameter values for traffic streams are given in Table 6.7 with timescale defined in the column titled "Change of Mean IntArr" whist timescale for streams used by Hybrid traffic stream is as per individual stream with timescale for stream change defined in the column titled "Change of Stream". For a given stream, the timescale guarantees that a change of mean interarrival time will take place randomly somewhere within the interval. The same applies to change of stream for Hybrid traffic stream type. The default simulation model parameter values for the Self-Organising algorithm are given in Table 6.2 and 6.3 whilst Tables 6.4, 6.5 and 6.6 depict default parameter values for the Optimal Omniscient, Optimal-Static Omniscient and Greedy algorithms. Earlier in Section 6.3.2, we outlined that the mean service rate ($\bar{\mu}$) is a function of demand intensity, size of the server ensemble and mean arrival rate ($\bar{\lambda}$) of a given traffic stream type for a specific service co-hosting instance. Therefore, we used the mean service rate defined for each level of demand intensity in Table 6.8 for respective traffic streams and hosting instance for simulation runs. The mean service rate is the reciprocal of the service time and as such modifies the mean exponentially distributed service time accordingly. We also modified the mean exponentially distributed sleep time to reflect the new service times. The respective algorithm's modified parameter values from its default values are depicted in Table 6.14. Finally, the simulation model parameters values used for the baulk manager are given in Table 6.9.

We begin by considering the adaptive revenue stream of Self-Organising against the Optimal Omniscient, Optimal-Static Omniscient and Greedy orchestration algorithms. We ran simulations with the hosting centre managed by each orchestration algorithm in turn. The demand for service was driven by Step, Heavy, Real, Inhomogeneous Poisson and Hybrid streams at four levels of demand intensity (low, medium, high and super high). For the reason outlined in Section 6.2, the simulation runs considered the cases of co-hosting only two, three and four internet services with revenue stream performance data collected over a twenty-four hour simulation time horizon. For each hosting instance, there are 20 simulation runs for each combination of traffic stream and intensity. Overall, there are 60 simulation runs aggregated across traffic streams, demand intensities and hosting. These runs considered single unique seed values for distribution used in traffic stream generation, baulk manager and respective

Server Orchestration Algorithms	
Parameter	**Value**
Self-Organising	
Exp. dist. mean service time	As per hosting-stream-intensity instance
Exp. dist. mean sleep time	As per hosting-stream-intensity instance
Advertisement posting threshold	0.04
Scouts	1 across the hosting centre
Pay-Per-Request-Served:	Pence:
2 services co-hosted	0.1, 0.155 (Service 0,1)
3 services co-hosted	0.1, 0.14166, 0.1833 (Service 0,1,2)
4 services co-hosted	0.1, 0.133, 0.166, 0.2 (Service 0,1,2,3)
Optimal Omniscient, **Optimal-Static Omniscient** and **Greedy**	
Exp. dist. mean service time	As per hosting-stream-intensity instance
Exp. dist. mean sleep time	As per hosting-stream-intensity instance
Pay-Per-Request-Served:	Pence:
2 services co-hosted	0.1, 0.155 (Service 0,1)
3 services co-hosted	0.1, 0.14166, 0.1833 (Service 0,1,2)
4 services co-hosted	0.1, 0.133, 0.166, 0.2 (Service 0,1,2,3)

Table 6.14: Modified Simulation model parameters from its default value for respective server orchestration algorithms.

Hosting	$\frac{\sum_{i=1}^{n} \text{Self-Org}_i}{\sum_{i=1}^{n} \text{Opt-Omni}_i}$	$\frac{\sum_{i=1}^{n} \text{Self-Org}_i}{\sum_{i=1}^{n} \text{Greedy}_i}$	$\frac{\sum_{i=1}^{n} \text{Self-Org}_i}{\sum_{i=1}^{n} \text{Opt-Stat-Omni}_i}$
2-Services	0.85	1.11	1.00
3-Services	0.80	1.07	0.98
4-Services	0.83	1.02	1.04
Overall	0.83	1.07	1.01

Table 6.15: Normalised revenue stream performance of the Self-Organising algorithm aggregated across traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 20$ and for overall, $n = 60$.

orchestration algorithms. In Table 6.15, we give the performance of the Self-Organising algorithm normalised against the Optimal Omniscient, Greedy and Optimal-Static Omniscient algorithms. Given that the Optimal Omniscient algorithm represents an upper bound on possible revenue earned by the hosting centre, the Self-Organising algorithm performs within 15% (two services), 20% (three services), 17% (four services) and 17% (overall). It performs with distinction against the Greedy algorithm where it outperforms by 11% (two services), 7% (three services), 2% (four services) and 7% (overall). The performance is competitive against Optimal-Static Omniscient for 2 and 3 services co-hosted where it performs equitably for two services and underperformed by 2% for three services whilst outperforming by 4% (four services) and 1% (overall).

The simulation runs performed above considered only a single initial seed value for distributions used in traffic streams, baulk manager and orchestration algorithms. Therefore, to further our understanding of statistical significance of the above adaptive revenue stream performance, we now extend the experiment where 10 simulation runs are performed for each instance of hosting (two, three and four), traffic stream (Step, Heavy, Inhomogeneous Poisson and Hybrid) and demand intensity (low, medium, high and super high) with unique random [Haa05] initial seed values for distributions used in traffic streams, baulk manager and orchestration algorithms. The hosting centre is managed, in turns, by Self-Organising, Greedy and Optimal-Static Omniscient algorithms. We did not consider Optimal Omniscient for these runs since it is apparent that it will always outperform other algorithms and to avoid its time-space intensive offline policy computation. Our extended experiment, therefore, entailed $10 \times 3 \times 4 \times 4 = 480$ different seed values with a total of 1440 simulation runs for three respective algorithms. In Table 6.16, we give the normalised revenue stream performance of the Self-Organising algorithm against the revenue stream of the Greedy and Optimal-Static Omniscient algorithms. We observe that the Self-Organising algorithm outperforms Greedy across each hosting instance and overall whilst being competitive against Optimal-Static Omniscient where it outperforms for two and three services co-hosted whilst underperforms for four services co-hosted. In Table 6.17, we give the statistical significance of the adaptive performance using the Wilcoxon matched-pairs signed ranks test. We observe that the adaptive revenue stream performance of the Self-Organising compared to the Greedy is highly significant overall. However, against Optimal-Static Omniscient, it is better than 95% significant for two and three services co-hosted in favour of Self-Organising but in favour of Optimal-Static

Hosting	$\frac{\sum_{i=1}^{n} \text{Self-Org}_i}{\sum_{i=1}^{n} \text{Greedy}_i}$	$\frac{\sum_{i=1}^{n} \text{Self-Org}_i}{\sum_{i=1}^{n} \text{Opt-Stat-Omni}_i}$
2-Services	1.03	1.01
3-Services	1.12	1.01
4-Services	1.03	0.99
Overall	1.06	1.00

Table 6.16: Normalised revenue stream performance of the Self-Organising algorithm aggregated across traffic streams (Step, Heavy, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 160$ and for overall, $n = 480$.

Wilcoxon Matched-Pairs Signed Ranks Test								
Host	**(Self-Org)-(Greedy)**				**(Self-Org)-(Opt-Stat-Omni)**			
	W^+	W^-	N	$\leq p$-value	W^+	W^-	N	$\leq p$-value
2	8951	2677	152	7.978e-09	7572	3904	151	6.591e-04
3	12502	378	160	5.433e-25	6834	4491	150	2.801e-02
4	8863	4017	160	3.676e-05	4528	6948	151	2.465e-02
Overall	91823.50	19802.50	472	6.237e-34	54127	48251	452	2.904e-01

Table 6.17: Wilcoxon matched-pairs signed ranks test for Self-Organising against Greedy and Optimal-Static Omniscient aggregated across traffic streams (Step, Heavy, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 160$ and for overall, $n = 480$.

Omniscient for four services co-hosted whilst being insignificant overall.

In Table 6.18, we illustrate the normalised adaptive revenue stream performance of the Self-Organising algorithm against Greedy and Optimal-Static Omniscient algorithms whilst illustrating the statistical significance of the adaptive performance in Table 6.19. The adaptive performance is an aggregate across the four traffic stream types and grouped in terms of demand intensity and services co-hosted at the hosting centre. Under this performance partitioning scheme, the Self-Organising algorithm consistently outperforms the Greedy in overall category for all four levels of demand intensity with statistical significance better than 99%. In the hosting category, it performs better than the Greedy for high and super high demand intensity with statistical significance better than 99%, whilst its performance is better than 99% significant for low and medium intensity, except when demand intensity is low and two or four services are hosted or for medium demand intensity when four services are hosted. Against the Optimal-Static Omniscient, it performs competitively for low, high and super high demand intensity whilst being outperformed for medium intensity in the overall category. However, performance is not statistically significant for low intensity whilst being

better than 95% statistically significant for medium, high and super high intensity. In the hosting category, it is outperformed for medium and high demand intensity when four services are hosted whilst performing competitively for all other instances. However, only on the four hosting instance, the result is at least 95% significant.

Next, we partition the adaptive performance by aggregating across demand intensity and group in terms of traffic streams and hosting. In Table 6.20, we depict the normalised performance of the Self-Organising algorithm against the Greedy and Optimal-Static Omniscient. The statistical significance of the performance given in Table 6.20 is computed using the Wilcoxon matched-pairs signed ranks test and depicted in Table 6.21. In the overall category, the Self-Organising outperforms the Greedy across all four traffic stream types (Step, Heavy, IP and Hybrid) with statistical significance better than 99%. In the same category, it outperforms the Optimal-Static Omniscient for Heavy stream with statistical significance better than 99% but it underperforms for IP and Hybrid stream with statistical significance less than 99% whilst being statistically insignificant for Step stream. In the hosting category, the Self-Organising consistently outperforms the Greedy with statistical significance better than 98% for Heavy and IP stream types whilst its performance is mixed for Step and Hybrid stream types as far as statistical significance is concerned. Evaluating performance of the Self-Organising against Optimal-Static Omniscient in the hosting category, we observe that it performs well for Heavy stream type with statistical significance greater than 99%. The performance is statistically insignificant for Step stream (two, three and four service hosting) and for IP and Hybrid streams (two and three service hosting). In fact, the Optimal-Static Omniscient outperforms the Self-Organising for IP and Hybrid stream types (four service hosting) with statistical significance greater than 99%.

We qualitatively observed, from the revenue stream results obtained above with an experiment environment comprising of four medium time scale traffic stream types, four level of demand intensity, three different service co-hosting and unique random seed values for each run, that the Self-Organising algorithm exhibits a highly adaptive performance in comparison to the conventional Greedy algorithm. We also qualitatively observe that its performance against the Optimal-Static Omniscient algorithm is competitive but mixed. We emphasise that the Optimal-Static Omniscient algorithm is theoretical and cannot be implemented in practise as it requires knowledge of the future. We believe that certain traffic stream behaviours coupled with its omniscient property confer an advantage to the Optimal-Static

ρ	Hosting	$\frac{\sum_{i=1}^{n} \text{Self-Org}_i}{\sum_{i=1}^{n} \text{Greedy}_i}$	$\frac{\sum_{i=1}^{n} \text{Self-Org}_i}{\sum_{i=1}^{n} \text{Opt-Stat-Omni}_i}$
Low	2-Services	0.99	1.01
Low	3-Services	1.04	1.01
Low	4-Services	1.02	1.00
Low	Overall	1.02	1.01
Medium	2-Services	1.04	1.00
Medium	3-Services	1.16	1.00
Medium	4-Services	1.00	0.95
Medium	Overall	1.07	0.98
High	2-Services	1.05	1.02
High	3-Services	1.21	1.02
High	4-Services	1.04	0.99
High	Overall	1.10	1.01
Super High	2-Services	1.05	1.01
Super High	3-Services	1.20	1.03
Super High	4-Services	1.06	1.02
Super High	Overall	1.10	1.02

Table 6.18: Normalised revenue stream performance of the Self-Organising algorithm aggregated across traffic streams (Step, Heavy, Inhomogeneous Poisson and Hybrid) where for each co-hosting instance (two, three and four services), $n = 40$ and for overall, $n = 120$.

Omniscient algorithm.

We hypothesise that traffic streams with medium or shorter timescale lead to high variability traffic stream behaviour such that the Self-Organising algorithm is penalised for being responsive, that is, adapting to a condition which may only have short duration. It is better to be unresponsive when conditions are highly dynamic as characterised by the Optimal-Static Omniscient algorithm which makes a single optimal static orchestration decision based on complete knowledge of the future traffic stream arrival behaviour and does not attempt to adapt for the rest of the time horizon. Therefore, under highly variable (short timescale) arrival behaviour, the Optimal-Static Omniscient is likely to outperform the Self-Organising algorithm. Conversely, if the traffic stream exhibits low variability (long timescale) then it is advantageous for the Self-Organising algorithm to be responsive and adapt to changes in conditions whilst the Optimal-Static Omniscient will be penalised for being unresponsive. In the next two sections, we design experiments that use traffic streams with short and long timescale to test the hypothesis.

Wilcoxon Matched-Pairs Signed Ranks Test									
ρ	Host	**(Self-Org)-(Greedy)**				**(Self-Org)-(Opt-Stat-Omni)**			
		W^+	W^-	N	$\leq p$-value	W^+	W^-	N	$\leq p$-value
Low	2	278	250	32	8.007e-01	353	143	31	4.058e-02
Low	3	717	103	40	3.795e-05	258	207	30	6.071e-01
Low	4	535	285	40	9.424e-02	194	302	31	2.944e-01
Low	Overall	4429	1849	112	2.414e-04	2161	2117	92	9.333e-01
Med	2	672	148	40	4.400e-04	407	413	40	9.732e-01
Med	3	809	11	40	8.510e-08	373	447	40	6.237e-01
Med	4	380	440	40	6.917e-01	187	633	40	2.784e-03
Med	Overall	5595	1665	120	2.683e-07	2714	4546	120	1.650e-02
High	2	670	150	40	4.867e-04	647	173	40	1.479e-03
High	3	818	2	40	4.328e-08	588	232	40	1.704e-01
High	4	686	134	40	2.131e-04	317	503	40	2.138e-01
High	Overall	6605	655	120	6.773e-15	4323	2937	120	6.974e-02
SHigh	2	724	96	40	2.512e-05	538	282	40	8.657e-02
SHigh	3	814	6	40	5.856e-06	614	206	40	6.232e-02
SHigh	4	616	124	40	1.243e-04	543	277	40	7.492e-01
SHigh	Overall	6629	631	120	4.112e-15	4965	2295	120	4.774e-04

Table 6.19: Wilcoxon matched-pairs signed ranks test for Self-Organising against Greedy and Optimal-Static Omniscient aggregated across traffic streams (Step, Heavy, Inhomogeneous Poisson and Hybrid) where for each co-hosting instance (two, three and four services), $n = 40$ and for overall, $n = 120$.

Short Timescale

In this section, we experiment with traffic streams that are controlled to exhibit high variability by using a short time scale. We test the hypothesis that the Self-Organising algorithm is likely to be outperformed by the Optimal-Static Omniscient algorithm. We modify the timescale interval defined in the column titled "Change of Mean IntArr" of Table 6.7 to 720–900 seconds as well as the "Change of Stream" interval defined for Hybrid stream. The new mean arrival rate ($\bar{\lambda}$) for each traffic stream is recorded by running a simulation for each hosting instance and the service rates used in the experiment are calculated for each instance of traffic stream, hosting and demand intensity using Equation 6.2. The simulation model parameter values, with exception of modification discussed above, for hosting centre, orchestration algorithms, traffic streams and baulk manager are set as per medium timescale experiment. The experiment considers the case of the hosting centre being managed, in turns, by Self-Organising, Greedy and Optimal-Static Omniscient algorithms with demand for service being driven by Step, Heavy, Real, Inhomogeneous Poisson and Hybrid traffic streams

Stream	Hosting	$\frac{\sum_{i=1}^{n} \text{Self-Org}_i}{\sum_{i=1}^{n} \text{Greedy}_i}$	$\frac{\sum_{i=1}^{n} \text{Self-Org}_i}{\sum_{i=1}^{n} \text{Opt-Stat-Omni}_i}$
Step	2-Services	1.04	≈ 1.00
Step	3-Services	1.11	≈ 1.00
Step	4-Services	1.01	0.97
Step	Overall	1.05	0.99
Heavy	2-Services	1.02	1.05
Heavy	3-Services	1.17	1.08
Heavy	4-Services	1.06	1.07
Heavy	Overall	1.08	1.07
IP	2-Services	1.04	1.01
IP	3-Services	1.16	0.99
IP	4-Services	1.02	0.94
IP	Overall	1.07	0.98
Hybrid	2-Services	≈ 1.00	≈ 1.00
Hybrid	3-Services	1.07	0.99
Hybrid	4-Services	1.01	0.97
Hybrid	Overall	1.03	0.99

Table 6.20: Normalised revenue stream performance of the Self-Organising algorithm aggregated across demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 40$ and for overall, $n = 160$.

at low, medium, high and super high demand intensity. We performed 60 simulation runs for each instance of hosting, traffic stream and demand intensity per algorithm with single unique initial seed values for distribution used in the traffic streams, algorithm and baulk manager. We performed an additional 60 simulation runs per algorithm with different random initial seed values. Therefore, across three algorithms, the experiment required 360 simulation runs.

In Table 6.22, we give the normalised revenue stream performance aggregated across traffic streams and demand intensity for the Self-Organising algorithm against the Greedy and Optimal-Static Omniscient algorithms. We observe that the Self-Organising algorithm is outperformed by the Optimal-Static Omniscient algorithm overall and for each instance of hosting. It performs better against the Greedy algorithm overall and the hosting instance of three and four services. The Wilcoxon statistical significance test results are given in Table 6.23, where we notice, overall, that Optimal-Static Omniscient algorithm outperforms the Self-Organising algorithm with significance greater than 99% whilst outperforming the Greedy algorithm with significance better than 97%. The results confirm the hypothesis that the responsive nature of Self-Organising algorithm will suffer in performance when traffic streams exhibit high variability in a small timescale. Essentially, it does not pay to adapt to a situa-

Wilcoxon Matched-Pairs Signed Ranks Test									
Stream	Host	**(Self-Org)-(Greedy)**				**(Self-Org)-(Opt-Stat-Omni)**			
		W^+	W^-	N	$\leq p$-value	W^+	W^-	N	$\leq p$-value
Step	2	509	19	32	4.83e06	256	272	32	8.885e-01
Step	3	820	0	40	3.717e-08	217	248	30	7.577e-01
Step	4	530	290	40	1.082e-01	137	359	31	3.036e-02
Step	Overall	5531	797	112	6.448e-12	1783	2588	93	1.235e-01
Heavy	2	592	228	40	1.470e-02	808	12	40	9.166e-08
Heavy	3	811	9	40	7.332e-09	757	63	40	3.206e-06
Heavy	4	543	277	40	7.492e-02	764	56	40	2.022e-06
Heavy	Overall	5914	1346	120	2.237e-09	6768	492	120	2.115e-16
IP	2	748	72	40	5.727e-06	501	319	40	2.238e-01
IP	3	791	29	40	3.153e-07	336	484	40	3.232e-01
IP	4	583	237	40	2.042e-02	69	751	40	4.727e-06
IP	Overall	6256	1004	120	6.200e-12	2276	4984	120	3.932e-04
Hybrid	2	460	360	40	5.058e-01	357	423	39	6.502e-01
Hybrid	3	770	50	40	1.352e-06	359	461	40	4.973e-01
Hybrid	4	553	267	40	5.544e-02	156	664	40	6.560e-04
Hybrid	Overall	5293	1967	120	1.339e-05	2475	4665	119	3.703e-03

Table 6.21: Wilcoxon matched-pairs signed ranks test for Self-Organising against Greedy and Optimal-Static Omniscient aggregated across demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 40$ and for overall, $n = 160$.

tion that very is likely to change in the very near future. In the next section, we assess the performance of the three algorithms by conducting an experiment with traffic streams that are controlled to exhibit variability along a long timescale.

Long Timescale

In the previous section, we experimented with traffic streams that exhibited high variability. In this section, we experiment with traffic streams that are controlled to exhibit low variability by using a long timescale. Here, we test the hypothesis that low variability behaviour is advantageous to the responsive Self-Organising algorithm as opposed to the Optimal-Static Omniscient algorithm. The time scale interval in column titled "Change of Mean IntArr" in Table 6.7 is modified to 5760–7200 seconds. The new timescale interval value is also used for "Change of Stream" interval defined for Hybrid stream. Again, the new mean arrival rate ($\bar{\lambda}$) for each traffic stream is recorded by running a simulation for each hosting instance and the service rates used in the experiment are calculated as described for short timescale experiment. The remaining simulation model parameter values are set as per the short time scale

Hosting	$\frac{\sum_{i=1}^{n} \text{Self-Org}_i}{\sum_{i=1}^{n} \text{Greedy}_i}$	$\frac{\sum_{i=1}^{n} \text{Self-Org}_i}{\sum_{i=1}^{n} \text{Opt-Stat-Omni}_i}$
2-Services	0.98	0.99
3-Services	1.02	0.97
4-Services	1.02	0.97
Overall	1.01	0.97

Table 6.22: Normalised revenue stream performance of the Self-Organising algorithm aggregated across short timescale traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 40$ and for overall, $n = 120$.

Wilcoxon Matched-Pairs Signed Ranks Test								
Host	**(Self-Org)-(Greedy)**				**(Self-Org)-(Opt-Stat-Omni)**			
	W^+	W^-	N	$\leq p$-value	W^+	W^-	N	$\leq p$-value
2	276	390	36	3.747e-01	281	460	38	1.968e-01
3	571	249	40	3.098e-02	259	521	39	6.859e-02
4	535	285	40	9.424e-02	273	507	39	1.040e-01
Overall	4230	2556	116	2.120e-02	2396	4390	116	6.046e-03

Table 6.23: Wilcoxon matched-pairs signed ranks test for Self-Organising against Greedy and Optimal-Static Omniscient aggregated across short timescale traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 40$ and for overall, $n = 120$.

experiment. As described previously for the short timescale experiment, the experiment also considers the case of the hosting centre being managed by respective algorithms with demand for service driven by Step, Heavy, Real, Inhomogeneous Poisson and Hybrid traffic streams at low, medium, high and super high demand intensity. We also performed 60 simulation runs for each instance of hosting, traffic stream and demand intensity per algorithm with single unique initial seed values for distribution used in the traffic streams, algorithm and baulk manager. We also performed an additional 60 simulation runs per algorithm with different random initial seed values. Therefore, across three algorithms, this experiment also required performing 360 simulation runs.

The normalised revenue stream performance of the Self-Organising algorithm versus the Greedy algorithm and the Optimal-Static Omniscient algorithm is depicted in Table 6.24 whilst the Wilcoxon statistical significance test results are depicted in Table 6.25. We note that the Self-Organising algorithm consistently outperforms the Optimal-Static Omniscient algorithm overall as well as in the hosting instance category with statistical significance better than 99%. It outperforms the Greedy algorithm in overall and three services hosted category

Hosting	$\frac{\sum_{i=1}^{n} \text{Self-Org}_i}{\sum_{i=1}^{n} \text{Greedy}_i}$	$\frac{\sum_{i=1}^{n} \text{Self-Org}_i}{\sum_{i=1}^{n} \text{Opt-Stat-Omni}_i}$
2-Services	≈ 1.01	1.06
3-Services	1.07	1.06
4-Services	1.05	1.04
Overall	1.04	1.05

Table 6.24: Normalised revenue stream performance of the Self-Organising algorithm aggregated across long timescale traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 40$ and for overall, $n = 120$.

Wilcoxon Matched-Pairs Signed Ranks Test								
Host	**(Self-Org)-(Greedy)**				**(Self-Org)-(Opt-Stat-Omni)**			
	W^+	W^-	N	$\leq p$-value	W^+	W^-	N	$\leq p$-value
2	447	219	36	7.456e-02	639	27	36	1.592e-06
3	707	113	40	6.741e-05	617	124	38	3.604e-04
4	526	294	40	1.205e-01	576	165	38	2.950e-03
Overall	4905	1881	116	3.129e-05	5324	1004	112	3.642e-10

Table 6.25: Wilcoxon matched-pairs signed ranks test for Self-Organising against Greedy and Optimal-Static Omniscient aggregated across long timescale traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 40$ and for overall, $n = 120$.

with statistical significance greater than 99%. Although it outperforms Greedy in terms of normalised revenue performance, the Wilcoxon matched-pairs ranks test is below the 95% significance level. The performance results confirm the hypothesis that the responsive nature of Self-Organising algorithm is beneficial when traffic streams exhibit variability on a long timescale. In such circumstances, the downtime penalty paid by the Self-Organising algorithm for orchestrating servers amongst co-hosted services is compensated by the increased revenue from a long lasting shift in traffic patterns.

6.4.2 Versatility

In this section, we turn our attention to designing, conducting and analysing results of experiments to illustrate the versatility of the Self-Organising server orchestration algorithm. First, we consider a tuning study that involves limited external environment and small perturbation of internal parameters. Next, we expand the external environment and consider a differentiated SLA hosting fee schedule. Under this environment, we conduct extensive tuning studies

of the Self-Organising and Greedy algorithms.

6.4.2.1 Improved and Degraded Self-Organising Algorithm

We begin with an experiment design that considers the case of co-hosting two, three and four services on an internet hosting centre employing the Self-Organising algorithm to orchestrate its server ensemble with demand for service driven by Step stream defined by 15:1 (peak-to-trough) behaviour, Heavy and Real traffic streams without any notion of demand intensity. Examples of all three traffic streams have been depicted earlier in Figures 6.10, 6.11 and 6.12. We select eight internal parameters of the Self-Organising algorithm and design a 2^{k-p} fractional factorial experiment where $k = 8$ and $p = 3$ using the JMP statistical discovery software tool from SAS Institute Inc. [JMP05]. The eight selected parameters, as depicted in Table 6.26, are set at high and low values with the default untuned parameter values shown as their mid-values. The remaining simulation model parameters for the Self-Organising algorithm are set as given in Tables 6.2 and 6.3. The simulation model parameters for Step, Heavy and Real traffic streams are as given in Table 6.7 with a modification of "Mean Interarrival Range" to 0.00033 and 0.005 second. The baulk manager's simulation model parameters are as given in Table 6.9. The 2^{8-3} fraction factorial experiment design generates 32 permutations of eight parameters set at either high or low value. Therefore, for each instance of hosting and traffic stream there will be 32 simulation runs with a total of $3 \times 3 \times 32 = 288$ simulation runs over 3 instances of hosting (2, 3 and 4 services) and traffic stream (Step, Heavy and Real).

Self-Organising Server Orchestration			
Parameter	**Low Value**	**Mid-Value**	**High Value**
Advert post thresh	0.05	0.1	0.15
Advert scale comp intv	2 sec	5 sec	8 sec
Server earning rate comp intv	2 sec	5 sec	8 sec
Hosting centre earning rate comp intv	2 sec	5 sec	8 sec
Mean sleep time	10 msec	15 msec	20 msec
Advert reading scale	5	10	15
Advert display duration scale	0.25	0.5	0.75
Scouts	1/2/3	2/3/4	3/4/5

Table 6.26: High and low simulation model parameter values for 2^{8-3} fractional factorial experiment.

The experiment generates 32 revenue streams for each instance of hosting and traffic stream from which we selected algorithm parameter settings that led to the best and second best

revenue streams. Accordingly, this gave us a 9 by 18 matrix of parameter settings from which we detected two parameters that influenced better revenue streams consistently when either was set at high or low value. We fixed these parameter values involving dynamic update rules and its advertisement reading threshold values, for the next stage of the fraction factorial experiment. We designed a 2^{6-3} fractional factorial experiment involving six parameters with high and low values as depicted in Table 6.27. This experiment design generated eight permutations of six parameters set to high or low values giving a total of $3 \times 3 \times 8 = 72$ simulation runs over three instances of hosting and traffic stream. From the revenue streams obtained, we computed a mean revenue for each parameter when it is set to high and low value. Next, we compared across traffic streams and hosting to identify four parameters for which either the high or low value resulted in better revenue streams in a sense that at least seven out of nine cases resulted in improvement of revenue stream whilst no case resulted in statistically significant degradation of revenue stream. We used these six parameter settings to obtain an improved version of the algorithm whilst inverting these parameter settings to obtain a degraded version of the algorithm. By inverting the parameter setting, we mean that if a parameter was set to a high value then an inverted form of the parameter will be its low value.

Next, we ran the final stage of the experiment with the hosting centre being managed by both versions of the algorithm, in turn, for each instance of hosting and traffic stream resulting in a further 18 simulation runs. The revenue streams from these runs are depicted in Table 6.28 along with the revenue streams from an untuned version of the algorithm. We also depict revenue streams from the Optimal-Static Omniscient algorithm and the Greedy orchestration algorithm for comparison. We do not include the Optimal Omniscient algorithm's revenue stream as it will always outperform other orchestration algorithms. We observe that the improved version of the algorithm consistently does better than the degraded version, thus confirming the best and worst tuned parameter settings.

Our principle observation is the following: despite efforts to degrade the performance of the Self-Organising algorithm, it performed well. For all 27 cases of revenue streams across three versions of the Self-Organising algorithm, hosting and traffic streams, it outperformed the Greedy algorithm in 25 cases whilst outperforming the Optimal-Static Omniscient algorithm in all 27 cases. The mean revenue performance of improved and degraded versions differs by only 5.5%, thus indicating that the Self-Organising algorithm is versatile, being competitive

against the Optimal-Static Omniscient algorithm and the Greedy algorithm, across a large range of parameter settings. These are internal parameters settings for the Self-Organising algorithm only, we did not degrade the other algorithms.

Self-Organising Server Orchestration		
	Dynamic Update Rules	**Parameter/Value**
Level	**Earning Rate (E)**	**Read Threshold**
Low	$E_{server} \leq 0.4E_{hosting\text{-}centre}$	0.50
Mid	$E_{server} \leq 0.5E_{hosting\text{-}centre}$	0.60
High	$E_{server} \leq 0.6E_{hosting\text{-}centre}$	0.70
Low	$0.4E_{hosting\text{-}centre} < E_{server} \leq 0.75E_{hosting\text{-}centre}$	0.10
Mid	$0.5E_{hosting\text{-}centre} < E_{server} \leq 0.85E_{hosting\text{-}centre}$	0.20
High	$0.6E_{hosting\text{-}centre} < E_{server} \leq 0.95E_{hosting\text{-}centre}$	0.30
Low	$0.75E_{hosting\text{-}centre} < E_{server} \leq 1.05E_{hosting\text{-}centre}$	0.01
Mid	$0.85E_{hosting\text{-}centre} < E_{server} \leq 1.15E_{hosting\text{-}centre}$	0.02
High	$0.95E_{hosting\text{-}centre} < E_{server} \leq 1.25E_{hosting\text{-}centre}$	0.03
Low	$1.05E_{hosting\text{-}centre} < E_{server}$	0.00
Mid	$1.15E_{hosting\text{-}centre} < E_{server}$	0.00
High	$1.25E_{hosting\text{-}centre} < E_{server}$	0.00

Table 6.27: High and low simulation model parameter values for 2^{6-3} fractional factorial experiment.

6.4.2.2 Tuning Self-Organising and Greedy Algorithms

In the experiment above, we demonstrated that the Self-Organising algorithm is versatile whilst being competitive against the Greedy algorithm and the Optimal-Static Omniscient algorithm when its internal parameter were perturbed over a defined range and demand for service were driven by Step, Heavy and Real traffic streams. In this experiment, we expand the range of the internal parameter perturbation and use differentiated pay-per-request-served SLA fees as well as increase the complexity of the external environment to encompass Step, Heavy, Real, Inhomogeneous Poisson and Hybrid traffic streams at low, medium, high and super high levels of demand intensity for the hosting centre. We focus on tuning the Self-Organising algorithm and the Greedy algorithm however, we do not consider the Optimal Omniscient algorithm or the Optimal-Static Omniscient algorithm here since both are theoretical algorithms and, as such, cannot be implemented in practise. The experiment is a

Revenue($\pounds \times 10^6$)						
Hosting	**Stream**	**Self-Organising**			**Opt-Stat-Omni**	**Greedy**
		Imp.	**Untuned**	**Deg.**		
2	Step (15:1)	0.97	0.91	0.88	0.86	0.84
2	Heavy	1.05	1.03	1.01	0.92	0.98
2	Real	1.21	1.05	1.17	0.84	1.04
3	Step (15:1)	1.05	1.01	0.96	0.89	0.91
3	Heavy	1.13	1.10	1.04	0.98	0.98
3	Real	1.25	1.24	1.24	1.11	0.82
4	Step (15:1)	1.09	1.01	0.98	0.95	0.95
4	Heavy	1.17	1.14	1.10	1.02	1.16
4	Real	1.28	1.29	1.25	1.24	1.17

Table 6.28: Revenue streams for three versions of the Self-Organising algorithm indicating its versatile performance.

systematic study of relationships or interactions amongst external parameters of the environment and internal parameters of the respective algorithms. Tailoring internal parameters of an algorithm to match the external environment parameters is called tuning and if an algorithm has low sensitivity to tuning then this indicates high degree of versatility. The experiment employs an iterative design-of-experiment approach that extracts significant parameters and searches for interactions. The fewer and the less significant interactions detected, the less the sensitivity to tuning indicated by the experiment. In essence, the experiment, therefore, compares the susceptibility of the two algorithms to tuning and hence compares versatility of the two algorithmic techniques.

The experiment considers the case of co-hosting two, three and four services on a hosting centre with employment, in turns, of the Self-Organising algorithm and the Greedy algorithm to orchestrate its server ensemble. The demand for service is driven by Step, Heavy, Real, Inhomogeneous Poisson and Hybrid streams at four demand intensity levels: low, medium, high and super high. We utilise a well known Design of Experiment (DOE) statistical method [CT04, Jai91] to determine the interaction between external and internal parameters that control the revenue stream of the hosting centre. First, we give details of the tuning methodology followed by tuning experiments for the Self-Organising and Greedy algorithms respectively.

Tuning Methodology

The DOE method entails constructing a process model of an internet hosting centre co-hosting multiple internet services on its server ensemble managed by an orchestration algorithm. A process model describes the total variation in one quantity y by partitioning into deterministic components that are a function of one or more quantities $x_1, x_2, \ldots$ and a random component with a given probability distribution. The general form of the model is $y = f(\vec{x}; \vec{\beta}) + \epsilon$ where y denotes the response variable, the function $f(\vec{x}; \vec{\beta})$ has two parts known as predictor variables $\vec{x}$ with its parameters $\vec{\beta}$ and ϵ denotes random error. The response variable and predictor variables are observed and the parameters are estimated by the model. Linear least square regression is a widely used modelling method which can be used to fit the data with any function of the form $f(\vec{x}; \vec{\beta}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots$ and unknown value of the parameters $\vec{\beta}$ in the regression function are estimated by determining numerical values for the parameters such that the sum of the squared deviations between the observed responses and the model function is minimised. That is, a least square criterion minimised to obtain the parameter estimates $\hat{\vec{\beta}}$ is $\sum_{i=1}^{n} = [y_i - f(\vec{x}; \hat{\vec{\beta}})]^2$. Under the DOE paradigm, a process is modelled as a black box with several internal (discrete or continuous) parameters which can be varied along with external (discrete or continuous) parameters and an output response as depicted in Figure 6.22. One of the most important parts of process modelling is collecting data for analysis by way of experiment design. In any given experiment, one or more process variables are changed in order to observe its effect on the response variable. A linear empirical model is often used to fit the experimental data. Consider, for instance, a linear model with three parameters x_1, x_2, x_3, the response y would be modelled by

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_{12} x_1 x_2 + \beta_{13} x_1 x_3 + \beta_{23} x_2 x_3 + \beta_{123} x_1 x_2 x_3 + Error \qquad (6.3)$$

if all terms are included in the model. The terms $\beta_1 x_1$, $\beta_2 x_2$, $\beta_3 x_3$ are the main effects, the terms $\beta_{12} x_1 x_2$, $\beta_{13} x_1 x_3$, $\beta_{23} x_2 x_3$ are two-way interaction effects whilst $\beta_{123} x_1 x_2 x_3$ is the three-way interaction effect. The DOE technique can be used in situations such as selecting key parameters affecting a response or maximising/minimising a response and entails five basic steps:

1. Choose an objective of the experiment or specific questions that the experiment is in-

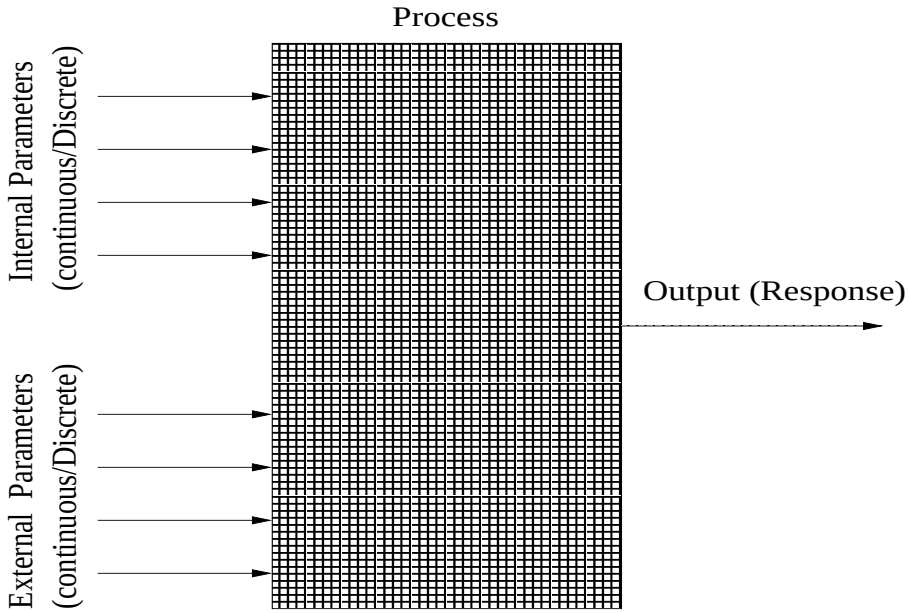


Figure 6.22: A process model for design of experiments tuning methodology.

tended to answer.

2. Choose process parameters. These include important internal parameters that are likely to affect the output response and their values (high, mid and low).

3. Choose an experimental design. This will be dependent on the number of process parameters and the objective of the experiment.

4. Conduct experiments defined by the experiment design and collect data.

5. Analyse data. This entails plotting various different graphs to fit and interpret the model in order to check if the objective questions have been answered or not.

Our objective for the experiment design is two-fold:

1. To screen important parameters which affect the response from the set of internal process parameters.

2. Determine relationships between important internal and external parameters.

Tuning Self-Organising Algorithm

In this section, we focus on tuning the Self-Organising algorithm. Let the process be an internet hosting centre employing a Self-Organising server orchestration algorithm to manage

multiple co-hosted internet services. The internal and external parameters as well as output response are identified and depicted in Table 6.29. The minimum and maximum level of a continuous parameter represent the range of perturbation used in the experiment whilst discrete parameters are varied at defined discrete values (e.g. demand intensity (ρ) which has four discrete levels of low, medium, high and super high). The internal parameters of the self-Organising algorithm represents various parameters that control the behaviour of servers in the hosting centre and rates of gathering information needed by the self-organising feedback loops. Given the number of internal and external process parameters, a fractional factorial experiment design is recommended for screening important parameters. To generate the design, we employed the JMP statistical discovery software tool from SAS Institute Inc. [JMP05]. The tool enables building of the process model via definition of process parameters, which is subsequently used to generate an experiment design with specific modelling effects and objectives. A typical fractional factorial experiment set generated by JMP is depicted in Figure 6.23 and Figure 6.24 where each line represents a simulation run with parameter setting defined by levels -1 (minimum level), +1 (maximum level) or 0 (middle level). In the process parameter Table 6.29, we did not give the parameter value associated with level 0 but it is the middle level of a parameter and represents the mid-value between minimum and maximum parameter level values.

In the DOE set, each simulation run is an instance of the hosting centre being managed by the Self-Organising server orchestration algorithm with specific setting for internal and external parameters. Examples of traffic streams are given in Section 6.3.1 with its parameter setting in Table 6.7. The default simulation model parameter values of the hosting centre are given in Table 6.1 whilst default simulation model parameter values for the Self-organising algorithm are given in Table 6.2. The mean exponentially distributed service and sleep time are set according to service rates ($\bar{\mu}$) given in Table 6.8 for each instance of traffic stream and demand intensity. The SLAs are modified for differentiated pay-per-request-served fees as follows (pence): 0.1, 0.155 (two services), 0.1, 0.1466, 0.1833 (three services) and 0.1, 0.133, 0.166, 0.2 (four services). For each simulation run, DOE sets internal and external parameter values to either high (+1), low (-1) or mid-value (0) as defined in Table 6.29. The number of simulation runs required are significant given the number of internal and external process model parameters. Therefore, in order to reduce simulation runs, the screening experiments were run initially without the parameter demand intensity (ρ). Given that the number of

```
TrafficStream,ServicesHosted,DownTime,Scouts,SleepTime,P.Post,P.Read,
Scale.AdDur,Intv.AdScale,Scale.AdRead,Intv.ServerProfit,Intv.HostProfit,
Real,2,-1,-1,1,1,-1,1,1,1,-1,1,.
Heavy,3,1,-1,-1,1,-1,-1,-1,1,-1,1,.
IP,2,0,0,0,0,0,0,0,0,0,0,.
Real,3,0,1,0,0,0,-1,-1,0,0,-1,.
Hybrid,4,1,1,-1,-1,-1,1,1,-1,1,1,.
Hybrid,4,-1,-1,1,-1,1,1,1,1,1,-1,.
IP,4,1,1,-1,1,-1,1,-1,1,1,1,.
Heavy,2,-1,-1,-1,-1,1,-1,1,-1,-1,-1,.
IP,4,0,0,0,0,0,0,0,0,0,0,.
IP,2,-1,0,0,0,-1,0,0,1,-1,0,.
Real,3,1,-1,-1,1,1,1,-1,1,1,-1,.
IP,3,1,-1,1,1,1,-1,-1,1,1,-1,.
IP,2,1,-1,-1,1,-1,-1,1,-1,-1,-1,.
Heavy,2,1,1,1,1,1,1,-1,1,1,-1,.
Step,2,-1,0,0,0,-1,-1,0,1,1,0,.
Hybrid,4,-1,1,1,1,-1,1,1,1,-1,-1,.
Real,2,1,0,-1,0,0,0,0,0,1,0,.
Real,4,-1,1,1,1,-1,1,1,-1,-1,-1,.
Hybrid,3,1,-1,1,-1,1,-1,1,-1,-1,-1,.
Step,2,1,-1,-1,-1,1,1,-1,1,-1,-1,.
Step,4,1,1,-1,1,1,1,1,-1,-1,-1,.
Hybrid,2,-1,1,-1,-1,1,1,-1,1,1,1,.
IP,4,0,0,0,1,1,1,-1,0,-1,0,.
Real,4,1,1,1,1,1,1,-1,-1,1,-1,-1,.
Real,3,1,1,1,1,1,1,-1,1,-1,1,1,.
Step,3,-1,-1,-1,0,0,1,1,0,0,1,.
Step,4,1,1,-1,-1,1,1,1,-1,-1,-1,.
IP,3,0,1,0,-1,1,0,-1,1,0,0,.
Hybrid,3,-1,-1,-1,1,-1,1,1,-1,-1,1,.
IP,3,0,0,0,0,0,0,0,0,0,0,.
Hybrid,3,1,1,-1,1,-1,-1,1,1,-1,1,.
IP,2,1,1,-1,1,-1,1,1,1,-1,-1,1,.
Heavy,2,1,1,1,-1,-1,1,-1,-1,-1,1,.
IP,4,1,-1,1,-1,1,1,1,1,-1,1,1,.
Hybrid,2,0,0,0,0,0,0,0,0,0,0,.
Real,3,-1,1,-1,-1,1,1,-1,1,1,1,-1,.
Heavy,3,0,0,0,0,0,0,0,0,0,0,.
Hybrid,2,1,0,0,-1,1,0,1,0,0,-1,.
⋮
```

Figure 6.23: An excerpt of a fractional factorial screening experimental design for Self-Organising algorithm.

```
TrafficStream,Rho,Hosted,DownTime,P.Read,Scale.AdDur,Intv.AdScale,
Scale.AdRead,Intv.ServProfit,Intv.HostProfit,Honey/Opt-Static
Heavy,Low,2,1,-1,1,-1,-1,1,1,.
Heavy,Low,3,-1,1,-1,-1,-1,-1,1,.
Heavy,Low,4,-1,-1,-1,-1,-1,-1,-1,.
Heavy,high,2,-1,-1,-1,-1,1,1,1,.
Heavy,high,3,-1,-1,-1,-1,1,1,-1,.
Heavy,high,4,-1,-1,1,1,1,-1,-1,.
Heavy,med,2,-1,-1,-1,1,1,1,-1,.
Heavy,med,3,-1,1,1,-1,-1,1,1,.
Heavy,med,4,-1,-1,1,1,-1,-1,1,.
Heavy,shigh,2,-1,-1,-1,1,-1,1,1,.
Heavy,shigh,3,-1,-1,1,-1,-1,1,-1,.
Hybrid,Low,2,-1,1,-1,-1,1,-1,-1,.
Hybrid,Low,4,-1,1,1,1,-1,-1,-1,.
Hybrid,high,2,1,-1,1,1,-1,1,-1,.
Hybrid,high,3,-1,1,1,1,-1,-1,-1,.
Hybrid,high,4,-1,1,-1,-1,1,1,-1,.
Hybrid,shigh,2,1,1,1,1,-1,-1,-1,.
Hybrid,shigh,3,1,1,-1,-1,1,-1,-1,.
Hybrid,shigh,4,1,-1,-1,-1,-1,-1,1,.
IP,high,2,1,-1,-1,1,-1,-1,1,.
IP,high,3,-1,-1,1,1,-1,1,1,.
IP,med,2,1,1,1,-1,-1,-1,1,.
IP,med,3,-1,1,1,1,1,-1,-1,.
IP,med,4,-1,-1,-1,1,-1,-1,1,.
IP,shigh,2,1,1,1,1,1,1,1,.
IP,shigh,3,-1,-1,-1,1,-1,-1,-1,.
IP,shigh,4,-1,1,1,-1,1,-1,-1,.
Real,Low,2,1,1,1,1,-1,1,-1,.
Real,Low,3,-1,-1,1,1,-1,1,1,.
Real,Low,4,-1,-1,-1,1,1,-1,1,.
Real,high,2,1,1,1,1,1,1,1,.
Real,high,4,1,1,1,-1,-1,-1,1,.
Real,med,3,-1,-1,1,-1,-1,-1,-1,.
Real,med,4,-1,-1,1,1,1,1,1,.
Step,Low,2,1,-1,-1,1,1,1,-1,.
Step,Low,3,1,1,1,1,-1,1,-1,.
Step,Low,4,1,-1,1,1,-1,1,1,.
Step,high,2,1,1,-1,-1,-1,1,-1,.
Step,high,3,-1,-1,1,-1,1,1,-1,.
Step,high,4,-1,-1,-1,1,-1,-1,-1,.
Step,med,2,1,-1,-1,-1,1,1,1,.
Step,med,3,-1,1,-1,-1,1,1,-1,.
Step,med,4,-1,1,1,-1,1,-1,-1,.
⋮
```

Figure 6.24: An excerpt of a typical fractional factorial interaction experimental design for Self-Organising algorithm.

Process: Self-Organising Algorithm and an internet hosting centre					
Internal			**External**		**Output**
	Level			*Levels*	
Parameter	*Min*	*Max*	*Parameter*	Disc./Cont.	*Response*
Scouts	1	10	Downtime	60-1200 sec	Revenue
Idle Sleep Time	1 msec	60 msec	Hosting	2,3,4	
Ad. Posting Threshold	0.04	0.2	Demand Intensity (ρ)	Low Medium High Super High	
Ad. Reading Threshold	0.4 0.133 0.0133	0.8 0.267 0.0267	Traffic Streams	Step Heavy Real IP Hybrid	
Advert Dur. Scale	0.1	4.0			
Advert Scaling Intv.	2 sec	40 sec			
Advert Reading Scale	2 sec	40 sec			
Server E.Rate Intv	2 sec	40 sec			
Centre E.Rate Intv	2 sec	40 sec			

Table 6.29: Internal and external process model parameters for Self-Organising orchestration algorithm and an internet hosting centre.

process parameters is greater than five, we chose a fractional factorial or Plackett-Burman design [CT04] to create the experiment set. We ran a 264 simulation run experiment and collected the revenue streams data to be analysed using the JMP statistical discovery software tool. We used a linear fit model with main effects since we are interested in determining important main effect parameters whilst discarding internal parameters that do not influence the revenue stream. The revenue stream response is defined in terms of the Optimal-Static algorithm, that is the difference between the Self-Organising algorithm and the Optimal-Static Omniscient algorithm's revenue streams divided by the Optimal-Static Omniscient algorithm's revenue stream where the Optimal-static Omniscient algorithm's revenue stream is obtained by running a simulation with exactly the same external parameters of the hosting centre. The output of the analysis with linear fit model parameters is depicted in Figure 6.25 and Figure 6.26 where the scaled estimates of the model parameter are given with significance t-test. Amongst the internal parameters, Scouts, Intv.AdScale, Intv.ServerProfit are significant ($(prob > |t|) < 0.05$). In addition, parameter P.Read is close to being significant. Based on these results, we fixed the following parameters for subsequent DOE iterations: Scouts, since

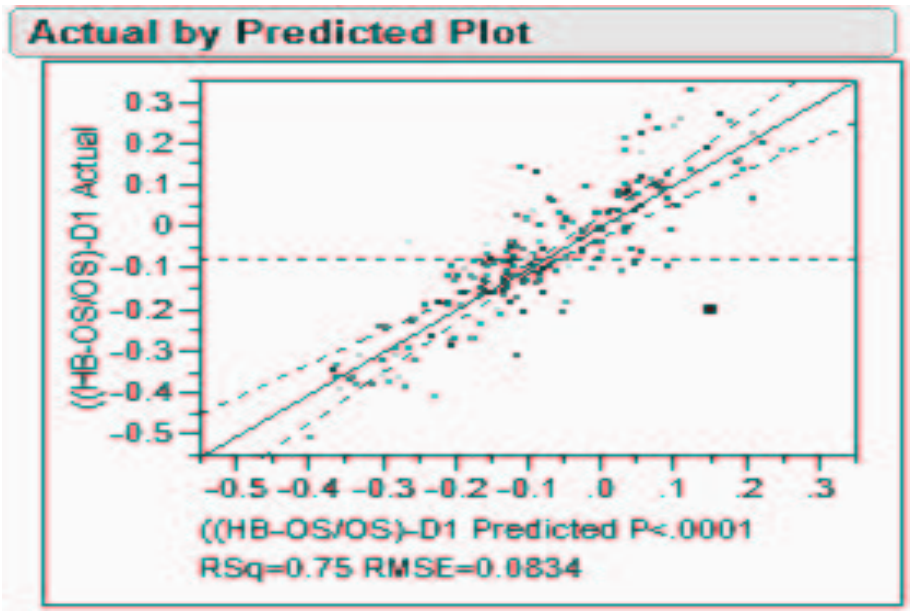


Figure 6.25: Linear model fit for the Self-Organising screening experiment design.

a low number of foragers are scouts in real honeybee colonies and the Figure 6.28 bears out this conclusion as the response is better when the number of scouts are low. The parameter SleepTime is insignificant so we froze it at mid-value whilst P.Post was frozen at low value since the response is slightly better and also a low proportion of returning foragers dance. Finally, Intv.HostProfit is insignificant and we froze it at minimum level.

For the next DOE, we set the advertisement posting threshold to 0.04, the total number of Scouts to 1 and the interval for computing hosting centre's earning rate to two seconds. The number of internal parameters was reduced by four. Adding the demand intensity (ρ) parameter Rho, we used the JMP tool to generate experiment designs for two-way interaction effects among internal and external parameters in the manner described above (an excerpt of a typical design is depicted in Figure 6.24). Again, we took an iterative approach to designing and running the simulation of experiment design. We ran the number of simulations to deduce relationships between internal and external parameters which would enable tuning the revenue stream of the Self-Organising algorithm.

The collected revenue stream data for the final experimental design with 260 simulation runs were analysed by linear modelling of main effects as well as all two-way interactions effects between internal and external parameters using the JMP tool. The analysis output is depicted in Figure 6.27, Figure 6.29 and Figure 6.30. The revenue stream response is normalised to the revenue stream of the Optimal-Static Omniscient algorithm. In Figure 6.29, a scaled estimate of the model parameters is given along with significance t-test. For external

Scaled Estimates

Nominal factors expanded to all levels

Continuous factors centered by mean, scaled by range/2

Term	Scaled Estimate	Std Error	t Ratio	Prob>|t|
Intercept	-0.08004	0.005788	-13.83	<.0001
TrafficStream[Heavy]	0.0106754	0.011547	0.92	0.3564
TrafficStream[Hybrid]	-0.058055	0.011567	-5.02	<.0001
TrafficStream[IP]	0.011083	0.011617	0.95	0.3413
TrafficStream[Real]	0.0008046	0.011832	0.07	0.9459
TrafficStream[Step]	0.0354923	0.011789	3.01	0.0030
ServicesHosted[2]	0.0264959	0.008188	3.24	0.0014
ServicesHosted[3]	0.0048469	0.008303	0.58	0.5601
ServicesHosted[4]	-0.031343	0.008315	-3.77	0.0002
DownTime	-0.082725	0.00655	-12.63	<.0001
Scouts	-0.092983	0.006454	-14.41	<.0001
SleepTime	-0.001087	0.007645	-0.14	0.8871
P.Post	-0.008094	0.006635	-1.22	0.2240
P.Read	0.0122065	0.006539	1.87	0.0635
Scale.AdDur	-0.006388	0.006584	-0.97	0.3332
Intv.AdScale	0.0720203	0.007627	9.44	<.0001
Scale.AdRead	-0.007066	0.006559	-1.08	0.2827
Intv.ServerProfit	0.0151852	0.006467	2.35	0.0199
Intv.HostProfit	-0.010154	0.006562	-1.55	0.1234

Figure 6.26: Scaled estimates of the model parameters for the Self-Organising screening experiment design.

parameters, TrafficStream, Hosted and DownTime are significant ($(prob > |t|) < 0.05$) as far as main effects are concerned. For internal parameters, Scale.AdRead is significant in terms of main effect. In terms of two-way interaction effects, there are a number of significant effects between internal and external parameters. These interaction effects are depicted in Figure 6.30 where a dotted line indicates that the effect was not modelled by the experiment design. The solid parallel lines indicates a lack of interaction between the two parameters in question when varied at respective levels whilst non-parallel lines indicate an interaction effect. We did not find any high degree interactions from the interaction plots. We observed a few low degree interaction effects amongst Scale.AdRead-DownTime, Scale.AdRead-Hosted and Scale.AdRead-Rho parameters as exhibited in respective plots by non-parallel lines. The great majority of interaction effects are negligible as exhibited by parallel or near-parallel lines in the respective plots. Overall the results indicate a remarkable dearth of tuning opportunity. This illustrates a considerable versatility of the Self-Organising algorithm to widely varying external and internal parameter settings.

The result confirms the versatility of the Self-Organising algorithm and leads one to speculate that mimicry of a method evolved over many millennia confers a versatile characteristic. Does the Greedy algorithm offer comparable versatility in the face of widely varying external parameters as well as internal parameter settings? We investigate this question in the next section using the same methodology and experimental design technique used for process

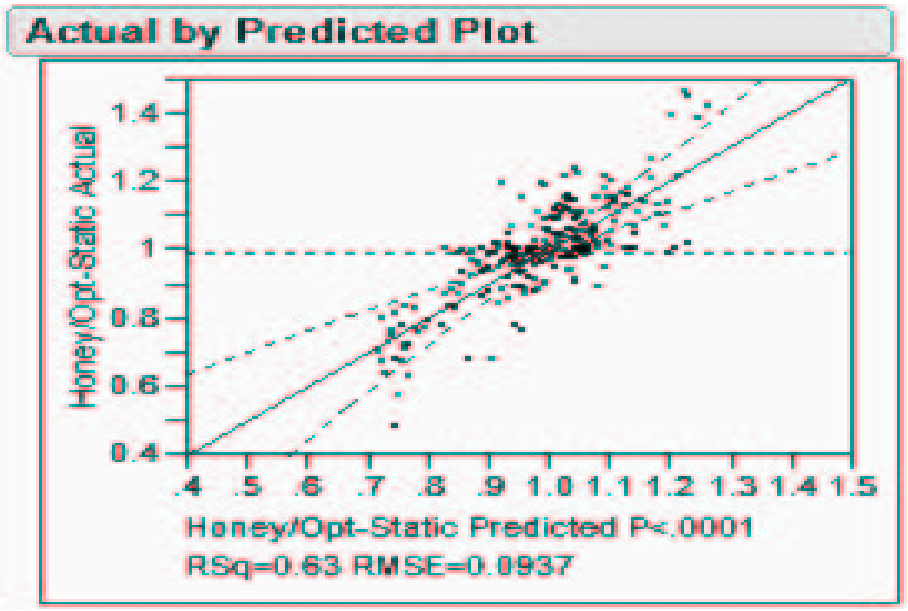


Figure 6.27: Linear model fit for the Self-Organising interaction effects experiment design.

modelling the Self-Organising algorithm and an internet hosting centre.

Tuning Greedy Algorithm

In this section, we turn our attention to the Greedy algorithm, so as to compare its tuning sensitivities with the Self-Organising algorithm. As before, we design a process model of an internet hosting centre with multiple co-hosted internet services managed by the Greedy algorithm. We employ tuning methodology in exactly the same manner as we did for the Self-Organising algorithm. Our objective for the experimental design was to determine relationships between internal and external parameters. The experiments should reveal whether there are two-way interaction effects which would suggest tuning opportunities to improve revenue stream. As we have stated earlier, lack of tuning opportunity would suggest a versatile characteristic of an algorithm as its revenue stream is immune to widely varying changes in parameter settings. On the other hand, tuning opportunities would suggest that the algorithm's revenue stream is dependent on setting internal parameters at specific values to achieve optimum performance in the face of uncontrolled external environment parameters.

Let the process be an internet hosting centre being managed by the Greedy orchestration algorithm with multiple services hosted on its server ensemble. The internal parameter of the Greedy algorithm and the external parameters of the hosting centre are identified and depicted in Table 6.30 where the meaning of maximum and minimum is as before.

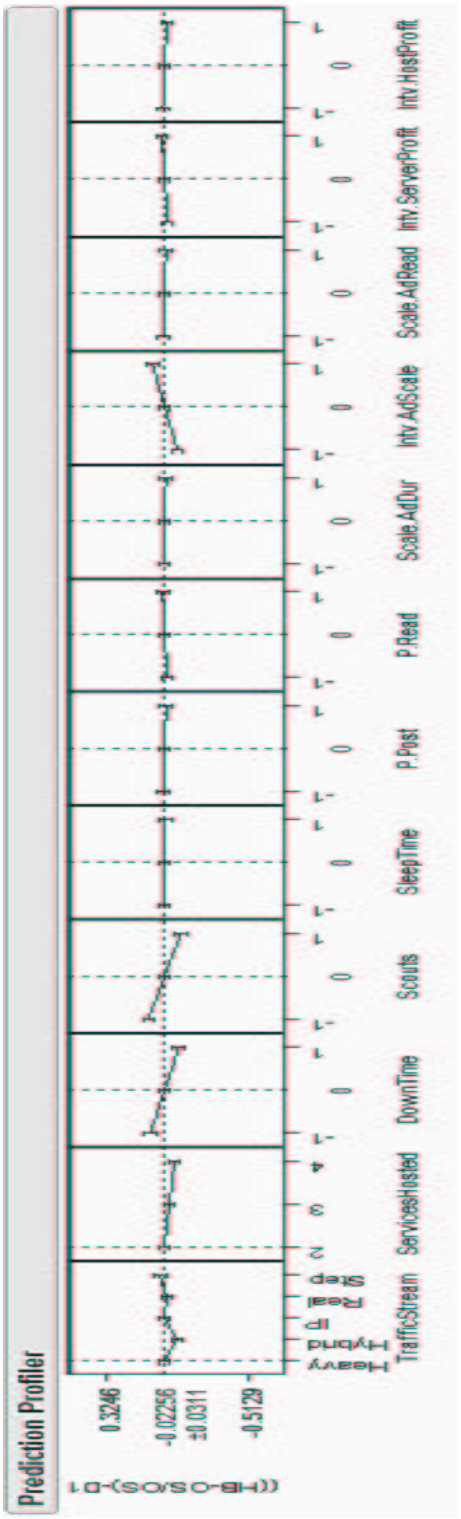


Figure 6.28: Prediction profile for the Self-Organising screening experiment design.

Process: Greedy algorithm and an internet hosting centre					
Internal			**External**		**Output**
	Level			*Levels*	
Parameter	*Min*	*Max*	*Parameter*	Disc./Cont.	*Response*
Allocation Interval	1800 sec	7200 sec	Downtime	60-1200 sec	Revenue
Forecast Interval	10 sec	1800 sec	Hosting	2,3,4	
			Demand Intensity (ρ)	Low Medium High Super High	
			Traffic Streams	Step Heavy Tail Real IP Hybrid	

Table 6.30: Greedy algorithm and internet hosting centre process parameters.

The internal parameter Intv.AllocPeriod defines the interval between changes in orchestration policy whilst the parameter Intv.ForecastPeriod defines the interval preceding an orchestration policy change during which the request forecast data is collected and subsequently used to formulate orchestration policy for the next Intv.AllocPeriod. Again, we use the JMP statistical discovery software tool to design an experiment with specific effects modelled. An excerpt of the experiment design is depicted in Figure 6.31.

In the experiment design set, each experiment is an instance of an internet hosting centre managed by Greedy orchestration algorithm with a specific parameter setting for internal and external process parameters. The demand for service is driven by Step, Heavy, Real, Inhomogeneous Poisson (IP) and Hybrid streams. Examples of traffic streams are depicted in Section 6.3.1 with their parameter settings in Table 6.7. The default simulation model parameter values of the hosting centre are given in Table 6.1 whilst default simulation model parameter values for the Greedy algorithm are given in Table 6.6. The mean service and sleep time are set according to mean service rate ($\bar{\mu}$) given in Table 6.8 for the respective instance of traffic stream and demand intensity. The pay-per-request-served SLA fees are differentiated as follows (pence): 0.1, 0.155 (2 services), 0.1, 0.1466, 0.1833 (3 services) and 0.1, 0.133, 0.166 and 0.2 (4 services). For each simulation run, DOE sets internal and external parameter values to either high (1), low (-1) as defined in Table 6.30 or mid-value (0) given by low average of the low and high values.

We ran simulations for experiment design sets comprising 300 runs and collected revenue

Scaled Estimates

Nominal factors expanded to all levels

Continuous factors centered by mean, scaled by range/2

Term	Scaled Estimate	Std Error	t Ratio	Prob>|t|
Intercept	0.993725	0.005863	169.50	<.0001
TrafficStream[Heavy]	0.0595404	0.01193	4.99	<.0001
TrafficStream[Hybrid]	-0.021944	0.011811	-1.86	0.0645
TrafficStream[IP]	-0.046225	0.011544	-4.00	<.0001
TrafficStream[Real]	0.0461247	0.011865	3.89	0.0001
TrafficStream[Step]	-0.037496	0.011751	-3.19	0.0016
Rho[Low]	0.0083252	0.010266	0.81	0.4183
Rho[high]	0.0052096	0.010116	0.51	0.6071
Rho[med]	-0.003039	0.01019	-0.30	0.7658
Rho[shigh]	-0.010496	0.0102	-1.03	0.3046
Hosted[2]	-0.024701	0.008336	-2.96	0.0034
Hosted[3]	0.005381	0.008396	0.64	0.5223
Hosted[4]	0.0193199	0.008369	2.31	0.0219
DownTime	-0.093157	0.00677	-13.76	<.0001
P.Read	-0.010143	0.006695	-1.51	0.1312
Intv.AdScale	-0.010005	0.006705	-1.49	0.1371
Scale.AdRead	0.0432058	0.006682	6.47	<.0001
Intv.ServProfit	-0.010389	0.00669	-1.55	0.1219
TrafficStream[Heavy]*Intv.AdScale	-0.032004	0.013699	-2.34	0.0204
TrafficStream[Hybrid]*Intv.AdScale	0.0102727	0.013586	0.76	0.4504
TrafficStream[IP]*Intv.AdScale	0.0194482	0.01329	1.46	0.1448
TrafficStream[Real]*Intv.AdScale	-0.009664	0.013632	-0.71	0.4791
TrafficStream[Step]*Intv.AdScale	0.0119473	0.013557	0.88	0.3792
TrafficStream[Heavy]*Scale.AdRead	-0.00613	0.013541	-0.45	0.6512
TrafficStream[Hybrid]*Scale.AdRead	0.0247347	0.013547	1.83	0.0693
TrafficStream[IP]*Scale.AdRead	0.0006262	0.013139	0.05	0.9620
TrafficStream[Real]*Scale.AdRead	-0.020128	0.013618	-1.48	0.1409
TrafficStream[Step]*Scale.AdRead	0.0008969	0.013526	0.07	0.9472
TrafficStream[Heavy]*Intv.ServProfit	-0.000812	0.013613	-0.06	0.9525
TrafficStream[Hybrid]*Intv.ServProfit	-0.001083	0.013532	-0.08	0.9363
TrafficStream[IP]*Intv.ServProfit	-0.024144	0.01314	-1.84	0.0675
TrafficStream[Real]*Intv.ServProfit	-0.009221	0.013581	-0.68	0.4979
TrafficStream[Step]*Intv.ServProfit	0.0352599	0.013487	2.61	0.0096
Rho[Low]*Intv.AdScale	0.0037105	0.011825	0.31	0.7540
Rho[high]*Intv.AdScale	-0.010267	0.011523	-0.89	0.3739
Rho[med]*Intv.AdScale	0.019182	0.011784	1.63	0.1050
Rho[shigh]*Intv.AdScale	-0.012626	0.011694	-1.08	0.2815
Rho[Low]*Scale.AdRead	0.0044574	0.011654	0.38	0.7025
Rho[high]*Scale.AdRead	0.0163643	0.011573	1.41	0.1588
Rho[med]*Scale.AdRead	-0.007099	0.01167	-0.61	0.5436
Rho[shigh]*Scale.AdRead	-0.013723	0.011657	-1.18	0.2404
Rho[Low]*Intv.ServProfit	0.0186785	0.011752	1.59	0.1134
Rho[high]*Intv.ServProfit	0.0068616	0.011537	0.59	0.5526
Rho[med]*Intv.ServProfit	-0.02469	0.011686	-2.11	0.0358
Rho[shigh]*Intv.ServProfit	-0.00085	0.011645	-0.07	0.9419
Hosted[2]*P.Read	0.0145173	0.00962	1.51	0.1327
Hosted[3]*P.Read	0.0027822	0.009597	0.29	0.7722
Hosted[4]*P.Read	-0.017299	0.00955	-1.81	0.0715
Hosted[2]*Scale.AdRead	-0.006873	0.009545	-0.72	0.4723
Hosted[3]*Scale.AdRead	-0.012084	0.009654	-1.25	0.2120
Hosted[4]*Scale.AdRead	0.0189569	0.009526	1.99	0.0478
Hosted[2]*Intv.ServProfit	0.0144663	0.009553	1.51	0.1314
Hosted[3]*Intv.ServProfit	-0.004371	0.009617	-0.45	0.6499
Hosted[4]*Intv.ServProfit	-0.010096	0.009583	-1.05	0.2933
DownTime*Scale.AdRead	0.0311045	0.006769	4.59	<.0001
DownTime*Intv.ServProfit	0.0167388	0.006757	2.48	0.0140

Figure 6.29: Scaled estimates of the model parameters for the Self-Organising interaction effects experiment design.

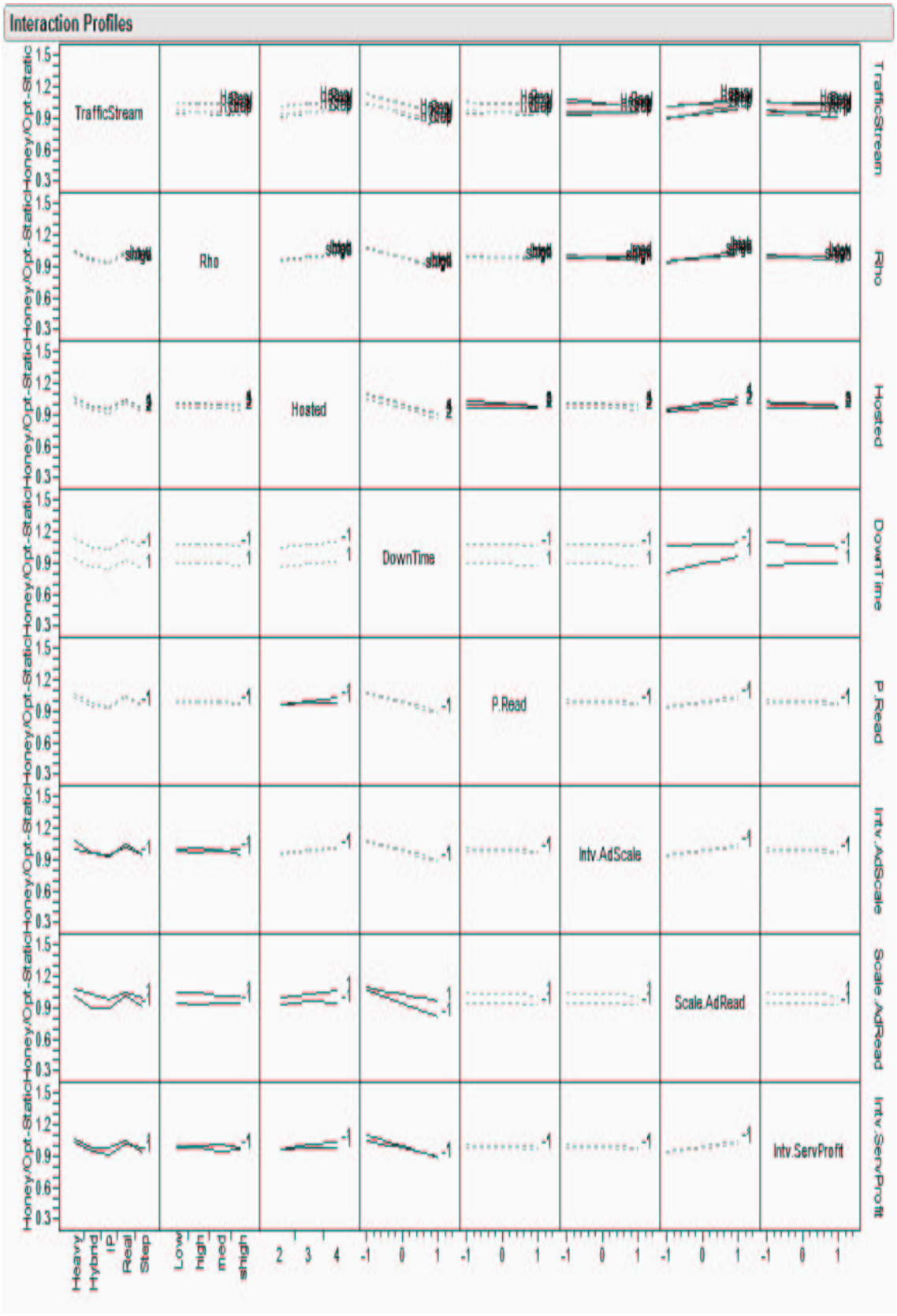


Figure 6.30: Interaction plots for the Self-Organising interaction effects experiment design where superimposed labels in rows are as follows: 1) TrafficStream(Heavy, Hybrid, IP, Real and Step); 2) Rho(low, med, high and shigh); 3) Hosted(2, 3 and 4) whilst the remainder have the labels high (+1) and low (-1).

```
TrafficStream,Rho,Hosted,DownTime,Intv.AllocPeriod,Intv.ForecastPeriod
Heavy,Low,2,-1,1,-1,
Heavy,Low,3,0,0,0,
Heavy,high,4,-1,-1,-1,
Heavy,med,2,0,0,0,
Heavy,shigh,2,-1,1,1,
Heavy,shigh,3,1,-1,1,
Heavy,shigh,4,-1,-1,-1,
Hybrid,Low,2,1,1,-1,
Hybrid,med,2,0,0,0,
Hybrid,med,3,-1,1,-1,
Hybrid,med,4,-1,1,1,
Hybrid,shigh,2,-1,-1,1,
Hybrid,shigh,3,-1,-1,1,
Hybrid,shigh,4,0,0,0,
IP,Low,2,1,1,1,
IP,Low,3,-1,-1,-1,
IP,Low,4,-1,-1,-1,
IP,high,2,1,1,-1,
IP,med,4,0,0,0,
IP,shigh,2,-1,-1,1,
IP,shigh,3,1,1,-1,
IP,shigh,4,-1,-1,-1,
Real,Low,2,-1,-1,-1,
Real,Low,3,-1,-1,1,
Real,high,4,-1,-1,1,
Real,med,2,1,1,-1,
Real,med,3,-1,1,1,
Real,med,4,1,1,1,
Real,shigh,2,-1,-1,-1,
Step,Low,4,-1,-1,-1,
Step,high,2,1,1,1,
Step,high,3,1,1,1,
Step,shigh,3,-1,1,1,
Step,shigh,4,-1,-1,1,
⋮
```

Figure 6.31: An excerpt of a fractional factorial interaction experimental design for Greedy algorithm.

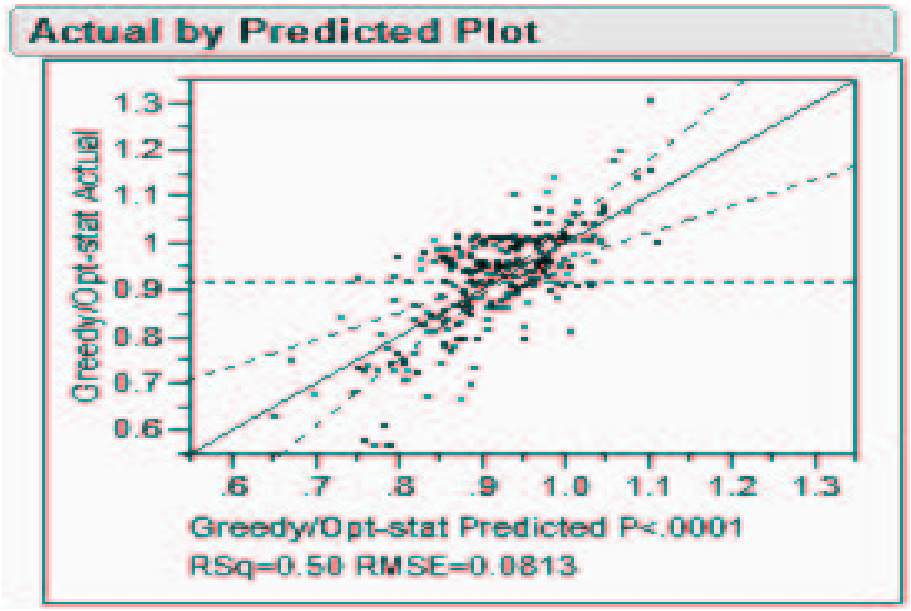


Figure 6.32: Linear model fit for Greedy process interaction experiment design.

stream data. The revenue response is normalised by the Optimal-Static Omniscient's revenue stream and data are analysed using the JMP statistical tool. The output of the analysis is depicted in Figure 6.32, Figure 6.33 and Figure 6.34. We used a linear model with main effects and all two-way interaction effects between internal and external parameters of the process. The Greedy revenue response is normalised to the Optimal-Static's revenue response. The scaled estimate of the model parameters are given in Figure 6.33 with significance t-test. We observe that the main effects TrafficStreams, Rho, Hosted, Downtime, Intv.AllocPeriod and Intv.ForecastPeriod are significant ($(prob > |t|) < 0.05$). Also, the two-way interaction effects TrafficStream-Intv.AllocPeriod, TrafficStream-Intv.ForecastPeriod, Rho-Intv.ForecastPeriod, Rho-Intv.AllocPeriod, Hosted-Intv.ForecastPeriod, Hosted-Intv.AllocPeriod, DownTime-Intv.ForecastPeriod and DownTime-Intv.AllocPeriod are all significant. In addition, there is a significant two-way interaction effect for Intv.AllocPeriod and Intv.ForecastPeriod. From the interaction plots given in Figure 6.34, we notice that non-parallel lines exist for most of the interactions between internal and external parameters. This suggests that the Greedy algorithm offers innumerable opportunities to tune its revenue stream.

The analysis of the tuning results thus finds a salient difference between the Self-Organising and greedy algorithms. The latter exhibits many significant interactions between internal (tuning) and external parameters, but the former exhibits almost none. The Greedy algorithm appears much less versatile to dynamic changes in internal and external parameters than

Scaled Estimates

Nominal factors expanded to all levels

Continuous factors centered by mean, scaled by range/2

Term	Scaled Estimate	Std Error	t Ratio	Prob>|t|
Intercept	0.9189133	0.004699	195.54	<.0001
TrafficStream[Heavy]	0.0316095	0.00934	3.38	0.0008
TrafficStream[Hybrid]	-0.065758	0.009464	-6.95	<.0001
TrafficStream[IP]	-0.023131	0.009455	-2.45	0.0151
TrafficStream[Real]	0.0717552	0.009405	7.63	<.0001
TrafficStream[Step]	-0.014475	0.009345	-1.55	0.1226
Rho[Low]	0.0182664	0.008141	2.24	0.0257
Rho[high]	0.0063066	0.008107	0.78	0.4373
Rho[med]	-0.003118	0.008151	-0.38	0.7024
Rho[shigh]	-0.021455	0.008194	-2.62	0.0093
Hosted[2]	0.0060762	0.006629	0.92	0.3602
Hosted[3]	0.0203688	0.006676	3.05	0.0025
Hosted[4]	-0.026445	0.006665	-3.97	<.0001
DownTime	-0.02161	0.005277	-4.10	<.0001
Intv.AllocPeriod	0.016525	0.005259	3.14	0.0019
Intv.ForecastPeriod	0.022985	0.005256	4.37	<.0001
TrafficStream[Heavy]*Intv.AllocPeriod	-0.007681	0.010438	-0.74	0.4625
TrafficStream[Hybrid]*Intv.AllocPeriod	0.0139789	0.010604	1.32	0.1886
TrafficStream[IP]*Intv.AllocPeriod	0.0219007	0.010593	2.07	0.0397
TrafficStream[Real]*Intv.AllocPeriod	-0.032061	0.010516	-3.05	0.0025
TrafficStream[Step]*Intv.AllocPeriod	0.0038622	0.010435	0.37	0.7116
TrafficStream[Heavy]*Intv.ForecastPeriod	0.0186925	0.010437	1.79	0.0744
TrafficStream[Hybrid]*Intv.ForecastPeriod	0.0235021	0.010612	2.21	0.0276
TrafficStream[IP]*Intv.ForecastPeriod	-0.02904	0.010594	-2.74	0.0065
TrafficStream[Real]*Intv.ForecastPeriod	-0.007358	0.010518	-0.70	0.4848
TrafficStream[Step]*Intv.ForecastPeriod	-0.005797	0.010435	-0.56	0.5790
Rho[Low]*Intv.AllocPeriod	0.0045695	0.009099	0.50	0.6160
Rho[high]*Intv.AllocPeriod	-0.000879	0.009069	-0.10	0.9229
Rho[med]*Intv.AllocPeriod	0.0053957	0.009121	0.59	0.5546
Rho[shigh]*Intv.AllocPeriod	-0.009087	0.009168	-0.99	0.3225
Rho[Low]*Intv.ForecastPeriod	0.0208798	0.009105	2.29	0.0226
Rho[high]*Intv.ForecastPeriod	-0.009545	0.009069	-1.05	0.2936
Rho[med]*Intv.ForecastPeriod	0.0064262	0.009124	0.70	0.4818
Rho[shigh]*Intv.ForecastPeriod	-0.017761	0.009188	-1.93	0.0543
Hosted[2]*Intv.AllocPeriod	-0.008875	0.007412	-1.20	0.2322
Hosted[3]*Intv.AllocPeriod	0.0152489	0.007476	2.04	0.0424
Hosted[4]*Intv.AllocPeriod	-0.006373	0.007469	-0.85	0.3943
Hosted[2]*Intv.ForecastPeriod	-0.017801	0.007415	-2.40	0.0170
Hosted[3]*Intv.ForecastPeriod	-0.018346	0.007483	-2.45	0.0149
Hosted[4]*Intv.ForecastPeriod	0.0361469	0.007468	4.84	<.0001
DownTime*Intv.AllocPeriod	0.0150667	0.005279	2.85	0.0047
DownTime*Intv.ForecastPeriod	-0.01262	0.005287	-2.39	0.0177
Intv.AllocPeriod*Intv.ForecastPeriod	-0.024869	0.005258	-4.73	<.0001

Figure 6.33: Scaled estimates of the model parameters for the Greedy process interaction experiment design.

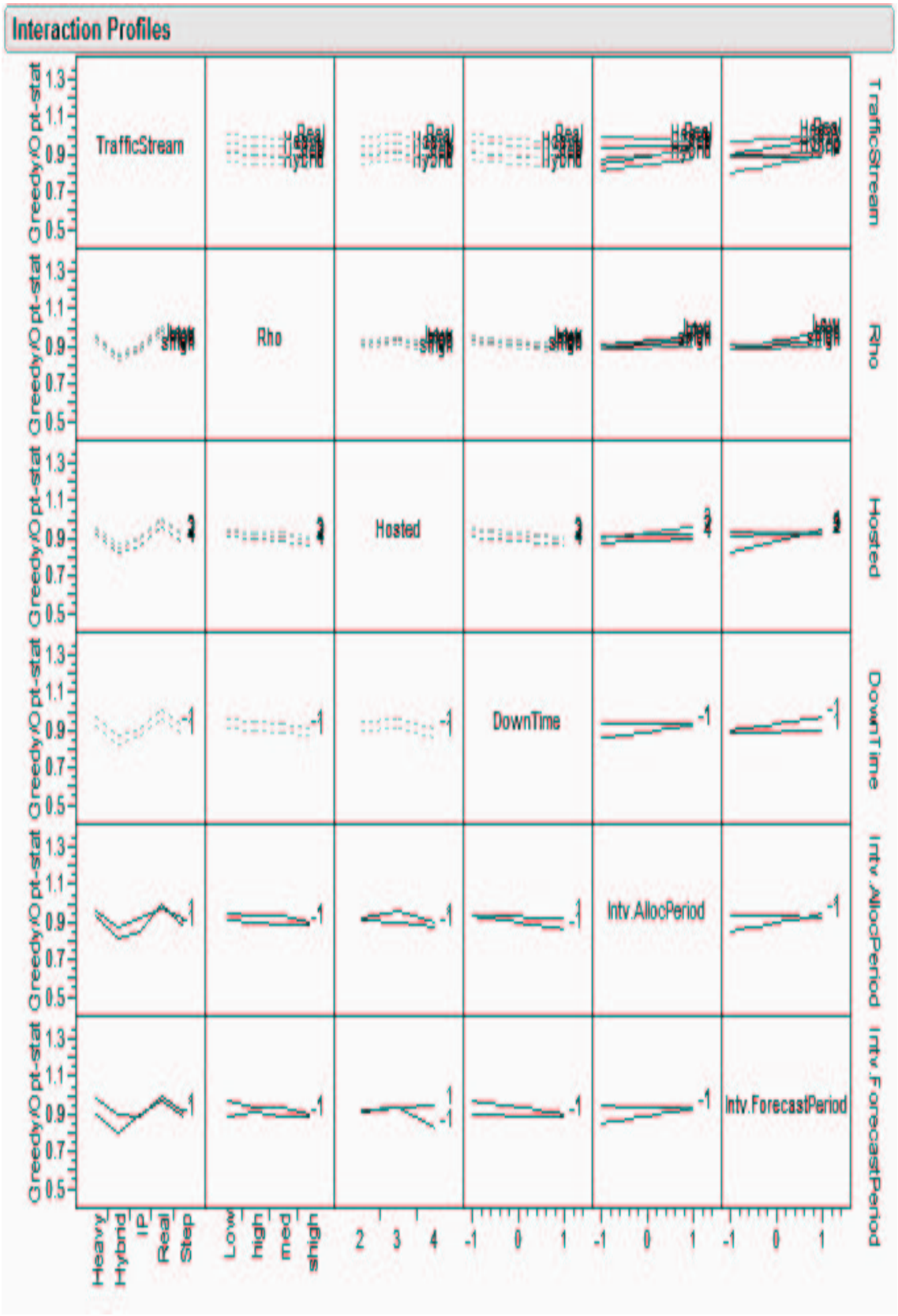


Figure 6.34: Interaction plots for the Greedy process interaction experiment design where superimposed labels in rows are as follows: 1) TrafficStream(Heavy, Hybrid, IP, Real and Step); 2) Rho(low, med, high and shigh); 3) Hosted(2, 3 and 4) whilst the remainder have the labels high (+1) and low (-1).

the Self-Organising algorithm. The tuning experiments with respect to both orchestration algorithms provide empirical evidence that the Self-Organising algorithm is superior to the Greedy algorithm in terms of versatility. In the next section, we design an experiment to quantify the observed difference in versatility.

6.4.2.3 Quantitative Analysis of Self-Organising Versus Greedy

In the preceding section, a greater interaction effect between internal parameters of an algorithm and the external parameters of the environment is taken to imply that the algorithm in question is less versatile. The interaction effects suggest ways to tune the algorithm, by setting internal parameters, to optimise the revenue stream for a given set of external parameters that are not under our control. In contrast, revenue stream of a more versatile algorithm could not be greatly improved by tuning it differently for different cases.

In this section, we devise an experiment for additional comparison of the two algorithms. This is a less comprehensive, but more direct, test of versatility than the previous experiment. In particular, it is designed to permit a more quantitative statistical assessment. The basic idea of the assessment is as follows:

1. We assemble a set of internal tunings, each represented by a choice of values for the internal parameters of the algorithm and a group of external environment settings comprising traffic stream, demand intensity, service hosting and downtime.

2. Denote a tuning represented by specific settings of internal parameter values for a given algorithm that results in the best mean revenue stream performance across the entire group of external settings as the best-single-tuning. For each specific external environment setting in a group, each choice of internal parameter settings of an algorithm yields a revenue stream from which the highest yielding revenue stream is selected.

3. Denote the mean yield of all these highest yielding revenue streams (across the entire external group setting) as the individually-tuned revenue stream. The hypothesis is that the gap between individually-tuned and best-single-tuning is smaller for the Self-Organising algorithm than for the Greedy algorithm.

As in the case of tuning experiments earlier, we design a process model of an internet hosting centre with services co-hosted and managed by the Self-Organising or Greedy orchestration algorithm. In order to compare both algorithms on equal terms, we had to reduce the

Process: Self-Organising algorithm and an internet hosting centre					
Internal			**External**		**Output**
	Level			*Levels*	
Parameter	*Min*	*Max*	*Parameter*	Disc./Cont.	*Response*
Advert Reading Scale	1 sec	40 sec	Downtime	60-1200 sec	Revenue
Server E.Rate Intv	1 sec	40 sec	Hosting	2, 3	
			Demand Intensity (ρ)	Medium High	
			Traffic Stream	Heavy Real IP	

Table 6.31: Reduced Self-Organising algorithm and internet hosting centre process parameters.

Process: Greedy algorithm and an internet hosting centre					
Internal			**External**		**Output**
	Level			*Levels*	
Parameters	*Min*	*Max*	*Parameters*	Disc./Cont.	*Response*
Allocation Interval	1800 sec	7200 sec	Downtime	60-1200 sec	Revenue
Forecast Interval	300 sec	1800 sec	Hosting	2, 3	
			Demand Intensity (ρ)	Medium High	
			Traffic-Streams	Heavy Real IP	

Table 6.32: Greedy algorithm and reduced internet hosting centre process parameters.

number of internal parameters of the Self-Organising algorithm's to match that of the Greedy algorithm's. We used the results of the *Tuning Self-Organising Algorithm* experiment to choose the most significant internal parameters in terms of the main and two-way interaction effects. The parameters absent from the process model were either set to the tuned values as discussed in the tuning experiment of the Self-Organising algorithm if available. The remaining parameters were set to the default untuned values. We also reduced external parameters to manage the simulation workload given that we intended to explore the entire tuning space of the internal parameter settings. The process model of the internet hosting centre employing the Self-Organising algorithm is given in Table 6.31 whilst the model employing the Greedy algorithm is given in Table 6.32.

The tuning methodology is applied in exactly the same manner with the design-of-experiment generated by JMP statistical discovery tool. For this experiment, we use the full factorial experiment design which gives us 324 simulation runs for each algorithm resulting from three

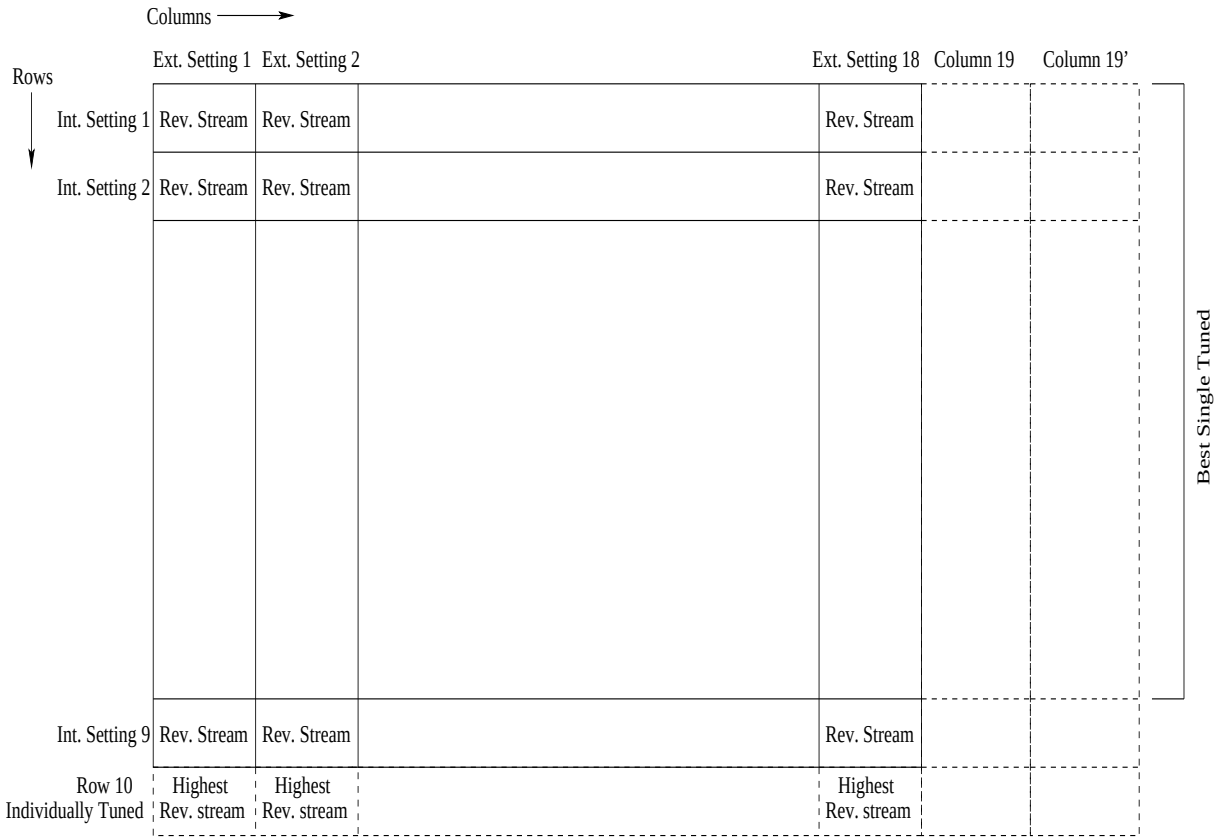


Figure 6.35: Schematic for individually-tuned and best-single-tuned quantitative analysis.

types of traffic streams, two demand intensity levels, two service hosting, three setting for downtime as well as three settings for two internal parameters. Therefore, a total of 648 simulation runs were performed over both algorithms and revenue stream data were collected.

Each of the two internal parameters has three possible settings, yielding nine possible internal settings. There are three possible traffic stream types, two possible demand intensity levels and three possible downtime levels giving eighteen possible external parameter settings per hosting instance. Hence, simulation runs using a combination of internal and external parameter settings generate a 9 by 18 table of revenue streams as depicted in Figure 6.35. We append to this table row 10 and fill it with highest of the nine revenue streams (individually-tuned) arising from respective columns of an external setting and nine internal settings. We also create an additional column 19 and fill it with the sum of eighteen revenue streams arising from the respective nine rows of an internal setting and eighteen external settings per row. We compute the mean revenue stream of column 19 (over rows $1, \ldots, 9$) and normalise each summed revenue stream by the mean revenue. The table element of row 10 and column 19 is filled by normalised revenue stream given by sum of revenue streams from row 10 (columns

Normalised Tuned Revenue ($\pounds \times 10^6$)			
Algorithm	**Individually (I)**	**Best Single (BS)**	**(I-BS)**
Self-Org	21.01	20.71	0.30
Greedy	21.46	20.30	1.16

Table 6.33: Normalised tuned revenue streams for Self-Organising and Greedy algorithm aggregated across traffic streams (Heavy, Real, Inhomogeneous Poisson), demand intensities (medium and high) and co-hosting instance (two and three services).

$1, \ldots, 18$) divided by mean revenue stream of row 10 (over columns $1, \ldots, 18$). The table and analysis described above considered a single hosting instance. Since the experiment considers co-hosting two and three services, we perform the same analysis on revenue stream data related to another instance of hosting. Next, we compute column $19'$ that is representative of normalised revenue streams in column 19 of each hosting instance table by summing corresponding values (column 19, rows $1, \ldots, 10$) from each table. We have normalised revenue streams in each table to obtain meaningful values of best-single-tuned and individually-tuned revenue streams since each hosting instance offers differing revenue availability. In column $19'$, we select the highest normalised revenue stream (best-single-tuned) from rows $1, \ldots, 9$ and compare it with the individually-tuned normalised revenue stream (row 10). We apply the analysis to revenue stream data of the Self-Organising and Greedy algorithms and depict the result in Table 6.33. Below, we extend the experiment to obtain standard deviations of the revenue stream performance in order to obtain the statistical significance of the result.

Here, we extend the experiment in order to compute variances for the individually-tuned and best-single-tuned normalised revenue stream values. The experiment conducted above yields two tables (9 by 18) with revenue streams for each combination of internal and external parameter setting. We select six columns from each table representing specific external settings that are representative of the traffic streams, demand intensity and downtime resulting in a new table of 9 by 12. We chose the top three tunings in terms of highest from the nine possible tunings in each column. The resulting thirty six combination (3×12) of external and internal settings were each run with 10 different initial random seed values for distributions used in the orchestration algorithms whilst using the same seed values for distributions used in generating traffic streams and the baulk manager. This extension of the experiment uses 360 different initial random seed values with a total of 720 simulation runs over the Self-Organising and Greedy algorithms. From the revenue stream performance data collected for each algorithm, the variances of the best-single-tuned and individually-tuned revenue stream were estimated

Normalised Tuned Revenue ($£ \times 10^6$)					
Algorithm	**Individually (I)**	$\frac{\sigma_I}{\sqrt{n}}$	**Best Single (BS)**	$\frac{\sigma_{BS}}{\sqrt{n}}$	**(I-BS)**
Self-Org	21.01	0.004378	20.71	0.008570	0.30
Greedy	21.46	0.000381	20.30	0.003244	1.16

Table 6.34: Normalised tuned revenue streams with sample standard deviation for Self-Organising and Greedy algorithms aggregated across traffic streams (Heavy, Real, Inhomogeneous Poisson), demand intensities (medium and high) and co-hosting instance (two and three services).

and standard deviations ($\frac{\sigma}{\sqrt{n}}$) of the sample means were computed accordingly.

In Table 6.34, we depict best-single-tuned and individually-tuned mean yields along with their standard deviations. Each mean value in Table 6.34 is the sum of many independent random values. Therefore, a normal approximation can be used to obtain confidence intervals. The confidence intervals for the differences between individually-tuned and best-single-tuned computed using 2.33 standard deviations (99% significance) for each algorithm resulted in the following: an upperbound of $(21.01 + (2.33 \cdot 0.004378)) - (20.71 - (2.33 \cdot 0.008570)) = 0.33$ for the Self-Organising algorithm; a lowerbound of $(21.46 - (2.33 \cdot 0.000381)) - (20.30 + (2.33 \cdot 0.003244)) = 1.15$ for the Greedy Algorithm. We conclude with 0.99 confidence that the Self-Organising algorithm is more than three times as versatile as the Greedy algorithm, in terms of the measure we defined previously. We point out that the quantitative test is biased against the Self-Organising algorithm since the internal parameter difference, as depicted in Table 6.31, between low and high values is much higher than for the Greedy algorithm's internal parameters as depicted in Table 6.32.

The main result of this study is a strong confirmation of the hypothesis that the Self-Organising algorithm has a smaller difference between individually-tuned and best-single-tuned revenue stream performance than the Greedy algorithm. The null hypothesis is rejected at p-value < 0.01. The result of the study strongly indicate that the Self-Organising algorithm exhibits a much higher degree of versatility than the classical Greedy algorithm.

6.5 Summary

In this chapter, we developed a simulation framework for an internet hosting centre managed by the Self-Organising, Greedy, Optimal Omniscient and Optimal-Static Omniscient server

orchestration algorithms as well as a framework for the dynamic environment of the Internet representing behaviour of traffic streams and internet service users. We assessed the comparative revenue stream performance of the Self-Organising server orchestration algorithm against three orchestration algorithms across multiple service hosting scenario under different traffic stream types, demand intensities and SLA hosting fee schedules.

We designed and conducted experiments on the simulation framework to explore the adaptiveness and versatility characteristics of the Self-Organising algorithm. The experiment results indicate that the Self-Organising algorithm is able to adapt to widely varying external circumstances and achieve better revenue stream performance over the Greedy or Optimal-Static Omniscient algorithms whilst being close to an upperbound revenue stream performance of the Optimal Omniscient algorithm. Under traffic stream variations along the peak-to-trough demand intensity ratios, the Self-Organising algorithm exhibits a better degree of adaptiveness at high ratio whilst being outperformed by the Greedy and Optimal-Static Omniscient algorithm at low ratio. Primarily, this is due to the hosting centre having enough capacity to handle a low ratio of demand intensity and downtime cost paid by the responsive nature of Self-Organising algorithm in orchestrating servers where demand may not warrant it or last long enough to recover overhead costs associated with orchestration. If a traffic stream exhibits low demand variability then employing an adaptive orchestration algorithm would be an overkill since a static algorithm would perform well in such circumstances given that it would not be subject to any overheads borne by the former algorithm.

When the traffic stream is considered along timescale variations, the Self-Organising algorithm displayed a higher degree of adaptiveness against the Greedy or Optimal-Static Omniscient algorithm for a longer timescale but underperformed against the Optimal-Static Omniscient algorithm for a shorter timescale. The results underscore the point that an adaptive algorithm will perform well in circumstances that exhibit steady state behaviour for a reasonable time duration before changing (long timescale) since such behaviour will allow the algorithm to better trade-off overhead against reward whilst circumstances that exhibit short timescale steady state behaviour is counter productive as far as cost-reward trade-off is concerned.

Extensive tuning experiments indicated that the Self-Organising algorithm is versatile to widely varying external circumstances when its own internal circumstances are changed. The superior versatility exhibited by the Self-Organising algorithm based on the biomimetic

approach that has evolved over many millennia against classical approach of the Greedy algorithm is highlighted by the dearth of tuning opportunities available in the former algorithm whilst tuning opportunities abound in the latter. Finally, quantitative analysis of the revenue stream performance revealed that the Self-Organising algorithm to be more versatile than the Greedy algorithm even when internal circumstances of the former algorithm are changed to a much greater degree.

In the next chapter, we propose a self-tuning feedback extension to the Self-Organising algorithm in order to overcome limitations of orchestration overhead. The basic idea of the self-tuning extension is to allow a server to be more susceptible to orchestration if there exist better revenue earning opportunities elsewhere and its current revenue rate does not compare well across the hosting centre. On the other hand, if revenue opportunities are moderate the server in question will be less susceptible to orchestration thereby avoiding downtime overhead cost.

Chapter 7

Self-Tuning Server Ensemble Orchestration

There is only one corner of the universe you can be certain of improving . . . and that's your own self.

Aldous Leonard Huxley *(1894 - 1963)*

7.1 Introduction

In the preceding chapters, we introduced the biomimetic Self-Organising server ensemble orchestration algorithm and assessed its efficacy against algorithms based on classical and omniscient approaches. We observed the superior adaptive and versatile performance of our algorithm over widely varying external circumstances. However, we noted that under certain external circumstances, the responsive nature of the Self-Organising algorithm becomes a liability due to orchestration overheads.

In this chapter, we propose a self-tuning feedback extension that attempts to temper the responsive nature of the Self-Organising algorithm when certain conditions are met. In the new Self-Tuning algorithm, a server changes its orchestration "mood" based on a revenue wellness criterion that compares its own revenue and the revenue opportunity elsewhere in the hosting centre. Essentially, a server will become exuberant about orchestration if its own revenue does not compare well with the revenue of the hosting centre and there are good revenue opportunities elsewhere. On the other hand, its orchestration "mood" will become unenthusiastic if its own revenue does not compare well with the revenue of the hosting centre

but there are mediocre revenue opportunities elsewhere, thus avoiding unnecessary overhead costs associated with orchestration. In the Self-Organising algorithm described in Chapter 4, this decision process is implicitly characterised by the probability of reading the advertisement board in Table 4.3. The proposed self-tuning feedback extension is an explicit refinement of this decision process.

The remainder of this chapter is organised as follows. In Section 7.2, we propose a Self-Tuning algorithm that builds on the Self-Organising algorithm by identifying key parameters that are central to the tuning feedback control mechanism. In Section 7.3, we first describe the simulation model of the Self-Tuning algorithm and the intensity manager that provides hosting centre wide information on the demand intensity. Finally, we assess the adaptive revenue stream performance of the Self-Tuning algorithm against the Self-Organising algorithm under an external environment comprising service hosting, traffic stream and demand intensity.

7.2 Self-Tuning Server Orchestration Algorithm

In this section, we investigate the possibility of improving the adaptiveness of the Self-Organising algorithm. Earlier we observed, from the analysis of results from the tuning fractional factorial experiment design for an internet hosting centre managed by the Self-Organising algorithm, that there were few opportunities for tuning its performance. However, we did observe some two-way factor interaction effects between internal and external factors such as Scale.AdRead-DownTime, Scale.AdRead-Hosted and Scale.AdRead-Rho as depicted by non-parallel lines in Figure 6.30. Therefore, we focus our attention on exploiting such interaction effects to improve adaptiveness. We introduce an additional feedback control loop to the Self-Organising algorithm and call the new algorithm Self-Tuning.

The Self-Organising orchestration algorithm given in Figure 4.6 has an exponentially distributed mean advertisement reading interval T_r which determines the rate at which a server reads the advertisement board i.e. seeks opportunity to earn revenue at other virtual servers in the hosting centre. The interval value is drawn from an exponential distribution for each successive attempt to read the advertisement board. The distribution mean is a function of Scale.AdRead and DownTime for the algorithm and it is used across the hosting centre by all servers irrespective of whether a server is serving requests from a service queue that may or may not be receiving requests at high demand intensity (ρ). Intuitively, we suggest that,

providing there is a scope for opportunity to earn revenue elsewhere and a server is making revenue below par, then it should be more exuberant in seeking revenue opportunities at other virtual servers. On the other hand, if the scope for revenue opportunities elsewhere is limited then it should temper its exuberance or become unenthusiastic for re-orchestration. We also suggest that the rate at which a server changes its mood for seeking revenue opportunities elsewhere should take into account the downtime cost of orchestration from one service (virtual server) to another.

We propose an improvement to the Self-Organising orchestration algorithm based on the analysis of the fractional factorial experiment design and our intuition. We call the new improved algorithm Self-Tuning since it attempts to adapt to dynamic changes in external parameters demand intensity (ρ) and DownTime to improve its revenue stream performance. The Self-Tuning server orchestration algorithm builds on the Self-Organising algorithm by adding a feedback control loop to change the server behaviour when certain conditions arise. The Self-Tuning server orchestration algorithm is depicted in Figure 7.1 where lines 7–26 and 45–60 are not shown but are as per lines 7–26 and 28–43 respectively of the Self-Organising algorithm depicted in Figure 4.6. The algorithm requires following additional parameters: meanRho, Downtime, and thresholds values for high and low demand intensity (ρ) which would trigger the self-tuning adaptive behaviour. The meanRho parameter is continuously updated and defines the demand intensity across the hosting centre averaged over the current and last interval. Whenever an interval given by Downtime elapses, a server compares its own earning rate with that of the hosting centre in line 28. If its earning rate is less than half of the hosting centre's earning rate than a server embarks on self-tuning in lines 29–41 so that either it becomes exuberant or unenthusiastic to reading the advertisement board thereby increasing or decreasing its chance of looking for earning opportunities elsewhere in the hosting centre. It gauges whether to become exuberant or unenthusiastic by comparing the demand intensity (meanRho) to demand intensity thresholds (RhoHighThresh,RhoLowThresh) such that if the demand intensity is greater than the high threshold then it will become exuberant by reducing the advertisement reading interval in lines 29–30 whilst demand intensity less than the low threshold will result in a server becoming unenthusiastic by increasing the advertisement reading interval in lines 31–32. When the request intensity falls between the low and high thresholds, a server will return to its blasé mood. That is, the advertisement reading interval reverts to its default value by successive increases if it is in an unenthusiastic mood

or successive decreases if it is in an exuberant mood (lines 34–39). We point out that just as foragers change their behaviour in response to a cue or a signal, servers in the Self-Tuning algorithm change their mood over time in response to feedback.

7.3 Efficacy of Self-Tuning Server Ensemble Orchestration

In this section, we assess the effectiveness of the Self-Tuning algorithm by comparing its adaptive revenue stream performance with that of the base Self-Organising algorithm's over an external environment which includes multiple service hosting, traffic stream behaviours and demand intensities. We describe the simulation model of the Self-Tuning algorithm along with the external environments followed by the experiment design to assess adaptiveness of the new algorithm.

7.3.1 Simulation Model

We have described the simulation model of the Self-Organising server orchestration algorithm in Section 6.2. The simulation model of the Self-Tuning server orchestration algorithm builds on the Self-Organising model with an addition of feedback control loop to adapt server behaviour in response to earning potential and demand intensity of the hosting centre. As described earlier, each virtual server in the hosting centre is an instance of a hosted service and any server that belongs to such a virtual server will be installed with service application. Servers can be orchestrated from one virtual server to another with some fixed downtime penalty. A server identified as a scout will randomly orchestrate itself to some other virtual server in the hosting centre whilst a server identified as a forager will orchestrate itself in response to an advertisement read from the advertisement board. Both type of servers will post advertisements on the advertisement board with a virtual server identifier and display expiry time. The advertisement board manager ensures that the advertisements do not get displayed beyond their display expiry time.

The Self-Tuning feedback control loop modifies the behaviour of scout or forager servers in the hosting centre. Each server checks at regular interval to see if its earning rate is less than or equal to 50% of the hosting centre's. If so, it becomes exuberant about an earning opportunity at some other virtual server by successively increasing the frequency of reading the advertisement board provided the demand intensity of the hosting centre is above a

```
SELF_TUNING_ORCHESTRATE(id, V_j, Q_j, AdBoard, E_centre, S, meanRho, Downtime)
 1  Initialise: Adboard reading interval T_r, MaxT_r, MinT_r, T_default
 2  Initialise: RhoHighThresh, RhoLowThresh, MoodSwing, k = MoodSwing
 3  forever
 4    while Q_j ≠ Empty do
 5      serve request
 6      collect earning
 ⋮  // self-organising pseudo code
27      if Downtime Elapsed then
28        if e < 0.5E_centre then // earning rate-- e:server's, ECentre:hosting
29          if meanRho > RhoHighThresh then // change server mood to exuberant
30            T_r = T_r − (MoodSwing · T_default) // minimum Tr = MinTr
31          elseif meanRho < RhoLowThresh then // change mood to unenthusiastic
32            T_r = T_r + (MoodSwing · T_default) // maximum Tr = MaxTr
33          else // meanRho is in default range
34            if T_r > T_default // server in unenthusiastic mood
35              T_r = T_r − k(T_r − T_default) // approach blase mood
36              k = k + MoodSwing //
37            elseif T_r < T_default // server in exuberant mood
38              T_r = T_r + k(T_default − T_r) // approach blase mood
39              k = k + MoodSwing //
40            endif
41          endif
42        endif
43      endif
44    endwhile
     //request queue empty -- idling
 ⋮  // self-organising pseudo code
61    sleep // idling-- goto sleep for a bit
62      if Downtime Elapsed then
63        if e < 0.5E_centre then // earning rate-- e:server's, Ecentre:hosting
64          if meanRho > RhoHighThresh then // change server mood to exuberant
65            T_r = T_r − (MoodSwing · T_default) // minimum Tr = MinTr
66          elseif meanRho < RhoLowThresh then // change mood to unenthusiastic
67            T_r = T_r + (MoodSwing · T_default) // maximum Tr = MaxTr
68          else // meanRho is in default range
69            if T_r > T_default // server in unenthusiastic mood
70              T_r = T_r − k(T_r − T_default) // approach blase mood
71              k = k + MoodSwing //
72            elseif T_r < T_default // server in exuberant mood
73              T_r = T_r + k(T_default − T_r) // approach blase mood
74              k = k + MoodSwing //
75            endif
76          endif
77        endif
78      endif
79  endforever
```

Figure 7.1: Self-Tuning server orchestration algorithm.

defined high threshold. Alternatively, it becomes unenthusiastic about an earning opportunity at some other virtual server by remaining associated with the present virtual server through successively decreasing the frequency of reading the advertisement board provided the demand intensity of the hosting centre is below a defined low threshold. Given the exuberant or unenthusiastic mood of a server, if the demand intensity of the hosting centre falls between the high and low thresholds, the server returns to its default mood by adjusting the frequency of reading the advertisement board up or down to its default value. The adjustment of reading frequency is triggered by defined intensity threshold levels and the mean demand intensity of the hosting centre. As depicted in Figure 7.2, an intensity manager computes mean intensity of the hosting centre over current and previous intervals. The new simulation model parameter values are chosen as follows: The demand intensity computation interval is chosen to be 60 seconds in order to provide update at a reasonable rate. The values for high and low demand intensity thresholds as well as MoodSwing were chosen arbitrarily after some preliminary investigation revealed their dependence on the number of services co-hosted. During this investigation, it was discovered that the difference between high and low thresholds increases as the number of services co-hosted increases whilst the rate at which mood of a server swings decreases. The details of simulation models for the hosting centre, traffic streams and the baulk manager have been described earlier in Section 6.2 along with default parameter values for the respective simulation models.

7.3.2 Comparative Assessment of Self-Tuning Server Orchestration Algorithm

In this section, we design an experiment to assess the adaptiveness of the Self-Tuning algorithm to the Self-Organising algorithm. The experiment design considers the case of co-hosting two, three and four services on an internet hosting centre managed, in turns, by the Self-Tuning algorithm and the Self-Organising algorithms. The demand for service is driven by Step, Heavy, Real, Inhomogeneous Poisson and Hybrid traffic streams at low, medium, high and super high levels of demand intensity. The simulation parameter values for the hosting centre are set as given in Table 6.1 whilst the traffic streams and baulk manager parameters are set as depicted in Tables 6.7 and 6.9. The default simulation parameter values for the Self-Organising algorithm are illustrated in Tables 6.2 and 6.3. The Self-Tuning algorithm uses the same parameters as the Self-Organising algorithm in addition to new parameters that control

INTENSITY_MANAGER(*meanServiceTimes[], meanInterArrivalTimes[], VS[], RhoIntv*)

1 Initialise – mean Rho compute interval T_ρ = *RhoIntv*,
2 Initialise – M = virtual servers in the hosting centre
3 Initialise – Hosting centre: meanRho = meanRhoPrev = meanRhoCur = 0.0
4 Initialise – First interval IntvZero = TRUE
5 // meanInterArrivalTimes[i] = mean inter arrival time of requests arrived
// during T_ρ for i^{th} virtual server
6 // meanServiceTimes[i] = mean service time for requests at i^{th} virtual server
7 // VS[i] = Number of server currently serving request for i^{th} virtual servers
8 // mean intensity over two intervals is updated by writing to global meanRho
9 **forever**
10 Sleep(T_ρ)
11 meanRhoCur = $\left(\frac{1.0}{\sum_{i=1}^{i=M} VS[i]} \cdot \sum_{i=1}^{i=M} \left\{\frac{\text{meanServiceTime[i]}}{\text{meanInterArrivalTimes[i]}}\right\}\right)$
12 $\forall_i$ Reset(meanInterArrivalTimes[i]) // `collect arrival data for next T_rho`
13 **if** *IntvZero* **then** // `first interval, report current intensity`
14 meanRho = meanRhoPrev = meanRhoCur
15 *IntvZero* = *FALSE*
16 **else** // `Other intervals, intensity averaged over cur+prev intervals`
17 meanRho = 0.5(meanRhoCur + meanRhoPrev)
18 meanRhoPrev = meanRhoCur
19 **endforever**

Figure 7.2: Demand intensity manager.

the tuning behaviour. In Table 7.1, we give parameters for both algorithms whose values are either modified from the default values or are new parameters. The mean service and sleep times are set according to inverse of mean service rates ($\bar{\mu}$) defined in Table 6.8. The service rate is a function of the mean arrival rate ($\bar{\lambda}$) of the hosting centre for a given traffic stream type and number of services co-hosted as well as specific demand intensity level.

The permutation of the experiment design in which there are three service hosting instances, five traffic stream types and four demand intensity levels gives 60 simulation runs. In addition, our experiment considers running each simulation run with 10 different initial random seed values for distributions used in traffic streams, orchestration algorithms and baulk manager. Therefore, the experiment comprises 600 simulation runs per algorithm with each run using different random initial seed values. Since we are comparing the Self-Tuning algorithm against the Self-Organising algorithm, the total simulation workload is 1200 runs. The simulation runs were performed for a twenty-four hour time horizon and revenue stream performance data were collected.

Server Orchestration Algorithms	
Parameter	**Value**
Self-Tuning	
Exp. dist. mean service time	As per hosting/stream/intensity instance
Exp. dist. mean sleep time	As per hosting/stream/intensity instance
Advertisement posting threshold	0.04
Scouts	1 across the hosting centre
Pay-Per-Request-Served:	Pence:
2 services co-hosted	0.1, 0.155 (Service 0,1)
3 services co-hosted	0.1, 0.14166, 0.1833 (Service 0,1,2)
4 services co-hosted	0.1, 0.133, 0.166, 0.2 (Service 0,1,2,3)
Demand intensity (ρ) comp. interval	60.0 seconds
High demand intensity (ρ) threshold:	
2 services co-hosted	0.80
3 services co-hosted	1.35
4 services co-hosted	1.55
Low demand intensity (ρ) threshold:	
2 services co-hosted	0.20
3 services co-hosted	0.50
4 services co-hosted	0.25
MoodSwing:	
2 services co-hosted (incr/decr)	0.080 (min: 0.520, max: 1.560)
3 services co-hosted (incr/decr)	0.076 (min: 0.544, max: 1.532)
4 services co-hosted (incr/decr)	0.068 (min: 0.524, max: 1.544)
Self-Organising	
Exp. dist. mean service time	As per hosting/stream/intensity instance
Exp. dist. mean sleep time	As per hosting/stream/intensity instance
Advertisement posting threshold	0.04
Scouts	1 across the hosting centre
Pay-Per-Request-Served:	Pence:
2 services co-hosted	0.1, 0.155 (Service 0,1)
3 services co-hosted	0.1, 0.14166, 0.1833 (Service 0,1,2)
4 services co-hosted	0.1, 0.133, 0.166, 0.2 (Service 0,1,2,3)

Table 7.1: Modified and new simulation model parameters for Self-Tuning and Self-Organising server orchestration algorithms.

Host	**Revenue**	**Wilcoxon Matched-Pairs Signed Ranks Test**			
	$\frac{\sum_{i=1}^{n} \text{Self-Tune}_i}{\sum_{i=1}^{n} \text{Self-Org}_i}$	**(Self-Tune)-(Self-Org)**			
		W^+	W^-	N	$\leq p$-value
2	≈ 1.00	9995	6295	180	8.242e-03
3	1.01	10371.50	7206.50	187	3.280e-02
4	1.01	9831	6640	181	2.384e-02
Overall	1.01	89474.50	60951.50	540	1.203e-04

Table 7.2: Normalised revenue stream performance and Wilcoxon matched-pairs signed ranks test for Self-Tuning against Self-Organising algorithms aggregated across traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), n and number of matched pairs is 200 whilst for overall, n and number of matched pairs is 600.

In Table 7.2, we depict the revenue stream of the Self-Tuning algorithm normalised by revenue stream of the Self-Organising algorithm aggregated across all five traffic stream types and four demand intensity levels. We note that the Self-Tuning algorithm either performs equitably or outperforms the self-organising when revenue streams are partitioned in terms of service co-hosting instance or grouped across all hosting instance. We also depict, in the same table, the results of the Wilcoxon matched-pairs signed ranks test where we observe that adaptive performance of the Self-Tuning algorithm over the Self-Organising algorithm is at least better than 96% significant across the performance partitioning range whilst being better than 99% significant in the overall category comprising three service co-hosting instances. The adaptive performance of the Self-Tuning algorithm over the Self-Organising algorithm confirms the advantage of dynamic behaviour in response to the global information conferred by the new feedback loop.

To ascertain the performance of the Self-Tuning algorithm when the hosting centre is operating under different demand intensity levels, we partitioned performance data in terms of low, medium, high and super high demand intensity levels irrespective of traffic stream types. We further partitioned the data into individual and overall hosting instances for each demand intensity level. This enabled us compare the effectiveness of the new tuning feedback loop across the demand intensity spectrum. We normalised the revenue streams of the Self-Tuning algorithm by revenue stream of the Self-Organising algorithm and applied the Wilcoxon matched pairs signed ranks test according to the performance data partitioned as per scheme described above. We depict the normalised revenue streams and Wilcoxon sta-

ρ	Host	**Revenue** $\frac{\sum_{i=1}^{n} \text{Self-Tune}_i}{\sum_{i=1}^{n} \text{Self-Org}_i}$	**Wilcoxon Signed Ranks Test** **(Self-Tune)-(Self-Org)**			
			W^+	W^-	N	$\leq p$-value
Low	2	≈ 1.00	305	160	30	1.386e-01
Low	3	1.01	520	183	37	1.126e-02
Low	4	1.02	409	152	33	2.219e-02
Low	Overall	1.01	3611	1439	100	1.898e-04
Medium	2	1.01	912	363	50	8.169e-03
Medium	3	≈ 1.00	705	570	50	5.178e-01
Medium	4	1.01	717	558	50	4.457e-01
Medium	Overall	1.01	6701	4624	150	5.147e-02
High	2	≈ 1.00	736	539	50	3.441e-01
High	3	1.01	756	519	50	2.547e-01
High	4	≈ 1.00	586	590	48	9.877e-01
High	Overall	≈ 1.00	6098	4928	148	2.632e-01
Super High	2	≈ 1.00	682	593	50	6.710e-01
Super High	3	≈ 1.00	605	670	50	7.574e-01
Super High	4	1.01	758	518	50	2.467e-01
Super High	Overall	≈ 1.00	6021	5304	150	5.018e-01

Table 7.3: Normalised revenue stream performance and Wilcoxon matched-pairs signed ranks test for Self-Tuning algorithm against Self-Organising algorithm aggregated across traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) where for each co-hosting instance (two, three and four services), n and number of matched pairs is 50 whilst for overall, n and number of matched pairs is 150.

tistical test results in Table 7.3. Here, we observe for the overall hosting instance that the Self-Tuning algorithm's revenue stream performance is better by 1% and statistically significant for low ($> 99\%$) and medium ($> 94\%$) demand intensity. However, for the same hosting instance, revenue stream performance of the Self-Tuning algorithm is no better than performance of the Self-Organising algorithm. Although, it wins more often than the Self-organising algorithm in the Wilcoxon test but results are statistically inconclusive. As for the individual hosting instance, the Self-Tuning algorithm performs better in three instances with statistical significance better than 97% whilst performance is insignificant on three other instances across low and medium demand intensity levels. In cases of high and super high demand intensity levels, the Self-Tuning algorithm's performance is statistically insignificant across two, three and four hosting instance. We believe that at low and medium demand intensity levels when

earning opportunities are scarce, the self-tuning feedback loop is able to take advantage of any newly arising earning opportunities by changing the server to exhibit an exuberant behaviour. At the other end of the demand spectrum, there is greater likelihood of earning opportunities everywhere. Therefore, changing the server to exhibit an unenthusiastic behaviour appears ineffectual in terms of improving revenue stream performance but it does not appear to harm its chances of being just as competitive as the Self-Organising algorithm.

7.4 Summary

This chapter introduced the new Self-Tuning server orchestration algorithm by building on the Self-Organising server orchestration algorithm. The new algorithm presented a tuning feedback loop as an extension that attempts to subdue the responsive nature of the Self-Organising technique when it is considered advantageous to do so under certain external circumstances.

We designed and conducted a limited set of experiments to assess the efficacy of the new algorithm against the base Self-Organising algorithm under a complex external environment comprising multiple services hosting, traffic stream behaviours and demand intensities. The experiment results illustrated a small improvement in adaptive revenue stream performance of the new algorithm in comparison to the Self-Organising technique. An improvement in adaptive performance not only highlights the value of the self-tuning feedback loop but it also suggests that the Self-Organising algorithm based on the collective intelligence of a honeybee colony is already well adapted to dealing with dynamic external conditions. The experiment results also highlighted the effectiveness of the self-tuning feedback loop across the demand intensity range. It is effective in improving adaptiveness at low demand intensity whilst ineffective in improving performance at high demand intensity but it does not degrade the performance. On the other hand, the self-tuning feedback loop relies on global information, thus making the algorithm less decentralised.

In the next chapter, we address the problem of power consumption in an internet hosting centres and introduce an energy-aware feedback extension to the Self-Tuning algorithm that enables servers in the hosting centre to conserve power when certain external conditions arise. The new feedback loop enables servers to respond to waxing and waning behaviour of demand for service. The fundamental idea behind the feedback loop is to exploit circumstances when

demand for service wanes by allowing servers to hibernate in order to reduce overall power consumption of the hosting centre.

Chapter 8

Energy-Aware Server Ensemble Orchestration

And God said, 'Let there be light' and there was light, but the Electricity Board said He would have to wait until Thursday to be connected.

Spike Milligan *(1918 - 2002) - Digby, 3 Musketeers*

8.1 Introduction

The recently emerging market trend towards building centralised internet hosting infrastructure facilities poses a significant challenge in provisioning server ensemble, managing heat and power, and dealing with facility space constraints [Inf05]. According to Cowden [Cow04], the operating cost of an internet hosting centre comprises facility lease, property tax, corporate tax, payroll, benefits and electricity usage. A comparative operating cost analysis of an internet hosting facility spread over 87,200 ft^2, employing 50 people and using 8,417,700 KWh of electricity per year revealed an operating cost of $£17,421,422$ to $£29,584,063$ (using currency conversion of $£1 = \$1.80$) over five years depending on the geographical location in the USA. The electricity component of the operating cost also depended on the geographical location and ranged from $£800,616$ to $£3,042,063$. The energy consumption cost across the eight locations ranged from 4.6% to 14.8% of the operating cost. An internet hosting infrastructure facilities has many energy consuming components including servers, storage, switches, interruptible power supply, auxiliary equipment as well as heating, cooling and lighting systems. As the trend is likely to continue given the economic benefit of the outsourcing model, the in-

ternet hosting infrastructure will become a substantial consumer of energy [CATV01, MJ01]. The energy demand projection for the internet hosting infrastructure facilities in the U.S. for 2005 places requirements of approximately 40 TWh (£2B at £50/MWh) per year to operate twenty-four hours a day [CD01] coupled with operational and capital costs, proportional to power demand, attributed to auxiliary equipment in respect of cooling and backup power. This energy requirement as well as environmental concerns of limiting carbon dioxide emissions as per Kyoto Protocol makes a compelling case for reducing energy consumption.

In this chapter, we address the issue of reducing server ensemble energy consumption in the hosting centre by introducing an energy-aware feedback extension to the Self-Tuning algorithm. The biomimetic inspiration for the new algorithm comes from the fact that natural honeybee colonies possess feedback control that matches the allocation of foragers to the supply of nectar. It has been observed that on sunny days when there is an abundance of nectar available due to many flowers in bloom, a large number of foragers leave the colony to forage. Conversely, on days when the weather is gloomy and there is low nectar availability owing to lack of flowers in bloom and increased expense of travel due to body temperature regulation, a small number of foragers leave the colony to forage [Tov04]. The feedback control would also appear to conserve the nectar reserve of the colony as any superfluous allocation of foragers would increase the consumption of reserve nectar since active foragers would require more nectar intake to sustain their foraging activity. However, the allocation would not necessarily translate into efficient collection of the available nectar supply from the surrounding countryside. It is known that a significant portion of the nectar reserve is consumed to meet the energy demands of foragers that explore at least 100 km^2 of the surrounding countryside [See95]. We call the new algorithm Energy-Aware since it attempts to conserve energy, on an ongoing basis, by sensing when demand for service wanes. It exploits such opportunities to place servers in hibernation mode and given that hibernation mode consumes significantly less power than full operational mode, the new algorithm reduces overall energy requirements of the hosting centre. Specifically, each server in the hosting centre continually monitors its own idleness and if a threshold is breached then it places itself in hibernation mode for some fixed time duration. The natural feedback consequence of such behaviour across the hosting centre leads to a reduction in the idleness of servers that are in operational mode since they now have to bear the future service demand load that would have been borne by the hibernating servers. This has the desired effect of servers reaching an

equilibrium point for the number of servers in operational mode to the number of servers in hibernation mode for a given service demand intensity.

The organisation for the rest of the chapter is as follows. First, we review relevant work on energy saving and compare with our work. In Section 8.3, we introduce the Energy-Aware server orchestration algorithm that builds on the Self-Tuning technique. We assess the efficacy of the new algorithm in Section 8.4 where we, first, describe the simulation model for the Energy-Aware algorithm and give its parameters. Next, we assess effectiveness of the new algorithm against the Self-Tuning technique in terms of power-on-time, adaptive revenue stream performance and hosting-centre-wide mean service waiting time. The experiment exposes both algorithms to circumstances that entail co-hosting two, three and four services with the demand for service driven by five traffic stream behaviours at four demand intensity levels.

8.2 Related Work

The issue of energy consumption by internet hosting infrastructure facilities has attracted considerable interest from the media, industry as well as researchers given the current focus on global warming and the reduction of greenhouse gas emission as defined by the Kyoto Protocol accord [Che02, Cor05, Kin02, MJ01, RR02].

Early efforts in energy saving were directed at mobile systems with emphasis on techniques aimed at the circuit level as well as the architecture level. All circuit level techniques are based on dynamic frequency and voltage scaling that entails adjusting the voltage and frequency to reduce energy consumption whilst architectural level techniques entail optimising the processing load such that the system subcomponents such as memory modules and disks can be in idle state and become subject to operating in low power mode (see Lefurgy *et al.* [LRR+03] and references therein).

Recent efforts in energy saving for server ensemble have concentrated on simply switching the entire server on and off. Various research efforts have concentrated on decision techniques, that is, how to arrive at the decision to switch servers on and off. The decision techniques usually entail considering demand load, concentrating request distribution and server utilisation [BR03, CATV01, CD01]. Our work similarly relies on decision being made on the basis of demand load, however we do not have a centralised decision making process. In our approach

each server in the hosting centre is autonomous and makes its own decision whether to switch to energy saving mode. Although we have cast the problem in terms of servers being placed in hibernation mode, they could just as easily be placed in other modes as specified in Advanced Configuration and Power Interface Specification (ACPI) standard or switched off.

8.3 Energy-Aware Server Orchestration Algorithm

We propose an energy conserving server orchestration algorithm that builds on the Self-Tuning algorithm given in Figure 7.1. The new algorithm introduces a feedback loop to enable individual servers to suspend the fully operational mode and enter the hibernation mode thereby reducing energy consumption. We call the new algorithm Energy-Aware as it attempts to reduce overall energy consumption of the hosting centre in response to lack of demand for service. The pseudo code of the Energy-Aware orchestration algorithm is depicted in Figure 8.1 where lines 7–26 are not shown but are as per Self-Organising pseudo code depicted in Figure 4.6 whilst lines 27–43 are as per Self-Tuning pseudo code depicted in Figure 7.1. Similarly, lines 54–68 are equivalent to lines 28-43 of the Self-Organising pseudo code of Figure 4.6 and lines 70–86 are equivalent to lines 62–78 of the Self-Tuning pseudo code of Figure 7.1. The algorithm requires additional parameters IdleIntv, IdleThresh, HibProb and HibIntv to trigger the energy-aware behaviour. As depicted in Figure 8.1, whenever an exponentially distributed interval, T_{idle}, whose mean is given by IdleIntv elapses (lines 44 and 87), a server computes the percentage of time it was idle over the interval T_{idle} in lines 45 and 88. In lines 46 and 89, it checks whether it was idle for greater than or equal to IdleThresh and, if so, it randomly decides with threshold HibProb to suspend operational mode and enter hibernation mode for an exponentially distributed interval, T_{hib}, drawn in lines 47 and 90, thereby suspending servicing of any requests. When this interval expires, a server in question exits hibernation mode and restores itself to operational mode in lines 49 and 92 and resumes serving requests. When a server makes a transition from hibernation to operational mode, there is a time cost involved due to system devices becoming fully operational upon power on and restoration of system state. Finally, next exponentially distributed interval, T_{idle} is drawn in lines 51 and 94 to begin the cycle again.

The algorithm exploits the feedback relationship between number of servers at a given virtual server and arrival behaviour of requests seeking service. If a given demand load is

ENERGY_AWARE_ORCHESTRATE($id, V_j, Q_j, \ldots, IdleIntv, IdleThresh, HibProb, HibIntv$)

```
1   Initialise: T_idle - Exp. distributed with mean IdleIntv
                T_hib - Exp. Distributed with mean HibIntv
                Hibernation Threshold HibProb
                Percent idle over IdleIntv - IdlePercent
2   Initialise: RhoHighThresh, RhoLowThresh, ......
3   forever
4     while Q_j ≠ Empty do
5       serve request
6       collect earning
⋮   // Self-organising + self-tuning pseudo code
44      if T_idle Elapsed then
45        Compute IdlePercent // %idle over T_idle
46        if Hibernate(HibProb) && IdlePercent ≥ IdleThresh then
47          T_Hib ← ExpDistHibernate.Draw() // exp. dist. hibernation interval
48          Hibernate_Mode(T_hib) // hibernate for time T_hib
49          Operational_Mode() // Delay before becoming fully operational
50        endif
51        T_idle ← ExpDistIdle.Draw() // exp. dist. next idle interval
52      endif
53    endwhile
     //request queue empty -- idling
⋮   // self-organising pseudo code
69    sleep // idling-- goto sleep for a bit
⋮   // self-tuning pseudo code
87      if T_idle Elapsed then
88        Compute IdlePercent // %idle over T_idle
89        if Hibernate(HibProb) && IdlePercent ≥ IdleThresh then
90          T_Hib ← ExpDistHibernate.Draw() // exp. dist. hibernation interval
91          Hibernate_Mode(T_hib) // hibernate for time T_hib
92          Operational_Mode() // Delay before becoming fully operational
93        endif
94        T_idle ← ExpDistIdle.Draw() // exp. dist. next idle interval
95      endif
96  endforever
```

Figure 8.1: Energy-Aware server orchestration algorithm.

serviced by a small number of servers at a virtual server, there will be almost no server idling before finding a request in the service queue to serve, but not all demand will be met. Conversely, if a virtual server has excessive servers for a given demand load then demand will be met, but each server is likely to idle considerably before finding a request in the queue to serve. Under the latter circumstances, the energy-aware feedback loop will place some servers in hibernation mode thus saving energy and enabling the remaining servers to serve requests efficiently from a revenue earning rate perspective. In essence, the feedback loop allows orchestrated servers at virtual servers to tend towards a balance-point of servers that simultaneously serves the current request arrival load well and, concurrently, conserves energy.

8.4 Efficacy of Energy-Aware Server Ensemble Orchestration

This section, firstly, presents the simulation model of the Energy-Aware algorithm and its parameters as well as models for the hosting centre and internet environment. Secondly, the simulation models are used in an experimental study to ascertain the performance effectiveness of the new algorithm against the base Self-Tuning scheme. We consider performance metrics of power-on-time, revenue streams and mean service waiting time experienced by users across the hosting centre. The environment of our experiment study consists of co-hosting up to four services, five types of traffic stream behaviour and four levels of hosting centre demand intensity.

8.4.1 Simulation Model

We have described the simulation model of the Self-Organising orchestration algorithm in Sections 6.2 and the Self-Tuning orchestration algorithm in Section 7.3.1. The simulation model of the Energy-Aware server orchestration algorithm builds on the Self-Organising and Self-Tuning models by introducing an additional feedback loop to enable servers to save energy by entering hibernation mode when there is lack of demand for service. As given previously, each service is hosted on a group of servers called virtual server where each server is installed with a common application of the hosted service. A virtual server can be augmented or reduced by orchestrating servers between virtual servers in the hosting centre at a cost of fixed down-time. A scout server randomly chooses another virtual server to augment and orchestrates itself accordingly whilst a forager orchestrates itself to another virtual servers according to an

advertisement read. Each server, whether scout or forager, posts advertisements on the advertisement board with a virtual server identifier and display expiry time. The advertisement board manager ensures that advertisements are expunged upon expiry of their display time. Each server is prone to modify its mood from exuberant, default or unenthusiastic as a result of the self-tuning feedback based on the current demand intensity of the hosting centre, high and low intensity threshold levels and its present earning rate as well as earning rate of the hosting centre.

The Energy-Aware feedback loop enables a forager server to adapt its behaviour to conserve energy. Whenever an exponentially distributed time interval elapses, each forager server in the hosting centre computes the percentage of time it was idle over the elapsed interval. If it was idle above or equal to a defined threshold then it randomly decides with hibernation threshold HibProb to hibernate for an exponentially distributed hibernation interval. During the hibernation interval, the server in question switches off power to subsystems such as disks and processor to conserve energy. When the hibernation interval expires, the server in question switches on power to return to the operational mode and resume serving requests. There is a fixed delay between exiting the hibernation mode and becoming fully operational due to time involved in system devices reaching ready status upon power switch on and restoration of system state. Another interval over which to measure idle time is drawn from an exponential distribution and so begins the next cycle of energy-aware feedback loop. We point out that servers that are designated as scouts are not prone to energy-aware feedback since we think that some servers in the hosting centre must always be in operational mode to report changes in demand for service across the hosting centre. The parameters values that control the feedback loop were chosen using scale reasoning. The mean exponentially distributed idle computation interval was selected to be 600 seconds in order to allow the servers to establish sufficient history over which to measures its idle time. We arbitrarily chose the hibernation threshold value of 0.4 by reasoning that if a server is busy for less than or equal to 60% of the time then there are too many servers serving a current demand load. The hibernation interval value of 240 seconds is derived from the fact that if a server is idle for at least 40% of the mean exponentially distributed 600 seconds interval then it should hibernate for a $0.4 \times 600 = 240$ seconds mean exponentially distributed interval. In order to avoid the knee-jerk reaction to an idle condition, we chose the hibernation threshold value of 0.33 such that one in three attempts to hibernate will be successful. Finally, delay between the exiting hibernation mode

and resuming the operational mode time for advanced power management techniques such as Intel's Instantly Available PC (IAPC) [Int02] is five seconds but we have chosen a value 90.0 seconds to allow for variability in the power management technology employed. We have already described the details of simulation models for the hosting centre, traffic streams and baulk manager earlier in Section 6.2 along with default parameter values for respective simulation models.

8.4.2 Comparative Assessment of Energy-Aware Server Orchestration Algorithm

In this section, we design experiments to assess performance of the Energy-Aware algorithm in terms of the overall server ensemble power on time in the hosting centre as well as effects of the energy-aware feedback loop on revenue stream and mean service waiting time. We assess the performance of the new algorithm by comparing it with performance of the Self-Tuning algorithm. The experiment design considers the complex external environment comprising two, three and four services co-hosted with the demand for service driven by five types of traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) at low, medium, high and super high levels of demand intensity. The hosting centre's simulation model parameters are set as depicted in Table 6.1 whilst simulation model parameters for five traffic stream types and the baulk manager are set as given in Tables 6.7 and 6.9 respectively. Given that the Energy-Aware algorithm builds on the Self-Tuning algorithm which in turn builds on the Self-Organising algorithm, the default simulation model parameters values for the Self-Organising algorithm are illustrated in Tables 6.2 and 6.3. The Energy-Aware algorithm uses the same parameters as the Self-Tuning algorithm with the addition of new parameters for the feedback loop that controls the energy saving behaviour. In Table 8.1, we give those simulation model parameters values for the Energy-Aware and Self-Tuning algorithms whose values have either been modified from default or are new values. We reiterate here that the mean service and sleep times are set according to the reciprocal of mean service rates ($\bar{\mu}$) defined in Table 6.8. The service rate is a function of the mean arrival rate ($\bar{\lambda}$) of the hosting centre for a given traffic stream type and the number of services co-hosted as well as specific demand intensity level.

Server Orchestration Algorithms	
Parameter	**Value**
Energy-Aware	
Exp. dist. mean service time	As per hosting-stream-intensity instance
Exp. Dist. mean sleep time	As per hosting-stream-intensity instance
Advertisement posting threshold	0.04
Scouts	1 across the hosting centre
Pay-Per-Request-Served:	Pence:
2 services co-hosted	0.1, 0.155 (Service 0,1)
3 services co-hosted	0.1, 0.14166, 0.1833 (Service 0,1,2)
4 services co-hosted	0.1, 0.133, 0.166, 0.2 (Service 0,1,2,3)
Exp. dist, mean idle comp. interval	600 seconds
Idle threshold	0.40
Hibernation threshold	0.33
Exp. dist mean hibernation interval	240 seconds
Delay (Mode: hibernation $\rightarrow$ operational)	90 seconds
Self-Tuning	
Exp. dist. mean service time	As per hosting-stream-intensity instance
Exp. dist. mean sleep time	As per hosting-stream-intensity instance
Advertisement posting threshold	0.04
Scouts	1 across the hosting centre
Pay-Per-Request-Served:	Pence:
2 services co-hosted	0.1, 0.155 (Service 0,1)
3 services co-hosted	0.1, 0.14166, 0.1833 (Service 0,1,2)
4 services co-hosted	0.1, 0.133, 0.166, 0.2 (Service 0,1,2,3)
Demand intensity (ρ) comp. interval	60 seconds
High demand intensity (ρ) threshold:	
2 services co-hosted	0.80
3 services co-hosted	1.35
4 services co-hosted	1.55
Low demand intensity (ρ) threshold:	
2 services co-hosted	0.20
3 services co-hosted	0.50
4 services co-hosted	0.25
MoodSwing:	
2 services co-hosted (incr/decr)	0.080 (min: 0.520, max: 1.560)
3 services co-hosted (incr/decr)	0.076 (min: 0.544, max: 1.532)
4 services co-hosted (incr/decr)	0.068 (min: 0.524, max: 1.544)

Table 8.1: Modified and new simulation model parameters for Energy-Aware and Self-Tuning server orchestration algorithms.

Host	**Power-On-Time**	**Wilcoxon Matched-Pairs Signed Ranks Test**			
	$\frac{\sum_{i=1}^{n} \text{Energy-Aware}_i}{\sum_{i=1}^{n} \text{Self-Tune}_i}$	**(Energy-Aware)-(Self-Tune)**			
		W^+	W^-	N	$\leq p$-value
2	0.95	0	20100	200	1.494e-34
3	0.95	0	20100	200	1.494e-34
4	0.94	0	20100	200	1.494e-34
Overall	≈ 0.95	0	180300	600	6.514e-100

Table 8.2: Normalised power-on-time performance and Wilcoxon matched-pairs signed ranks test for Energy-Aware algorithm against Self-Tuning algorithm aggregated across traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), n and number of matched pairs is 200 and for overall, n and number of matched pairs is 600.

As stated earlier, our experiment design considers three hosting instances, five traffic stream types and four levels of demand intensity resulting in 60 simulation runs per orchestration algorithm used to manage the hosting centre. Furthermore, each of the 60 simulation runs is performed with 10 different initial random seed values for distributions used in traffic streams, the baulk manager and respective orchestration algorithms. Therefore, the total simulation space considered by our experiment is 600 simulation runs per algorithm with 1200 simulation runs overall since we are comparing the Energy-Aware algorithm with the Self-Tuning algorithm. We performed the simulation runs for a twenty-four hour time horizon and collected performance data in respect of power on time, revenue stream and mean service waiting time.

In Table 8.2, we depict power-on-time and Wilcoxon test statistics for results aggregated across traffic streams and demand intensities by partitioning into two, three, four and overall co-hosting instances. The power-on-time is the the sum of time, over a twenty-four hour hosting centre operation, during which all servers in the hosting centre were in operational mode and, therefore, consuming power. In case of the Self-Tuning algorithm, the power-on-time is simply the number of servers in the hosting centre multiplied by twenty-four hours since it does not attempt to save power. The Energy-Aware algorithm attempts to save power by placing servers in the hibernation mode. Therefore its total power-on-time for the hosting centre will be less than or equal to that of the Self-Tuning algorithm for any simulation run instance. We normalise the power-on-time of the Energy-Aware algorithm by the Self-Tuning

Hosting	**Revenue**	**Mean Service Waiting Time**
	$\frac{\sum_{i=1}^{n} \text{Energy-Aware}_i}{\sum_{i=1}^{n} \text{Self-Tune}_i}$	$\frac{\sum_{i=1}^{n} \text{Energy-Aware}_i}{\sum_{i=1}^{n} \text{Self-Tune}_i}$
2-Services	0.99	1.02
3-Services	0.99	1.01
4-Services	0.99	1.02
Overall	0.99	≈ 1.02

Table 8.3: Normalised revenue stream and mean waiting time for service performance of the Energy-Aware algorithm against Self-Tuning algorithm aggregated across traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), $n = 200$ and for overall, $n = 600$.

algorithm's power-on-time and observe a 5% power-on-time saving for the overall hosting instance whilst for each individual hosting instance, we note power-on-time reduction in the range of 4%–5%. These results are highly significant as shown by the Wilcoxon matched-pairs signed ranks test. Despite the complexity of traffic stream types and demand intensity levels, the Energy-Aware algorithm is able to reduce power-on-time on each occasion as highlighted by W^+ column in the table. We have used 90 seconds delay for a server to transition from hibernation mode to operational mode. However, if we use 10 seconds delay instead then the power-on-time reduction would be 5.3%–6.7%.

Given that servers do not take any part in serving requests whilst in hibernation mode, an increase in demand for service may lead to performance degradation in terms of revenue stream and service waiting time. In Table 8.3, we give performance results for the revenue stream and mean waiting time by partitioning into two, three, four and overall co-hosting instance. These results are aggregated across five traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and four demand intensity levels (low, medium, high and super high). We normalise the Energy-Aware algorithm's revenue streams by revenue streams of the Self-Tuning algorithm and observe that the revenue stream performance degradation is 1% for overall and individual hosting instance. We define service waiting time to be the difference between arrival time of a request at the hosting centre and time a server begins servicing it. For each simulation run in our experiment, we recorded the mean service waiting time for hosting centre when managed by the Energy-Aware or Self-Tuning algorithms. We normalised the mean service waiting time of the Energy-Aware algorithm by the Self-Tuning algorithm's mean service waiting time. We observe that the mean waiting time performance degrades

Wilcoxon Matched-Pairs Signed Ranks Test								
Host	**Revenue**				**Mean Service Waiting Time**			
	(Energy-Aware)-(Self-Tune)				**(Energy-Aware)-(Self-Tune)**			
	W^+	W^-	N	$\leq p$-value	W^+	W^-	N	$\leq p$-value
2	5348	14317	198	3.177e-08	14009	6091	200	1.367e-06
3	7270	12830	200	6.953e-03	12258	7842	200	7.070e-03
4	5684	14416	200	1.003e-07	12075	8025	200	1.350e-02
Overall	56092	123009	598	2.480e-15	113629	66671	600	3.265e-08

Table 8.4: Wilcoxon matched-pairs signed ranks test for Energy-Aware algorithm against Self-Tuning aggregated across traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) and demand intensities (low, medium, high and super high) where for each co-hosting instance (two, three and four services), number of matched pairs is 200 and for overall, number of matched pairs is 600.

2% for overall hosting instance whilst degrading 1%–2% for individual hosting instances. To determine statistical significance of the revenue stream and mean waiting performance degradation, we applied the Wilcoxon matched-pairs signed ranks test by partitioning the performance data as before and illustrate the result of the test in Table 8.4. We note that the revenue stream performance degradation of the Energy-Aware algorithm is highly significant ($> 99\%$) irrespective of which hosting instance is considered. As for the mean service waiting time, we observe that performance degradation is at least better than 98% significant irrespective of the hosting instance.

To determine the behaviour of the energy-aware feedback loop when the hosting centre is exposed to various levels of demand intensity, we partitioned the performance data based on four demand intensity levels irrespective of traffic stream types. We further partitioned each demand intensity level performance data along individual and overall hosting instances. We normalised the power-on-time of the Energy-Aware algorithm by power-on-time of the Self-Tuning algorithm and show the results in Table 8.5. We have not included Wilcoxon statistical test result since the Energy-Aware algorithm is able to decrease power-on-time in all external environment considered and, therefore, results are highly significant as illustrated in Table 8.2. We observe from the results that the Energy-Aware algorithm is consistently able to achieve less power-on-time than the Self-tuning algorithm across all four demand intensity levels. We anticipated that the energy-aware feedback loop will be active in low to medium demand intensity range where there is ample opportunities for servers to hibernate but we did not expect such opportunities when the demand intensity is in the high to super high range. However, closer examination of the normalised power-on-time suggests diminishing

Normalised Power-On-Time		
ρ	Hosting	$\frac{\sum_{i=1}^{n} \text{Energy-Aware}_i}{\sum_{i=1}^{n} \text{Self-Tune}_i}$
Low	2-Services	0.91
Low	3-Services	0.91
Low	4-Services	0.91
Low	Overall	0.91
Medium	2-Services	0.95
Medium	3-Services	0.95
Medium	4-Services	0.94
Medium	Overall	0.95
High	2-Services	0.96
High	3-Services	0.96
High	4-Services	0.96
High	Overall	0.96
Super High	2-Services	0.97
Super High	3-Services	0.97
Super High	4-Services	0.97
Super High	Overall	0.97

Table 8.5: Normalised Power-On time performance for Energy-Aware algorithm against Self-Tuning algorithm aggregated across traffic streams (Step, Heavy, Real, Inhomogeneous Poisson and Hybrid) where for each co-hosting instance (two, three and four services), $n = 50$ whilst, for overall, $n = 150$.

opportunities to hibernate as demand intensity increases. We note that at a low demand intensity level, the normalised power-on-time is 0.91 whilst, at a super high level, power-on-time drops to 0.97. We point out that if the transition delay time of 10 seconds is used instead of 90 seconds then the power-on-time would be 0.88 at low demand intensity level and 0.96 at super high level. Typical mean power consumption for a PIII 500 MHz PowerEdge with 3 disks using either FreeBSD 4.0, Windows 2000 or Solaris 2.7 operating system is 124 watts when in the operational mode whilst only 2.5 watts in the hibernation mode [CATV01].

8.5 Summary

We introduced the Energy-Aware server ensemble orchestration algorithm in this chapter that builds on the Self-Tuning algorithm. The new algorithm is based on the energy saving feedback

loop that senses when demand for services wanes and places servers in the hibernation mode in response to this slack in demand. Given that demand for service waxes and wanes across the hosting centre over a lengthy time horizon and servers are placed in hibernation mode, overall reduction in energy consumption is achieved by the new algorithm since the hibernation mode consumes significantly less energy than the full operational mode.

The new algorithm was tested for effectiveness against the base Self-Tuning scheme for energy saving, adaptive revenue stream performance and mean service waiting time experienced by users across the hosting centre. The effectiveness was tested on an environment of co-hosting two, three and four services, five traffic stream behaviours and four demand intensities. Our experiments demonstrated that the energy-aware feedback is able to exploit wanes in demand for service and achieve at least 5% time spent in hibernation mode across the server ensemble for the external environment considered. The time spent in the hibernation mode translates into significant energy saving since consumption between the operational and hibernation mode can vary by a factor of up to 50. The experimental results also demonstrated that the energy saving is not entirely cost free as the revenue stream of the new algorithm degraded by a small margin of 1% whilst the mean service waiting time experienced by users across the hosting centre increased by a small margin of approximately 2% in comparison to performance of the Self-Tuning technique. Finally, results of our experimental study show that energy saving achieved by the new algorithm degrades as demand intensity increases. This is intuitive since higher demand intensity across the hosting centre presents fewer opportunities to place servers in the hibernation mode. Nevertheless, our algorithm is able to achieve energy savings even when the hosting centre is subject to the super high level of demand intensity. Of course, there may be a circumstance where it may or may not make sense to reduce energy cost. Consider, for example, an internet hosting centre with energy cost representing 10% of the operating cost and reducing it by 5% results in revenue loss of 1%. Clearly, if reduction in revenue results in net loss then it does not make any sense to try to save energy cost. However, if the price of electricity fluctuates significantly as observed in California ($£55 - £277$ per MWh) [CATV01] then it definitely makes sense to save energy during price peaks.

Chapter 9

Conclusions

To know, is to know that you know nothing. That is the meaning of true knowledge.

Confucius *(551-479 BC)*

9.1 Contribution Summary and Concluding Remarks

The unrelenting growth of internet services in almost every aspect of human activity has led to the dependence on its infrastructure akin to dependence on electric, telecommunication or water infrastructures. However, the internet service infrastructure is differentiated by an open and dynamic environment of the Internet in which demand is highly unpredictable along a time as well as magnitude axes. As a result, a trend of co-hosting multiple services for a fee on a collection of servers in a centralised facility has emerged to address performance issues with respect to the infrastructure dependence. The new hosting paradigm has gained a significant foothold in the commercial domain with players such as the American Telegraph and Telephone, British Telecom, International Business Machines Corporation and Microsoft Corporation investing heavily in the infrastructure. The total revenue for the hosting industry is estimated to be £22 billion for the year 2005. In this context, the internet hosting centres face the problem of orchestrating their collection of servers amongst co-hosted services to optimise revenue in the face of an unpredictable future demand.

This thesis proposed a new biomimetic server ensemble orchestration algorithm for internet hosting centres. The novel approach is based on exploiting the remarkable homology between internet hosting centres and honeybee colonies. Our approach is distributed, robust, adaptive, versatile and inherits the autonomic property of the honeybee colonies. The remainder of this

section summarises the major contributions and conclusions of the thesis.

1. A collection of algorithms based on the biomimetic approach for orchestrating server ensemble in internet hosting centres has been developed. Each proposed algorithm is utilised as a base for successive introduction of new biomimetic algorithms to address issues of revenue stream and energy saving.

- The Self-Organising algorithm addresses optimisation of the hosting fee revenue stream by exploiting the self-organising inner workings of the forager orchestration in honeybee colonies. It mimics key interactions and mechanisms of forager orchestration in the honeybee colony and implements them onto the internet hosting centre. Each server in the new algorithm is autonomous and mimics the behaviour of the forager whilst orchestration decisions are made independently of any centralised control or *a priori* knowledge of service demand.

- The Self-Tuning algorithm also addresses optimisation of revenue stream by extending the Self-Organising algorithm to improve revenue stream performance. It successfully exploits the responsive nature of the Self-Organising algorithm and attractiveness of revenue opportunities elsewhere in the hosting centre to introduce additional feedback mechanisms that either makes each server in the ensemble exuberant or unenthusiastic to orchestration. Essentially, if a server believes that its revenue rate is mediocre and there exists better opportunities that is, grass-is-greener-elsewhere then the server in question will become exuberant whilst if it believes that grass-is-really-not-greener-elsewhere then it will become unenthusiastic to orchestration.

- The Energy-Aware algorithm addresses the reduction in energy consumption of the server ensemble whilst minimising degradation in revenue stream. It extends the Self-Tuning algorithm by introducing additional feedback mechanisms to successfully exploit opportunities over a lengthy time horizon to place servers in energy saving mode when demand for service wanes. It also successfully benefits from the natural feedback that occurs as a consequence of the new energy-aware feedback to match the number of servers required for a given service demand load. It is able to achieve this since any future service demand load that would have been borne by servers in energy saving mode will be handled by the rest of servers thus making them resistant to being placed in energy saving mode.

2. A collection of algorithms based on classical or omniscient approach for orchestrating

servers in the hosting centre has been developed. Each algorithm uses demand forecasting to formulate successive orchestration policies to address the issue of revenue stream. The classical approach uses forecast based on historical demand whilst omniscient approach assumes complete knowledge of future demand and, therefore, has an ideal forecast.

- The Greedy algorithm is based on the classical approach and uses demand forecast from recent history to make orchestration decisions for uncertain future demand. It always chooses an orchestration policy that would be optimal if the future demand matched the forecast. The time horizon is divided into equal length discrete time steps and, at beginning of each step, a new orchestration policy is successfully formulated using demand data collected from previous step. Therefore, the servers in the ensemble are not independent and act under the strict central authority of the Greedy "conductor".

- The Optimal Omniscient algorithm has an ideal forecast owing to its awareness of future demand and, being prescient, enables the algorithm to make optimal server orchestration decisions. The time horizon is split into discrete time step of equal length and with knowledge of demand at beginning of each step, the algorithm computes a sequence of optimal orchestration policies using a dynamic programming formulation. Given the sequence of optimal orchestration policies, it provides an upperbound on the possible revenue stream. We have successfully implemented a parallelised non-recursive approximation to the infeasible recursive dynamic programming formulation. As before, servers in the ensemble are orchestrated in a regimented order under the Optimal Omniscient algorithm's central command.

- The Optimal-Static Omniscient also has an ideal forecast and, therefore, makes an optimal server orchestration decision. As before, demand for service is known in advance for a time horizon partitioned into equal length discrete time steps. However, unlike the Optimal Omniscient algorithm, this algorithm computes a single orchestration policy that optimises the revenue stream over the time horizon. Therefore, it provides an upperbound on the possible revenue stream for *ad hoc* static techniques. The server ensemble is orchestrated according to the optimal static orchestration policy at the beginning and remains in effect for the entire time horizon.

3. Both Greedy and, in particular, Optimal Omniscient algorithms need to compute optimal

orchestration policy frequently such that exploring complete state space becomes computationally prohibitive especially as number of services hosted increases. We have successfully developed a Local Neighbour Search algorithm that significantly reduces computation time for determining an optimal orchestration policy for a given state of the hosting centre and a known demand for service.

4. An extensive simulation framework for an internet hosting centre being managed by biomimetic, classical and omniscient algorithms as well as for external Internet environment that drive the hosting centre has been implemented. Its features include the following:

- An internet hosting centre can be configured with a given size of server ensemble to co-host up to four services with pay-per-request-served SLAs hosting fee schedule. The management of server ensemble can be configured to use the Self-Organising, Self-Tuning, Energy-Aware, Greedy, Optimal Omniscient and Optimal-Static Omniscient server orchestration algorithms.

- The co-hosted services are driven by individual users submitting requests over the Internet. The aggregation of requests assume certain traffic stream behaviour whilst individual user's response to delay in receiving service assume a baulking behaviour. The external environment can be configured to take on five types of traffic stream behaviour– Step, Heavy, Real, Inhomogeneous Poisson and Hybrid as well as configurable baulking behaviour.

- The hosting centre's demand intensity can be configured to any value by setting the mean service time parameter across the server ensemble based on the demand intensity value desired and mean arrival rate for a given traffic stream.

5. In order to assess effectiveness of the Self-Organising algorithm over the Greedy and omniscient approaches, we have performed two extensive performance studies to determine the adaptiveness and versatility of the Self-Organising algorithm. The studies involved co-hosting up to four services, four levels of demand intensity and five types of traffic stream behaviour. We draw the following conclusions from the studies:

- First and foremost, the biomimetically inspired Self-Organising algorithm is able to adapt very well to widely varying external environments in an autonomic manner. It is able to outperform the Greedy and Optimal-Static Omniscient algorithms whilst performing

close to the upperbound Optimal Omniscient algorithm. On many external environment circumstances comprising traffic stream type, service hosting and demand intensity, the conclusion is supported with confidence better than 99%.

- The Self-Organising algorithm adapts very well to dynamic traffic stream behaviours. However, its responsive nature is counter-productive when traffic streams exhibit mean arrival rates that take on different values on a short timescale. On the other hand, it proves to be beneficial when traffic streams exhibit behaviours in which the mean arrival rates take on different values on a long timescale. Conversely, the non-responsive Optimal-Static Omniscient algorithm benefits when traffic stream behaviour is characterised by a short timescale whilst it is at a disadvantage when traffic stream behaviour is characterised by a long timescale. The conclusion is supported with confidence at least better than 99% for traffic streams behaviour controlled using short, medium and long timescale.

- The traffic stream behaviour suggests an en-masse hysteria on the part of the service users. However, the biomimetic algorithm uses a randomised and decentralised strategy of smoothing traffic bursts to orchestrate server ensemble in response and achieves highly successful adaptive performance.

- The Self-Organising algorithm exhibits significantly superior versatility to the Greedy algorithm. The conclusion is supported with confidence greater than 99%. We posit that the biomimetic approach derives its versatility from the random way in which foragers are orchestrated and the overall self-organising property of the technique that has evolved over many millennia.

6. The efficacy of the Self-Tuning algorithm is assessed by an extensive performance study to determine its adaptiveness in comparison to the base Self-Organising algorithm. The study entailed an external environment comprising five types of traffic stream behaviour, four levels of demand intensity and up to four services co-hosted. We conclude following from the study:

- The Self-Tuning algorithm is successful in demonstrating the virtue of tuning self-organising orchestration behaviour through a simple feedback mechanism.

- Earlier, we highlighted the superior versatility property of the Self-Organising algorithm which limits opportunity to tune performance through internal parameter settings. Having

demonstrated better comparative performance of the Self-Tuning algorithm over the Self-Organising algorithm, we suggest that the performance of the biomimetic approach can be tuned by introducing appropriate additional feedback mechanisms that alter server's orchestration behaviour.

7. The performance of the Energy-Aware algorithm is comparatively assessed with the base Self-Tuning algorithm. Again, the performance study used an external environment that consisted of five types of traffic stream behaviour, four levels of demand intensity and co-hosting of up to four internet services. The results of the performance study leads to the following conclusions:

- The Energy-Aware algorithm is highly successful in exploiting opportunities to conserve energy consumed by the server ensemble.
- The reduction in energy consumption is not cost free. The Energy-Aware algorithm trades-off revenue stream and user waiting time for energy saving.
- The collective behaviour borne out of the simple actions of an individual is very powerful. This is demonstrated by the Energy-Aware algorithm where an individual server acts autonomously on local information which generates feedback that enables the server ensemble to collectively reach an equilibrium point for matching servers required to meet demand for service.

8. The novel approach to server orchestration proposed in this thesis is inspired from swarm intelligence of honeybee colonies. Consequently, the approach benefits from inheriting numerous properties with desirable implications for applicability in practise:

- The self-organising property of the approach means that the server orchestration algorithm is autonomic. That is, servers orchestrate themselves spontaneously as a result of internal stimuli only. This is highly beneficial in terms of managing large server ensemble complexity since apart from reducing cost associated with operational staff, the servers may detect problems and react to them faster, in real-time.
- The robustness property of the approach means that the server orchestration algorithm is resilient to server failures. The algorithm is able to function independently of the size of server ensemble since the server behaviour is exactly the same irrespective of number of

servers, number of services hosted and parameter values. Naturally, the algorithm's ability to function in the face of failures is highly advantageous in practise.

- The distributed property of the approach means that each server in the ensemble function independently. That is, there is no centralised coordinator controlling behaviour of servers in the ensemble. In practise, such a property translates into scalability by independent behaviour and enhancement of robustness by absence of single point of failure.

- We have demonstrated that the approach inherits adaptiveness and versatility properties from swarm intelligence of honeybee colonies. These properties enable the server orchestration algorithm to perform well in widely varying internal and external circumstances without requiring specific parameter values. Clearly, these properties lend themselves well to application in a dynamic operating environment.

- Above, we highlighted practical significance of the properties of the honeybee inspired swarm intelligence approach for server ensemble orchestration in internet hosting centres. Given that swarm intelligence approaches are generally characterised by such properties, one can imagine some other artificial domain benefiting from an application of a suitable biomimetic algorithm. For example, consider the wireless network domain where mobile *ad hoc* networks are formed dynamically and temporally by autonomous collection of mobile devices that communicate over bandwidth constrained wireless links with no fixed infrastructure or centralised coordination [SYJ04]. The number of mobile device participating in the network is dynamic and, therefore, the topology may change rapidly and unpredictably over time. Typical application of such a network include sensor data array [BLB02], disaster relief [ZS03], rural Internet [PFH02] and animal tracking [SH03]. These application require mobile *ad hoc* networks to function under variable degree of dynamic environment and constraints. Therefore, it requires efficient distributed algorithms to determine network topology, link scheduling and routing of messages [NIS05]. In such a domain, self-organising, distributed and robustness properties would lend themselves well in practise: the network topology is unpredictable over time thus requiring ongoing organisation; mobile devices are distributed and autonomous; and prone to failure owing to limited battery life. Similarly, adaptiveness and versatility would offer significant advantage in practise as the mobile devices can adapt to widely varying circumstances and enables them to operate autonomously once deployed. Finally, certain applications in the domain may not offer the opportunity to

tune algorithm's parameters post deployment and, for such applications, versatility would be a valuable property.

9.2 Future Research Directions

There are several ways in which the biomimetic approach could be refined and extended. We envisage further investigation into the following significant research directions:

- The case studies involving biomimetic algorithms explored performance over co-hosting of two, three and four services only. However, the scalability of performance to a scenario where large number of services are co-hosted will be highly significant. Based on the widely varying dynamic environmental conditions faced by honeybee colonies and their survival over many millennia, our intuition is that the biomimetic algorithm has the potential to scale to a greater number of co-hosted services.

- The comparative assessment study highlighted the fact that the Self-Organising algorithm outperformed the Greedy algorithm by a significant margin when the hosting centre co-hosted three services. It appears that there is a region of parameter space where the Greedy algorithm's performance degrades considerably when compared with co-hosting of two and four services. This issue needs to be explored further to determine the cause of performance degradation.

- The performance study of the Self-Tuning and Energy-Aware algorithms measured their efficacy against the Self-Organising algorithm. Although the study provides comparison against the original biomimetic algorithm, it lacks completeness since conventional algorithms were not included in the study. Therefore, the performance study should be extended to compare performance against the respective counterpart (self-tuning and energy-aware) conventional algorithms.

- We outlined the trade-off involved in choosing co-hosting, discrete time step length and the time horizon for the memory and CPU intensive Optimal Omniscient algorithm in Section 5.4.1. Consequently, we chose a discrete time step length that is greater than the re-purposing downtime by a factor of 6. The choice of discrete time step length may have an effect on the system behaviour. Therefore, a study encompassing different values of

discrete time step length should be performed to ascertain its impact on the performance of the respective algorithms.

- The current implementation of the Optimal Omniscient algorithm is prohibitive especially when the number of services co-hosted increases. An alternative implementation, possibly based on a network flow approximation, would be helpful for the scalability performance study discussed above.

- We assessed the efficacy of the Self-Organising algorithm against algorithms based on classical and omniscient approaches. A performance study that includes the algorithm of Jayram *et al.* [JKK+01] would further validate effectiveness of the biomimetic approach.

- In the current implementation of the biomimetic algorithms, the server ensemble utilises a centralised single advertisement board to communicate global information. Clearly, this represents a single point of failure and study encompassing fault-tolerant techniques such as the distributed message passing approach would be beneficial.

- The Self-Tuning and Energy-Aware algorithms demonstrated their effectiveness in spite of the fact that the parameter setting was based on common sense reasoning. A performance tuning study would highlight their sensitivity or versatility to the internal parameter settings.

- Finally, we do not currently have a tool to visualise activities of the internet hosting centre. A tool that enables visualisation of key activities related to orchestration, demand load and algorithms would enable greater insight into the performance of algorithms.

Bibliography

[AB99] Tarek F. Abdelzaher and Nina Bhatti. Web Server QoS Management by Adaptive Content Delivery. http://kabru.eecs.umich.edu/servers/, 1999.

[ACP95] Thomas E. Anderson, David E. Culler, and D. Patterson. A Case for NOW (Network of Workstations). *IEEE Micro*, 12(1):54–64, February 1995.

[ADC97] A. Arpaci-Dusseau and D. Culler. Extending Proportional-Share Scheduling to a Network of Workstations. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, June 1997.

[ADMC] Jussara Almeida, Mihaela Dabu, Anand Manikutty, and Pei Cao. Providing Differentiated Levels of Service in Web Content Hosting. http://www.cs.wisc.edu/~cao/papers/diff-QoS.html.

[AFF+01] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D.P. Pazel, J. Pershing, and B. Rochwerger. Oceano - SLA Based Management of a Computing Utility. In *7th IFIP/IEEE International Symposium on Integrated Network Management*, 2001.

[AJ99] Martin Arlitt and Tai Jin. Workload Characterization of the 1998 World Cup Web Site. http://citeseer.nj.nec.com/article/arlitt99workload.html, September 1999.

[AL00] Tarek F. Abdelzaher and Chenyang Lu. Modeling and Performance Control of Internet Servers. In *IEEE Conference on Decision and Control*, 2000.

[Alt00a] What Next for Unmetered Access? http://news.bbc.co.uk/hi/english/business/, August 22nd, 2000.

[Alt00b] Altavista Blames BT for 'fiasco'. http://news.bbc.co.uk/hi/english/business/, August 22nd, 2000.

[ALT03] Alteon Web Switches. http://www.nortelnetworks.com/products/01/alteon/webswitch/, 2003.

[Apa03] Apache HTTP Web Server. http://www.apache.org/httpd.html, August, 2003.

[APB96] Eric Anderson, Dave Patterson, and Eric Brewer. The Magicrouter– An Application of Fast Packet Interposing. In *Proceedings of* 2^{nd} *symposium on Operating System Design and Implementation*, 1996.

[App03] Apple. The New Xserve. http://www.apple.com/xserve/, 2003.

[ASB] Tarek F. Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance Guarantees for Web Server End-System: A Control-Theoretic Approach. http://kabru.eecs.umich.edu/servers/.

[AT03] AT and T. Enterprise Hosting Service. http://www.business.att.com/products/, 2003.

[BAG00] R. Buyya, D. Abramson, and J. Giddy. Economy Driven Resource Management Architecture for Computational Power Grids. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA2000), Las Vegas,USA., 2000.

[BBC99] Online Auction Site Crashes. http://news.bbc.co.uk/hi/english/business/, June 11th, 1999.

[BBC00a] Inland Revenue Web Fiasco. http://news.bbc.co.uk/hi/english/business/, August 18th, 2000.

[BBC00b] How the Web Was Wounded. http://news.bbc.co.uk/hi/english/business/, February 10th, 2000.

[BBC01] UK Surfers Swamp News Sites. http://news.bbc.co.uk/1/hi/sci/tech/1540441.stm, 12 September, 2001,.

[BBC03] Glitch Hits Online Film Launch. http://news.bbc.co.uk/1/hi/entertainment/film/3085204.stm, 10 September, 2003.

[BBL00] Mark Baker, Rajkumar Buyya, and Domenico Laforenza. The Grid: International Efforts in Global Computing. In *Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*, l'Aquila, Italy, July 31st, 2000 - August 6th, 2000.

[BC96] A. Bestavros and C. Cunha. Server-initiated Document Dissemination for the WWW. *IEEE Data Engineering Bulletin*, 19(3):3–11, 1996.

[BCLM98] Azer Bestavros, Mark Corvella, Jun Lou, and David Marting. Distributed Packet Rewriting and its Application to Scalable Server Architectures. In *Proceedings of ICNP'98: The* 6^{th} *International Conference on Network Protocols*, Austin, Texas, October, 1998.

[BDK95] A. Baratloo, P. Dasgupta, and Z. M. Kedem. CALYPSO: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms. In *Proc. of the Fourth IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-4)*, pages 122–129, 1995.

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the* 3^{rd} *USENIX symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, February 1999.

[BDT99] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press, 1999.

[BF99] Nina Bhatti and Rich Friedrich. Web Server Support for Tiered Services. *IEEE Network*, 13(5):64–71, September 1999.

[BGM+02] Lawrence D. Brown, Noah Gans, Avishai Mandelbaum, Anat Sakov, Haipeng Shen, Sergey Zeltyn, and Linda H. Zhao. Statistical Analysis of a Telephone Call Center: A Queueing-Science Perspective. http://www-stat.wharton.upenn.edu/~haipeng/, November 2002.

[Bic97] Peter Bickford. Worth the Wait? Human Interface Online, October 1997.

[BIKZ98] Arash Baratloo, Ayal Itzkovitz, Zvi M. Kedem, and Yuanyuan Zhao. Just-in-time transparent resource management in distributed systems. Technical Report TR1998-762, Department of Computer Science, New York University, 1998.

[Bio05] Biomimicry. Case Studies. http://www.biomimicry.net/, 2005.

[BKB00] Anna Bouch, Allan Kuchinsky, and Nina Bhatti. Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service. Technical Report HPL-2000-4, HP Laboratories, 2000.

[BLB02] Allan Beaufour, Martin Leopold, and Philippe Bonnet. Smart-tag based data dissemination. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 68–77, Atlanta, Georgia, USA, 2002.

[BLFN95] T. Berner-Lee, R. T. Fieding, and H. F. Nielsen. Hypertext Transfer Protocol - HTTP/1.0. http://www.w3.org/pub/WWW/Protocols/HTTP1.0/draft-ietf-http-spec.html, 1995.

[Blo05] Michael Bloch. Ecommerce - Credit Card Fraud Costs and Statistics. http://www.tamingthebeast.net/articles2/credit-card-fraud.htm, 2005.

[BM00] Scott M. Baker and Bongki Moon. Distributed Cooperative Web Servers. http://www8.org/w8-papers/2a-webserver/distributed/distributed.html, August, 2000.

[BR03] Ricardo Bianchini and Ram Rajamony. Energy Conservation in Clustered Servers. *IEEE Computer*, 36(12):39–48, December 2003.

[Bre00] Eric A. Brewer.
Lessons from Giant-Scale Services. http://db.cs.berkeley.edu/~jmh/cs262, 2000.

[BSD01] A. Bouch, M. Sasse, and H. DeMeer. Of Packets and People: A User-Centered Approach to Quality of Service. In *IWQoS 2001*, 2001.

[BSTV93] John J. Barholdi, Thomas D. Seeley, Craig A. Tovey, and John H. Vande Vate. The pattern and Effectiveness of Forager Allocation Among Flower Patches by Honey Bess Colonies. *Journal of Theoretical Biology*, (160), 1993.

[BT03] BT. BTWebWorld. http://www.btwebworld.com/, 2003.

[CADAD+97] David E. Culler, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Brent Chun, Steven Lumetta, Alan Mainwaring, Richard Martin, Chad Yoshikawa, and Frederick Wong. Parallel Computing on the Berkeley NOW. In *Proceedings of the 9th Joint Symposium on Parallel Processing*, Kobe, Japan, 1997.

[CATV01] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, and Amin M. Vahdat. Managing Energy and Servers Resources in Hosting Centres. In *18th ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[CB97] Mark E. Crovella and Azer Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. *IEEE /ACM Transactions on Networking*, 5(6):835–846, 1997.

[CB98] Yatin Chawathe and Eric Brewer. System Support for Scalable and Fault Tolerant Internet Services. In *Proceedings of Middleware '98*, Lake District, UK, September 1998.

[CD98] Gianni Di Caro and M. Dorigo. AntNet: Distributed Stigmergetic Control for Communication Network. *Journal of Artificial Intelligence Research*, 9:317–365, 1998.

[CD01] Jeff Chase and Ron Doyle. Balance of Power: Energy Management for Server Clusters. In *Proceedings of the Eighth Workshop on Hot Topic in Operating Systems HotOS'2001*, May 2001.

[CDM+] A. Colorni, M. Dorigo, F. Maffioli, V. Maniezzo, G. Righini, and M. Trubian. Heuristics From Nature for Hard Combinatorial Optimization Problems. *International Transaction in Operational Research*, 2(1):1–21.

[CDM91] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. Distributed Optimization by Ant Colony. In *ECAL91– European Conference on Artificial Life*, Paris, France, 1991.

[CDN+96] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrel. A Hierarchical Internet Object Cache. In *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, CA, January, 1996.

[CF01] Fintan Culvin and Xristina Faulkner. Browsing the Web: Delay, Determination and Satisfaction. In *34th Hawaii International Conference on System Sciences*, 2001.

[CFK+98] Karl Czajkowski, Ian T. Foster, Nicholas T. Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A Resource Management Architecture for Metacomputing Systems. In *JSSPP*, pages 62–82, 1998.

[CH97] D. Costa and A. Hertz. Ant Can Colour Graphs. *Journal of Operational Research Society*, 48:295–305, 1997.

[Che99] Ludmilla Cherkasova. Scalable Web Hosting Service. Technical Report HPL-1999-52(R.1), HP Laboratories, 1999.

[Che02] Allen Chen. Data Centers Energy Use: Truth versus Myth. http://www.lbl.gov/Science-Articles/Archive/data-center-energy-myth.html, 16th July 2002.

[CID99] Jim Challenger, Arun Iyengar, and Paul Dantzig. A Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of the IEEE INFOCOM'99 Symposium*, March, 1999.

[CIS03] Cisco 400 series LocalDirectors. http://www.cisco.com/warp/public/cc/pd/cxsr/400/index.shtml, 2003.

[CKR98] Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford. Improving End-to-End Performance of the Web Using Server Volumes and Proxy Filters. In *ACM SIGCOMM*, pages 241–253, 1998.

[CM99] Xiangping Chen and Prasant Mohapatra. Providing Differentiated Service from an Internet Server. In *IEEE International Conference on Computer Communications and Networks*, 1999.

[Con80] W. J. Conover. *Practical Nonparametric Statistics.* John Wiley and Sons, 2^{nd} edition, 1980.

[Cor05] Hewlett Packard Corp. Energy Efficiency. http://www.hp.com/hpinfo/globalcitizenship/environment/, 2005.

[Cow04] Tim Cowden. Kansas City Area Development Council– Comparative Operating Cost Proforma. http://www.thinkkc.com/datacenters/proforma/proforma.htm, 2004.

[CT04] Carroll Croarkin and Paul Tobias, editors. *NIST/SEMATECH e-Handbook of Statistical Methods.* NIST/SEMATECH, http://www.itl.nist.gov/div898/handbook/, 2004.

[CY97] Kenichi Chinen and Suguru Yamaguchi. An Interactive Prefetching Proxy Server for Improvement of WWW Latency. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[CZ01] Xin Chen and Xiaodong Zhang. Coordinated Data Prefetching by Utilizing Reference Information at Both Proxy and Web Servers. *ACM SIGMETRICS Performance Evaluation Review*, 29(2), September 2001.

[CZB98] Pei Cao, Jin Zhang, and Kevin Beach. Active Cache: Caching Dynamic Contents on the Web. In *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 373–388, 1998.

[DAC96] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective distributed scheduling of parallel workloads. In *ACM SIGMETRICS'96 Conference on the Measurement and Modeling of Computer Systems*, 1996.

[DG97] M. Dorigo and L. M. Gambardella. Ant Colonies for the Travelling Salesman Problem. *BioSystem*, (43):73–81, 1997.

[DGLS99] Bill Devlin, Jim Gray, Bill Laing, and George Spix. Scalability Terminology: Farms, Clones, Partitions and Packs— RACS and RAPS. http://research.microsoft.com/~gray, December 1999. Technical Report, MS-TR-99-85.

[DKMT96] D. Dias, W. Kish, R. Mukherjee, and R. Tewari. A Scalable and Highly Available Web Server. In *Proceedings COMPCON'96*, pages 85–92, Santa Clara, 1996.

[DKR95] Partha Dasgupta, Zvi M. Kedem, and Michael O. Rabin. Parallel Processing on Networks of Workstations: A Fault-Tolerant, High Performance Approach. In *International Conference on Distributed Computing Systems*, pages 467–474, 1995.

[Dum81] Klaus Dumpert. *The Social Biology of Ants.* Pitman Publishing Limited, 1981.

[FCAB98] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. Tech. Rep. 1361, Department of Computer Science, University of Wisconsin at Madison, February 1998.

[FCLJ99] Li Fan, Pei Cao, Wei Lin, and Quinn Jacobson. Web Prefetching Between Low-Bandwidth Clients and Proxies: Potential and Performance. In *Measurement and Modeling of Computer Systems*, pages 178–187, 1999.

[FK99] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*, chapter 2: Computation Grids, pages 15–51. Morgan Kaufmann Inc, 1999.

[FKSR02] Daniela De Farias, A. J King, Mark Squillante, and Benjamin Van Roy. Dynamic Control of Web Server Farm. In *The 2nd Annual INFORMS Revenue Management Section Conference*, 2002.

[FS96] Bryan Ford and Sai Susarla. CPU Inheritance Scheduling. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105, 1996.

[GBHC00] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, Distributed Data Structures for Internet Service Construction. http://www.cs.berkeley.edu/~gribble, 2000.

[GTD99] L. M. Gambardella, E. D. Taillard, and M. Dorigo. Ant Colonies for the Quadratic Assignment Problem. *Journal of Operational Research Society*, 50:167–176, 1999.

[GWBC99] Steven D. Gribble, Matt Welsh, Eric A. Brewer, and David Culler. The MultiSpace: An Evolutionary Platform for Infrastructural Services. http://www.cs.berkeley.edu/~gribble, 1999.

[Haa05] Mads Haahr. Random Number Service. http://www.random.org/, 2005.

[HDL98] Jonathan M. D. Hill, Stephen R. Donaldson, and Tim Lanfear. Process Migration and Fault Tolerance of BSPlib Programs Running on Networks of Workstations. In *European Conference on Parallel Processing*, pages 80–91, 1998.

[HGKM98] Guerney D. H. Hunt, German S. Goldszmidt, Richard P. King, and Rajat Mukherjee. Network Dispatcher: A Connection Router for Scalable Internet Services. In *Proceedings of the* 7^{th} *International World Wide Web Conference*, April, 1998.

[HP03a] HP. ProLiant BL Servers. http://h18004.www1.hp.com/products/servers/platforms/index-bl.html, 2003.

[HP03b] HP. ProLiant Servers. http://h18004.www1.hp.com/products/servers/platforms, 2003.

[HPS97] James C. Hu, Irfan Pyarali, and Douglas C. Schmidt. Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-Speed Networks. In *Proceedings of the* 2^{nd} *Global Internet mini-conference held in conjunction with GLOBECOM'97*, Phoenix, Arizona, November 1997.

[HPS98] James C. Hu, Irfan Pyarali, and Douglas C. Schmidt. Applying the Proactor Pattern to High-Performance Web Servers. In *Proceedings of the* 10^{th} *International Conference on Parallel and Distributed Computing Systems, IASTED*, Las Vegas, Nevada, October 28-31 1998.

[IBM03a] IBM. Blade Servers. http://www-1.ibm.com/servers/eserver/bladecenter/, 2003.

[IBM03b] IBM. Cluster 1350. http://www-1.ibm.com/servers/eserver/clusters/hardware/1350.html, 2003.

[IBM03c] IBM. e-Server. http://www-1.ibm.com/servers/eserver/pseries/hardware/tour/, 2003.

[IBM03d] IBM. Managed Hosting. http://www-3.ibm.com/services/e-business/hosting/managed_hosting.html, 2003.

[IC97] Arun Iyengar and Jim Challenger. Improving Web Server Performance by Caching Dynamic Data. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, California, December, 1997.

[Inf05] Infoconomy. Alstom Creates a Virtual World. http://www.infoconomy.com/pages/search/group107168.adp, 2005.

[Int02] Intel. PC Energy-Efficiency Trends and Technologies. http://cache-www.intel.com/cd/00/00/10/27/102727_ar024103.pdf, 2002.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley- Interscience, New York, NY, April 1991.

[JK00] Paul John and Arminder Kaur. Web Server Load Balancing and Redundancy. http://www.microsoft.com/technet/ecommerce/duwwsr.asp, April, 2000.

[JKK+01] T. S. Jayram, Tracy Kimbrel, Robert Krauthgamer, Baruch Schieber, and Maxim Sviridenko. Online Server Allocation in a Server Farm via Benefit Task System. In *Proceedings of the STOC*, Hersonissos, Crete, Greece, July 6-8, 2001.

[JLDB95] M. Jones, P. Leach, R. Draves, and J. Barrera. Modular Real-Time Resource Management in the Rialto Operating System. Technical Report MSR-TR-95-16, Microsoft Research, May 1995.

[JMP05] JMP Statical Discovery Software. http://www.jmp.com/, 2005.

[JR00] Hani Jamjoom and John Reumann. QGuard: Protecting Internet Servers from Overload. http://kabru.eecs.umich.edu/servers/, 2000.

[JRR97] Michael B. Jones, Daniela Rosu, and Marcel-Catalin Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Symposium on Operating Systems Principles*, pages 198–211, 1997.

[KBM94] E. Katz, M. Butler, and R. McGrath. A Scalable HTTP Server: NSCA Prototype. *Computer Networks and ISDN Systems*, 27:155–163, 1994.

[Ker04] Sean Michael Kerner. Online Fraud Could Be Decreasing, But Remains a Worry. http://www.ecommerce-guide.com/solutions/secure_pay/article.php/3436641, 16^{th} November 2004.

[KEY01] News and Events: Press Releases. http://www.keynote.com/news_events/releases_2001/091101.html, September 11, 2001.

[Kin02] Rawlson King. Optimizing Data Center Energy Efficiency Results In Cost Savings . http://www.thewhir.com/features/dce.cfm, 10th June 2002.

[KKL+02] A. Keller, G. Kar, H. Ludwig, A. Dan, and J. L. Hellerstein. Managing Dynamic Services: A Contract Based Approach to Conceptual Architecture. In *IFIP/IEEE International Symposium on Network Operations and Management*, Florence, Italy, April 17^{th} 2002.

[KKO98] Mehmet Karaul, Yannis A. Korilis, and Ariel Orda. A Market-Based Architecture for Management of Geographically Dispersed, Replicated Web Servers. In *Proceedings of the First International Conference on Information and Computation Economics (ICE-98)*, pages 158–165, October, 1998.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 654–663, El Paso, Texas, 4-6 May 1997.

[KLM97] Tom M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *USENIX Symposium on Internet Technologies and Systems*, December 1997.

[KMR95] T. Kwan, R. McGrath, and D. Reed. NSCA's World Wide Web Server: Design and Performance. *IEEE Computer*, pages 68–74, November 1995.

[KW01] Balachander Krishnamurthy and Craig E. Wills. Improving Web Experience by Client Characterization Driven Server Adaptation, November 16 2001.

[LAS01] Chenyang Lu, Tarek F. Abdelzaher, and John A. Stankovic. A Feedback Control Approach for Guaranteeing Relative Delays in Web Server. In *IEEE RealTime Technology and Application Symposium (RTAS 2001)*, June, 2001.

[Lin67] Martin Lindauer. *Communication Among Social Bees.* Harvard University Press, 1967.

[Lin03] Linux Operating System. http://www.linux.org/, 2003.

[Lit94] Mark C. Little. C++SIM - Object Oriented Discrete Event Simulation. http://cxxsim.ncl.ac.uk, 1994.

[LRR+03] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W Keller. Energy Management for Commercial Servers. *IEEE Computer*, 36(12):39–48, December 2003.

[LTBL97] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report Tech. Rep. 1346, University of Wisconsin at Medison, April 1997.

[Mar03] John Markoff. I.B.M. Says It Benefits 2 Ways From On-Demand Computing. New York Times, November 13, 2003.

[MC98] E.P. Markatos and C. Chronaki. A Top-10 Approach to Prefetching on the Web. In *USENIX Symposium on Internet Technologies and Systems*, 1998.

[MC99] V. Maniezzo and A. Colorni. The Ant System Applied to Quadratic Assignment Problem. *IEEE Transaction on Knowledge Data Engineering*, 11:769–778, 1999.

[Mer04] Paulo Eduardo Merloti. Optimization Algorithms Inspired by Biological Ants and Swarm Behavior. http://www.merlotti.com/EngHome/Computing/AntsSim/ants.htm, 2004.

[MJ01] Jennifer D. Mitchell-Jackson. Energy Needs In An Internet Economy: A Closer Look at Data Centres. Master's thesis, Energy and Resource Group, University of California at Berkeley, July 2001.

[MLB95] R. Malpani, J. Lorch, and D. Berger. Making World Wide Web Caching Servers Cooperate. In *Proceedings of the Fourth International WWW Conference*, Boston, MA, December 1995.

[MMS88] M. Manasse, L. McGeoch, and D. Sleator. Competitive Algorithms for Online Problems. In 20^{th} *Annual Symposium on Theory of Computing*, pages 323–333, 1988.

[Mor54] Derek Wragge Morley. *The Evolution of an Insect Societies.* George Allen and Unwin Ltd, 1954.

[Nie00] Jakob Nielsen. *Designing Web Usability.* New Riders Publishing, 2000.

[NIS05] NIST. Mobile Ad Hoc Networks (MANETs). http://www.antd.nist.gov/wahn_mahn.shtml, 2005.

[PAB+98] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the ACM* 8^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, October, 1998.

[PDZ99] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the USENIX 1999 Annual technical conference*, Monterey, CA, June, 1999.

[PFH02] A. Pentland, R. Fletcher, and A. Hasson. A Road to Universal Broadband Connectivity. In *Proceedings of the Second International Conference on Open Collaborative Design for Sustainable Innovation; Development by Design (dyd02)*, Bangalore, December, 2002.

[PGH+01] S Parekh, N Gandhi, J. L. Hellerstein, D Tilbury, T. S. Jayram, and J Bigus. Using Control Theory to Achieve Service Level Objectives in Performance Management. In *IFIP/IEEE International Symposium on Integrated Network Management*, 2001.

[PH97] D. Povey and J. Harrison. A Distributed Internet Cache. In *Proceedings of the 20th Australasian Computer Science Conference*, Sydney, Australia, February 5-7, 1997.

[PM96] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. In *Proceedings of the ACM SIGCOMM '96 Conference*, Stanford University, CA, 1996.

[PM03] J. Palmer and I. Mitrani. Dynamic Server Allocation in Heterogeneous Clusters. Technical Report CS-TR: 799, School of Computing Science, University of Newcastle, May 2003.

[RCG98] M. Rabinovich, J. Chase, and S. Gadde. Not All Hits are Created Equal: Cooperative Proxy Caching Over a Wide-Area Network. In *Proceedings of the 3^{rd} International WWW Caching Workshop*, Manchester, UK, June, 1998.

[RE01] Ram Rajamony and Mootaz Elnozahy. Measuring Client-Perceived Response Time On the WWW. In *USENIX Symposium on Internet Technology and Systems*, San Francisco, California, March, 2001.

[Ric70] O. W. Richards. *The Social Insects.* Peter Smith, 1970.

[RR02] Chris Robertson and Joseph Romm. Data Centers, Power, and Pollution Prevention– Design for Business and Environmental Advantage. http://www.coolcompanies.org/images/DataCenterJune.pdf, June 2002.

[SCS91] T. D. Seeley, S. Camazine, and J. Sneyd. Collective Decision Making in Honey Bees: How Colonies Choose Among Nectar Sources. *Behaviour Ecology and Sociology*, (28):277–290, 1991.

[See85] Thomal D. Seeley. *Honey Bee Ecology: A Study of Adaptation in Social Life.* Princeton University Press, 1985.

[See95] Thomal D. Seeley. *The Wisdom of the Hive: The Social Physiology of Honey Bee Colonies.* Harvard University Press, 1995.

[See03] Thomas D. Seeley. Personal communication, 2003.

[SH03] Tara Small and Zygmunt Haas. The Shared Wireless Infostation Model – A New Ad Hoc Networking Paradigm (or Where there is a Whale, there is a Way). In *Proceedings of the ACM MOBIHOC'03*, Annapolis, Maryland, USA, June, 2003.

[SHBR96] R. Schoonderwoerd, O. E. Holland, J. Bruten, and L. Rothkrantz. Ant-based Load Balancing in Telecommunication Network. Technical Report HPL-96-76, HP Laboratories, 1996.

[squ00] Squid Proxy Cache Server. http://www.squid-cache.org/, August, 2000.

[STA01] Anees Shaikh, Renu Tewari, and Mukesh Agrawal. On the Effectiveness of DNS-based Server Selection. In *Proceedings of the IEEE INFOCOM*, Anchorage, Alaska, April, 2001.

[SUN03a] SUN. Fire B1600 Blade Platform. http://wwws.sun.com/servers/entry/blade/, 2003.

[SUN03b] SUN. Fire Servers. http://www.sun.com/lowcost/feature/v60-v65.html, 2003.

[SYJ04] S. Sesay, Z. Yang, and He J. A survey on mobile ad hoc wireless network. *Information Technology*, 3(2):168–175, 2004.

[SYZ00] Mark S. Squillante, David D. Yao, and Li Zhang. *System Performance Evaluation: Methodologies and Applications*, chapter 2: Internet Traffic: Periodicity, Tail Behavior and Performance Implications, pages 23–32. CRC Press, March 2000.

[TDVK99] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design Considerations for Distributed Caching on the Internet. In *Proceedings of the Twentieth International Conference on Distributed Computing Systems*, May, 1999.

[Tov04] Craig A Tovey. Personal communication, 2004.

[Tow03] Abigail Townsend. Tax Website Crash as Deadline Nears. http://news.independent.co.uk/uk/politics/story.jsp?story=476479, 2003.

[TS94] Craig A. Tovey and Thomas D. Seeley. Why Search Time to Find a Food-Storer Bee Accurately Indicates the Relative Rates of Nectar Collecting and Nectar Processing in Honey Bee Colonies. *Animal Behavior*, (47):311–316, 1994.

[VR98] Vinod Valloppillil and Keith W. Ross. Cache Array Routing Protocol v1.0, INTERNET-DRAFT, February 1998.

[Wak99] Russel C. Waker. *Introduction to Mathematical Programming.* Prentice-Hall Inc., 1999.

[Wan99] Jia Wang. A Survey of Web Caching Schemes for the Internet. Department of Computer Science, Cornell University, September, 1999. Technical Report TR99-1767.

[WC97a] Z. Wang and J. Crowcroft. CacheMesh: A Distributed Cache System for World Wide Web. In *Proceedings of NLANR Web Cache Workshop*, 1997.

[WC97b] D. Wessels and K. Claffy. Internet Cache Protocol (ICP), version, RFC 2186, National Laboratory for Applied Network Research/UCSD, September 1997.

[WE95] C. A. Waldspurger and Weihl. W. E. Stride Scheduling: Deterministic Proportional- Share Resource Management. Technical Report MIT/LCS/TM-528, Massachusetts Institute of Technology, 1995.

[WGBC00] Matt Welsh, Steven D. Gribble, Eric A. Brewer, and David Culler. A Design Framework for Highly Concurrent Systems. http://www.cs.berkeley.edu/~mdw/papers/events.pdf, 2000.

[Wig01] Richard W. Wiggins. The Effects of September 11^{th} on the Leading Search Engine. *First Monday*, 7(10), October 2001.

[WW94] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Operating Systems Design and Implementation*, pages 1–11, 1994.

[WY01] Joel L. Wolf and Philip S. Yu. On Balancing the Load in a Clustered Web Farm. *ACM Transactions on Internet Technology*, 1(2):231–261, November 2001.

[Yan01] Xueshi Yang. *Impulsive Self-Similar Processes with Applications in Broadband Communication System Modeling*. PhD thesis, Electrical and Computer Engineering, Drexel University, December 2001.

[YCE+97] Chad Yoshikawa, Brent Chun, Paul Eastham, Amin Vahdat, Thomas Anderson, and David Culler. Using Smart Clients to Build Scalable Services. In *Proceedings of the 1997 USENIX Annual Technical Conference*, pages 105–117, Anaheim, CA, January, 1997.

[Zeu03] Zeus Web Server. http://www.zeus.com, 2003.

[ZFJ97] Lixia Zhang, Sally Floyd, and Van Jacobson. Adaptive Web Cache. In *Proceedings of the* 2^{nd} *Web Cache workshop*, 1997.

[ZS03] G. Zussman and A. Segall. Energy Efficient Routing in Ad Hoc Disaster Recovery Networks. *Ad Hoc Networks*, 1(4):405–421, November 2003.