Techniques for Hosting Mobile Web Services on Resource Constrained Devices

by

Muhammad Asif, M.A.Sc (ECE), B.Sc. (EE)

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Carleton University

Ottawa, Ontario

Abstract

This thesis concerns hosting web services (WSs) on mobile devices. A number of challenges need to be addressed for hosting web services on such resource constrained devices. These include handling the diversity in the hardware configurations and operating systems for these devices, the execution of resource demanding web service applications and complex WS standards. A multi-dimensional approach has been proposed in this dissertation to address these challenges.

A web service execution environment (WSEE) that uses lightweight components is devised for hosting web services on mobile devices. The hosted web services can be accessed by multiple WS clients and can support a basic set of WS standards such as SOAP and XML Signature. To support more computationally complex and resource demanding WS standards such as the standards for security and transaction management, a partitioned WSEE is proposed.

Design time and runtime WS partitioning techniques are proposed to handle complex and resource demanding WS applications. The design time techniques use two graph-based algorithms for WS partitioning: Maximum Offloading Minimum Cost (MOMC) and Cluster based Application Partitioning (CAP). For the runtime WS partitioning technique, multiple execution plans, each of which corresponds to a specific partitioning of the system are determined first and then an appropriate execution plan is selected at runtime by using information on the current system load.

The effectiveness of these proposed techniques is investigated through a system prototype (using sample web services) as well as simulation (using randomly generated application graphs). The experimental results demonstrate that the partitioned systems achieved with the proposed techniques outperform the un-partitioned systems and the partitioned systems using the existing techniques. For fixed load scenarios, the design time WS partitioning techniques are observed to perform the best for a large number of WS clients. Among the design time techniques, MOMC shows the best performance for small to medium size applications whereas CAP exhibits its effectiveness for large sized applications. For systems with a large variability in the number of active WS clients and the workload parameters investigated, the run time partitioning technique outperforms the other techniques.

Acknowledgements

First and foremost, I would like to express my sincere thanks to my thesis advisor Prof. Shikharesh Majumdar for his continuous support, encouragement and help throughout the thesis. I fully appreciate your consistent enthusiasm and patience. I am thankful to Alcatel-Lucent, Natural Sciences and Engineering Research Council of Canada, Government of Ontario and Carleton University for providing financial support for this research. I am grateful to all reviewers whose comments helped me to shape this thesis.

I am thankful to my wife, Saima Arooj, who has always been understanding and supportive. Thanks for always being there for me and pushing me to reach this milestone. Thanks to my two daughters, Sana and Sara, my joy and my source of strength. Finally, I am close to finish my school and you two are the only one in my family who will go to school for next few years.

I dedicate this thesis to my father (who always encouraged me to go for the highest degree) and my mother who prayed for my thesis till she left for another world. I am forever indebted to my father and my mother for their support and prayers. I am also thankful to my sister who always asked for my progress and always wished me a good luck. In the end, I would to thank my colleague and friend Imran Ahmad who encouraged me in writing this thesis by setting milestones for every month in 2011. That really helped me in completing this document.

Table of Contents

Abstract ii
Acknowledgements iv
Table of Contents v
List of Tables ix
List of Figures x
List of Appendices xvii
List of Symbols xviii
List of Abbreviations and Acronyms xix
Chapter 1: Introduction 1
1.1 Overview 1
1.2 Motivations and Challenges 2
1.2.1 Applications 2
1.2.2 Challenges 4
1.3 Thesis Objectives 6
1.4 Thesis Contributions 7
1.5 Thesis Scope 9
1.6 Publications 9
1.7 Thesis Outline 11
Chapter 2: Background and Related Work 12
2.1 Mobile Web Services 12
2.1.1 Accessing Web Services on Mobile Devices (As a WS Client) 12
2.1.2 Providing Web Services from Mobile Devices (As a WS Provider) 16
2.1.3 Offloading the Entire WS Application to a Peer/Remote Node 19
2.2 Application Partitioning 21
2.2.1 Types of Application Partitioning 22
2.2.2 Application Partitioning Frameworks 29
2.2.3 Graph Theory based Approaches for Application Partitioning 30
2.3 Design Time versus Runtime WS Application Partitioning 33
Chapter 3: WS Execution Environments 36
3.1 Overview 36
3.2 Lightweight WSEE 37
3.2.1 Transport Layer 38
3.2.2 Service Layer 42
3.2.3 WS Standards Layer 48

3.3 Distributed SOAP Engine Based Partitioned WSEE 49
3.3.1 Motivation for the Proposed WSEE Partitioning Technique 50
3.3.2 Configurable Partitioning Scheme for Execution of WSManager Tasks 51
3.3.3 System Overview and Design 54
3.4 Experimental Analysis..58
3.4.1 Sample Web Services...58
3.4.2 Workload and System Parameters..61
3.4.3 Performance Metrics 61
3.4.4 Setup...62
3.4.5 Experimental Results..63
3.5 Summary..75
Chapter 4: WS Partitioning Frameworks 77
4.1 Overview 77
4.2 Mobile WS Partitioning Frameworks...80
4.2.1 Intermediate Node based Framework 80
4.2.2 Backend Node based Framework...82
4.2.3 Forwarding Node based Framework...84
4.3 System Design and Prototype Implementation 85
4.3.1 Web Service Execution Environment 87
4.3.2 Partition Coordination Subsystem...87
4.4 Sample Web Services and Experimental Setup.....................................89
4.4.1 Sample Web Services...89
4.5 Experimental Setup...92
4.5.1 Workload and System Parameters..93
4.5.2 Performance Metrics 94
4.6 Experimental Results..94
4.6.1 Performance Comparison of the Three Frameworks using Image WS 95
4.6.2 Performance Comparison of the Three Frameworks using Tracking WS 98
4.6.3 Performance Comparison of the Three Frameworks using NavigateMe WS 99
4.6.4 Performance Comparison of the Three Frameworks using π Calculator WS..101
4.7 Summary..119
Chapter 5: Design Time WS Partitioning 122

5.1 Overview 122
5.2 Design Time WS Application Partitioning Guidelines 124
5.3 Graph Terminologies 125
5.3.1 Graph 126
5.3.2 Edge Weight (W_E): 126
5.3.3 Vertex Weight (W_V): 127
5.3.4 Source Vertex 127
5.3.5 Vertex Distance 127
5.3.6 Graph Size (D_{max}) 127
5.3.7 Edge Cut 128
5.3.8 Degree of Benefit (β) 128
5.3.9 Beneficial Cut 128
5.4 Maximum Offloading Minimum Cost (MOMC) Algorithm 128
5.4.1 Assumptions 129
5.4.2 Objective Function 129
5.4.3 Algorithm Steps 130
5.5 Clustering based Application Partitioning (CAP) 134
5.5.1 Objective Function 135
5.5.2 Assumptions 135
5.5.3 Algorithm Steps 136
5.6 Using Remote Computing Node Information 141
5.7 Experimental Analysis using a System Prototype 142
5.7.1 Partitioning Techniques Used 142
5.7.2 Sample WS Application – Tracking WS 143
5.7.1 Workload and System Parameters 149
5.7.2 Performance Metrics 150
5.7.3 Experimental Setup 150
5.7.4 Performance Results 151
5.8 Experimental Analysis using a simulator 161
5.8.1 Random Graph Generator (RGG) 162
5.8.2 Simulator 167
5.8.3 Workload and system Parameters 170
5.8.4 Performance Metrics 171
5.8.5 Experimental Results 171
5.9 Summary 183
Chapter 6: Run Time WS Partitioning 186

6.1 Overview 186
6.2 Generating Execution Plans for a WS 188
6.2.1 Input Parameters..190
6.2.2 Upper Bound for Remote Partition of an Execution Plan................190
6.2.3 Objective Function 191
6.2.4 Algorithm Steps..193
6.3 Runtime Middleware for WS Partitioning...195
6.3.1 Web Service Execution Environment 196
6.3.2 Runtime WS Partitioning Engine (Runtime Engine) 198
6.3.3 Relationship between System Load Information and Execution Plans 199
6.3.4 Selecting an Execution Plan..200
6.3.5 Components of Runtime Partitioning Engine 202
6.3.6 Runtime Engine..202
6.3.7 Partition Manager...202
6.3.8 Load Monitor..202
6.3.9 Device Profile...203
6.4 Approach used for Performance analysis 203
6.4.1 Simulator 203
6.5 Performance Analysis...204
6.5.1 Input Parameters...204
6.5.2 Performance Metrics 205
6.5.3 Experimental Results...206
6.6 Summary...216
Chapter 7: Conclusions 218
7.1 Summary...218
7.1.1 Lightweight and Partitioned Web Service Execution Environments 218
7.1.2 WS Partitioning Frameworks...219
7.1.3 Design Time WS Partitioning Techniques...220
7.1.4 Runtime WS Partitioning Technique 221
7.2 Directions for Future Research..222
References 224

List of Tables

Table 3-1: Variation of Sample web service....64

Table 4-1: Workload and system parameters for performance comparison of the three frameworks using Image WS....96

Table 4-2: Workload and system parameters for performance comparison of the three frameworks using π Calculator WS....102

Table 4-3: Mean response time (msec) for requests and the mean CPU time (msec) for overheads....104

Table 5-1: Mean values/Value Range and default mean values used for workload and system parameters....171

Table 6-1: Input Parameters and their default value....205

List of Figures

Figure 2-1: Software components of Shopper-Kiosk application example [Mcf03]...18

Figure 2-2: Application partitioning architecture proposed by Chandra *et al.* [Cha02] 28

Figure 3-1: Proposed Architecture for Lightweight WSEE...39

Figure 3-2: Sequence diagram capturing interactions between WSManager and the components in the service layer and components of other layers...............................43

Figure 3-3: XML grammar for defining WS Mapping...47

Figure 3-4: Tasks of a SOAP engine (a) performed as a single SOAP engine (b) performed as a distributed SOAP engine...51

Figure 3-5: A sample partitioning scheme...53

Figure 3-6: A partitioned WSEE based on a distributed SOAP engine.......................55

Figure 3-7: Effect of the number of concurrent clients (C) on the response time (R) using different sample web services for lightweight WSEE 65

Figure 3-8: Effect of the number of concurrent clients (C) on the response time (R) using different sample web services for partitioned WSEE 66

Figure 3-9: Performance comparison of the lightweight WSEE and the partitioned WSEE using Image WS with average image size of 100 K, Tracking WS and π Calculator WS using complexity level of 50K 67

Figure 3-10: Scalability of partitioned WSEE when HTTP is used as a transport mechanism for exchanging SOAP messages between the intermediate node and the mobile node..69

Figure 3-11: Scalability of partitioned WSEE when TCP socket is used as a transport mechanism for exchanging SOAP messages between the intermediate node and the mobile node......70

Figure 3-12: Effect of using additional WS standards when Tracking WS is invoked using the two versions of WSEE......72

Figure 3-13: Effect of device processor speed on performance of Image WS with the lightweight WSEE and the partitioned WSEE......73

Figure 3-14: Effect of device processor speed on performance of Tracking WS with the lightweight WSEE and the partitioned WSEE......74

Figure 3-15: Effect of device processor speed on performance of π Calculator WS with the lightweight WSEE and the partitioned WSEE......74

Figure 4-1: The intermediate framework for WS partitioning......81

Figure 4-2: The backend framework for WS partitioning......83

Figure 4-3: The forwarding framework for WS partitioning......85

Figure 4-4: High level architecture of the system for hosting of partitioned mobile web services......86

Figure 4-5: Internal details of the Partition Coordinating Subsystem......88

Figure 4-6: Performance comparison of the three partitioning frameworks when the mobile device is operated at 624 MHz and (a) image size = 800x600 (b) image size = 640x480......97

Figure 4-7: Performance comparison of the three partitioning frameworks when the mobile device is operated at 624 MHz and Tracking WS is invoked......98

Figure 4-8: Performance comparison of the three partitioning frameworks when the mobile device is operated at 624 MHz and NavigateMe WS is invoked for (a) a 20x20 grid map (b) a 40x40 grid map 100

Figure 4-9: Response time for invoking the sample π Calculator WS with mobile device operated at 624 MHz and using an intermediate framework 106

Figure 4-10: Response time for invoking the sample π Calculator WS with mobile device operated at 624 MHz and using a backend framework 107

Figure 4-11: Response time for invoking the sample π Calculator WS with mobile device operated at 624 MHz and using a forwarding framework............................. 107

Figure 4-12: Performance comparison of the three partitioning frameworks when the mobile device is operated at 624 MHz and the sample π calculator WS (with N = 100000) is invoked by one WS client. 109

Figure 4-13: Performance comparison of the three partitioning frameworks when the mobile device is operated at 624 MHz and the sample π Calculator WS (with N = 1000) is invoked by a) six concurrent WS clients b) twelve concurrent WS clients. 111

Figure 4-14: Performance comparison of the three partitioning frameworks when the mobile device is operated at 624 MHz and the sample π WS (with N = 100000) is invoked by a) six concurrent WS clients b) twelve concurrent WS clients............... 113

Figure 4-15: The effect of the size of data exchanged between WS partitions on the performance of the three frameworks when the sample π Calculator WS with O = 50% is invoked by twelve concurrent WS clients (a) with N = 1000 (b) with N = 100000... 114

Figure 4-16: Effect of using WS security standard on the relative performance of the three partitioning frameworks when the mobile device is operated at 624 MHz and the sample π Calculator WS with O = 50% is invoked by twelve concurrent WS clients 116

Figure 4-17: Effect of CPU speed of the mobile device on relative performance of the three partitioning frameworks when the π Calculator WS (with N = 1000000) is invoked by a) one WS client b) twelve WS clients 118

Figure 5-1: The MOMC algorithm for mobile WS partitioning...............................131

Figure 5-2: The CAP algorithm for mobile application partitioning.........................137

Figure 5-3: Algorithm for computing a vertex with the maximum degree of benefit from a set of vertices..138

Figure 5-4: Partitioning of Tracking WS1 achieved with the three techniques: OEA, MinCut, MOMC and CAP...146

Figure 5-5: Partitioning of Tracking WS2 achieved with the three techniques: OEA, MinCut, MOMC and CAP...147

Figure 5-6: Partitioning of Tracking WS3 achieved with the four techniques: OEA, MinCut, MOMC and CAP...149

Figure 5-7: Performance of the partitioned systems achieved using different techniques (NPC, OEA, MinCut, MOMC and CAP) for Tracking WS1 153

Figure 5-8: Performance of the partitioned systems using different techniques (NPC, OEA, MinCut, MOMC and CAP) for Tracking WS2 155

Figure 5-9: Performance of the partitioned systems achieved using different techniques (NPC, OEA, MinCut, MOMC and CAP) Tracking WS3.......................155

Figure 5-10: Performance comparison of the partitioned systems achieved with NPC, OEA, MinCut and MOMC and invoked by multiple WS clients for Tracking WS1 157

Figure 5-11: Performance comparison of the partitioned systems achieved with NPC, OEA, MinCut and MOMC and invoked by multiple WS clients for Tracking WS2 157

Figure 5-12: Performance comparison of the partitioned systems achieved with NPC, OEA, MinCut and MOMC and invoked by multiple WS clients for Tracking WS3 158

Figure 5-13: Performance of the partitioned systems achieved with NPC, OEA, MinCut and MOMC for Tracking WS1 when mobile device is operated at CPU speed of (a) 624 MHz (b) 208 MHz........160

Figure 5-14: Sample directed graphs based on (a) Linear Template (b) Binary Template (c) Binary Meshed Template 163

Figure 5-15: Internal details of graph generating tool 166

Figure 5-16: Execution of application (a) on a single machine (b) on two machines169

Figure 5-17: Comparison of effectiveness of MOMC and CAP for different graph sizes........174

Figure 5-18: Comparison of MOMC and CAP with OEA and NP (no partitioning) 176

Figure 5-19: Comparison of efficacy of different partitioning techniques with the MinCut algorithm........177

Figure 5-20: Effect of the mean vertex weight on the mean execution time of the un-partitioned system and partitioned systems using the MOMC algorithm and the CAP algorithm........178

Figure 5-21: Effect of the mean edge weight on the mean execution time of the un-partitioned system and partitioned systems using the MOMC algorithm and the CAP algorithm........180

Figure 5-22: Effect of the mean variability factor on the mean execution time of the un-partitioned system and partitioned systems using the MOMC algorithm and the CAP algorithm 181

Figure 5-23: Effect of the speed up factor on the mean execution time of the un-partitioned system and partitioned systems using the MOMC algorithm and the CAP algorithm 182

Figure 5-24: Effect of using remote node information in computing partitions with the MOMC algorithm and the CAP algorithm 183

Figure 6-1: A set of sample execution plans 189

Figure 6-2: The runtime partitioning algorithm for achieving multiple partition plans 194

Figure 6-3: Architecture of the proposed middleware for runtime WS partitioning . 197

Figure 6-4: Comparison of performance of partitioned systems for various execution plans, C and for a graph size = 4 207

Figure 6-5: Comparison of performance of partitioned system when one execution plan is used at a time for graph size = 8 208

Figure 6-6: Comparison of performance of partitioned system when one execution plan is used at a time for graph size = 12 209

Figure 6-7: Performance comparison of the partitioned systems achieved with the three techniques using a fixed number of clients with a zero think time 210

Figure 6-8: Performance comparison of the partitioned systems achieved with the three techniques using fixed number of clients with mean think time = 1000 time units 212

Figure 6-9: Performance comparison of the three WS partitioning techniques with varying mean client think time and a) C = 10 and b) C = 20....................................213

Figure 6-10: Performance comparison of the three techniques using variable number of clients...215

List of Appendices

Appendix A: WS Partitioning Frameworks ……………………………….. 235

Appendix B: Profiling Output of a Few Sample Applications ……………. 239

Appendix C: Real Time Profiling for Device Profile Index ………………..241

List of Symbols

G	Graph
V	Set of vertices of a graph
E	Set of edges of a graph
W_E	Edge weight
W_V	vertex weight
s	Source vertex
S	Speed up factor
D	Vertex distance
D_{max}	Maximum vertex distance (graph size)
C	Number of active WS clients
C_{max}	Number of maximum active WS clients
R	Mean response time
τ	Mean execution time
Ƥ	Percentage improvement in execution time
ξ	Scalability metric of WSEE
β	Degree of benefit of a partiton
ω	Device speed
Δ	Variability factor
ρ	system load
χ	device profile index
N_E	Number of execution plans

List of Abbreviations and Acronyms

AMD	Advanced Micro Devices
API	Application Programming Interface
ARM	Advanced RISC Machine
CAP	Clustering Application Partitioning
CDC	Connected Device Configuration
CF (.NET)	Compact Framework
CLDC	Connected Limited Device Configuration
COM	Component Object Model
CPU	Central Processing Unit
DHCP	Dynamic Host Control Protocol
DOM	Document Object Model
GB	Giga (2^{30} = 1073741824) Bytes
GHz	Giga (10^{9}) Hertz
GPS	Global Positioning System
HTTP	Hyper Text Transfer Protocol
IAPPGA	Internet Accessible Program Packet for Graph Algorithms
J2ME	Java 2 Micro Edition
JCP	Java Community Process
JSR	Java Specification Request
JVM	Java Virtual Machine
KB	Kilo (2^{10} =1024) Bytes

MB	Mega (2^{20} = 1048576) Bytes
MHz	Mega (10^6) Hertz
MIDP	Mobile Information Device Profile
MinCut	Minimum Cut
MIPS	Microprocessor without Interlocked Pipeline Stages
MLRB	MultiLevel Recursive Bisection
MOMC	Maximum Offloading Minimum Cost
MSB	Multilevel Spectral Bisection
NP/NPC	No Partitoning / No Partitioning Case
OEA	Offloading Entire Application
OS	Operating System
PDA	Personal Digital Assistant
RAM	Random Access Memory
ROM	Read Only Memory
RPC	Remote Procedure Call
SH	SuperH
SMS	Short Message Service
SOAP	Simple Object Access Protocol
TCP	Transport Control Protocol
UDDI	Universal Description, Discovery, Integration
WML	Wirless Markup Language
WSA	Web Service APIs

WSDL	Web Service Description Language
WSEE	Web Service Execution Environment
WML	Wireless Markup Language
WNP	Wireless Network Provider
XML	eXtensible Markup Language

Chapter 1: Introduction

This chapter presents an overview of hosting web services on resource constrained handheld devices. It describes the motivations behind hosting of web services on handheld devices and enumerates different challenges that are encountered while hosting such services. Later, the scope and goals of this thesis are presented. At the end of this chapter, summary of the proposed contributions is presented.

1.1 Overview

A web service (WS) is a software component that can be accessed over the internet using standard protocols and well defined interfaces [Ws04]. Web services are accessed using the Simple Object Access Protocol (SOAP) [Soa08] and the Hyper Text Transfer Protocol (HTTP). WS interfaces are described in a well-defined format using the Web Service Description Language (WSDL) [Wsd01]. Normally, service providers publish their web services to a registry such as the Universal Description, Discovery, Integration (UDDI) registry [Udd05]. WS clients can search for web services by querying the UDDI registry. A WS requester or client is defined as an entity that consumes the WS. A WS provider is defined as an entity that provides access to software applications as web services.

There are a number of WS execution environments (WSEE) available for providing access to web services by hosting them on desktop machines that are connected to wired networks. Mobile devices such as smart phones, tablets, netbooks and Global Positioning System (GPS) based devices have resource constraints that make it difficult to host web services on them. This thesis discusses and investigates different techniques for hosting web services on such devices.

1.2 Motivations and Challenges

Web services are getting popular in the domain of business to business electronic commerce and in automating information exchange between business processes because of the interoperability they provide in a distributed heterogeneous environment. Most existing systems use web services that are hosted in fixed infrastructures. Hosting of web services on wireless handheld mobile devices is a relatively new concept. Researchers have discussed many interesting applications of hosting services on mobile devices.

1.2.1 Applications

There are many interesting applications where hosting of web services on handheld mobile devices is useful. A few examples of such applications are described next.

- A shipping company can provide tracking of the shipping items in a real time. The tracking is possible if the vehicle carrying the shipping items is equipped with a mobile device, a GPS receiver and a tracking WS hosted on the mobile device. The universal resource identifier of the hosted WS can be provided to the customer. The customer can directly access the hosted WS and track the shipping items in real time and estimate their delivery time.
- A supply chain management system is a network of different companies or departments for producing, handling and/or distributing a specific product to

the consumer. A person running a small business and using a tablet device (such as iPad or Galaxy Tab) or a netbook in the field can be a part of a supply chain system. The services offered by such a person in the field can be available through web services hosted on his/her device. The reason for hosting web services on his/her device is that the data to be used in a WS needs to be the most recent. Since s/he is always updating the data while working in the field, it makes sense to host web services on his/her device.

- Tracking skilled persons such as doctors in an emergency situation is another example. For such application, a WS is assumed to be hosted on his/her smart phone equipped with a tracking sensor (e.g. a GPS receiver). The WS can be invoked at a nursing station in the hospital and the exact location of the doctor can be determined.
- A smart GPS based device in a vehicle can host a traffic information service for peer GPS based devices in other vehicles to access. Any peer GPS based device in another vehicle can access such a service (on multiple devices) to collect information about traffic congestion, road blocks or accidents. Based on the collected information, the GPS based device can compute an alternate route for the driver.
- Use of mobile web services for study of animals' life from remote sites is another interesting application. In Australia, a biodefence project has been initiated to study the health of animals, their migration habits and the effect of environmental changes on their life. Mobile web services hosted on mobile devices that are either attached or inserted into an animal body can be used by researchers all across the world for studying the life in remote forests.

- Another interesting application is a mobile web service that is based on a mobile device and wireless sensors attached to a patient's body. Such a service can combine results of different sensors and perform computations on the mobile host to aggregate results into a summary before sending the response back to the requester. The WS requester can be a hospital or an independent agency that may be hired to monitor a person's health in real time. Such application scenarios are investigated in [Luq08] and [Aij10].
- Publically available wireless devices are often resource constrained as the mobile devices discussed earlier. For example, a company can provide live contents (e.g. text and/or images) as mobile web services from public places using such wireless devices. These web services can be accessed by news agencies or security persons. This type of web services can help in reducing cost of sending resources (e.g. camera, vehicle and persons) to a place where an event of public interest is happening.
- In pervasive computing, interaction among different devices can be managed conveniently if devices are offering services using WS technologies. The use of WS technology for interaction of devices is useful to integrate devices from different manufacturers even if they are using different platforms and different programming languages.

There are a number of challenges in hosting web services on mobile devices. These challenges are discussed next.

1.2.2 Challenges

The key challenges of hosting web services on resource constrained devices include limited resources on such devices, diversity in hardware and operating systems used on mobile devices, non-availability of WS toolkits and weak wireless signals. Each of these is discussed in the following subsections.

1.2.2.1 Limited Resources

The WS application may require complex algorithms to achieve its goals; complex cryptographic algorithms are executed, for example, if the WS application requires an end to end security at the message level. In many cases, intensive data processing is required to provide the desired functionality. For example, image format conversion requires a significant amount of resources. Moreover, accessing or providing a WS always incurs the overhead of SOAP/XML processing. The required resources for WS applications include CPU power, memory, battery and bandwidth. Unfortunately, handheld devices are limited in terms of each of these resources and this represents a big challenge for hosting WS applications that are typically resource demanding on handheld mobile devices.

1.2.2.2 Diversity

Another challenge in the domain of handheld devices is the diversity in hardware architectures such as Advanced RISC Machine (ARM) developed by ARM Holdings, Microprocessor without Interlocked Pipeline Stages (MIPS) developed by MIPS Computer Systems and SuperH (SH) developed by Hitachi. There is also diversity in the available operating systems (OSs) for mobile devices. These include Windows Mobile, iPhone OS, Symbian, Android OS and Blackberry OS.

1.2.2.3 Non Availability of WS Toolkits

There is a number of WS toolkits and execution environments available for desktop nodes. The most popular are the Apache Axis2 [Apa08], Glassfish Project [Gla09] for a Java runtime environment and AlchemySOAP [Alc07] for C/C++ runtime environment. However, these toolkits are not suitable for mobile devices because of the following reasons. First, these toolkits are too large and resource demanding for handheld devices. Second, these toolkits are developed for such runtime environments which are not optimized or not available for mobile devices. For example, Apache Axis2 requires a Java run time environment 1.4 or higher that is only available for desktop machines. Third, limited CPU power, communication bandwidth, battery and memory that characterize handheld mobile devices are often inadequate for using such heavy weight WSEEs. The mobile devices need a light weight environment for hosting web services so that devices can have enough free resources to perform the other core functions such as making or receiving phone calls.

1.2.2.4 Weak Signals

A challenge inherited from the wireless domain is the handling of weak wireless signals that can cause connections drops and loss of packets.

It can be argued that a solution developed in the native code of the device will be the most efficient. However, writing optimized programs in low level native codes for a large variety of handheld devices is an expensive solution because of the diversity of available handheld devices. Using a WS is attractive because WS technology provides interoperability in a heterogeneous environment in which the WS client and the WS provider may be implemented using different languages and may run on diverse platforms.

1.3 Thesis Objectives

This thesis concerns a number of research problems, and thus has multiple objectives. One of the objectives is to investigate techniques and frameworks for hosting web services on resource constrained handheld devices. Another objective is to investigate application partitioning algorithms for web service applications so that parts of WS applications can be executed on remote computing nodes for lowering the resource demand on the handheld device. WS partitioning is expected to improve the overall system performance by facilitating WS provisioning with a reduced response time in comparison to the case in which the entire WS applications are run on the handheld device. The partitioned web services also aim to provide system scalability to handle multiple clients.

1.4 Thesis Contributions

The contributions of this thesis are summarized next.

- A lightweight WSEE that can handle a set of basic WS standards for hosting WS on handheld devices is devised.
- Investigating and devising a configurable partitioned WSEE based on a distributed model of a SOAP engine that uses a WS specific configurable partitioning scheme for offloading the execution of certain tasks (related to the conformation with WS standards/specifications) to a more powerful intermediate node.
    - This novel concept of partitioned WSEE is very useful for conforming to the data and the resource intensive WS standards such as WS-Security and WS-AtomicTransaction. Such resource demanding WS standards are very hard to apply for web services hosted on handheld devices because of their limited resources.
- Analyzing performance of three application partitioning frameworks (Intermediate framework, Backend framework and Forwarding framework) for hosting of web services on handheld devices. The intermediate and backend frameworks were proposed in the literature for mobile and conventional applications. This thesis

analyzes their feasibility for hosting web services on handheld devices and also proposes a new framework (the forwarding framework). Details of the three frameworks are available in chapter 4.

- Investigating and devising two design time WS partitioning algorithms: Maximum Offloading Minimum Cost (MOMC) Algorithm and Clustering Based Application Partitioning (CAP) Algorithm. The majority of the algorithms available in the literature have different objectives and they usually target large scale scientific applications. These algorithms are either very complex or based on objectives of either only minimizing the communication cost between application components or dividing the application into partitions of similar sizes so that the partitions can be run in parallel efficiently. The algorithms proposed in this thesis focus primarily on offloading part or parts of an application from a handheld mobile device to a remote computing node. The proposed algorithms can also use the characteristics (such as the processing speed) of a remote computing node when this information is available. The details of the design time WS partitioning algorithms with performance analysis are discussed in Chapter 5.
- Devising a hybrid technique for runtime WS application partitioning: The design time WS partitioning is easy to use and is observed to be effective for a wide range of application scenarios, but the partitioned systems achieved with the design time approaches are generally insensitive to the variation in system load. The proposed technique for runtime WS application partitioning combines advantages of both the design time and the runtime application partitioning. The runtime technique first applies a graph based algorithm for achieving multiple execution plans for a WS application at design time. The technique then uses a

runtime middleware system which can select an appropriate execution plan based on the system load information and then uses that execution plan for executing the application partition.

1.5 Thesis Scope

For this research, both data-intensive and compute intensive applications are considered as web services. Applications involving streaming data such as multimedia applications are not considered because web services are not typically designed for such applications.

The WS partitioning techniques proposed in this thesis focus primarily on partitioning of mobile WS applications but these techniques can be applied to partition other types of mobile applications as well. However, the proposed partitioning techniques are not devised for data partitioning or partitioning of large scientific applications.

1.6 Publications

A number of experiments have been performed for detailed analysis of the performance of the proposed techniques of hosting web services on resource constrained devices. Based on the analysis and the results of experimentation, a number of research papers have been published which are listed next.

1. M. Asif and S. Majumdar, “A Runtime Partitioning Technique for Mobile Web Services”, In the Proceedings of the International Workshop on Applications of Wireless Ad hoc and Sensor Networks (AWASN'11), Taipei, Taiwan, Sep 13-16, 2011.

2. M. Asif and S. Majumdar, “Partitioning frameworks for mobile web services provisioning”, International Journal of Parallel, Emergent and Distributed Systems, 1744-5760, Taylor & Francis, 2011.
3. M. Asif and S. Majumdar, “Hosting Web Services on Mobile Devices”, Book Chapter in Mobile Web 2.0: Developing and Delivering Services to Mobile Phones, Taylor & Francis Group, 2010.
4. M. Asif and S. Majumdar, “A Graph-based Algorithm for Partitioning of Mobile Web Services”, In the proceeding of 17th Annual Meeting of the IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2009), London, UK, September 2009.
5. M. Asif and S. Majumdar, “Performance Analysis of Mobile Web Service Partitioning Frameworks”, In the Proceedings of the 16th International Conference on Advanced Computing and Communications (ADCOM 2008), Chennai, India. December 2008.
6. M. Asif, S. Majumdar, R. Dragnea, “Partitioning the WS Execution Environment for Hosting Mobile Web Services”, In the Proceedings of the 2008 IEEE International Conference on Services Computing (SCC 2008), Honolulu, HI, 8-11 July 2008.
7. M. Asif, S. Majumdar, R. Dragnea, “Application Partitioning for Enhancing System Performance for Services Hosted On Wireless Devices”, In the Proceedings of the Workshop on Service Oriented Engineering and Optimization 2007, Goa, India, December 2007.
8. M. Asif, S. Majumdar, R. Dragnea, “Hosting web services on resource constrained devices”, In the Proceedings of the 2007 IEEE International conference on web services, Salt Lake City, UT, July 9-13 2007.

1.7 Thesis Outline

The outline of this thesis document is presented. Chapter 2 discusses the background of this research and the related work in the area of mobile web services, in application partitioning. Chapter 3 discusses the lightweight WSEE and the distributed SOAP engine based partitioned WSEE for hosting of web services. Chapter 4 presents an analysis of different partitioning frameworks for hosting of partitioned web services. Chapter 5 discusses design time WS application partitioning techniques and a detailed performance analysis of the techniques using application prototypes as well as simulation models. Chapter 6 introduces run time WS application partitioning technique. Chapter 7 concludes the thesis and enumerates direction for future research.

Chapter 2: Background and Related Work

This chapter covers the existing work on hosting web services on resource constrained devices, application partitioning in general and WS partitioning in particular, application partitioning frameworks and graph based algorithms for WS application partitioning.

2.1 Mobile Web Services

The term 'mobile web service' is used both for web services that are *accessed* from handheld mobile devices and for web services that are *provided* from handheld mobile devices. In the first case, mobile WS clients are run on mobile devices to create requests (SOAP messages) for accessing web services. The web services may be hosted on other mobile devices or on more powerful computers in a fixed infrastructure. The term 'mobile web service' is used only when a web service is hosted on mobile devices. In the next subsections, a critical analysis of the related work in the area of accessing web services from a mobile device and providing web services from a mobile device is presented.

2.1.1 Accessing Web Services on Mobile Devices (As a WS Client)

There is a substantial amount of work done in the area of accessing web services from a mobile device. The web services may be hosted in a fixed infrastructure or on other mobile devices. Since this is not directly related to the research work presented in this thesis, so only a high level summary of the key papers is presented. The approaches proposed in the literature can be grouped into three categories. Each category is discussed in a separate subsection.

2.1.1.1 Accessing WSs Directly on Mobile Devices

In the first category of approaches, web services are accessed directly on mobile devices. Schall *et al.* provide an analysis of several toolkits that are proposed for accessing web services from mobile devices [Sch06]. These include gSOAP, kSOAP, JSR 172 (WSA) and support of web services in the .NET compact framework. A brief summary of these toolkits is presented next.

gSOAP is a C/C++ toolkit for web services [Eng03]. The toolkit includes a Web Service Description Language (WSDL) parser that creates header files for stub/skeleton based on the WSDL of a WS. The run-time library (stdsoap2) can be used to serialize outgoing WS requests and to de-serialize incoming WS responses. eSOAP is another toolkit available for C++ programmers to access web services from embedded systems [Eso04]. A limitation of these toolkits is that it is not easy to find a C++ compiler for all the different types of mobile devices.

kSOAP [Kso03] and JSR-172 [Wsa04] are proposed for the Java Micro Edition (Java ME) platform [Jme06]. Java ME is a set of standard Java APIs defined through the Java Community Process (JCP). Java ME has two configurations, the Connected Device Configuration (CDC) and the Connected Limited Device Configuration (CLDC). CDC is a subset of Java Standard Edition (Java SE) and is designed for products with resource constraints, typically 2 MB of RAM and 2.5 MB of ROM for the Java application environment [CDC05]. In contrast to CDC, CLDC is designed for products with very limited resources; typically 128 KB to 512 KB of RAM [CLD05].

CLDC provides libraries such as the Connection Framework which are suitable for devices with a small memory footprint (not part of J2SE). CLDC has a profile, Mobile Information Device Profile (MIDP), specifically designed for cell phones to provide the user interface, network connectivity, local data storage, and application management needed by these devices.

kSOAP is an open source SOAP library for devices with Java ME support. It provides a lightweight way to access SOAP based web services. However, kSOAP cannot generate client side stubs from the WSDL of a WS. JSR-172 (WSA) is a set of web services APIs (WSA) for Java ME available in Sun's wireless tool kit (WTK). In comparison to kSOAP, client side stubs can be generated from WSDL of a WS using WSA. This accelerates the development process of WS clients. WSA for J2ME has similar capabilities (e.g., a stub generator) as gSOAP's client runtime. However, it is important to note that all these toolkits are only suitable for *consumption* of web services on mobile devices. These toolkits do not provide APIs that can be used for *hosting* web services on mobile devices [Eng03, Kso03].

.NET Compact Framework (CF) [Net05] is a subset of Microsoft's .NET framework. With .NET CF, web services can be accessed in a synchronous or asynchronous manner. The steps of invoking web services using .NET CF and standard .NET framework are similar.

Rendon *et al.* discuss the use of a J2ME Web Services API (WSA) for devices with Java ME support and Short Message Service (SMS) for devises without the support of Java ME. The SMS based approach has very limited applications because it requires an intermediate node to translate text messages to SOAP requests for WS invocation and to convert WS response to an SMS text message for a mobile device. This approach is also not user friendly because a strict format needs to be followed for sending text messages from mobile devices.

2.1.1.2 Accessing WSs using Mobile Agents

In the second category, web services are accessed using mobile agents. A *mobile agent* is an autonomous entity that gathers information or accomplishes tasks without human interaction and can also self-migrate in a heterogeneous network [Bra05]. Cheng *et al.* propose a framework in which web services are accessed by using mobile agents [Che02]. The proposed framework requires different mobile agents for different categories of web services that make it unattractive from an implementation's point of view. Another framework based on mobile agents is proposed by Adacal *et al.* [Ada06]. In this framework, mobile web service (MWS) agents reside in the system of the wireless network provider (WNP). WNP creates a mobile WS agent as soon a new WS request is received from a mobile client. The newly created mobile WS agent is responsible for the complete invocation process for the WS. The invocation process of a WS includes discovery of web services, invocation of a work flow engine to execute web services according to a work flow document and in the end translating a WS response according to the device configuration. The proposed solution uses mobile WS agents that reside in a WNP environment.

2.1.1.3 Accessing WSs using Proxy Nodes

In the third category, web services are accessed through proxy nodes. A number of architectures and frameworks based on the use of a proxy node have been proposed. Park *et al.* propose a middleware (running on a proxy node) that converts the response of a web service to a wireless markup language (WML) or other formats that are suitable for mobile devices [Par06]. Similarly, a WS request sent in WML or a device specific format is converted to a SOAP message as required by the WS. The middleware serves merely as a translator. Steele *et al.* has gone a step ahead and suggest a new architecture for discovery and invocation of web services [Ste05]. The proposed architecture has a component available on a proxy node that uses WSDL to automatically generate a user interface (UI) for the access of a WS. The user interface is created on the fly based on the requirements of a WS and configuration of the mobile device. The frameworks proposed earlier are not more than intelligent translators. Lee *et al.* propose to use proxy or intermediary nodes for more than service discovery and XML processing [Lee06]. They also suggest providing authentication, auditing and management of clients in addition to XML processing and service discovery at the proxy. MacDonald and Mitchell have a different view for authentication and auditing of web services [Mac05]. They propose to use the WNP (which is a trusted third party) for authentication and handling of payment for the usage of web services. According to them, WNPs are the best choice because they already have trust relationships with mobile clients. They can ensure security between mobile clients and the service provider. Enhancing network providers for such tasks are expected to minimize service latency as well.

2.1.2 Providing Web Services from Mobile Devices (As a WS Provider)

Hosting of web services on mobile devices has started receiving attention recently. There are two types of approaches used in the literature for providing web services from mobile devices and are described in the following subsections.

2.1.2.1 Hosting Entire WS on a Mobile Device

A representative set of works that has been proposed for hosting web services on a handheld mobile device is discussed next.

The earliest research is done at IBM [Mcf03]. They develop a prototype for a shopper-kiosk application where a shopper comes to a store with his/her handheld device, purchases a few things and uses its wallet services to pay bills. On arriving at the store, the user's device connects to the store's network, determines the services offered by the store, presents its services to the store, securely uses and executes the services, and deregisters its services and itself from the store when the user leaves the store. The main components of the example application are shown in Figure 2-1. These include a mobile device, a wireless access point and a Dynamic Host Control Protocol (DHCP) server, a Universal Description, Discovery, Integration (UDDI) repository (registry), a servlet engine and a kiosk station. The access point provides connectivity to the mobile device and helps the mobile device to locate the store's services and the UDDI repository.

The servlet engine generates web pages to drive the interaction between the kiosk and the user. More details can be found in [Mcf03]. This research has successfully addressed a number of key issues in the context of hosting web services on mobile devices. For example, it addresses discovery of services by using the local UDDI registry in the shop. The shopper has to register its wallet service as soon he/she walks in the shop. Upon user's entry into the shop, his/her device is issued a unique IP address and is assigned an easy-to-remember name. This name is used to identify the user's device at the time of payment without entering the IP or the MAC address of the device. This demo application is tested with a personal digital assistant (PDA) equipped with a 100 MHz CPU and a Bluetooth wristwatch that is equipped with an 18MHz CPU. Security and trust between the shopper and shop system are established by using X509 security certificates. The shop system has to present its valid security certificate to the wallet service (on the device) before fetching the financial information of the user. The system uses Bluetooth as a communication link between the mobile device and the shop system. Use of Bluetooth makes its application very limited because of the short range of Bluetooth connection. Apart from the Bluetooth communication limitations, this research has opened doors for other interesting applications of hosting of WSs on mobile devices. For example, a doctor or a specialist can expose his/her special skills as a WS and can register it while boarding a flight/ship or entering a restaurant. In a situation of need or emergency, such persons can be located through their exposed WS very easily.

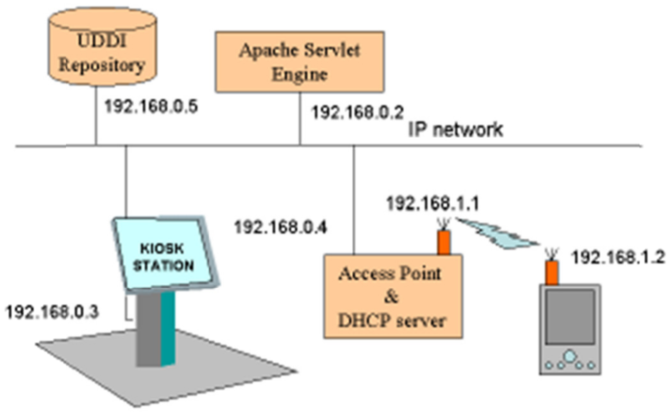


Figure 2-1: Software components of Shopper-Kiosk application example [Mcf03]

In another effort, Srirama *et al.* has implemented a prototype for hosting of web services for Sony Ericsson P800 smart phone [Sri06]. The sample WS prototype accesses a Bluetooth Global Positioning System (GPS) receiver and a file system. A WS handler is developed on top of a web server to handle WS requests. The authors use PersonalJava instead of J2ME because of the availability of a richer application environment and better interaction with the Sony Ericson P800 smart phone. Note that PersonalJava is a Java software environment to execute Java applications on handheld and mobile devices. PersonalJava has been superseded by the J2ME CDC [CDC05] and CLDC [CLD05]. kSOAP [Kso03] is used for XML processing because of its effective memory footprint. This work is extended in two different domains:

1. Providing a secure communication and an access control for the mobile WS provisioning domain ([Sri07-1] and [Sri07-2]).
2. Supporting mobile web service provisioning through cloud computing is considered [Sri11]. Web service provisioning through cloud computing is based on a mediation framework that is proposed in [Sri10].

Pham *et al.* [Pha05] propose a lightweight SOAP server architecture for mobile devices and provide an implementation for the micro edition of Java (J2ME). The proposed SOAP server is useful to provide the access to web services via HTTP.

The approaches discussed earlier focus mainly on hosting of relatively simple WS applications. A few researchers have explored the usability of mobile web services in a peer to peer environment ([Aij08] and [Geh05]). Some researchers have also proposed migration of the entire WS application to a remote node. In the domain of pervasive and mobile computing, researchers have also proposed partitioning of applications and executing part or parts of them on remote computing nodes. The work in these two categories is presented next.

2.1.3 Offloading the Entire WS Application to a Peer/Remote Node

Researchers have proposed techniques for migration of WS code entirely from one node to another. The main focus of these works is the selection of the appropriate target node either using context information [Hem05, Riv07] or using information of the target node [Yea05].

Yeon *et al.* present a lightweight framework for hosting web services on mobile devices [Yea05] that focuses on the processing of the SOAP messages, the execution and migration of services, the management of context and the service directory, and the publishing and discovery of services. SOAP processing is achieved by using PocketSOAP toolkit [Fel04] which is available for only windows family of operating systems. The key feature of this work is the migration of services in case of low battery power on the mobile device or weak signal strength. In such cases, the service (code) is migrated to another node. The target node is selected using a criterion that takes into account the context of the node and its capabilities such as CPU speed and available memory. The proposed framework is tested using real devices connected by Bluetooth.

Hemmati *et al.* propose a framework, which supports the migration of application codes and its execution states [Hem05]. The target node is selected based on context information for the framework, which is collected by a context manager from the other nodes. Riva *et al.* [Riv07] go a step further by proposing a mobile service framework that continuously monitors dynamic context changes in an ad-hoc wireless network. For example, the context of a service requester is used to transfer a mobile service to a node closer to the service requester. However, the proposed framework is required to monitor the context of the WS requester as well as the context of the framework.

In addition to these works on offloading WS code to a remote computing node for execution, existing work also addresses migration of general application code from mobile devices to remote computing nodes. For example, Chen *et al.* propose an offloading framework for a Java-based environment that dynamically decides whether to execute the code locally or remotely, based on the cost of Javacode compilation, computational complexity and communication channel conditions [Che04]. The Coign project proposes a system to use a MinCut algorithm to statically partition binary applications built from Microsoft's Component Object Model (COM) components [Hun99]. Li *et al.* suggest to construct a static cost graph and to apply a partitioning scheme to statically divide an application's tasks into client and server subtasks during application design [Li01].

2.2 Application Partitioning

Application partitioning (AP) is defined as the separation of an application into different components that can be executed on different nodes. Performance improvement with application partitioning can be achieved by executing the partitions on one or more remote computing nodes. The partitions can be deployed on remote computing nodes at design time or moved to remote computing nodes at run time. The process of moving or deploying the partitions on remote computing nodes is termed offloading. Offloading a part or parts of an application from a local node to a remote computing node can have two different objectives. Based on these objectives, the offloading is classified into two types: adaptive offloading and beneficial offloading. In case of *adaptive offloading*, the resources on the local node are not sufficient to execute the complete application, so a part or parts of the application are migrated to a remote computing node. Adaptive offloading may lead to degradation in overall system performance because of communication overheads that accrue when the partitions are executed on remote nodes. *Beneficial offloading* is the migration of a part or parts of the application from a local node to a remote computing node if this migration leads to an improvement in overall system performance. For beneficial offloading, the additional overheads of executing parts of an application on a remote computing node must be less than the performance improvement expected from such offloading.

There are a number of factors that can affect application partitioning. The most commonly used partitioning criteria include processing time of the application, available memory on the local node, bandwidth requirements for data exchange with a remote application and the frequency of executions of different application components [Chu04, Til02]. If the processing speed or the available memory on a device is not sufficient to execute an application, then a part of an application can be offloaded to a nearby node. Sometimes, it is beneficial to execute a part of the application closer to the data source if a great deal of data needs to be exchanged on a wireless link. Offloading such components of an application to a node which is closer to the data source is an example of adaptive offloading. In a typical application, a few components are executed more frequently than others. Moving such components to a node with more powerful processors can potentially lead to better performance of the application. Frequency of interactions between different components is another important factor for deciding which components to move to which partition. Components that inter-communicate heavily with one another should be kept in one partition or in partitions that execute on nodes with faster communication links.

2.2.1 Types of Application Partitioning

Application partitioning can be achieved at different levels. The existing research on application partitioning can be grouped into two main classes: static and dynamic.

Static application partitioning is a separation of application components at design time. *Dynamic* application partitioning is further categorized into two subclasses: one deals with application partitioning at compile time and the other focuses on partitioning at execution time. Details of the research work in these three classes of application partitioning are presented in the following subsections.

2.2.1.1 Design Time Application Partitioning

In design time partitioning of applications, which part of the application is executed on which node is decided during application design. Typical examples of this type of partitioning are client server applications. These types of applications are partitioned into clients and servers to achieve better performance by executing the application across multiple nodes in a distributed manner [Sil04, Tan02]. Clients are lightweight programs that can be executed on users' nodes. The server usually constitutes a major part of the application and is executed on a powerful node. Intelligent design of partitioning can also reduce network traffic. This criterion is used as a key objective by many researchers in the area of mobile computing. A representative set of existing works is presented next.

Watson [Wat95] proposes to partition the data and the functionality of an application into hyperobjects. The hyperobjects are linked hierarchical objects. The objective of hyperobjects is to utilize the bandwidth efficiently by using caching, pre-fetching and data reduction techniques.

Schill *et al.* [Sch99] also propose an approach for data optimization on wireless links but they use mobile agents to achieve their goals. The application partitioning model is based on two components. One of the components runs on the mobile device and the other on a proxy node. The component placed on the proxy node represents a mobile device and all communications from the mobile device are mediated by it. The application specific components use mobile agents for performing different tasks of the application. These mobile agents migrate independently to servers (for example an email server) and communicate with servers locally. Application specific components collect data from one or more mobile agents and optimize the data according to the configuration of the device. Use of mobile agents is not very beneficial in this work except that it provides local communication with the server. The disadvantage is the requirement of mobile platforms on the remote servers. In the real world, it is not a common practice to provide support for hosting mobile agents on general environments such as web servers, email servers and WS providers.

2.2.1.2 Compile Time Application Partitioning

In the second type of partitioning, applications are partitioned at compile time. Researchers use two types of approaches. The first approach is for existing applications which are already developed without any considerations for partitioning. The second approach is proposed for such applications that use annotations or some other marking schemes in the code to guide a partitioning tool. With both approaches, applications are partitioned into multiple parts based on an input configuration provided by a user. Input configuration is usually provided through an input file [Jam05] or from a user interface [Til02]. Different approaches have used different configuration parameters such as the number of partitions, partitioning criteria and association between different objects. A summary of existing research based on the two approaches discussed earlier is presented next.

Researchers have developed tools for automatic partitioning that relieve programmers from considering application partitioning at the time of application development. Popular tools include J-Orchestra [Til02, Lio04] and Protium [You01]. J-Orchestra is a GUI-based tool that takes a regular Java program as an input along with partitioning configuration parameters. The partitioning configuration parameters provide information such as how many partitions are required and what are the constraints and rules that need to be followed. The tool then rewrites the bytecode of the input Java program based on the partitioning configuration. J-Orchestra is an elegant tool but it has a number of limitations. For example, it needs a fair amount of guidance from the user. Moreover, it is not possible to partition complex applications without having knowledge of the internal structure of the program. J-Orchestra can only partition applications that are written in Java.

Protium [You01] is another application partitioning tool that is developed to partition desktop applications for remote access. Any desktop application can be partitioned into three entities such that it can be accessed and executed from a remote machine. The three partitioned entities are Viewers, Services and Application Specific Protocols. Viewers are run on a machine or on a device near the user. Services are run on a managed environment and are accessed through the network. Services are server programs running on a remote server machine. Application Specific Protocols are used for communication between Viewers and Services.

In addition to these tools, there are some approaches that use annotations for partitioning of applications. For example, Jamwal and Iyer introduce the idea of breakable objects (BoBs) for application partitioning in Java [Jam05]. BoBs are the entities in a program that can be split easily. Java applications can be written using BoBs. At compile time, a BoB partitioning tool uses a detailed configuration file to split the program into the desired number of partitions. However, use of BoBs has a number of limitations. Programmers have to follow strict rules for developments of applications using BoBs.

Chu *et al.* [Chu04] also propose an architecture that divides an application's components into local and remote groups. The components identified as local are executed on the mobile device and those marked as remote are executed on the server side. The architecture requires a complex algorithm to be executed to identify local and remote components. The details of the algorithm can be found in [Chu04].

2.2.1.3 Run Time Application Partitioning

In run time application partitioning, programs are partitioned at execution time. Partitioning may be triggered when a device has limited resources and the resources are in use by other applications running on the device. This type of partitioning is more challenging than the previous two types because there can be a number of factors that can affect partitioning decisions. In the literature, there are two types of approaches that have been adopted to handle application partitioning at run time. First is the enhancement of the execution environment so that the execution environment itself can decide to transfer an application or part of it to a nearby node. The partitioning of applications is done by the execution environment according to the execution needs and availability of resources [Mes02]. The second approach deals with migration of one object or a complete package to a nearby computing resource by the application itself. At run time, if resources on a handheld device become limited, then the application can decide to partition itself based on the resource availability [Cha02]. The partitioning criterion in both types of approaches is dynamic and is thus more challenging. Both types of approaches have their strengths and pitfalls. Extending an execution environment for application partitioning is a more generic and a better solution. Sometimes it is not possible, however, to enhance the execution environment if it is a proprietary product. The advantage of partitioning of an application by itself is that the programmer can have more control on the partitioning criteria but it demands a subsystem for partitioning with every application and thus requires additional efforts from a programmer.

Messer *et al.* propose a heuristic approach for application partitioning through the execution environment by enhancing the execution environment itself [Mes02]. Their partitioning algorithm is based on the execution history of the application. From this execution history, frequency of interactions between different components is determined. The partitioning algorithm uses the MinCut algorithm that is proposed by Stoer and Wanger [Sto97] for separating different components of the application into partitions. Chandra *et al.* [Cha02] propose to utilize the execution environment for an application in a different manner. The proposed architecture (see Figure 2-2) is based on two guidelines which are presented next.

- A proxy server is proposed to be present between a remote server and a mobile device as shown in Figure 2-2. If an application code is sent (for execution) to a mobile device, the proxy server intercepts this code and converts it into low-level native code of the device according to the device configuration. The native code runs faster on the device. In this way, the compilation process that is performed on a proxy server is separated from the execution process that runs the native code on the mobile device. The computation intensive compilation and optimization of the application is done at the proxy server. The mobile device only executes the generated native code that it receives from the proxy server.

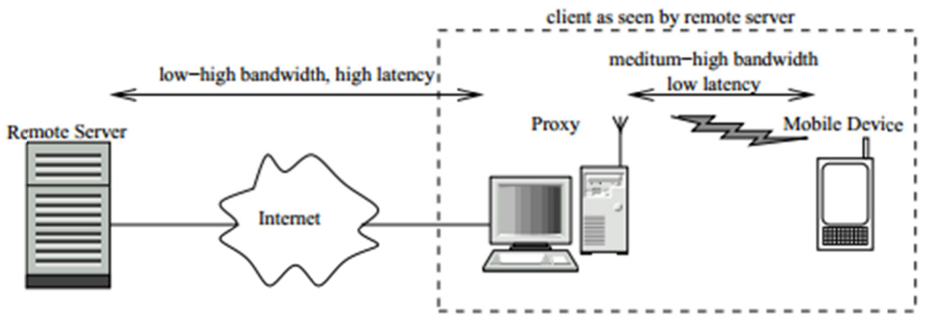


Figure 2-2: Application partitioning architecture proposed by Chandra *et al.* [Cha02]

- Partitioning of an application is proposed to be performed by the application itself. It is suggested that applications running on mobile devices partition themselves at object boundaries. Partitioning granularity depends on application type and it can vary from one object to a complete package of classes. The partitioning criterion that is used includes processing time, memory requirements and the frequency of execution of the different methods.

In another effort, Ou *et al.* [Ou07] propose an offloading middleware which provides runtime offloading services for resource constrained mobile devices. The middleware considers multiple constraints (i.e. memory, CPU and bandwidth) while applying the proposed (K+1) application partitioning algorithm.

Alshahwan *et al.* [Als11-1] extend the partitioned WSEE proposed for hosting of mobile web services in this dissertation (described in Chapter 3) for RESTful [Fie00] mobile web services. The authors have also performed an evaluation of the partitioned WSEE for SOAP based mobile web services and for RESTful mobile web services. The results of this evaluation demonstrate that the hosting of RESTful web services on mobile devices exhibits a better performance in comparison to the SOAP based mobile web services [Als11-2]. Note that the RESTful services do not support WS standards such as WS-Security which is critical when such services are used for enterprise applications.

2.2.2 Application Partitioning Frameworks

In addition to research on migration of WS code entirely to a remote computing node for execution or offloading part or parts of an application through some framework, there is a fair amount of work available in the literature that uses algorithms based on graph theory for application partitioning. A representative set of works in the area of graph partitioning is presented next.

Existing research has used two different types of partitioning frameworks for offloading partitions of an application to remote nodes. A few researchers propose to use an intermediate node for providing a mobile WS ([Cha02] and [Sri06]). The backend node based partitioning framework, first used by Messer *et al.* [Mes02], has never been used in the context of web services. However, it is used by a number of researchers for offloading a part or parts of mobile applications [Ou07] and also for applications in the area of pervasive computing [Mes02]. As already mentioned, Ou *et al.* [Ou07] propose an offloading middleware for mobile applications and WS clients.

The backend node based framework is a popular choice for migration of WS code from one node to the other. For example, Hemmati *et al.* [Hem05] propose a framework, which supports the migration of application codes and its execution states. Riva *et al.* [Riv07] go a step further by proposing a mobile service framework that continuously monitors dynamic context changes in an ad-hoc network. Kim *et al.* [Kim07] have proposed a lightweight framework for hosting web services that has a capability to migrate the web service code to peer mobile devices in case of low battery or weak signal strength. The authors propose a cost based algorithm for selecting the target node for WS code migration.

Han *et al.* [Han06] discuss interesting design paradigms for mobile computing. These includes downloading code on demand, remotely accessing resources, migrating a task to a remote environment through mobile agents and accessing mobile components remotely. But the approach discussed is not supported by any performance evaluation and it is thus hard to say which design paradigm is the best in terms of performance.

This thesis uses the intermediate node based framework and the backend node based framework for the execution of partitioned web services. Note that these two frameworks have never been used for WS application partitioning. This thesis also proposes a new WS partitioning framework that is based on a forwarding node. These three frameworks are compared by evaluating their performance by using a prototyping and measurement approach.

2.2.3 Graph Theory based Approaches for Application Partitioning

For graph theory based application partitioning, the application to be partitioned is modeled as a graph. A graph is represented by a set of vertices V and a set of edges E that connect the vertices of the graph. Graph partitioning is an NP-complete problem [Kur99]. A large variety of algorithms has been proposed for graph partitioning. The graph partitioning problem addresses the partitioning of the vertices of a graph in *p* roughly equal partitions such that the number of edges connecting the vertices in different partitions is minimized. Different types of algorithms have been proposed for graph partitioning. In this section, a representative set of graph-based algorithms that have been used for application partitioning is presented.

Although the Ford-Fulkerson algorithm [Ful56] is proposed for maximum flow problems, it is used by many researchers as a basic tool for finding the minimum cut (MinCut) to divide a graph into two partitions such that the network flow can be maximized. The Ford-Fulkerson algorithm is not easy to implement and it only partitions a graph into two parts. Recently, simpler algorithms have been proposed for finding a MinCut in a graph [Sto97]. A summary of the work based on the minimum cut algorithm is presented next.

The Coign [Hen99] project proposes a system that uses the MinCut algorithm to statically partition binary applications built from Microsoft's Component Object Model (COM) components. As already mentioned, Messer *et al.* [Mes02] also use the simple MinCut algorithm [Sto97] for partitioning of an application graph. This partitioning approach presented by Messer *et al.* is for run time partitioning and it has the limitation that it requires a modified Java execution environment instead of a standard Java Virtual Machine (JVM).

A number of algorithms are proposed to divide a graph into k partitions (also known as k-way partitioning). These include Spectral methods [Hen93], Multilevel Spectral Bisection (MSB) methods [Bar93] and Geometric methods ([Nou86] and [Mil93]). These algorithms are characterized either by very high computational complexity (the Spectral methods and the MSB methods) or poor output partitions (the Geometric methods) [Mil03]. Another popular approach of achieving k-way partitioning is recursive bisection. In recursive bisection, a 2-way partitioning of graph G is obtained first, and then a 2-way partitioning of each resulting partition is obtained recursively. After $\log_2 k$ phases, the graph G is partitioned into k partitions. Thus, the problem of performing a *k*-way partitioning is reduced to that of performing a sequence of bisections.

Researchers improve the recursive bisection algorithms by introducing coarsening and un-coarsening phases [Hen93, Kar99]. These are called multilevel recursive bisection (MLRB) algorithms that are characterized by moderate computational complexity and provide graph partitions that meet the partitioning objectives. The basic structure of a multilevel bisection algorithm is quite simple. A graph G is first coarsened down to a small number of vertices, a bisection of this much smaller graph is computed, and then this partitioning is projected back to the original graph by periodically refining the partitioning. Since the finer graph has more degrees of freedom, such refinements decrease weight of the edge cut. The edge cut is a set of edges that separate a graph into two parts. The complexity of the MLRB for producing a *k*-way partitioning of a graph $G = (V, E)$, is $O(|E| \log k)$ [Kar99]. The experiments presented in [Kar99] show that MLRB produces partitions that are significantly better and is an order of magnitude faster in execution in comparison to the partitions achieved with the state-of-the-art implementation of the well-known spectral bisection [Bar93]. A number of approaches are available for matching the vertices for the coarsening phase. These include random matching (selecting the two vertices randomly), heavy edge matching (selecting the two vertices that are connected through the edges with maximum weights) and light vertex matching (selecting the two vertices that are the lightest in weight).

Ou *et al.* propose an offloading middleware for mobile applications and WS clients [Ou07]. The middleware considers multiple constraints (i.e. memory, CPU and bandwidth) while applying the proposed (k+1) graph theory based application partitioning algorithm. The (k+1) partitioning algorithm is based on the algorithms previously proposed [Kur99] but uses a heavy edge and light vertex matching approach to divide the application into k balanced partitions in addition to a special partition that is constrained to execute on the local device.

The graph partitioning algorithms discussed earlier are very complex and based on an objective of either minimizing the communication cost between different application components or achieving partitions of the same size so that the partitions can be run in parallel. Moreover, these algorithms require a lot of computing resources and are characterized by large execution times for achieving partitions. Hence, these algorithms are not suitable especially for run time partitioning. Although the general application partitioning problem has been studied by many researchers, comparatively little work exists in the domain of WS application partitioning that has different constraints. For example, achieving partitions of the same size is not a key requirement in case of mobile WS applications. The size of the offloaded partition(s) is as important as the communication cost among different partitions. The algorithms proposed in this thesis consider such factors while devising that are important in WS application partitioning.

2.3 Design Time versus Runtime WS Application Partitioning

Application partitioning is generally performed either at design time or at run time. As discussed in Section 2.2, the algorithms proposed in the literature are mostly based on design time partitioning. This is because design time partitioning is easy to use and implement. Since the algorithms are executed in advance, the complexity of partitioning algorithms does not contribute to overall performance of the partitioned system. The main drawbacks of design time partitioning are 1) the partitioned system achieved may not be an optimal solution for various devices on which the system may get deployed and 2) the partitioned system is generally insensitive to the variation in system load. Performing an application partitioning at runtime is an attractive solution for such scenarios because it can use system load information and device characteristics for achieving an effective partitioned system. Achieving a partitioned system with runtime partitioning approach has a few limitations as well. For example, deciding when to run the application partitioning algorithm is an important question. Should it be run for every request arrival or run only when the system load changes significantly? In both cases, there may be a substantial overhead affecting the application response time. The benefit that accrues from running the application on a partitioned system must be able to offset this overhead. There are, however, many situations when it makes sense to use a runtime partitioning approach. A few examples of such situations are described next. The device may be running multiple applications at a time and these applications are sharing the device resources such as CPU and memory. The device may be receiving a large number of WS requests for the hosted web services. The device may be close to running out of battery power.

In such situations, using a runtime application partitioning is expected to be effective because the size of the partition to be executed on a remote computing node can be varied based on the situation. For example, in a situation when a device is running out of battery power, a substantial amount of application can be offloaded to preserve the device's battery power. If a large number of WS requests are waiting to be executed, running a large part of an application on a remote computing node can improve system response time. If a device is not experiencing any of these problems, it may be beneficial to execute a little part of an application (or even no part of an application) on a remote computing node.

This thesis investigates both the design time and the run time techniques for WS application partitioning. The design time WS partitioning algorithms are graph based. The proposed runtime WS partitioning combines advantages of both design time and the runtime application partitioning by achieving multiple execution plans each corresponding to a specific partitioning of the WS application in advance and then using a runtime middleware to select an appropriate execution plan based on the system load information.

A comprehensive simulation-based analysis is performed to analyze the performance of the proposed design time and run time WS partitioning techniques.

Chapter 3: WS Execution Environments

This chapter presents details of the system design for the lightweight WSEE and the configurable partitioned WSEE. The two different WS execution environments are proposed to provide toolkits for hosting web services on resource constrained devices (lightweight WSEE) and to provide support for conformation with a wide range of computationally demanding WS standards (partitioned WSEE). This chapter also presents a detailed performance analysis of the two proposed WSEEs using different sample web services hosted on mobile devices using wireless local area network environment.

3.1 Overview

As already mentioned in Chapter 1, hosting web services on resources constrained devices is challenging because of their resource constraints. A number of challenges need to be addressed for hosting web services on such devices. Diversity of device hardware/operating systems, execution of resource demanding web services, conformation to computationally demanding web services standards (such as WS-Security and WS-Policy), supporting multiple WS concurrent clients and providing reasonable response times are a few examples of such challenges. This chapter is focused on investigating the web service execution environment for hosting web services on resource constrained devices. The proposed web service execution environments are expected to fulfill the following objectives:

- Reducing the resource demand of the mobile device.
- Achieving a reasonable end-to-end response time when WS clients invoke web services provided from mobile devices.
- Facilitating the access to the hosted web service to a dozen or more concurrent WS clients.
- Supporting a large number of WS standards for the web services hosted on the mobile device.

3.2 Lightweight WSEE

The primary motivation for proposing the lightweight WSEE is to provide an environment for hosting web services for small handheld devices such as smart phones, pagers and personal digital assistants. The existing environments or toolkits for web service hosting (Apache Axis and Oracle (Formerly Sun Systems) JAX-WS, for example) cannot be used for resource constrained devices because these environments require a Java run time environment 1.4 or higher that is only available for desktop machines. The lightweight WSEE provides lightweight components to facilitate hosting of web services with a basic set of WS standards such as SOAP, XML Signature [W3c02-1] and XML Encryption [W3c02-1].

A WSEE essentially requires a set of software components that facilitate hosting of web services. The required software components include a transport component for data exchange with WS clients, a SOAP engine component for processing XML/SOAP messages and an application invocation engine that can execute requested WS applications on behalf of WS clients.

As already mentioned, the Java ME environment is selected for implementation of the lightweight WSEE prototype because of its ability to be operable on diverse platforms for mobile devices such as Windows Mobile OS, Android, Symbian OS and Blackberry OS. Java ME has two configurations: the Connected Device Configuration (CDC) (JSR 218) [CDC05] and the Connected Limited Device Configuration (CLDC) (JSR 139) [CLD05]. The lightweight WSEE prototype is capable of running under both the CDC and the CLDC configurations. For resource constrained devices, a lightweight version of each of these components is devised.

Architecture of the proposed lightweight WSEE is based on four layers: a transport layer, a service layer, a WS standards layer and a WS applications layer. A high level architectural overview of the proposed lightweight WSEE is shown in Figure 3-1.

The transport layer is responsible for receiving and sending SOAP messages to WS clients. The service layer de-serializes incoming SOAP messages, conforms to WS standards if required, executes the request WS application and serializes the outgoing WS response messages. The WS standard layer is a collection of the modules that are implemented for conformation to the WS standards. The WS application layer represents the deployed WS applications. The details of these layers and their key components are discussed next.

3.2.1 Transport Layer

In the lightweight WSEE, the transport layer is capable of exchanging SOAP messages using HTTP or TCP protocols. Note that there is no specific transport protocol associated with exchange of SOAP messages. SOAP messages can be exchanged between nodes using any transport mechanism. This allows WS applications to select any appropriate transport mechanism according to the availability of resources and the quality of service requirements.

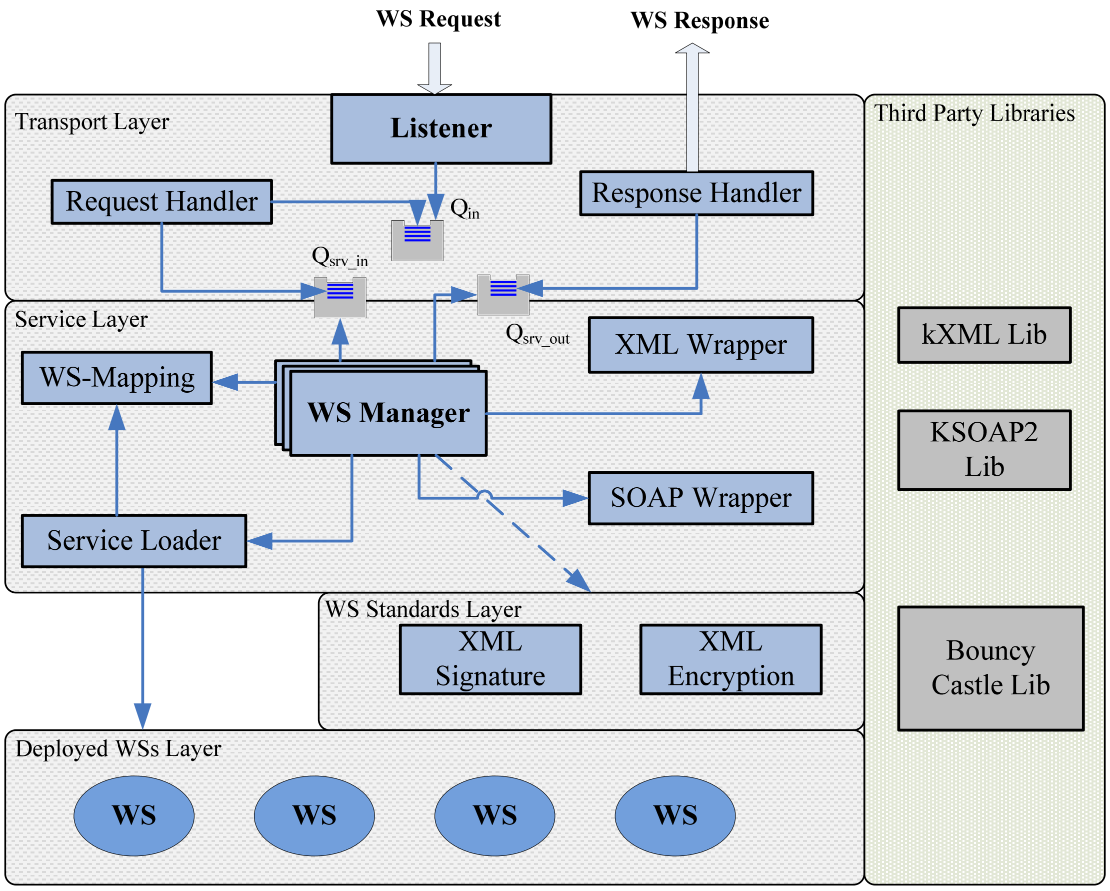


Figure 3-1: Proposed Architecture for Lightweight WSEE

3.2.1.1 Initial Setup

As the lightweight WSEE application starts, the transport listener (Listener in Figure 3-1), which is a main program, performs the following steps for the initial setup of the WSEE:

- Load WSEE environment variables from a java property file ('wsee_config.properties'). These environment variables include

    - Number of minimum (N_{min}) and number of maximum (N_{max}) threads of WSManager (explained in the next subsection).
    - Transport mode: HTTP or TCP.
    - Port number at which the transport listener is listening.
    - Timeout after which the transport listener disconnects from a non-responsive client.
    - Root directory name for the deployed web service applications.
- Create a thread (Request Handler in Figure 3-1) for handling WS requests.
- Create a thread (Response Handler in Figure 3-1) for handling outgoing WS response messages.
- Create WSManager threads the number of which is provided through environment variable $N_{min.}$
- Initialize the Web Service Mapping (WS-Mapping) component. During the initialization stage, details of all the deployed web services are loaded in memory. The details of WS mapping component are described in the next section.
- Create a server socket and start listening at a port specified in the properties file.

On receiving an incoming request from a WS client, the listener creates an object of Java Socket class. This newly created object is put in the Q_{in} queue and the request handler is notified. The listener starts listening again at the specified port.

3.2.1.2 Request Handler

The request handler thread is either in a wait state or in a run state. The request handler is in the wait state when there is no Socket object in Q_{in} (see Figure 3-1). On arrival of a new Socket object in Q_{in}, the request handler is notified by the listener. On receiving the notification, the request handler goes into the run state. The request handler remains in the run state when it is in the process of handling the new request. At the completion of processing of the request, the request handler checks Q_{in}. If Q_{in} is empty, the request handler goes into the wait state and waits for a new request. If Q_{in} is not empty, the request handler selects the next request (Socket object) waiting in the queue and starts its processing. The key objective of the request handler is to separate out the SOAP message from the request.

Once the SOAP message is extracted, an object of SoapRequest class is created. The SoapRequest is an application data object that holds information about the SOAP request message. The SoapRequest object is put into Q_{srv_in} (see Figure 3-1) queue. After this, the request handler fetches a new request from Q_{in} (if Q_{in} is not empty) or goes into the wait state (if Q_{in} is empty).

3.2.1.3 Response Handler

Once the WS request is processed by the service layer, the response SOAP message (if any) is placed in a separate queue, Q_{srv_out} (see Figure 3-1). The response handler works in a similar manner as the request handler. The response handler thread is in a wait state when there is no response message in Q_{srv_out} (see Figure 3-1). On arrival of a new response message in Q_{srv_out}, the response handler is notified by the service layer. On receiving such a notification, it fetches the response message from Q_{srv_out} and starts its processing. The response message contains the response SOAP message and information about the WS client. A WSEE application class, WSResponse, is used to represent the response message.

The WS client's address and the port number at which the client is expecting to receive the response is stored in WSResponse object by WSManager. An attribute 'response' contains the response SOAP message. There are two attributes (isFault and FAULT) for indication of any fault that occurs while invoking the requested WS. After fetching the WSResponse object from Q_{srv_out}, the response handler opens a socket connection with the WS client and sends the response SOAP message. Once the response message is sent, the response handler fetches a new response SOAP message from Q_{srv_out} (if it is not empty) or enters into the wait state.

3.2.2 Service Layer

This is a core layer of the lightweight WSEE. The primary responsibilities of this layer are the parsing of the incoming SOAP messages, executing the WS application and then wrapping the results of the WS application into a response SOAP message.

Currently, the service layer is using a Remote Procedure Call (RPC) style as a WSDL binding. The WSDL binding describes how a WS is bound to a messaging protocol. The document style binding is another popular style of the binding and can be added in future versions. The reason for using RPC style of binding is that it is easy to implement and the WSDL generated using the RPC style binding is straight forward and easy to understand. Note that RPC style and document style bindings are not programming models. These bindings only help to translate a WSDL to a SOAP message. The subcomponents of the service manager are described next.

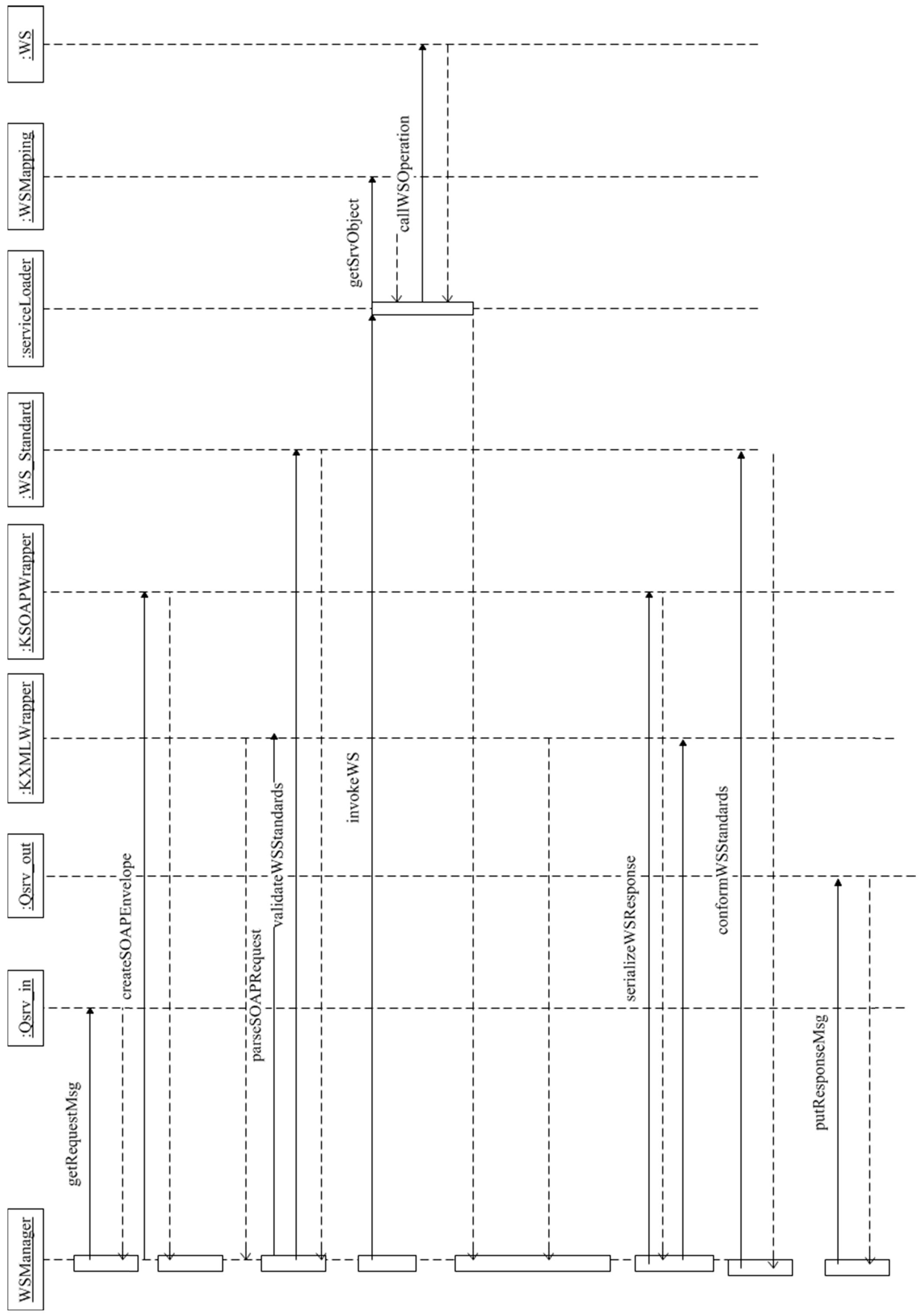


Figure 3-2: Sequence diagram capturing interactions between WSManager and the components in the service layer and components of other layers

WSManager (shown in Figure 3-1) is a main controller of the service layer. Multiple threads of WSManager are available in a thread pool. These threads are used to process WS requests waiting in Q_{srv_in}. The maximum number of threads that can be created for WSManager is provided as an input parameter in a 'wsee_config.properties' file. The WSManager uses different components such as XML Wrapper, SOAP Wrapper, a service loader, WS-Mapping and interacts with the components of the WS standards layers for conforming to different WS standards. The interactions of WSManager with other components are captured in a UML sequence diagram presented in Figure 3-2. Different operations shown in the sequence diagram (see Figure 3-2) are explained next.

Whenever there is a new request waiting in queue (Q_{srv_in}), WSManager starts by getting a waiting request object (WSRequest) from Q_{srv_in} (see *getRequestMsg* operation in Figure 3-2). After getting the WSRequest object, WSManager creates an object of SoapSerializationEnvelope class (see KSOAP documentation [Kso03] for more details) and is shown as *createSOAPEnvelope* operation in Figure 3-2. WSManager interacts with SOAPWrapper for this operation. SOAPWrapper uses an open source KSOAP2 library to create a SoapSerializationEnvelope object. SoapSerializationEnvelope is a comprehensive object that can hold the SOAP request header, the SOAP request body, the SOAP response header and the SOAP response body separately.

After creating the SoapSerializationEnvelope object, WSManager uses XMLWrapper (see *parseSOAPRequest* operation in Figure 3-2) which in turn uses an open source kXML library to parse the SOAP request message. After parsing the SOAP message, WSManager interacts with the WS Standards layer and is shown as *validateWSStandards* operation in Figure 3-2. For the lightweight WSEE, this operation performs the tasks that are required for the verification of XML Signature (WS-Security) of WS clients and signing of response SOAP messages. After performing the tasks that are required for the conformation with the given WS standard(s), if any, WSManager interacts with the service loader for invoking a particular method of the requested WS class (see *invokeWS* operation in Figure 3-2). The service loader interacts with WS-Mapping to locate the Java class of the requested WS (shown as *getSrvObject* operation in Figure 3-2). Once the Java class of the requested WS is identified, the service loader first uses this operation either to create an instance of the class or to locate an already created instance from application context and then invokes the requested method of the Java class. This operation is shown as *callWSOperation* in Figure 3-2. Next, WSManager serializes the response of the requested WS application into a SOAP message. For this step, WSManager uses XMLWrapper and SOAPWrapper for serialization and is shown as *serializeWSResponse* operation in Figure 3-2. WSManager interacts with the WS standard layer again to perform the actions required to conform to a given WS standard if required (see *conformWSStandards* operation in Figure 3-2. For example, the response SOAP message may be required to be encrypted (using XML encryption). Signing the response SOAP message is another example of performing an action to conform to a WS standard before sending it to the WS client. In the lightweight WS standard, this operation is used to sign the response SOAP message using XML Signature specifications. The last operation (*putResponseMsg*) performed by WSManager is to store the final response SOAP message in Q_{srv_out} and to notify the response handler (transport layer) that the response is ready to be send to the WS client.

In the next subsections, different components of the service layer are described.

3.2.2.1 XML Wrapper

For XML parsing and serialization, a lightweight XML parser, KXML [Kxm03], is used. The KXML parser is based on a pull parsing technique and is an implementation of the XMLPULL parser API [Xml02]. XML Pull Parsing is a process of parsing XML as a stream rather than building a Data Object Model (DOM) tree or push parsing in which events are pushed out to a client code. The pull XML parsers are fast and more memory efficient in comparison to the parsers that are based on DOM. To access this third party library, a wrapper component (KXMLWrapper) is introduced in the service layer. The XML Wrapper provides an interface to WSManager for accessing XML parsing methods.

3.2.2.2 SOAP Wrapper

For SOAP processing, a KSOAP [Kso03] library is used. KSOAP is an open source library for WS clients to access web services. To use it on the server side for processing of SOAP messages, the open source library is extended in this research. The extension is introduced for the invocation of web services using the remote procedure call (RPC) style binding. For accessing core methods of the KSOAP library and its extension (devised for the lightweight WSEE), a wrapper (the SOAP Wrapper) is introduced.

3.2.2.3 Service Loader

This component is used for invocation of the requested web services. WSManager supplies the SOAP Action parameter and the parsed SOAP request message to the service loader. The service loader interacts with WS-Mapping to get the Java class name (with path) based on the value of the SOAP Action.

3.2.2.4 WS-Mapping

WS-Mapping is a static component that manages an in-memory list of deployed web services. The light WSEE assumes that all deployed web services are available in directory whose path is provided in the 'wsee_config.properties' file. Individual web services are assumed to be available in subfolders of the web services root directory. A WS is required to provide a Java implementation of the WS application and a 'ws.xml' file in a folder under the root directory of web services. The ws.xml file is required to follow the following XML grammar.

In this XML format shown in Figure 3-3, the <webservice> tag is the root node of 'ws.xml' document. <ws-uri> represents the SOAP Action that is used as a key for identification of the web service. Value inside the <ws-uri> node is required to be unique. <ws-class> node refers to the name of the Java class of the WS application. <ws-standards> node can contain multiple child nodes. Each child node corresponds to the actions that need to be performed to conform to a specific WS standard. The support for the custom data types as WS parameters is planned for a future version. The current version of the lightweight WSEE supports primitive data types and collections as WS parameters. At the time of initialization, WS-Mapping reads 'ws.xml' files for all deployed web services and loads the relevant data of the deployed WSs in memory.

```
<webservice>
    <ws-uri> … </ws-uri>
    <ws-class> … </ws-class>
    <ws-standards> … </ws-standards>
</webservice>
```

Figure 3-3: XML grammar for defining WS Mapping

After invoking the requested WS application, WSManager serializes the results into a response SOAP message. If the requested WS requires conformation with additional WS standards such as the one that requires the verification of a XML Signature, WSManager interacts with the component that provides the verification of the XML Signature.

3.2.3 WS Standards Layer

To support additional WS standards, another layer is introduced in the lightweight WSEE system. This layer comprises the components that perform actions that are required to conform to different WS standards. For testing purposes and as a proof of concept, support for verification of the XML Signature of incoming SOAP messages and signing of the outgoing SOAP messages are provided in the lightweight WSEE. For verification and signing of XML Signatures, a lightweight version of a third party cryptography library (Bouncy Castle) is used. A brief overview of the steps involved in verification of the XML Signature and signing of SOAP messages are discussed next.

3.2.3.1 XML Signature Verification

The process of verifying the XML Signature is based on the specifications of XML Signature standard [W3c02-2].The first step of verification of the XML Signature is the extraction of the digest, the signature and the public key elements form the SOAP header. Note that the digest, the signature and the public key elements are placed in the SOAP header by WS clients. The digest of a message is a unique number which is created by using a hashing algorithm for representing the message. If the message is changed, the digest value will also be changed. The signature is an encrypted form of a message that is obtained by using the private key of a user and the message itself. In the second step, the digest of the message is computed and compared with the digest value already extracted from the SOAP header. In the third step, the signature extracted in the first step is verified by using the public key information. Note that this key information is also extracted in the first step. If the computed digest does not match the digest value extracted in the first step or the signature is not verified by using the public key, the XML Signature verification is said to have failed and a SOAP fault is sent back to the WS client. A SOAP fault is an optional part of a SOAP envelope (in addition to the header and the body) used for reporting errors.

3.2.3.2 XML Signature Signing

Signing a SOAP message is also a three step process. First, the digest for the contents to be signed (the body or any element of a SOAP message) is computed. In the second step, the signature of the contents to be signed is calculated using the private key of a user. In the last step, the digest, the signature and the public key information are inserted in the SOAP header. The XML Signature specifications describe structure of elements that can be used to insert the digest, the signature and the public key information in the SOAP header.

3.3 Distributed SOAP Engine Based Partitioned WSEE

In the lightweight WSEE, only the verification and the signing of the XML Signature are supported as a proof of concept. Other WS standards are proposed by the WS community for security, reliability and transactions. A WSEE that supports a large number of WS standards is difficult to deploy on mobile devices with limited resources. For such resource demanding requirements, it is proposed to partition the execution environment and deploy the two partitions on two different nodes (an intermediate node and the mobile device node). The intermediate node is a node that acts as a proxy for the hosted WS on a mobile computing node and it provides the partial functionality of the WSEE. The proposed technique of WSEE partitioning is based on the splitting of the SOAP engine (WSManager in Figure 3-6) functionality. The functionality of WSManager is divided into two partitions only: one for the intermediate node and the other for the mobile device. The intermediate node is the one that receives the WS requests from WS clients on behalf of the mobile device.

3.3.1 Motivation for the Proposed WSEE Partitioning Technique

WSManager performs a series of tasks for invoking a WS. The sequence of tasks related to the invocation of a WS that are performed by the distributed WSManager, is described. The typical tasks performed by a WSManager thread are decryption of incoming SOAP message (T1), verification of the identity of the WS client (T2), verification of the integrity of the message (T3), invocation of the WS (T4), signing the response message with the service provider's certificate (T5) and encryption of the response message (T6) before sending it back to the WS client. The sequence of tasks as they are performed by a WSManager thread is captured in Figure 3-4-a. The dotted lines with an arrow head in Figure 3-4 represent the order of execution of tasks. The order of execution implies that T1 is required to be performed before T2 and T2 is required to be performed before T3 and so on.

As already mentioned, the functionality of WSManager is divided into two partitions: one of the partitions is handled by an *intermediate WSManager* and the other by a *mobile WSManager*. Note that both the nodes (intermediate as well the mobile device node) uses the same implementation of WSManager. But WSManager on the intermediate node does not need to interact with the service loader and the WS-Mapping components. The objective of having the same implementation of

WSManager on the two nodes is to make the partitioning of WSManager tasks configurable. The two WSManager differ only in their executions. One WSManager executes one set of tasks and the other executes another set of tasks. The intermediate WSManager can be assigned such tasks that demand more resources. The mobile WSManager is responsible for processing the tasks that require local resources on the mobile device or use confidential information available on it. For example, if encrypting or signing a response message requires a security certificate of the device's owner, then it is not a good practice to delegate such tasks to an intermediate node. With a distribution of tasks shown in Figure 3-4-b, the intermediate WSManager decrypts the incoming message (T1), verifies the identity of the WS client (T2) and the integrity of the message (T3). The mobile WSManager executes the task for invoking the requested WS (T4) and the task for signing the response message (T5). In the end, the intermediate node performs the task for encrypting the response message (T6).

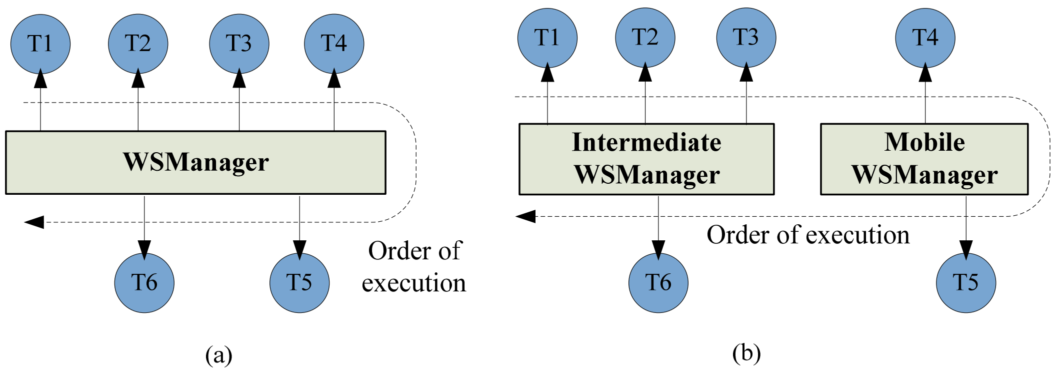


Figure 3-4: Tasks of a SOAP engine (a) performed as a single SOAP engine (b) performed as a distributed SOAP engine

3.3.2 Configurable Partitioning Scheme for Execution of WSManager Tasks

To make it possible to allocate different sets of tasks for execution to the intermediate WSManager and the mobile WSManager for a given WS, a configurable partitioning scheme is introduced. The *partitioning scheme* can be defined as an agreement between the two distributed components of WSManager. This partitioning scheme helps the two components to identify which tasks need to be executed by which component. The partitioning scheme is required to be submitted at the time of deployment of a new WS. The submitted partitioning scheme is available to both components of WSManager (the intermediate WSManager and the mobile WSManager) at the time of execution. A sample partitioning scheme for execution of WSManager tasks is presented in Figure 3-5. Note that this arrangement has the flexibility of using different partitioning schemes for different web services hosted on the mobile device.

This scheme uses an XML schema that defines the XML elements with attributes for each WSManager task. The scheme shown in Figure 3-5 is designed in such a way that the most of the tasks related to security are processed by the intermediate WSManager running on an intermediate node. In this scheme, the intermediate WSManager is assigned the following tasks

1- Decrypting of the incoming SOAP message.
2- Verifying of identity of the WS client (IDENTITY).
3- Validating the integrity of the message (INTEGRITY) and
4- Encrypting the response message (ENCRYPTION).

The mobile WSManager is assigned the following tasks.

1- Invocation of the requested WS application (WEBSERVICE).
2- Signing the response message.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!--    A sample partitioning scheme to split task execution between an
        intermediate node and a mobile node
-->

<webservice uri= ‘http://www.carleton.ca/sce/mobileweb/sample1’>
  <!-- tasks assigned to static SOAP engine -->
  <intermediate>
        <DECRYPTION required="false">
        <IDENTITY required="true"/>
        <INTEGRITY required="true"/>
  </intermediate>

  <!-- tasks assigned to mobile SOAP engine -->
  <mobile>
        <WS_OPERATION class-name=""/>
        <SIGNATURE required="true"/>
  </mobile>
  <intermediate>
        <ENCRYPTION required="false">
    </intermediate>
</webservice>
```

Figure 3-5: A sample partitioning scheme

The sequence of tasks performed by the two components of WSManager follows the same order as specified in the partitioning scheme. In Figure 3-5, there are two ‘intermediate’ and one ‘mobile’ XML blocks of elements. An ‘intermediate’ block includes a list of tasks that are assigned to the intermediate WSManager. A ‘mobile’ block represents a set of tasks for the mobile WSManager. The ‘intermediate’ block that comes after the ‘mobile’ block contains a list of tasks that have to be performed on the intermediate node after the execution of tasks by the mobile WSManager.

The partitioning scheme is made configurable because the partitioning of tasks depends on both the nature of a WS and the resource availability on the mobile device and on the intermediate node. If the device is not capable of performing a task such as verification of XML Signature of WS clients due to limited resources, the partitioning scheme can be configured in such a way that such tasks are performed by the intermediate node. In another case, if the owner of a more powerful mobile device requires performing these security-related tasks on the device to avoid any security risks or because of business requirements, a different partitioning scheme can be used.

3.3.3 System Overview and Design

The lightweight WSEE (discussed earlier) is enhanced to achieve the partitioned WSEE. The execution of different WSManager tasks is achieved through a chain of handlers. Each handler is represented by an XML element (under 'intermediate' or 'mobile' blocks of elements in Figure 3-5) in the partitioning scheme. Each handler contains a set of operations for performing one particular WSManager task. After performing the assigned task, the handler passes the control of execution to the next handler in the chain. If the XML element attribute 'required' is set to 'false' for a handler in the partitioning scheme, then the control of execution is passed to the next handler without executing any code for that handler.

After processing part of the incoming SOAP message, the intermediate WSManager delivers the rest of the message to the mobile WSManager. Note that the parts of a SOAP message that have been processed and not required by mobile WSManager tasks can be eliminated from the message before forwarding it to the mobile WSManager. This will add processing overheads but can potentially save bandwidth. The partitioned WSEE forwards a complete SOAP message to the mobile WSManager.

The internal details of the partitioned WSEE based on two components of WSManager are shown in Figure 3-6. The functional details of the components for the transport layer, the WS standards layer and the WS applications layer are the same as discussed for the lightweight WSEE in Section 3.2. Thus, only a high level description is provided for these layers. In this section, the role and functionality of the service layer is discussed in more detail.

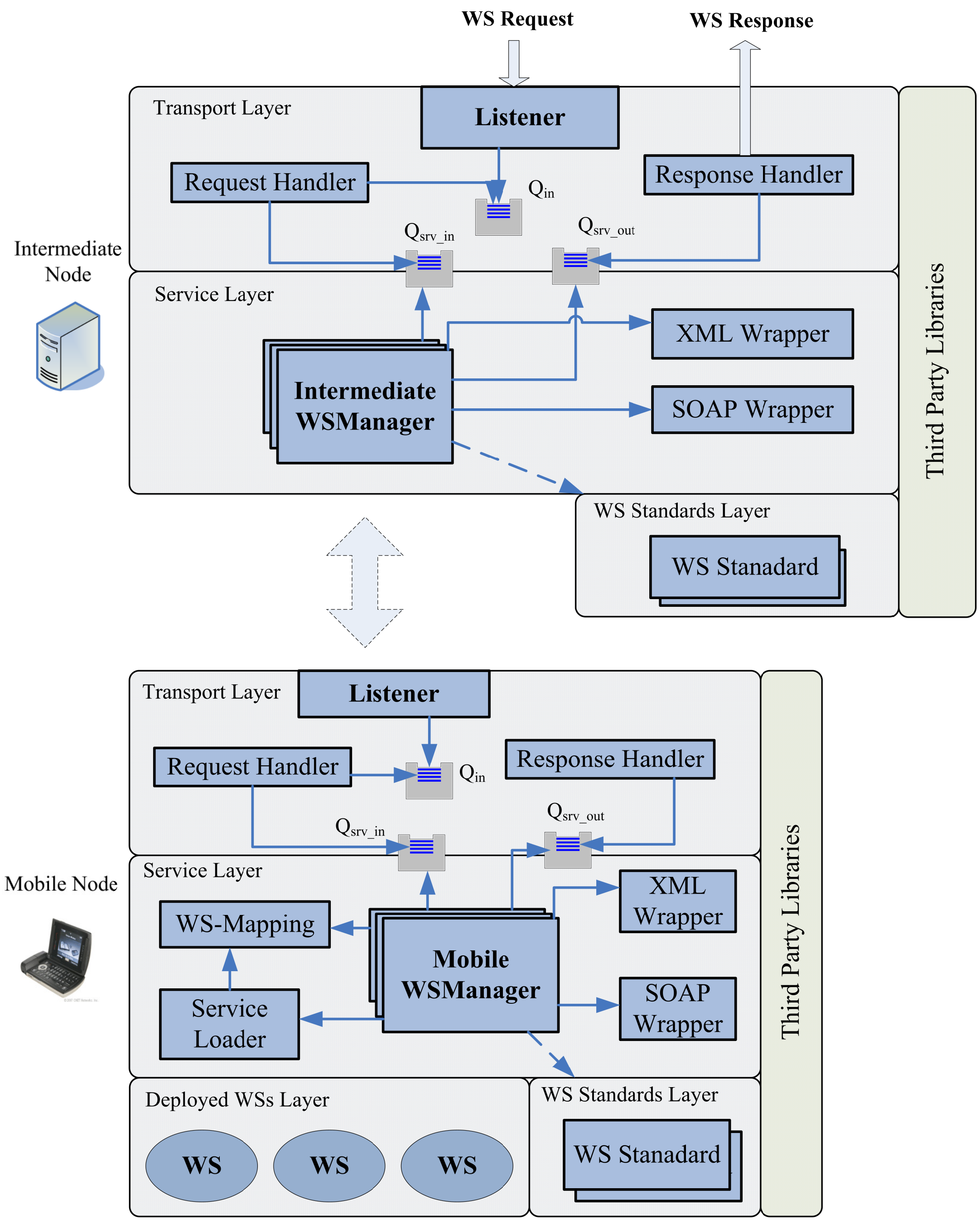


Figure 3-6: A partitioned WSEE based on a distributed SOAP engine

On receiving a new WS request, the transport listener of the WSEE deployed on an intermediate node puts the incoming WS requests in a queue shown as Q_{in} in Figure 3-6. The Request Handler thread parses the SOAP message from the incoming WS requests and puts the parsed message in another queue (shown as Q_{srv_in} in Figure 3-6) which is an input queue for the intermediate WSManager.

The intermediate WSManager applies a series of operations (similar to the one shown in Figure 3-5). Note that the intermediate WS Manager performs only those operations that are assigned to it based on the partitioning scheme. For application of these operations, the intermediate WS Manager interacts with the components in its own layer such as XML Wrapper, SOAP Wrapper and also with components of the WS standards layer. These components may use third party libraries for providing the required functionality. The typical tasks assigned to the intermediate WSManager are the actions that are required for conformation to different WS standards.

It is important to mention that there are WS standards such as WS-Security that requires execution of resource demanding algorithms. Even if a mobile device is capable of executing such resource demanding algorithms, the repeated invocations of such algorithms will not only hinder the device to perform its core functions (such as voice calls) but it will also consume its battery power significantly.

Note that the service layer deployed on the intermediate node does not include WS-Mapping and the service loader components. These components are required for the invocation of requested web services. The service layer of the WSEE deployed on the intermediate node is mainly responsible for performing actions that are required for conformation to resource demanding WS standards.

In the next step, the intermediate WSManager forwards the SOAP message to the part of the WSEE deployed on the mobile device node. On receiving the SOAP message from the intermediate node, the transport layer of WSEE deployed on the mobile node passes it to its service layer using the Q_{in} queue, the Request Handler thread and the Q_{srv_in} queue in the same manner as discussed in Section 3.2. The mobile WSManager only executes those tasks that are assigned to it by the partitioning scheme. An example partitioning scheme was presented in Figure 3-5. The mobile WS Manager interacts with the sibling components (XML Wrapper, SOAP Wrapper) and with components of the WS standards layer and also with the deployed WS layer. Note that the service layer of the WSEE deployed on the mobile node includes WS-Mapping and the service loader. After performing the actions required for conformation to WS standards (if required), the mobile WSManager uses the service loader for invocation of the requested WS. The steps of executing the WS were already explained in the discussion of the lightweight WSEE in Section 3.2.2.

The results of executing the requested WS are serialized into a SOAP message (response SOAP message) by the mobile WSManager. The response SOAP message is returned to the intermediate WSManager. On receiving the response SOAP message, the intermediate WSManager can execute rest of the task(s) (if any) assigned to it. In the last step, the WSEE hosted on the intermediate node sends the final response message to the original WS client.

Note that the WS clients only communicate with the intermediate node. The communication between the intermediate WSManager (intermediate node) and the mobile WSManager (mobile node) is transparent to the WS clients. Thus, the use of a wireless environment friendly transport mechanism for exchanging SOAP messages between the two components of WSManager is expected to improve system performance.

3.4 Experimental Analysis

The feasibility and the performance of the two proposed WSEEs are analyzed for a number of scenarios. A list of sample web services that are used for a detailed performance analysis of the two proposed WSEEs is presented next.

3.4.1 Sample Web Services

Three sample web services are used: Image service, Tracking service and π Calculator service. Image WS is a data intensive service that is used to fetch an image data from a mobile device. Tracking WS is a lightweight service that can be used to fetch the location information of the device. π Calculator WS is a computationally intensive service that computes the value of π.

3.4.1.1 Image WS

This service is designed for fetching image data from a mobile device based on three input parameters: location, keyword and the desired image size. At the time of saving an image that was captured by a built-in camera on the device, the user or the device is saving location information, one to three keywords, and a textual description of the image. The image that matches the most number of input parameters is selected. In case, there are multiple images which match the same number of input parameters, the image is selected randomly from that list of images. If an appropriate match is not found, a null response is returned.

This service can also return a resized image based on the input parameter. This additional feature contributes to more computation on the mobile device.The WS has three main components: *SearchImage, ResizeImage* and *PackData*. The *SearchImage* component queries a local database to look for the path of an image file based on the location parameter. The *ResizeImage* component is responsible for resizing the image according the value of the image size parameter. For resizing the image, a bilinear interpolation algorithm is used. Bilinear interpolation considers the closest 2x2 neighborhood of known pixel values for calculation of the interpolated value. *PackData* serializes the resized image data and the time information when the image was captured in a response SOAP message.

For the experiments presented in Section 3.4.5, three different sets of images of different sizes are retrieved. The three different sets used have sizes in the range 10 KB, 100 KB and 500 KB. In a single experiment, only one set of images is used.

3.4.1.2 Tracking WS

This WS runs on the mobile device of an emergency responder such as a doctor and provides the address, elevation and time zone of the location. The WS has three main components: *GetCoordinate*, *GetLocDetail* and *PackData*. The *GetCoordinate* component fetches the actual Global Positioning System (GPS) coordinates from a GPS receiver. For this experiment, the step of getting GPS coordinates is emulated by fetching a random set of coordinates from a local file containing more than a thousand locations. The *GetLocDetail* component queries a database of locations to find the details of a location that is closest to the GPS coordinates. The location database is downloaded from a well-known geographical organization (GeoNames [Geo08]) serializes the location information as a response message and sends it back to the WS requester.

Currently, there is a large variety of mobile phones from Apple, Blackberry and Nokia that come with a built-in GPS receiver. Since the device used (Dell Axim) in the performance experiments involving Tracking WS does not come with a built-in GPS receiver, the step of obtaining GPS coordinates is replaced with a random selection of a database record from a city database (available from GeoNames [Geo08]). In this synthetic application, a database record is selected randomly for every new request and every record has the same probability of being selected. This WS is designed to be a lightweight application and is less data intensive as compared to the image WS.

3.4.1.3 π Calculator WS

Calculation of π is not a typical candidate for a sample WS, but it is chosen because its computational intensity can be varied conveniently for performance analysis. It has been used by other researchers for a similar analysis [Ou07]. For this WS, the classic technique of calculating the value of π using the Gregory-Leibniz Series [Wel86] is deployed. According to the Gregory-Leibniz series, a value of π can be calculated by the following series

$$\pi = 4 * \sum_{r=1}^{\infty} \frac{(-1)^{r+1}}{2r-1}$$

The accuracy of π depends on the number of terms used in the summation of the series. More the number of terms used, the more accurate is the value of π. The number of terms (N) used to calculate π is made an input parameter for this WS and is used to compute the CPU time consumed. For the experiments presented in Section 3.4.5, the values of N used are 10 K, 50K and 100K.

3.4.2 Workload and System Parameters

To analyze the performance of the two WSEEs, following workload and system parameters are varied.

3.4.2.1 WS Complexity:

Performance of the partitioned WSEE is observed by invoking different sample WSs that were discussed earlier. Note that the sample WSs are characterized by different levels of computational complexity. In addition to that, each sample WS is invoked using different values of parameters that are required to access them. For example, the π calculator service can be invoked using different values of number of terms (N) to be used for calculation of π.

3.4.2.2 Number of Concurrent WS Clients (C):

The number of WS clients (C) invoking a sample web service at the same time is varied to investigate the scalability of the system. C is varied from 1 to 20.

3.4.2.3 Mobile Device Speed (ω):

The experiments are run on real handheld devices of different processing speeds. The Dell Axim PDA device used in our experiments can be run with different processing speed (208 MHz and 624 MHz). The default speed of the device used is 624 MHz.

3.4.3 Performance Metrics

The performance of the system is analyzed by measuring the end to end ***response time (R)***. Response time is defined as the difference between the time when a WS response is received by the WS client and the time when the WS client sends the corresponding SOAP message request to a WS provider.

Scalability metric for WSEE (ξ) is measured by measuring the number of WS clients that can be serviced concurrently with a reasonable response time that is assumed to be approximately 30 seconds. The value of 30 seconds is chosen based on the most commonly used value of the timeout used by http servers. The value can be different for different applications.

3.4.4 Setup

Separate prototypes for the lightweight WSEE and the partitioned WSEE are implemented in Java ME. The WS client programs are run on a machine that is equipped with a 2.4 GHz Pentium 4 processor and a memory of 2 GB. The client node is running under Windows XP Professional operating system. The intermediate node is a desktop computer equipped with a Pentium 4 processor that is running under the Ubuntu Linux operating system. Its CPU speed is 3 GHz and 3 GB of RAM is available on this node. The lightweight WSEE with sample web services is deployed on the Dell Axim x51v PDA without using any intermediate node. In case of the partitioned WSEE, the WSEE package is installed both on the intermediate node and on the mobile device (Dell Axim x51v PDA). Note that the sample web services are only available on the mobile device. The PDA used as the mobile device has an Intel XScale ARM processor (PXA270) that can be run at multiple speeds (208 MHz, 520 MHz and 624 MHz). The default processor speed used is 624 MHz in all experiments. The PDA used is equipped with a RAM of 64 MB and is running the Windows Mobile 5.0 operating system. The Java ME environment (J9) available on the PDA is a JVM provided by IBM for Java ME CDC devices [Ren09]. The installed J9 (JVM) is based on the specification of CDC 1.1 [CDC05]. The client machines and the intermediate node communicate with the PDA hosting the sample WSs using a wireless local area network (IEEE 802.11b standard). The measurements were made on a dedicated network where the experiments ran without any interference from other applications.

A closed system model is used for experimentation. Each client (a Java thread) operates cyclically and sends one request at a time. As soon as the response is received, the client repeats the cycle. The system is stressed by increasing the number of concurrent WS clients. For a single experiment, each client generates 10000 requests. So for an experiment with 10 WS clients, for example, the response time is calculated by taking the average of response times of 100,000 requests. Each experiment is repeated 15-25 times to obtain sufficiently small confidence intervals for the average values. For the experiments presented next, confidence intervals of ±5% (or less) for mean response time were obtained at a confidence level of 95%.

3.4.5 Experimental Results

Results of experiments carried out to investigate the performance of the lightweight WSEE and the partitioned WSEE are presented. In the first experiment, the performance of the lightweight WSEE is compared with the performance of the partitioned WSEE. In the second experiment, scalability and the effect of the transport mechanism on the performance of partitioned WSEE is investigated. The impact of using additional WS standards on the lightweight WSEE and the partitioned WSEE is analyzed in the third experiment. In the fourth experiment, the impact of the speed of the processor of the mobile device on performance is investigated. In all experiments, only one a specific sample web service is accessed at a time.

3.4.5.1 Lightweight WSEE versus Partitioned WSEE

This experiment is carried out using the three sample web services. The input parameters for Image WS and π Calculator WS are summarized in Table 3-1. The average image sizes used for Image WS are 10KB, 100 KB and 500 KB. The average values of N used for π Calculator WS are 10K, 50K and 100K.

In this experiment, first the three sample web services are accessed through the lightweight WSEE (see Figure 3-7) and then the partitioned WSEE (see Figure 3-8) is used. The number of WS clients accessing a sample WS is varied from 1 to 20. It was not possible to experiment with more than 20 WS clients using the lightweight WSEE because of the Window Mobile operating system limitations on the Dell Axim PDA. Window Mobile operating system has a limit of accepting that many requests (socket connections).

Table 3-1: Variation of Sample web service

Sample WS	***Variation 1***	***Variation 2***	***Variation 3***
Image WS	ImageSize =10 KB	ImageSize=100 KB	ImageSize = 500 KB
π Calculator WS	N = 10 K	N = 50 K	N= 100 K
Tracking WS	-	-	-

HTTP is used to exchange SOAP messages between the two partitions of the SOAP engine in case of the partitioned WSEE. For the two WSEEs, the response time is measured for different values of the concurrent clients (varied from 1 to 20) and is plotted in Figure 3-7 and Figure 3-8. A comparison of the mean response times for the two versions of the WSEE is shown in Figure 3-9 for different sample web services.

Note that the graphs presented in Figure 3-7, Figure 3-8 and Figure 3-9 are plotted using a $\log_{10}$ scale for the response time (Y-axis).

In Figure 3-7, as the number of concurrent clients is increased, the response time increases as well. The mean response time observed for Tracking WS is the smallest in comparison to other sample WS because Tracking WS uses less computing resources and requires a very small amount of data to transfer. Image WS with three different sets of images (10K, 100K and 500K) is observed to show high values of the mean response time because of the network delays. As the complexity level of π Calculator WS is increased, the mean response time is observed to increase as well for both versions of WSEE.

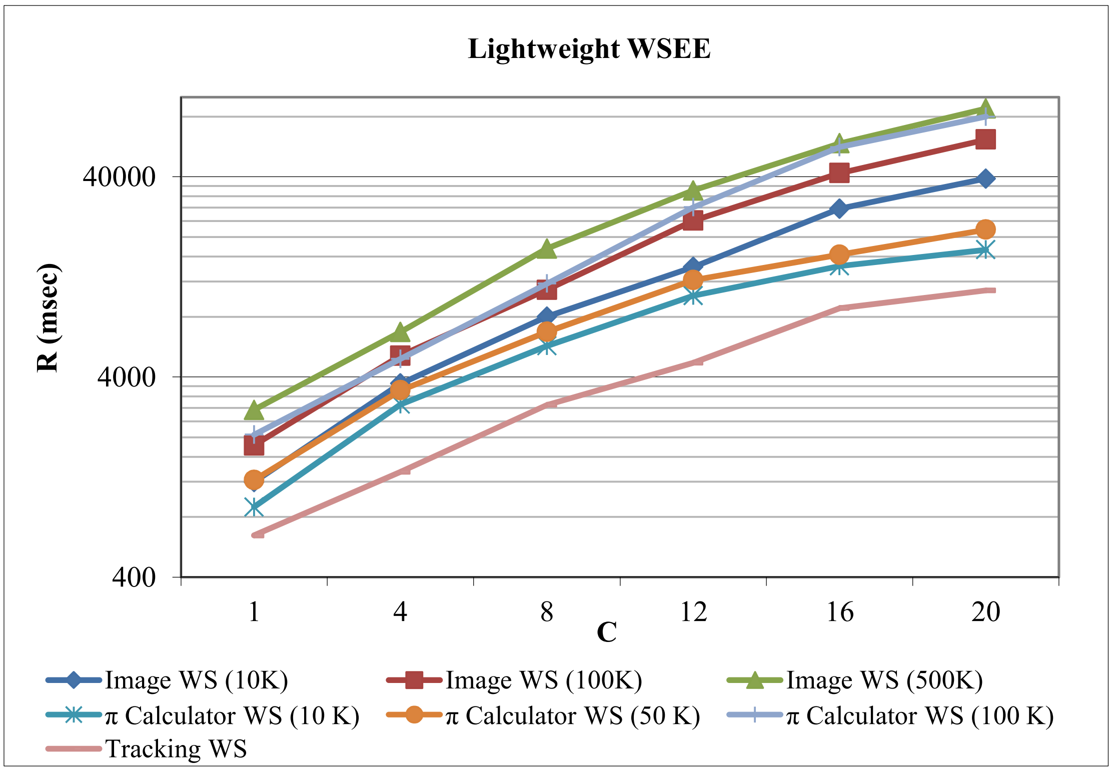


Figure 3-7: Effect of the number of concurrent clients (C) on the response time (R) using different sample web services for lightweight WSEE

For both the WSEE versions, Figure 3-9 shows that the contention for resources increases with an increase in the number of concurrent clients and results in the increased mean response time.

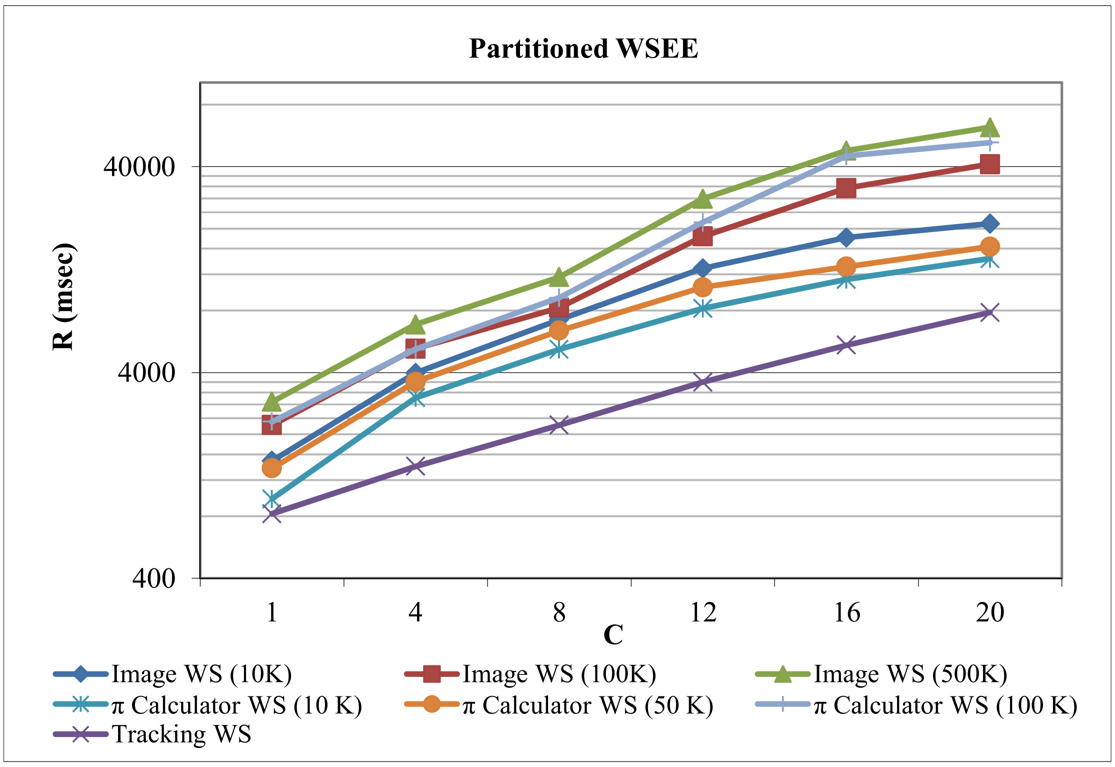


Figure 3-8: Effect of the number of concurrent clients (C) on the response time (R) using different sample web services for partitioned WSEE

The results presented in Figure 3-9 show an interesting pattern. For a higher number of concurrent clients (C > 4) the mean response time for the partitioned WSEE is significantly lower than the mean response time for the lightweight WSEE. For a lower number of concurrent clients (1 and 4, for example) the mean response time observed for the sample web services is observed to be lower for the lightweight WSEE. When a lower number of concurrent clients is used, the resource contention is not very significant. Since the lightweight WSEE uses a single direct connection with the clients, the network delays are expected to be lower in comparison to the partitioned WSEE and thus results in a relatively better performance. Another interesting behavior is observed when the number of concurrent clients is increased for both the WSEE versions. For the lightweight WSEE, the contention for resources on the mobile device increases significantly. This results in a large increase in the mean response time when the sample WSs are invoked with the lightweight WSEE in comparison to the partitioned WSEE

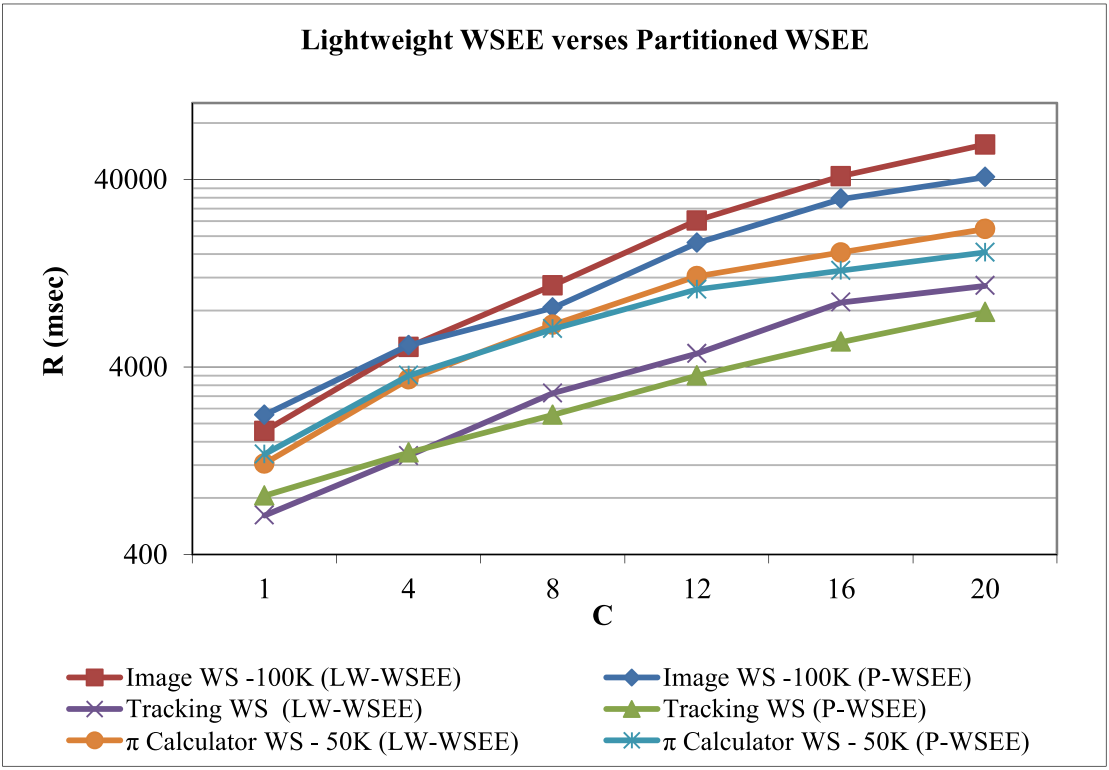


Figure 3-9: Performance comparison of the lightweight WSEE and the partitioned WSEE using Image WS with average image size of 100 K, Tracking WS and π Calculator WS using complexity level of 50K

3.4.5.1 Scalability and Effect of Transport Mechanism between Intermediate Node and Mobile node

Scalability of the partitioned WSEE is an important feature because the lightweight WSEE can support a limited number of WS clients due to resource constraints and limitations of the operating system (OS) used on the mobile device. The partitioned WSEE is designed with a feature that queues up excessive requests on part of the WSEE deployed on the intermediate node. The number of concurrent clients is varied between 1 and 50. For mobile applications, 50 concurrent clients are considered a large number and can load the system significantly.

For this experiment, an additional experiment is performed to study the effect of the transport mechanism used between the part of the partitioned WSEE deployed on the intermediate node and the part of the partitioned WSEE deployed on the mobile node. The communication between the intermediate node and the mobile node is based on SOAP messages but this communication is not visible to external world. This provides an opportunity of using a transport mechanism between the intermediate node and the mobile node that gives rise to a lower overhead in comparison to that achieved by using HTTP.

The transport mechanisms used between the intermediate node and the mobile node are HTTP and TCP (Sockets). Note that the transport mechanism used between WS clients and the intermediate node was HTTP. The experiments are performed using sample web services as described in Table 3-1. The results reported in Figure 3-10 and Figure 3-11 are only for Image WS with an image set of size 100K, Tracking WS and π Calculator WS with N= 50K. Again, the graphs presented in Figure 3-10 and Figure 3-11 are plotted using a $\log_{10}$ scale for the mean response time (Y-axis).

As indicated in Figure 3-10 and Figure 3-11, the mean response time of the partitioned WSEE (using TCP and HTTP) scales effectively with an increase in the number of clients. The performance of Image WS is inferior to the other two WSs (Tracking WS and π Calculator WS) because of the increased transport delay due to exchange of the large size of data. For Tracking WS and π Calculator WS, as the number of clients increases, the response time increases sub-linearly. Partitioned WSEE with TCP sockets is observed to be more scalable with the number of concurrent clients in comparison to the partitioned WSEE with HTTP.

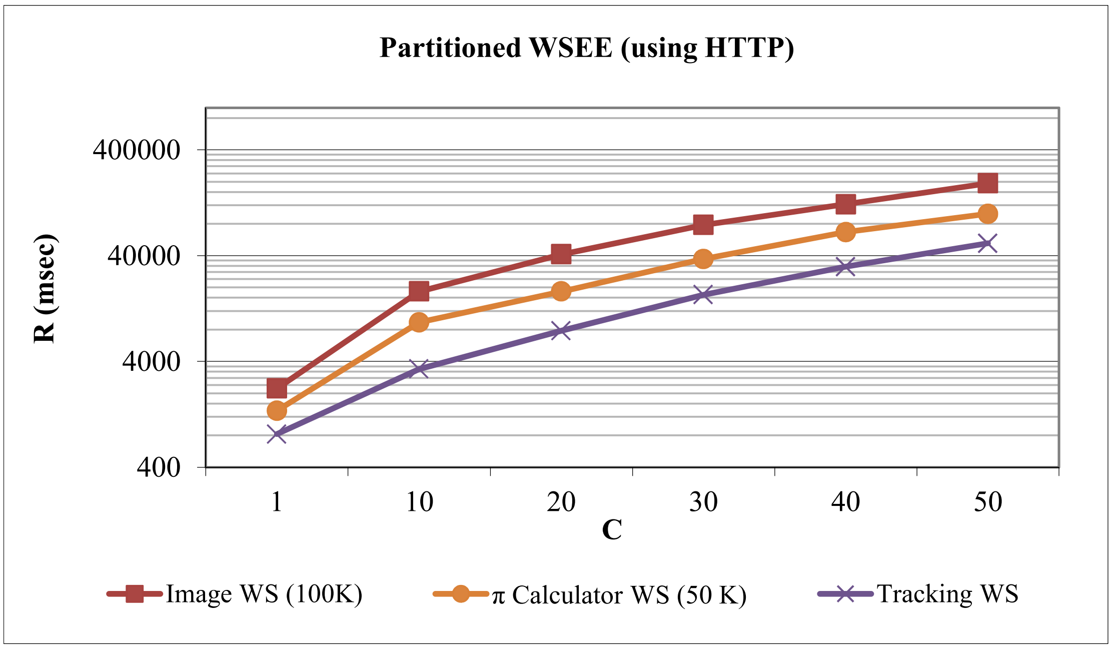


Figure 3-10: Scalability of partitioned WSEE when HTTP is used as a transport mechanism for exchanging SOAP messages between the intermediate node and the mobile node

As already mentioned the scalability of the lightweight WSEE is limited, thus the use of partitioned WSEE not only helps the scalability issue but it also exhibits a better performance in comparison to the lightweight WSEE.

This experiment also demonstrates that a heavy weight protocol such as HTTP may not be suitable for use on systems with resource constraints and having a short response time requirements. Modified versions of TCP for wireless environments are discussed in the literature [Ava02] and can also be used as a transport mechanism for exchanging SOAP messages between the intermediate node and the mobile node for further improvements of the mean response time.

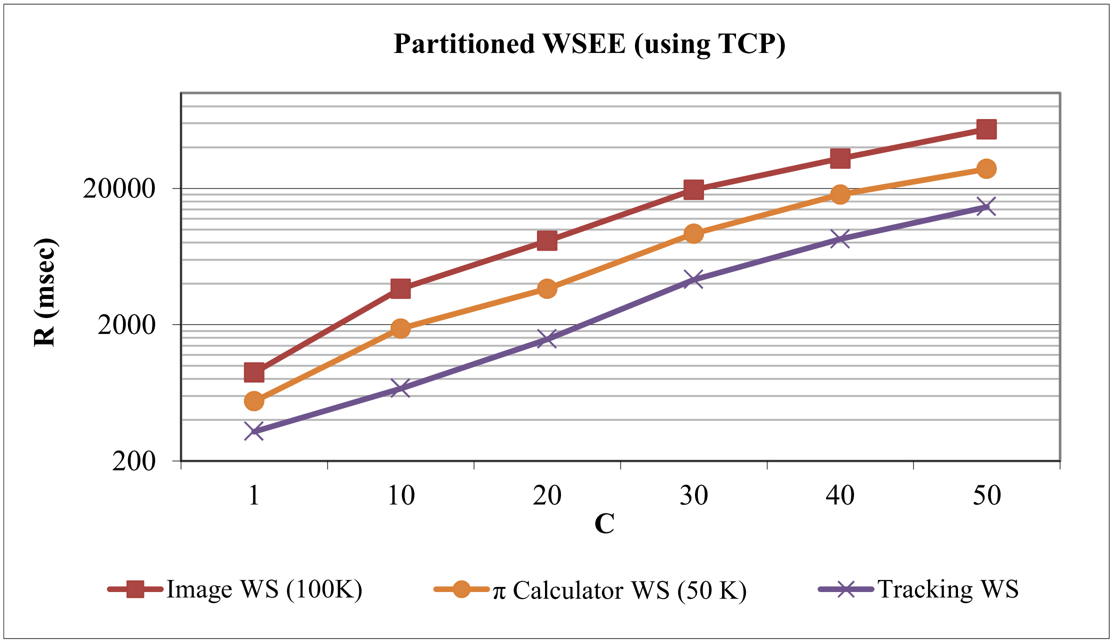


Figure 3-11: Scalability of partitioned WSEE when TCP socket is used as a transport mechanism for exchanging SOAP messages between the intermediate node and the mobile node

3.4.5.2 Effect of Number of WS Standards Used

One of the motivations for proposing the partitioned WSEE is to provide support for multiple WS standards especially those that require significant computing resources to achieve their goals. In this experiment, the sample WSs with two additional WS standards (verification of XML Signature and Signing of WS response message) are used and the performances of the lightweight WSEE and the partitioned WSEE are compared. The detailed steps of these two WS standards are discussed in Section 3.2.3. In the first part of this experiment, WSEE is required to verify the XML signatures only. The XML Signature is sent as a part of the WS request. In the second part of the experiment, the WSEEs are also responsible for signing the response message before sending it to the WS client. In case of the partitioned WSEE, the two WS standards are applied by the part of WSEE deployed on the intermediate node. This experiment is repeated using all the variations of sample web services mentioned in Table 3-1. The number of concurrent WS clients used are 1, 10 and 20. The results reported in Figure 3-12 only shows the effect of the additional WS on the response time for Tracking WS using the two versions of WSEE. A similar trend has been observed for the other sample web services.

In Figure 3-12, the first three bars in each set of results are representing the response time of invoking Tracking WS using the lightweight WSEE with no WS standard, with one WS standard (verification of XML Signature) and with two WS standard (verification of XML Signature and signing of WS response) respectively. For the lightweight WSEE, when one or more WS standards are used, the mean response time is observed to increase significantly with an increase in the number of concurrent clients. This is because the steps of verifying an XML Signature and signing a response message require execution of complex algorithms. Such algorithms need a lot of computing resources and thus results in an increased response time.

The last three bars in each set of results presented in Figure 3-12 are representing the response time of accessing Tracking WS using partitioned WSEE with no WS standard, with one WS standard (verification of XML Signature) and with two WS standards (verification of XML Signature and signing of WS response) respectively. The effect of using the two WS standards is not significant on the mean response of accessing Tracking WS when partitioned WSEE is used. The mean response with partitioned WSEE is observed to be significantly lower in comparison to the lightweight WSEE. The execution of the complex algorithms on a powerful intermediate node seems to give rise to a significant improvement in performance.

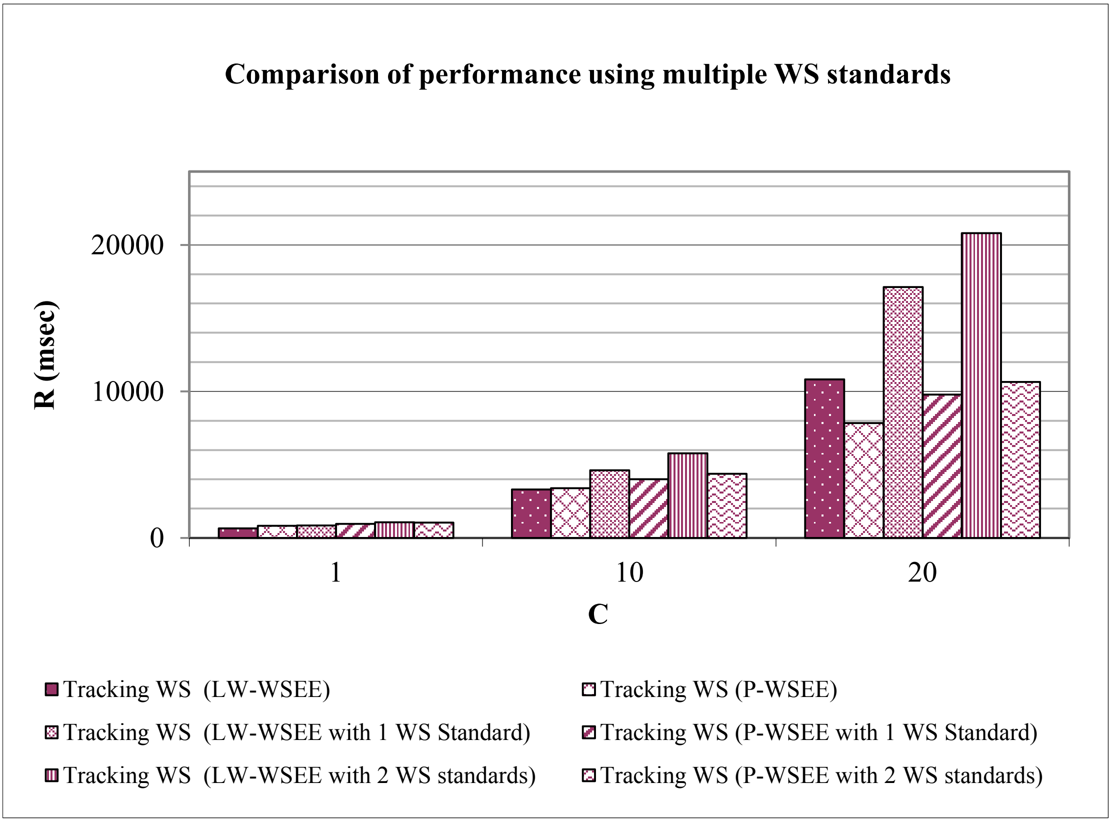


Figure 3-12: Effect of using additional WS standards when Tracking WS is invoked using the two versions of WSEE

3.4.5.3 Effect of the Speed of the Processing Resources (ω)

In this experiment, the CPU speed for the Dell Axim PDA is changed from 624 MHz to 208 MHz. Smart devices usually come with an option of operating at multiple processors speeds to conserve the battery power. This feature is utilized to emulate devices with different processing speeds. This experiment is performed using the sample web services (as mentioned in Table 3-1) deployed on the same device but operating the mobile device at two different processing speeds. As expected, the response time for the sample web services is increased when the processor speed of 208 MHz is used. The percentage increase in the response time is measured for Image

WS with an image set of size 100 K (Figure 3-13), Tracking WS (Figure 3-14) and π Calculator WS with N = 50K (Figure 3-15).

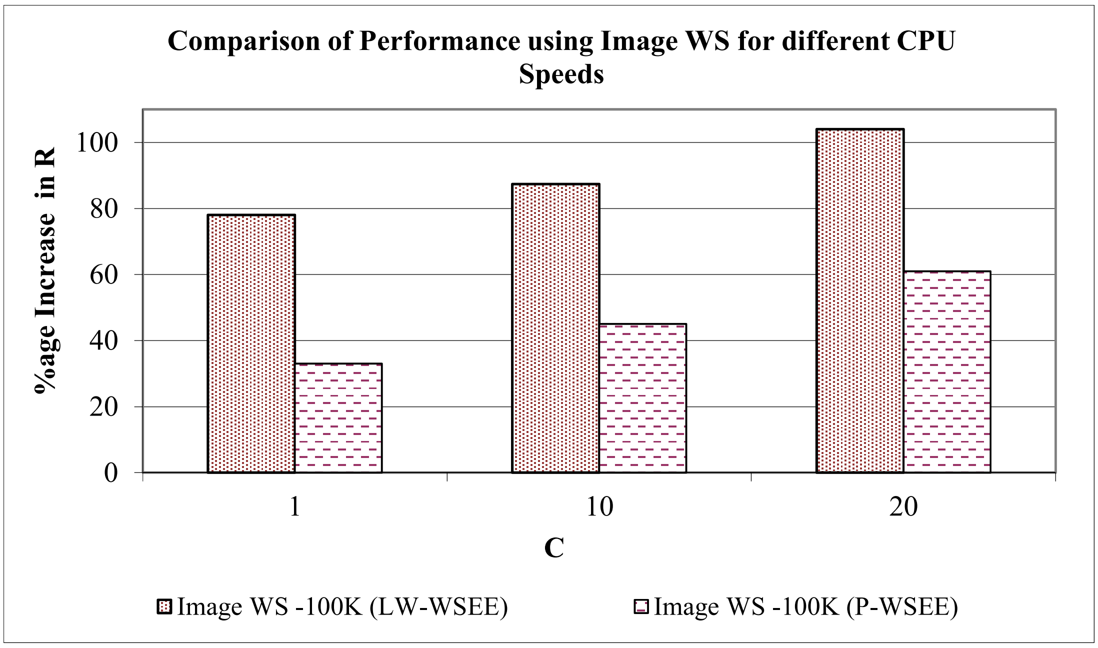


Figure 3-13: Effect of device processor speed on performance of Image WS with the lightweight WSEE and the partitioned WSEE

The increase in the response time for the lightweight WSEE is of the order of 70-80% for a single WS client (see first bar in all three graphs), 80-85 % for 10 WS clients (see third bar in all three graphs) and 90-105% for 20 WS clients (see 5th bar in all three graphs). The increase in the mean response time for the partitioned WSEE is in the range of 30% to 60% for the three sample web services using 1, 10 and 20 WS clients as indicated by the 2nd, 4th and 6th bars in all three graphs. This increase in the mean response time is small in comparison to the decrease in the processing speed by a factor of 3 (changed from 624 MHz to 208 MHz).

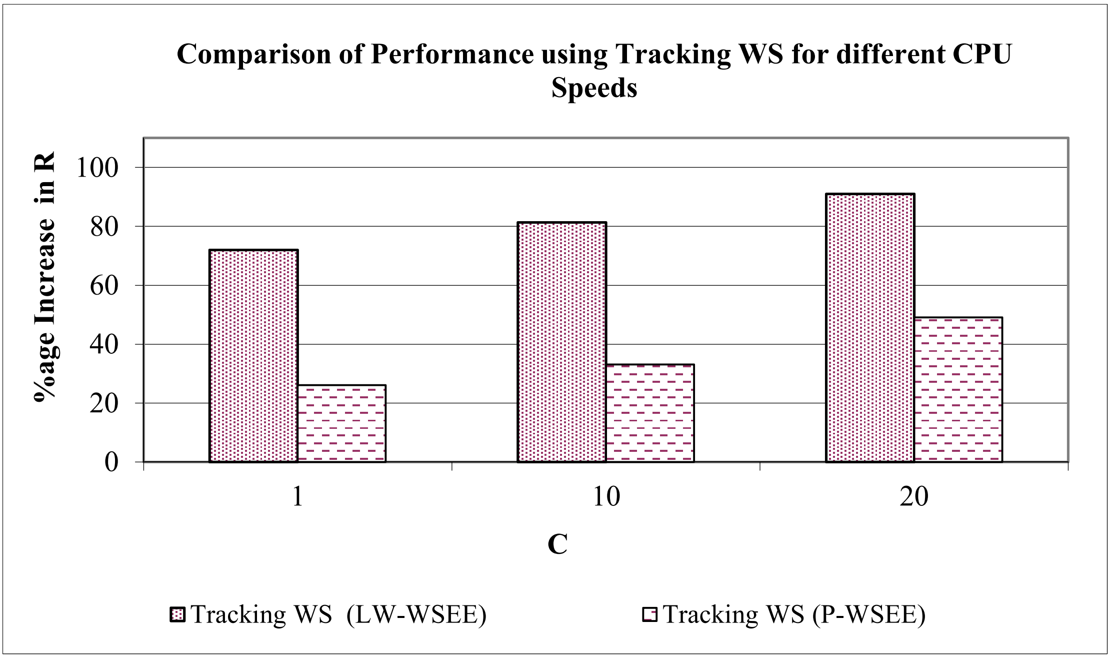


Figure 3-14: Effect of device processor speed on performance of Tracking WS with the lightweight WSEE and the partitioned WSEE

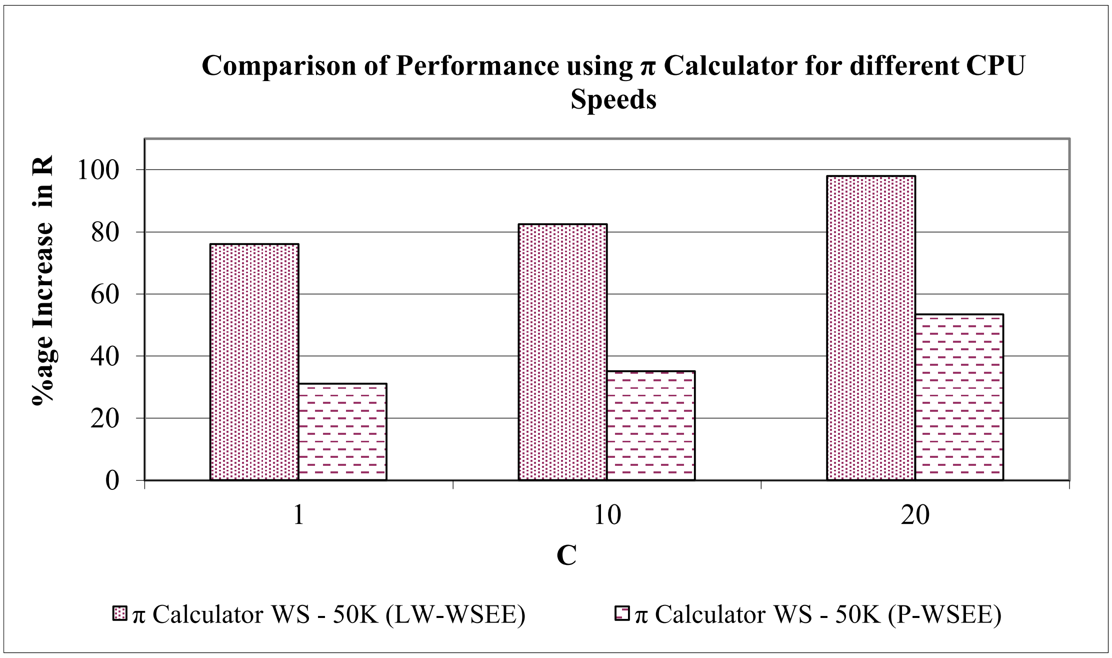


Figure 3-15: Effect of device processor speed on performance of π Calculator WS with the lightweight WSEE and the partitioned WSEE

The increase in the mean response time for the lightweight WSEE is large in comparison to the increase in the mean response time when the partitioned WSEE is used (see Figure 3-13, Figure 3-14 and Figure 3-15). In case of the lightweight WSEE, as the CPU is operated at a lower speed, the resource contention is significant because the execution of both the WSEE and the sample WSs is performed on the device. When the partitioned WSEE is used, CPU demanding tasks are executed on the intermediate node thus lowering the demand on resources of the mobile node and results in only a small increase in the mean response time.

3.5 Summary

Key characteristics of the lightweight WSEE and the partitioned WSEE are observed based on the implementation and detailed experimental analysis and are summarized next.

- The lightweight WSEE has a very small memory footprint that is less than 100K.
- For a lower number of concurrent clients and a basic set of WS standards, the lightweight WSEE is observed to be demonstrating better performance in comparison to the partitioned WSEE.
- For a larger number of concurrent clients or when additional WS standards are required, the partitioned WSEE exhibits a better performance in comparison to the lightweight WSEE.
- For mobile devices with very limited resources such as pagers and conventional mobile phones, the performance for partitioned WSEE is observed to be less sensitive with an increase in the number of clients in comparison to the lightweight WSEE.

- The partitioned WSEE is observed to be scalable with the number of concurrent clients. In comparison to using HTTP as a transport mechanism between the intermediate node and the mobile node, TCP seems to be showing a superior performance in terms of mean response time and scalability.

Chapter 4: WS Partitioning Frameworks

This chapter discusses three different frameworks for WS partitioning. Two of the three frameworks, the intermediate node based framework and the backend node based framework, have been used by researchers for deployment of partitioned systems involving mobile and conventional applications. This thesis investigates these two frameworks for mobile web services for the first time. This thesis also introduces a third partitioning framework based on a forwarding node. A performance comparison for the three partitioning frameworks is performed using system prototyping.

4.1 Overview

The complexity of a WS can vary depending on its goals. Some applications such as checking the availability of resources in a resource pool require only a few lines of programming. However in certain applications, such as image format conversion used in image processing applications, a WS can be a complex business process that may involve a number of software components and execute complex algorithms to achieve its goals. Invoking such a service for a number of times by multiple concurrent clients can lead to temporarily stopping the mobile device from performing its core functionalities such as voice services. The repeated invocation of a complex service also increases the probability of the device going out of battery power more quickly.

Hosting of such complex WS applications on mobile devices is facilitated by using WS partitioning. WS Partitioning is performed to divide a WS application into multiple components so that execution of computationally complex components can be offloaded to a remote computing node.

Hosting web services requires a WS execution environment (WSEE). If the CPU time required by a WSEE is denoted by T_{WSEE}, the CPU time required by a WS application itself by T_{AP} and the network and queuing delay by T_{ND}, then the overall response time R can be computed by the following equation

$$R \cong T_{WSEE} + T_{AP} + T_{ND} \qquad 4\text{-}1$$

T_{WSEE} includes the CPU time spent on sending and receiving SOAP messages and executing WS protocols. If a WS application can be partitioned into 'n' components where T_{P1}, T_{P2}, T_{P3} ...T_{Pn} are the CPU times required by each component, then the total CPU time required by the WS application will be

$$T_{AP} = T_{P1} + T_{P2} + T_{P3} + \ldots. + T_{Pn} = \sum_{i=1}^{n} T_{Pi} \qquad 4\text{-}2$$

Equation 4-2 only considers CPU time of a WS application while running all partitions sequentially on a single node. If we group WS partitions into two sets: one set for partitions that are to be executed on a mobile device and the other set for offloaded partitions that are to be executed on a remote node, then equation 4-2 can be written

$$T_{AP} = \sum_{i=1}^{m} T_{Pi} + \sum_{j=m+1}^{n} T'_{Pj} \qquad 4\text{-}3$$

where i = 1, 2… m are partitions executed locally on a mobile node and j= m+1, m+2,…n are the partitions that are to be executed remotely. Running mobile WS partitions on multiple nodes adds two overheads: the overhead corresponding to the CPU time required for coordination of different WS partitions and the overhead corresponding to the CPU time required for exchange of application data between the different WS partitions. If we denote the two overheads by δ_{coord} *and* δ_{comm} respectively, then equation 4-3 can be rewritten as

$$T_{AP} = \delta_{coord} + \delta_{comm} + \sum_{i=1}^{m} T_{Pi} + \sum_{j=m+1}^{n} T'_{Pj} \qquad 4\text{-}4$$

The overall response time of a WS application can be computed by combining equations (4-1) and (4-4)

$$R \cong T_{WSEE} + T_{ND} + \delta_{coord} + \delta_{comm} + \sum_{i=1}^{m} T_{Pi} + \sum_{j=m+1}^{n} T'_{Pj} \qquad 4\text{-}5$$

Equation (4-5) gives an estimation of the time required to execute a WS that is partitioned across two sets. Note that in this research, the executions of partitions on a mobile node and on a remote node are not concurrent.

In this chapter, an analysis of deploying of the offloaded WS partitions using three types of partitioning frameworks is presented. The first framework uses an intermediate node to intercept the requests (from a WS client to a mobile device) and processes it partly before forwarding it to the WS provider running on the mobile device. The second framework is based on a backend node for execution of offloaded parts of an application. The backend node is used to execute offloaded partitions of an application on request for the WS provider (mobile node) and send the result back to the WS provider that uses it to satisfy the client request. These two frameworks have been used by different researchers (see [Cha02], [Mes02], [Ou07], [Sri07-1] and [Sri07-2] for example) for offloading partitions of conventional applications to additional nodes but none of the existing research has investigated their effectiveness in the context of mobile WSs. The third framework uses a forwarding node. The forwarding node is used to execute offloaded partitions of an application, aggregate overall results and forward the response to the WS client instead of sending the result back the mobile device. The implementation of the prototypes for each of the framework is based on Java Standard Edition (Java SE) and Java Micro Edition (Java ME). A detailed description is provided in the next section.

4.2 Mobile WS Partitioning Frameworks

WS partitioning can be achieved with a backend or an intermediate node. An important objective of application partitioning is that the partitioning of the WS should be transparent to the WS client and the execution of the partitioned application is experienced as if it were running on a single node. In this section, a brief discussion of the three proposed partitioning frameworks for mobile web services is presented. It is important to note that for all the frameworks discussed next, the same WSEE is deployed on all types of nodes (mobile node or the intermediate or the backend node). Different partitions of sample web services are executed on different nodes.

The proposed frameworks can handle both wireless and wired WS clients. Only wired devices are used in the experiments reported in Section 4.6. Since the delay between the client and a framework is expected to be the same for all the three frameworks, the performance ranking of the different frameworks on systems with wireless clients is expected to remain the same as that reported in this thesis.

4.2.1 Intermediate Node based Framework

For an intermediate node based framework (referred to as intermediate framework), an intermediate node on a wired network works both as a service proxy and as a surrogate node. In the context of the work presented in this thesis, a surrogate node is the one that performs execution of code on behalf of another node. For the client, a service seems to be hosted on the intermediate node. A high level overview of this framework is shown in Figure 4-1.

WS clients send their requests to the intermediate node. On receiving the WS requests, the intermediate node parses the parameters (if any) and executes one or more partitions of a WS application locally and offload execution of the other parts of the application to a mobile node through a wireless network. The execution of a part or parts of a WS application locally or on a mobile device is based on a partitioning plan. A workflow language such as Business Process Execution Language (BPEL) [BPE03] can be used to define the order of execution of the different partitions.

In addition to coordinating the execution of the different partitions of a WS application, the intermediate node is also responsible for aggregating results (a final response) and sending it to the WS client. The total number of messages exchanged among the three nodes (the WS client, the intermediate node and the WS provider) is four (see Figure 4-1).

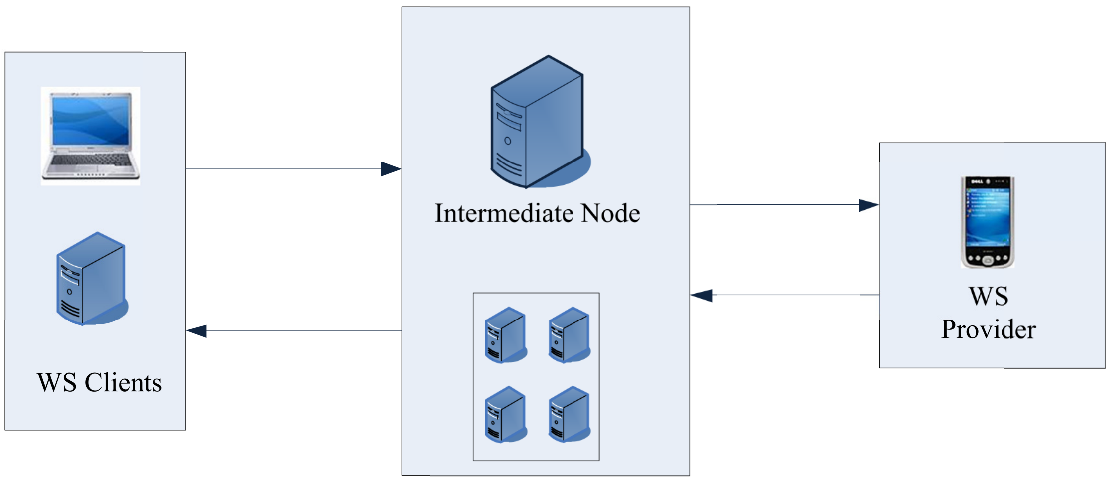


Figure 4-1: The intermediate framework for WS partitioning

With this framework, majority of the WSEE components can be run on a powerful intermediate node. By analyzing equation (4-5) in reference to this framework, T_{WSEE}

is expected to be lower in comparison to a situation in which the WS is fully hosted on a resource constrained mobile node. Since the coordination of the different partitions of the WS application is also performed on the intermediate node, δ_{coord} is expected to be lower in comparison to a situation in which the coordination is performed on a mobile node.

The use of an intermediate node as a service proxy ensures availability of the service all the time in comparison to hosting the service directly on a mobile device. This is because mobile devices occasionally drop connections or may not be accessible when they are operating in an area with non-existent or weak wireless signals. The processing of SOAP/XML to support different complex WS standards can be performed on the intermediate node, thus reducing the resource requirements of the mobile device. This framework is suitable for design time application partitioning strategies. This is because WS requests are received directly by the intermediate node and the responsibilities of the intermediate node need to be determined in advance. In this framework, the control of the execution flow for the WS application is delegated to the intermediate node. The mobile device owner, however, has to trust the intermediate node for the exchange of the application data. The applications that involve exchange of personal data or require access to local resources of the mobile devices more often are not good candidates for using this partitioning framework.

4.2.2 Backend Node based Framework

A backend node based framework (referred to as a backend framework) does not involve any intermediate node between a WS client and a WS provider. Instead, a WS client request is directly received by a resource constrained mobile device. The interactions of the different components in this framework are shown in Figure 4-2.

On receiving a WS request, a WS application on the mobile device runs a set of the partitions of the requested WS locally as determined by the execution plan and then offloads the execution of the rest of the WS application to the backend node. As shown in Figure 4-2, it is the responsibility of the mobile device to collect the results from the remote WS partitions running on the backend node, aggregate the results and send the final response back to the WS client. As shown in Figure 4-2, the total number of messages exchanged between the three nodes is four.

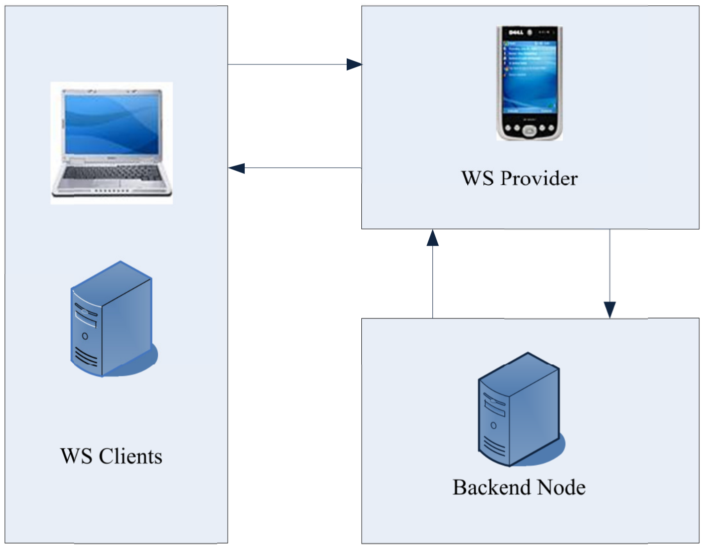


Figure 4-2: The backend framework for WS partitioning

The mobile devices typically come with CPUs that have a very limited processing power. Since the mobile device itself is responsible for receiving the WS requests directly, executing a number of the WSEE components locally, coordinating the executing of the different WS partitions and supporting the exchange of data among different WS partitions, the values of T_{WSEE}, δ_{coord} and δ_{comm} are expected to be higher for this framework in comparison to the intermediate framework. The factor that can contribute to improve the performance of this framework is the large value of $\sum_{i=1}^{n} T_{pi} - (\sum_{i=1}^{m} T_{pi} + \sum_{j=m+1}^{n} T'_{pj})$. It can be achieved either by using a powerful backend node (that will lower the value of $\sum_{j=m+1}^{n} T'_{pj}$) or when more parts of the WS application can be offloaded to the backend node.

Regardless of its performance, this framework is suitable for WS applications that require the control of the application to be on the mobile device. Since the WS requests are received directly by the mobile device, this framework can be used either with a design time partitioning or with a runtime application partitioning approach.

4.2.3 Forwarding Node based Framework

The objective of this framework is to alleviate the drawbacks of obtaining the results from the backend node and then forwarding them (after combining the results of a local execution if any) to the WS client. In this forwarding node based framework (referred to as the forwarding framework), local partitions are executed on the mobile device first and then the results are forwarded to a backend surrogate node that executes the remaining partitions. The responsibility of sending the final WS response to a WS client is delegated to the forwarding node as well. A high level description of the interaction of the different components of this framework is presented in Figure 4-3.

In comparison to the backend framework, this framework is expected to experience a lower overhead corresponding to the coordination of the execution of different WS partitions (δ_{coord}) and the exchange and processing of the data between the mobile node and the backend node (δ_{comm}). It is because the task of collecting the results from the backend node is eliminated and a lower number of messages are exchanged between the three nodes.

A major difference between this framework and the previously discussed frameworks is that the total number of messages exchanged between the three nodes is only three. The forwarding framework allows applications to keep the control of application flow on the mobile device and reduces overheads by delegating the task of aggregating results and sending the consolidated response back to the WS client to the backend node. This framework is suitable only for such applications in which WS partition on the mobile device is to be executed first.

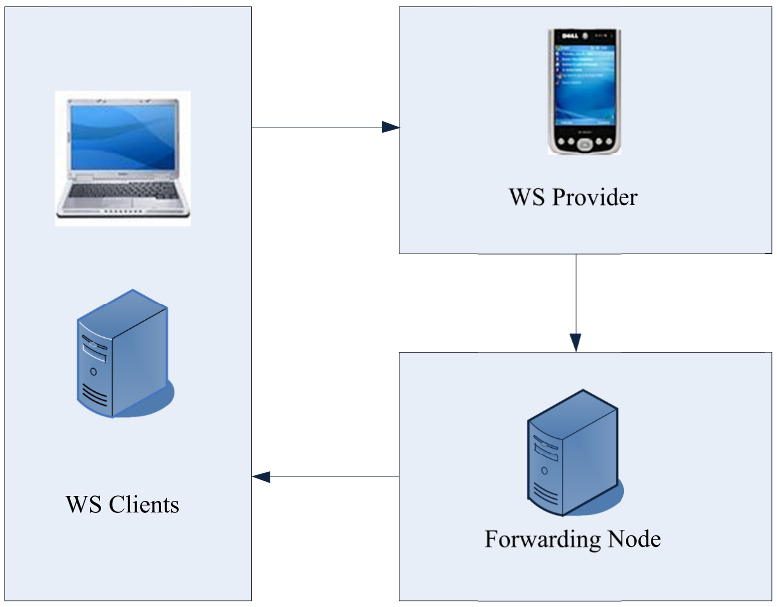


Figure 4-3: The forwarding framework for WS partitioning

A prototype of each of the three frameworks is implemented and an in-depth analysis of their performances is performed. The design and implementation of the prototypes are presented in Section 4.3 while a discussion of their performance is covered in the following sections. Note that in case of the forwarding framework, the mobile device (WS provider) also forwards the IP address and the TCP port number on which the WS client is waiting for a response from the forwarding backend node.

4.3 System Design and Prototype Implementation

To analyze the performance of the three partitioning frameworks discussed in the previous section, a prototype of each of the frameworks using a lightweight WS execution environment for hosting the different partitions of a WS application is built. To coordinate the execution of the different partitions of the WS application that are to be executed on multiple nodes, a partition coordination subsystem is introduced. A high level overview of the system that includes a WSEE and a partition coordination subsystem is captured in Figure 4-4. The different components of Figure 4-4 are discussed next.

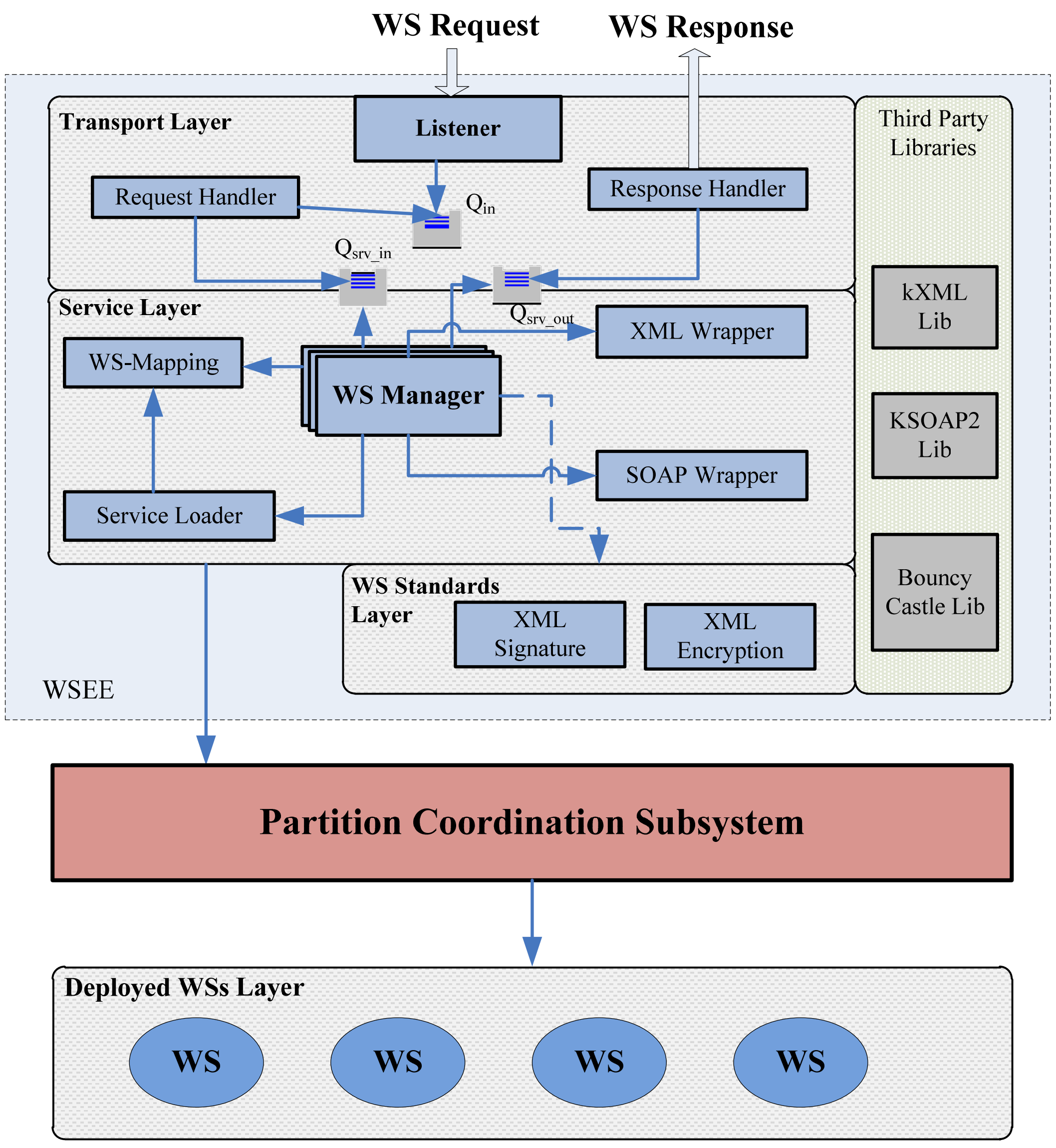


Figure 4-4: High level architecture of the system for hosting of partitioned mobile web services

4.3.1 Web Service Execution Environment

Since this work is focused on the investigation of WS partitioning, the lightweight WSEE is used for handling of WS requests and WS responses. The same lightweight WSEE is used on the intermediate/backend node and on the mobile node for the three partitioning frameworks described in the last section. The functional details of the components of the lightweight WSEE were presented in Section 3.2. The partitioned coordination systems subsystem is described next.

4.3.2 Partition Coordination Subsystem

The partition coordination subsystem is responsible for coordinating the execution of the different partitions of a WS application. For a performance analysis of the three partitioning frameworks, a design time WS application partitioning strategy is used. The design time partitioning strategy is simple to implement for all the three partitioning frameworks and is useful for studying the effect of different workload and system parameters on the overall system performance. In the design time partitioning strategy, the partitioning plan that describes which partitions to run locally on a mobile device and which partitions to execute remotely is provided through an XML document for every deployed WS. At the time of starting the WSEE, the partitioning plan for every deployed WS is loaded into the system memory of each node. For each implementation of the three frameworks, the partition coordination subsystem is deployed on the intermediate node (for the intermediate framework) or on the mobile device node (for the backend and the forwarding frameworks). The WS partitions on remote nodes are deployed as child web services. The WS partitions deployed on a local node are invoked directly. The different components of the partitioning coordination subsystem are shown in Figure 4-5.

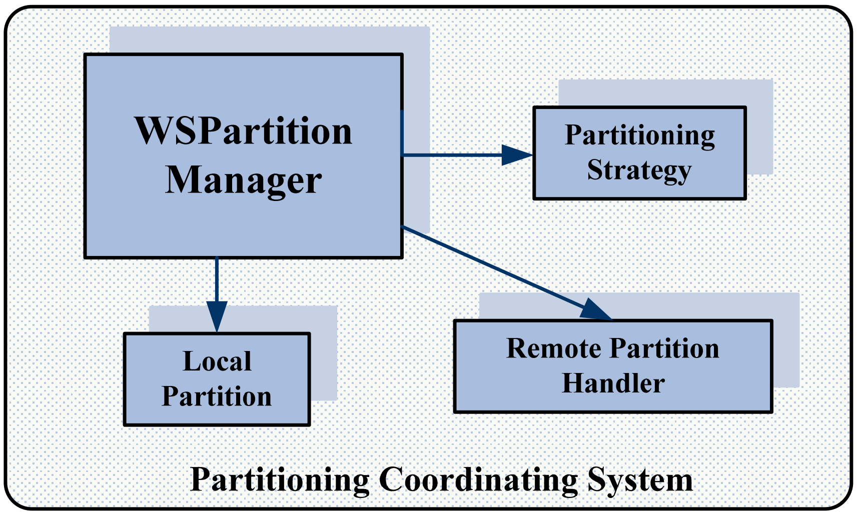


Figure 4-5: Internal details of the Partition Coordinating Subsystem

The WS partition manager is a key component of the partition coordination subsystem. It is an entry point for a WS and it facilitates the execution of the local partition(s) and the remote partition(s) according to the WS partitioning plan. The remote partition handler is responsible for communicating with the remote node (the mobile node in case of the intermediate framework or the backend node in case of the backend and the forwarding frameworks) for execution of the offloaded WS partitions.

A WS can also be partitioned into more than two components. This type of partitioning is only useful when offloaded partitions are targeted to run on multiple remote computing nodes. Since the partitioning frameworks discussed in Section 4.2 use only one remote computing node, only two partitions are considered: P1 and P2 (P1 running on a mobile device and P2 running on a surrogate node (the intermediate node or the backend node). For a system with two partitions, equation (4-5) reduces to

$$R \cong T_{wsee} + T_{ND} + \delta_{coord} + \delta_{comm} + T_{P1} + T'_{P2} \qquad 4\text{-}6$$

4.4 Sample Web Services and Experimental Setup

For experimental analysis, prototypes for the three partitioning frameworks are implemented. The WSEE used (the lightweight WSEE) for all the three frameworks is the same. The sample web services used are described next.

4.4.1 Sample Web Services

The four web services experimented with are Image WS, Tracking WS, NavigateMe WS, and π Calculator WS. The details of these web services are presented next.

4.4.1.1 Image WS

Image WS involves search to a local database of images, a resource demanding algorithm for resizing of a selected image and exchange of a large volume of data across the three nodes. This service fetches image data (may be resized) from a mobile device based on a set of input parameters: location and image size. A detailed description of this service is presented in Section 3.4.1.1.

4.4.1.2 Tracking WS

Tracking WS involves access to a local hardware resource for retrieval of GPS coordinates and querying of a database of considerable size. Tracking WS does not involve a large data exchange across the three nodes. A detailed description of this service is presented in Section 3.4.1.2.

4.4.1.3 NavigateMe WS

This WS is very similar to Tracking WS and provides directions for reaching the mobile device owner to the WS client. Such a WS is useful for WS clients with no GPS facility and hence no maps are available on their devices. The WS client can provide its location in terms of a nearby point of interest (POI) as an input parameter.

For the evaluation of this WS, grid maps of two different sizes (20x20 and 40x40) are used. For an AxA grid map, A^2 points of interest (POIs) are connected in a grid form. Such a grid map can model a neighborhood of interest in a city. Although a simple model is used, the lack of details is not expected to affect the relative performance of partitioning frameworks. This WS is chosen because, unlike Tracking WS, it can give rise to different execution times and different volumes of output data in multiple invocations.

The distances between the connected POIs are assigned using a uniform random number generator. The lower and upper bounds used for the uniform random number are 20 units and 100 units respectively. Note that the relative performances of the partitioning frameworks are not expected to be dependent on the specific values of the upper and lower bounds and the type of unit used.

This WS has four key components: *GetSrcCoordinates*, *GetDestCoordinates*, *FindDirections* and *PackData*. *GetSrcCoordinates* is used to get the input parameter and translate it to a closest point of interest on the grid map. *GetDestCoordinate* fetches the GPS coordinates of the destination device and maps them to the nearest POI on the grid map. *FindDirections* applies Dijkstra's algorithm for finding the shortest path from the source POI to the destination POI. The Dijkstra algorithm is chosen for this experimentation because of its wide use in the determination of the shortest path between two nodes. Once the shortest path is determined, the directions to reach the destination are serialized by *PackData* in a response message which is sent to the WS client. Since the WS is invoked with different input parameters, different execution times and different volumes of output data are produced in each invocation.

4.4.1.4 π Calculator WS

A brief discussion of π Calculator WS is presented in Section 3.4.1.3. The calculation of π is used for a detailed analysis of the three partitioning frameworks because its computation is easy to partition and its computational intensity can be varied conveniently for performance analysis. This is appropriate for the nature of the research presented in this chapter that focuses on the relative performances of the three partitioning frameworks.

The value of π is calculated using Gregory-Leibniz series [Wel86]

$$\pi = 4 * \sum_{r=1}^{\infty} \frac{(-1)^{r+1}}{2r-1}$$

The accuracy of π depends on the number of terms used in the summation of the series. More the number of terms used, the more accurate is the value of π. The two input parameters of this WS that are varied to study the performance of the three partitioning frameworks are described next.

Computational Intensity (N):

The number of terms (N) used to calculate π represents computational intensity. The mean value of N is varied from 10, 100, 1000, 10000 and 100000.

Different values of N in experiments are used to vary the computational intensity of this sample WS. A random number of terms is used in the computation of π in a single experiment. The Gaussian random number is generated using the value of N as a mean value and 0.1*N as its standard deviation. Using a Gaussian distribution instead of a fixed value of N introduces randomness in the computation intensity used in the experiments. The nature of the distribution used is not expected to change the relative performance of the frameworks presented in this chapter. The partitioning of π Calculator WS is achieved by splitting the calculation of term $\{(-1)^{r+1} /(2r-1)\}$ across two or more nodes.

Offloaded Partition Size (O):

It represents the part of the WS application (in terms of the number of terms used for calculation of π) that is offloaded from a mobile device to a remote surrogate (intermediate or backend) node for execution. For example, if the total number of terms to be used for calculation of π is N and the value of offloaded partition size (O) is X, then the computation of N *X/100 terms is offloaded to a remote surrogate node and the computation of remaining (N - N *X/100) terms is performed on the mobile device. This parameter is varied from 0% (no partitioning), 5%, 25%, 50%, 75% and 95%.

In addition to these two parameters, the size of the data exchanged between the two partitions of this sample WS is also varied. The impact of operating the mobile device at two different processing speeds (208 MHz and 624 MHz) is also investigated.

4.5 Experimental Setup

The WS client programs used to invoke the sample web services are written in standard Java. The WS clients are run on a desktop machine (running Windows XP) equipped with 1 GB of RAM and an Intel Core 2 processor with a speed of 2.4 GHz. The intermediate/ backend node is a desktop computer equipped with a Quad Core 2 processor that is running under Ubuntu 7.1 (Linux) operating system. Its CPU speed is 2.4 GHz and 2GB of RAM is available on this node. A Dell Axim x51v PDA is used as the mobile device. The PDA used as the mobile device has an Intel XScale ARM processor (PXA270) that can be run at multiple speeds (208 MHz, 520 MHz and 624 MHz). The PDA used is equipped with a RAM of 64 MB and is running under Windows Mobile 5.0 operating system. The Java ME environment (J9) available on the PDA is a Java Virtual Machine (JVM) provided by IBM for Java ME CDC devices. The client machine and the intermediate/backend node communicate with the mobile node (PDA) using a wireless local area network. The Axim PDA is equipped with wireless adaptors that use the IEEE 802.11-b standard.

4.5.1 Workload and System Parameters

To analyze the performance of the three WS partitioning frameworks, the following workload and system parameters are varied.

4.5.1.1 WS Complexity:

Performance of the partitioned WSEE is observed by invoking different sample WSs that are discusses earlier. Note that the sample WSs are characterized by different levels of computational complexity. In addition to that, each sample WS is invoked using different values of input parameters that vary the computational complexity of the sample WSs. For example, the π calculator service can be invoked using different values of number of terms (N) (for calculation of π).

4.5.1.2 Concurrent WS Clients (C):

The number of WS clients invoking the sample web services at the same time can be varied to investigate the scalability of the system.

4.5.1.3 Mobile Device Speed (ω):

The experiments are run on real handheld devices of different processing speeds. The Dell Axim PDA device used in our experiments can be run with different processing speeds.

4.5.2 Performance Metrics

The main performance metric of interest is the ***mean response time (R)***. Response time is defined as the difference between the time when a WS response is received by the WS client and the time when the WS client sends the corresponding SOAP message request to a WS provider.

4.6 Experimental Results

A number of experiments are carried out to investigate the performance of the three partitioning frameworks. These are listed next.

- In the first set of experiment, Image WS is used as a sample WS for performance comparison of the three partitioning frameworks.
- In the second set of experiments, Tracking WS is used.
- A performance analysis of an algorithm for finding the shortest path between two points of interest is presented in the third set of experiments. In the third set of experiments, NaviageMe WS is used as a sample WS.
- A more detailed performance analysis of the three partitioning frameworks is performed using π Calculator WS. π Calculator WS is used for a detailed analysis of the three partitioning frameworks because of the flexibility it provides for varying its computational intensity. Such a WS providing the facility for varying CPU execution time, an important parameter in our experiments, is appropriate for the nature of the research presented in this chapter.

The performance results are presented in the form of graphs. One parameter is changed at a time and is shown on the horizontal axis of each graph. The fixed factors of each experiment are shown in the caption of each figure. In all the experiments, only one type of a WS partitioning plan is used at a time. A closed system model is used for all the experiments. Each client operates cyclically and sends one request at a time. As soon as the response is received, the client repeats the cycle.

For the results presented in this chapter, each client generates 10000 requests. So, for an experiment with 12 concurrent WS clients, for example, the response time is calculated by taking the mean of response times of 120,000 requests. Each experiment is repeated 15-30 times to obtain sufficiently small confidence intervals for the mean values. For the experimental results presented next, confidence intervals of ±5% (or less) for mean response time were obtained at a confidence level of 95%.

4.6.1 Performance Comparison of the Three Frameworks using Image WS

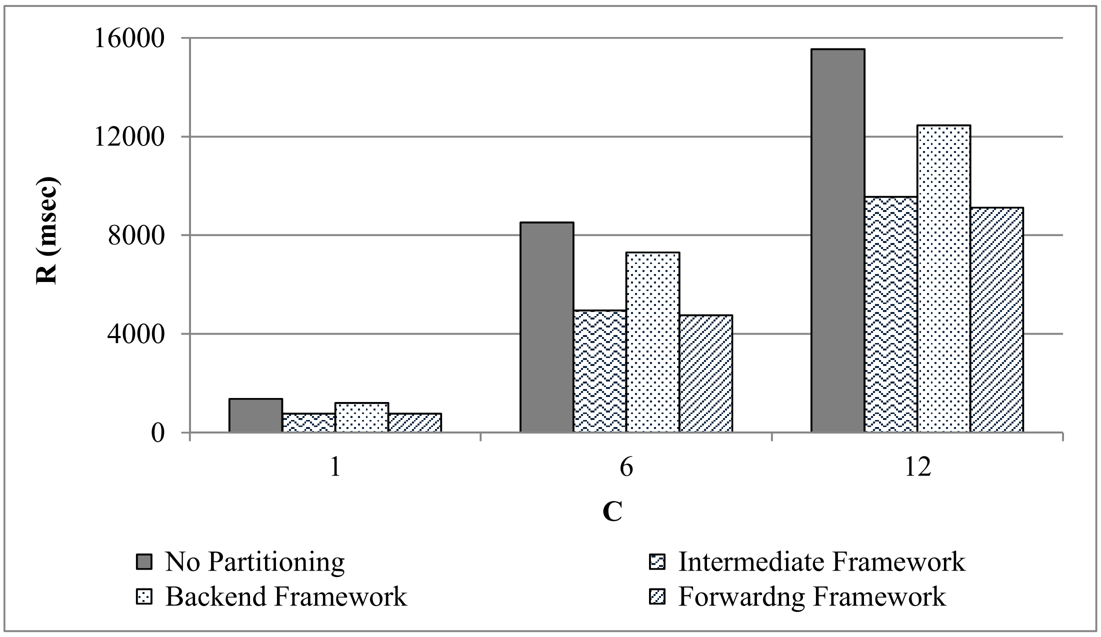
For this experiment, Image WS is partitioned into two partitions using the design time application partitioning. The *SearchImage* component is executed on the mobile node while the executions of the other two components (*ResizeImage* and *PackData*) are offloaded to a remote surrogate node (the intermediate or the backend node). The rationale for this design time partitioning is to keep the execution of components that have dependencies on local resources on the mobile device and offload the execution of components that have significant resource requirements to a remote surrogate node.

The workload and system parameters for this set of experiments are presented in Table 4-1. For this experiment, 100 images of size 1600x1200 (in pixels) are stored on the mobile device. The values used for the image size parameter of Image WS are 800x600 and 640x480. The mean response time is measured by invoking the WS with one, six and twelve WS clients. The results are shown in Figure 4-6 and the bars in the set of bars for each value of C are presented in the following sequence. The first bar corresponds to no partitioning case, the second bar to the intermediate framework, the third bar to the backend framework and the fourth bar to the forwarding framework. The image to be processed is selected randomly using a uniform distribution. A similar trend in relative performances of the three partitioning frameworks is observed for the two values of the image size parameter.

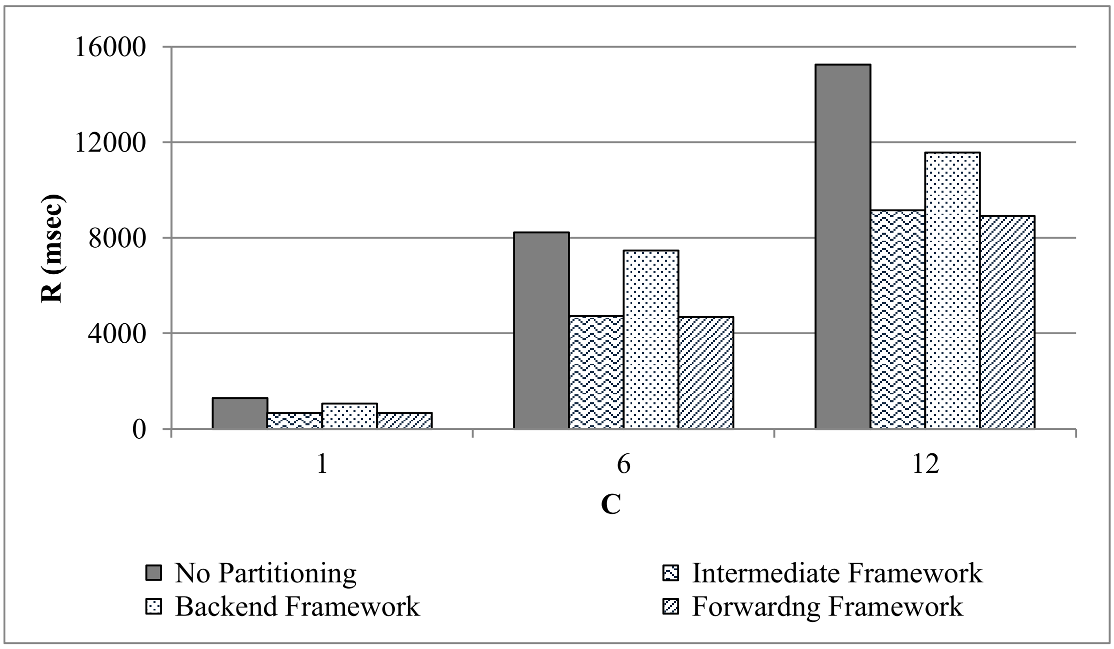
Table 4-1: Workload and system parameters for performance comparison of the three frameworks using Image WS

Parameter	***Range of Values***
Number of WS clients	1, 6, 12
Image Sizes	800x600 and 640x480
Number of Images used	100
Mobile Device Speed	624 MHz

The forwarding framework exhibits a slightly better performance in comparison to the intermediate framework especially when six and twelve WS clients are used. Note that the performance of the forwarding and the intermediate node frameworks is observed to be superior to the performance of the backend framework. The performance of the backend framework is observed to be marginally better than the no partitioning case when Image WS is invoked with only one WS client. But it is interesting to note that its relative performance improves considerably when Image WS is invoked for more than one WS client (six and twelve WS clients in this experiment). This is because the increase in mobile resource contention for higher number of concurrent clients offsets the large overheads of executing a partition on a backend node.



(a)



(b)

Figure 4-6: Performance comparison of the three partitioning frameworks when the mobile device is operated at 624 MHz and (a) image size = 800x600 (b) image size = 640x480

4.6.2 Performance Comparison of the Three Frameworks using Tracking WS

Image WS involves a significant amount of data exchange. The forwarding framework uses the lowest number of messages (three) exchanged between the three nodes, thus resulting in small network delays in comparison to the intermediate framework which leads to four message exchanges between the three nodes.

For this experiment, Tracking WS is partitioned into two partitions. The components (*GetCoordinate,* for example) with dependencies on local resources of a mobile device are put in one partition. The rest of the components (*GetLocDetail* and *PackData*) are grouped in another partition and are executed on a remote surrogate node (the intermediate or the backend node). The mean response time is measured by invoking the WS with one, six and twelve WS clients. The results are shown in Figure 4-7 and the bars in the set of bars for each value of C are presented in the following sequence.

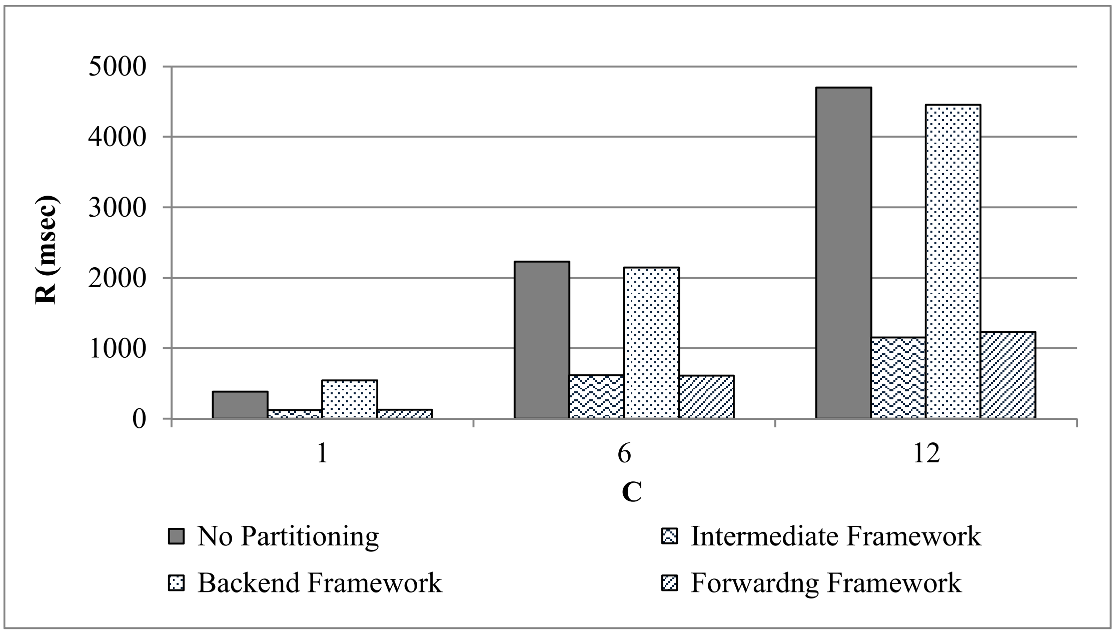


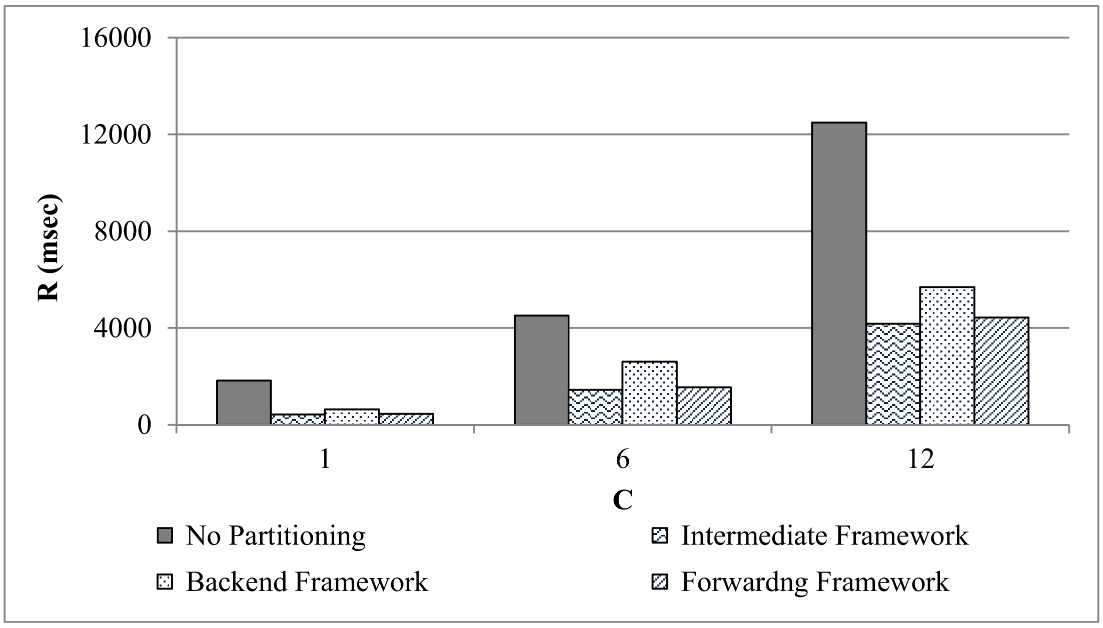
Figure 4-7: Performance comparison of the three partitioning frameworks when the mobile device is operated at 624 MHz and Tracking WS is invoked

It is interesting to note that the relative performance of the three partitioning frameworks is the same as observed for Image WS. Once again, the performance of the forwarding and the intermediate node framework is significantly superior to the performance of the backend framework. The performance of the backend framework is observed to be slightly better than the no partitioning case when Tracking WS is invoked by six and twelve WS clients. As discussed in Section 4.2.2, the mobile node in the backend framework is responsible for handing the message exchanges with WS clients and also coordinating execution of the different components of a partition. This results in a poor performance exhibited by the backend framework in comparison to the other partitioning frameworks. When a higher number of WS clients is used, the overheads accrued due to handing of message exchanges and the coordination of WS partitions are compensated by the offloading executions of more partitions and thus resulting in a slightly better performance than the no partitioning case.

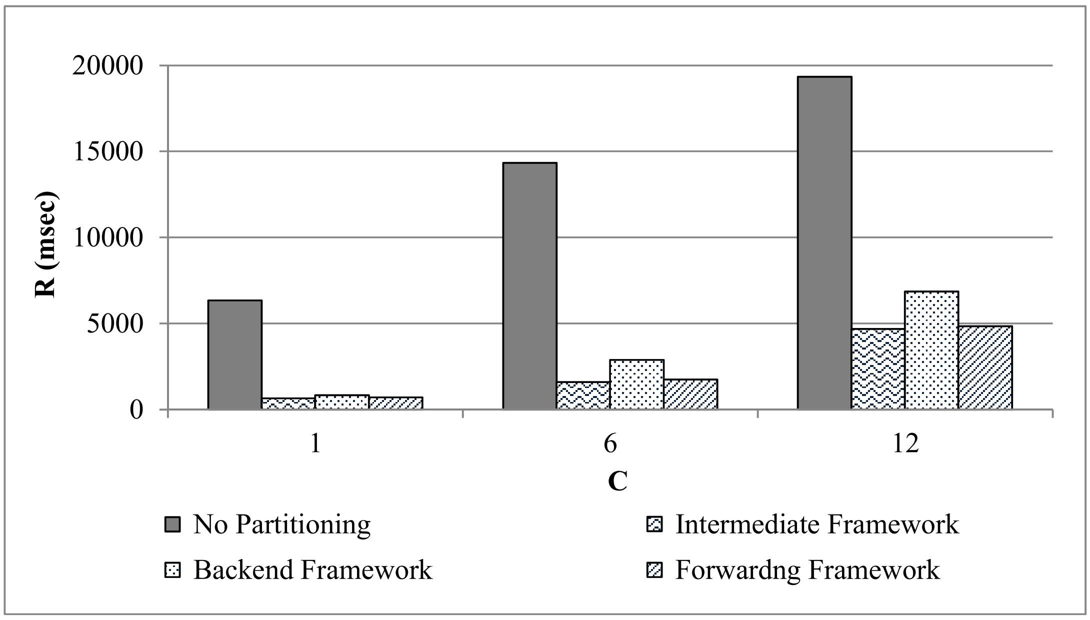
4.6.3 Performance Comparison of the Three Frameworks using NavigateMe WS

For this set of experiment, NavigateMe WS is partitioned into two partitions. The components (*GetSrcCoordinates* and *GetDestCoordinates,* for example) with dependencies on local resources of a mobile device are put in one partition. Rests of the components (*FindDirections* and *PackData*) are grouped into another partition that is executed on a remote surrogate node (the intermediate or the backend node).

The map data is assumed to be available on all nodes except the WS client device. The mean response time is measured using grid maps of sizes 20x20 and 40x40. The number of concurrent WS clients used is one, six and twelve. The results are shown in Figure 4-8.



(a)



(b)

Figure 4-8: Performance comparison of the three partitioning frameworks when the mobile device is operated at 624 MHz and NavigateMe WS is invoked for (a) a 20x20 grid map (b) a 40x40 grid map

The relative performance of the three partitioning frameworks for this WS application is similar to that observed for Tracking WS. The performance of the intermediate and forwarding frameworks is superior to the backend framework. The intermediate framework is observed to demonstrate the best performance because a powerful computing node (the intermediate node) is responsible for execution of the resource demanding components and coordination of the WS partitions. The backend framework is observed to perform better than the no partitioning case.

The experimental results presented in Figure 4-7 and Figure 4-8 show that for WSs that require a large exchange of data such as Image WS, the forwarding framework exhibits a better performance over the other frameworks (see Figure 4-6). For the other sample WSs that demand minimal data exchange, the intermediate framework exhibits the best performance (see Figure 4-7 and Figure 4-8).

4.6.4 Performance Comparison of the Three Frameworks using π Calculator WS

For a detailed analysis of system performance, the π Calculator WS is used. A number of experiments are performed with π Calculator WS and are described next. In the first experiment, different overheads are measured for the three partitioning frameworks. In the second experiment, the performances of the three partitioning frameworks are compared for different values of computational intensity (N) of the sample WS using a single WS client at a time. In the third experiment, the scalability of the three partitioning frameworks is analyzed by increasing the number of concurrent WS clients (C) for different values of N. In the fourth experiment, the effect of data size on the performance of the three partitioning frameworks is investigated. Note that the artificial data used in the fourth experiment, in addition to the name and parameters of the method in a WS partition, do not have any semantic value but are used to increase the size of the data exchanged. In all the other experiments, the artificial data is not required. In the fifth experiment, the effect of the overheads of the WSEE is investigated by using an additional WS security standard for verifying and signing the body of the SOAP messages. In the sixth experiment, the mobile device is operated at multiple speeds to evaluate the significance of partitioning frameworks for devices with limited processing speeds. In any experiment, only one parameter is varied at a time. The workload and system parameters that are varied are presented in Table 4-2.

Table 4-2: Workload and system parameters for performance comparison of the three frameworks using π Calculator WS

Parameter	***Range of Values***	***Default Value***
Number of WS clients	1, 6, 12	1
Computational Intensity (N)	10 - 1000000	1000, 100000
Offloaded Partition Size (O)	0-95%	50%
Mobile Device Speed	208 MHz, 624 MHz	624 MHz

In one experiment, only one of the parameters is varied at one time. All other parameters used in the experiment are set to their default values. The number of clients (C) is varied from one to twelve. The computational intensity for π Calculator WS is varied from 10 to 1000000. Two default values are listed in Table 4-2 because two sets of experiments, one with N = 1000 and the other with N = 100000, are performed. The offloaded partition size (O) is varied from 0 to 95% with a default value of 50%. The mobile device is operated at two processing speeds: 208 MHz and 624 MHz. 624 MHz is used as a default speed.

4.6.4.1 Overheads of the Partitioning Frameworks

In this experiment, the three partitioning frameworks are analyzed by measuring the processing time of the two components of the sample WS (T_{P1} and T'_{P2}), execution times of the WSEE (T_{WSEE}) and the CPU times corresponding to the overheads δ_{coord} and δ_{comm}. For this experiment, two values of computational intensity (N = 1000 and N = 100000) are used. The offloaded partition size of the sample WS, O, is set to 50%. The experiment is carried out using a single WS client and the experimental results are summarized in Table 4-3.

The results are observed to be in accordance with equation (4-5) and (4-6). Note that the mean response time observed is approximately equal to the sum of T_{P1}, T'_{P2}, T_{WSEE}, δ_{coord} and δ_{comm}. δ_{coord} and δ_{comm} are observed to be higher for the backend framework in comparison to the other two frameworks. It is because the resource constrained mobile device itself is responsible for receiving the WS requests, executing a number of the WSEE components locally, coordinating the execution of the different WS partitions and supporting the exchange of data among different WS partitions. As a result, the mean response time for the backend framework is substantially higher than the two other frameworks. It is also interesting to note that δ_{comm} is the smallest for the forwarding framework. This may be because of the lower number of messages exchanged between different nodes for this framework.

The mean response times for the intermediate and the forwarding framework are close to each other. The execution times of the partitioned applications (P_1 and P_2) are nearly the same (see Table 1) because for each framework the two partitions are run on the same mobile node and the same surrogate node (the intermediate or the forwarding node). The overheads due to WSEE (T_{WSEE}) is observed to be lower for the intermediate framework. This is because most of the SOAP/XML processing is performed on a powerful intermediate node. The overhead corresponding to data exchange (δ_{comm}) is observed to be lower for the forwarding framework because only three messages are exchanged for each invocation in comparison to the four messages exchanged for the intermediate framework. The benefit of a lower $\mathrm{T}_{\mathrm{WSEE}}$overhead for the intermediate framework seems to be compensated by a higher communication overhead (δ_{comm}). Overall, the performance of the intermediate framework turns out to be slightly superior to that of the forwarding framework.

Table 4-3: Mean response time (msec) for requests and the mean CPU time (msec) for overheads

	Response Time	T_{P1}	T'_{P2}	T_{WSEE}	δ_{coord}	δ_{comm}
Intermediate Framework	346.7	54.4	18.3	101.3	29.1	138.9
Backend Framework	854.1	89.3	21.1	219.4	104.5	405.8
Forwarding Framework	359.2	52.1	17.4	153.2	42.3	86.5

4.6.4.2 Performance of Partitioning Frameworks with a Single WS Client

For this experiment, a single WS client is used to generate WS requests. N is varied from 10 to 100000. Values of O are varied from 0 to 95%. Various combinations of these two parameters for the three partitioning frameworks were experimented with and the results are presented in Figure 4-9 (using the intermediate framework), Figure 4-10 (using the backend framework) and Figure 4-11 (using the forwarding framework). The five vertical bars in Figure 4-9 to Figure 4-11correspond to different values of O (shown as legends at bottom of each graph). In all the graphs of Figure

4-9 to Figure 4-11, the first set of bars (O = 0%) represents the no partitioning case and all the computation is done on the mobile device. The bars in the set of bars for each value of N are presented in the following sequence. The first bar corresponds to O = 0%, the second bar to O = 10%, the third bar to O = 25%, the fourth bar to O = 50%, fifth bar to O = 75% and sixth bar to O = 95%.

Figure 4-9 shows the mean response time for different values of O and N for the intermediate framework. As the value of O is increased, the mean response time is reduced. As more components of the sample WS application are offloaded to an intermediate node, a lower time is required to process the WS request resulting in a reduced response time. As the value of N is increased, the mean response time for invoking the WS is increased. This is because higher values of N imply more computation. It is important to note that for higher values of N, the mean response time decreases significantly as the value of O is increased beyond 25% (see set of bars for N = 10000 and N = 100000 in Figure 4-9). Note that performance of the intermediate framework using partitioning is degraded in comparison to the no partitioning case when the value of O = 5% and N ≤100. This is because a very small part of the WS application is offloaded to an intermediate node and the overheads of executing the partitioned WS application seems to be significant in comparison to the benefit achieved with partitioning.

Figure 4-10 shows that the performance of the backend framework is poor in comparison to the no partitioning situation for lower values of N (N < 1000) and when the offloaded partition size is less than 50% (O < 50%). This is because the size of the offloaded partition is not large enough to offset the overheads that accrued because of handling of exchange of WS messages and coordination of WS partitions on the mobile node. For π Calculator WS with value of N greater than 1000 (N = 10000 and N = 100000 in Figure 4-10), the backend framework with partitioning exhibits a better performance in comparison to the no partitioning situation when the value of O is increased to 50% or more.

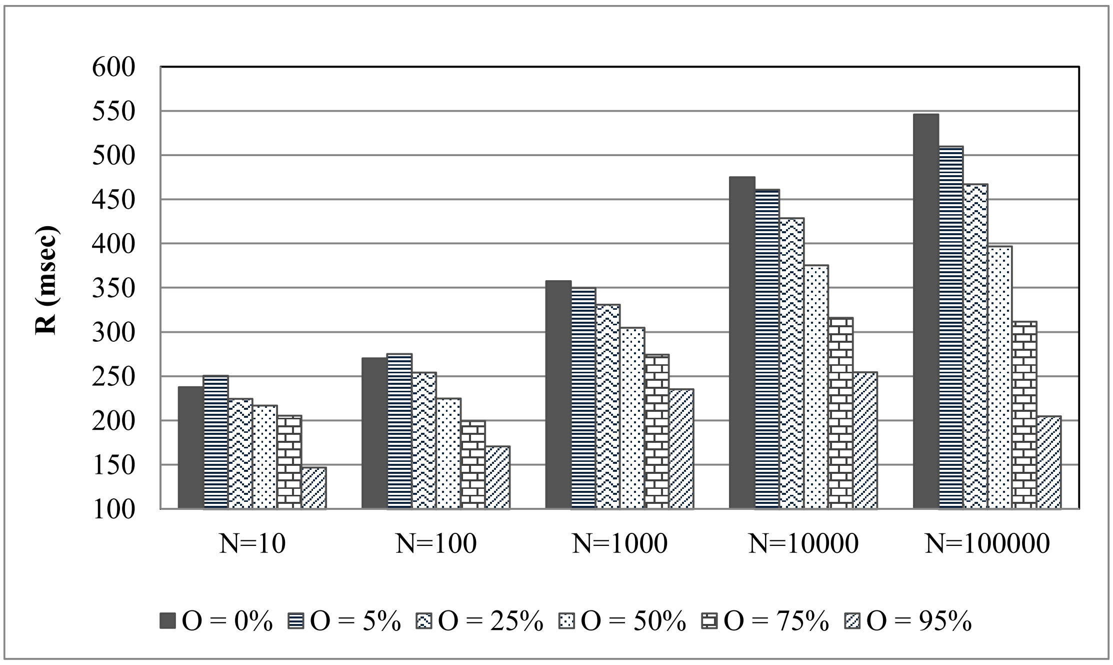


Figure 4-9: Response time for invoking the sample π Calculator WS with mobile device operated at 624 MHz and using an intermediate framework

The forwarding framework (see Figure 4-11) exhibits a performance degradation in comparison to the no partitioning case when O ≤ 25% and N < 1000. This is again because a small part of the WS is executed on a backend node and the performance benefit due to offloading a part of the WS is not high enough to offset the overheads of partitioning. But as O is increased beyond 25%, the mean response time is decreased significantly and a performance improvement over the no partitioning situation is observed.

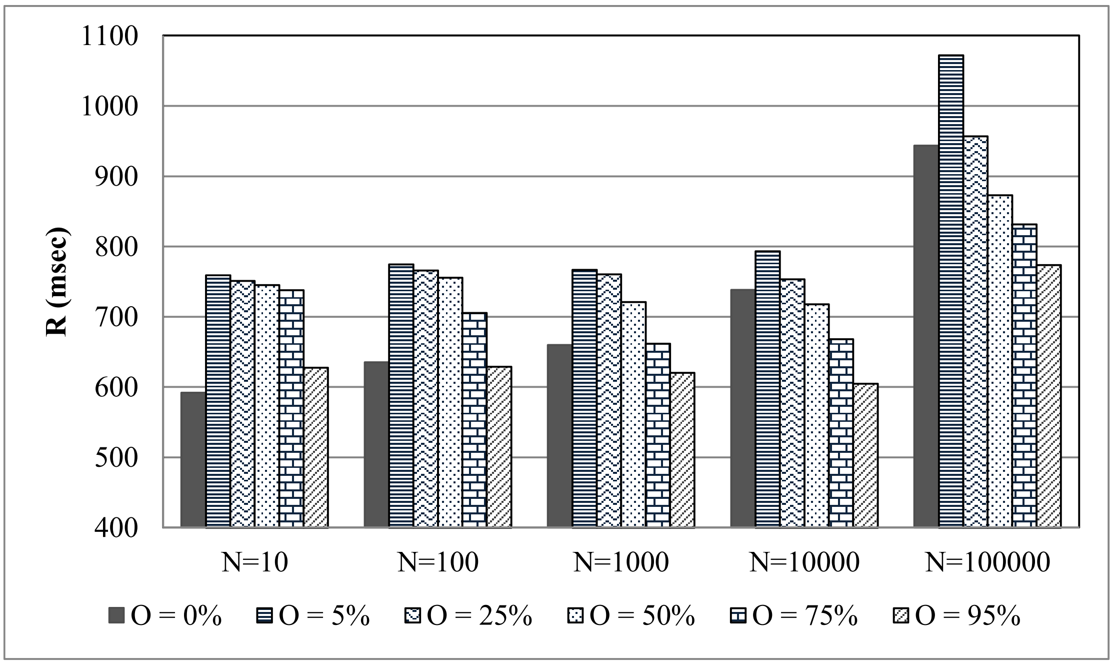


Figure 4-10: Response time for invoking the sample π Calculator WS with mobile device operated at 624 MHz and using a backend framework

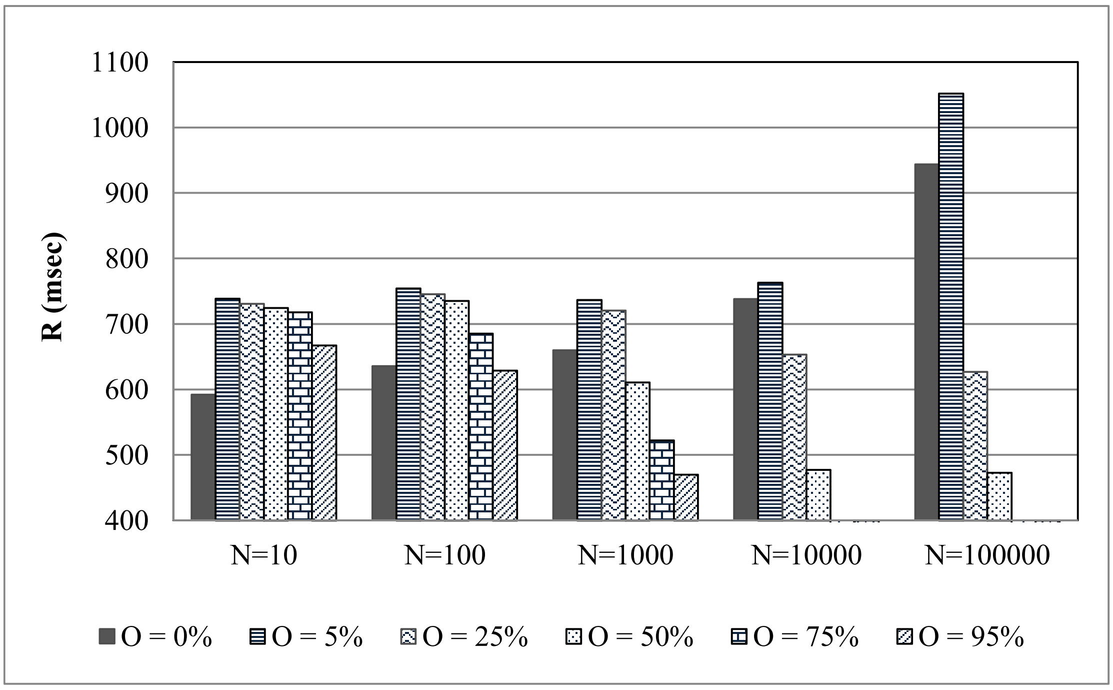


Figure 4-11: Response time for invoking the sample π Calculator WS with mobile device operated at 624 MHz and using a forwarding framework

In general, for lower values of computational intensity (10 and 100), there is hardly any improvement observed in performance for π Calculator WS unless a very large size of partition (O > 75%) is offloaded for execution. But for higher values of computation intensity (N > 1000), a significant performance improvement is observed especially when O ≥ 25%. This shows that WS partitioning is most effective for web services with a substantial execution time and when a significant part of the WS is offloaded to an intermediate or backend node.

Figure 4-12 presents a performance comparison of the three partitioning frameworks for the sample WS with computational intensity of N = 100000. Figure 4-12 displays a high performance degradation for the backend framework in comparison to the no partitioning situation when O ≤ 25%. The two other frameworks exhibit a much superior performance in comparison to the backend framework. The intermediate framework is the winner especially for lower values of O. The forwarding framework exhibits comparable performance with the intermediate framework for higher values of O (see the last set of bars in Figure 4-12).

4.6.4.3 Performance of Partitioning Frameworks with Multiple WS Clients

In this experiment π calculator WS is invoked by 6 and 12 concurrent clients. The value of O is varied from 0% to 95%. The experiment is repeated for N = 1000 and N= 100000. The results are presented in Figure 4-13 for N = 1000 and in Figure 4-14 for N = 100,000. The results are discussed with reference to the value of N and C. In all the graphs of Figure 4-13 and Figure 4-14, the bars in the set of bars for each value of O are presented in the following sequence. The first bar corresponds to the intermediate framework, the second bar to the backend framework and the third bar to the forwarding framework.

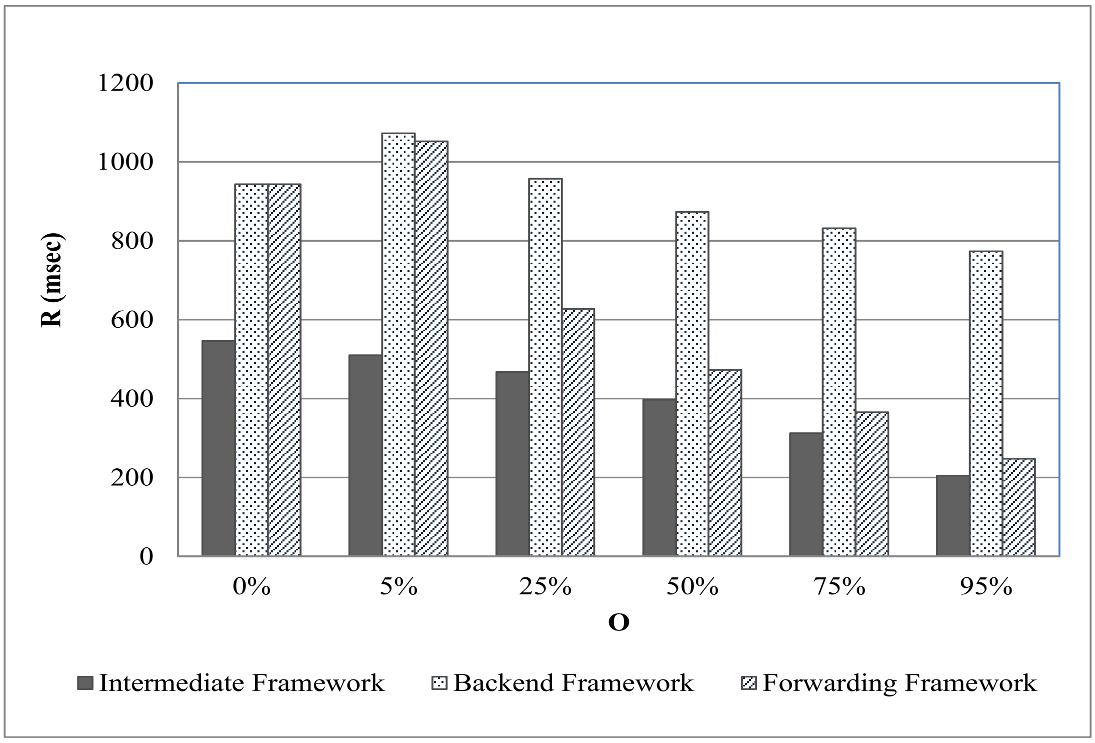


Figure 4-12: Performance comparison of the three partitioning frameworks when the mobile device is operated at 624 MHz and the sample π calculator WS (with N = 100000) is invoked by one WS client.

4.6.4.3.1 For N = 1000 and C = 6 (see Figure 4-13-a)

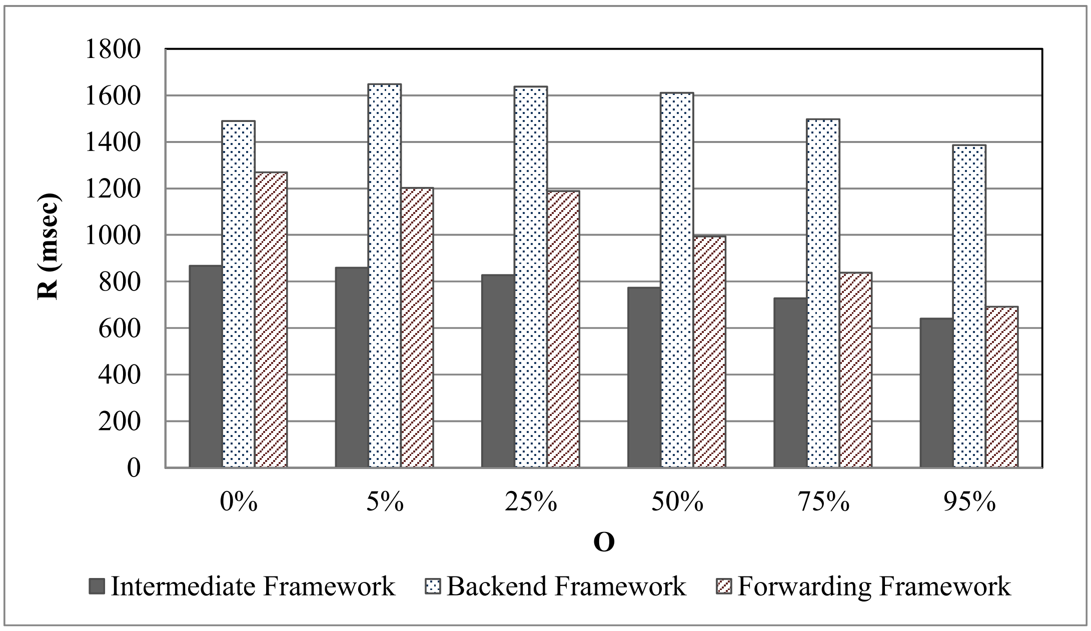
Figure 4-13-a shows that there is hardly any improvement in performance observed for the three partitioning frameworks in comparison to the no partitioning case when N = 1000 and the value of O ≤ 50%. In fact the performance of the backend framework is inferior to that of no partitioning case for O ≤ 50%. When the value of O is increased higher than 50%, the backend framework exhibits marginally better performance in comparison to the no partitioning case. The overheads associated with the backend framework are higher, thus it requires a significant part of application to be offloaded to a backend node to offset the overheads. The intermediate framework is the winner among the three frameworks followed by the forwarding framework.

4.6.4.3.2 For N = 1000 and C = 12 (see Figure 4-13-b)

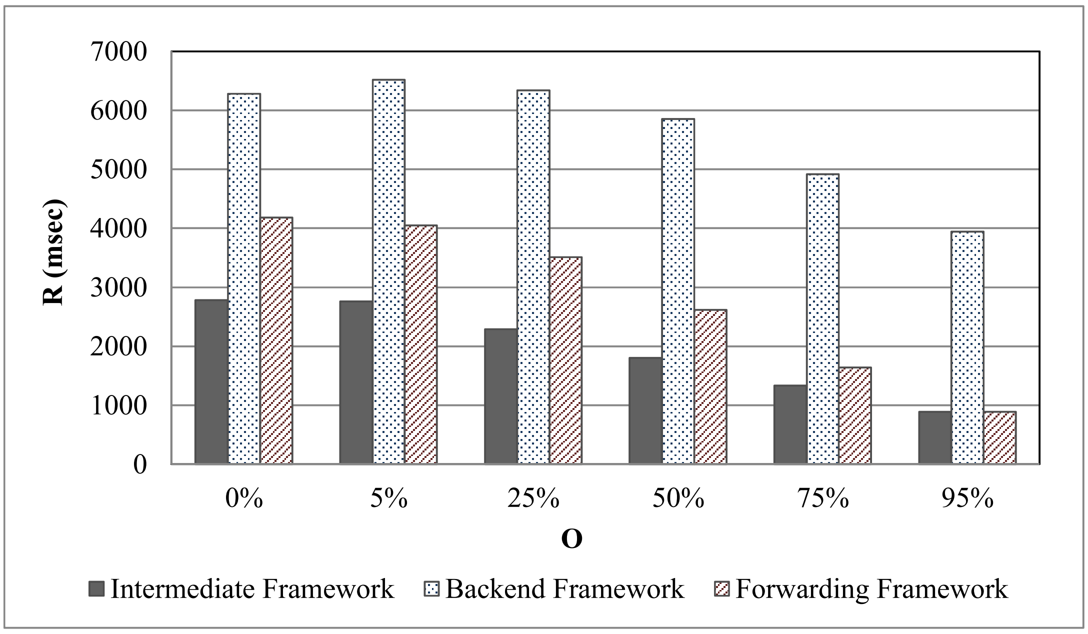
The relative performance of the three partitioning frameworks is observed to be the same when twelve WS clients are used. Performance of the forwarding framework is close to the intermediate framework when $O \geq 75\%$. The backend framework demonstrates a better performance in comparison to the no partitioning case when the value of O is used greater than 5%. The resource contention is expected to be high on the mobile node when a large number of WS clients is invoking the sample WS. Because of this increased resource contentions, the benefit of offloading a partition using the forwarding framework and the backend framework is large enough in comparison to the overheads associated with these frameworks. This results in improved performance of these two frameworks when such a large number (twelve clients in this case) of WS clients is used.

4.6.4.3.3 For N = 100,000 and C = 6 (see Figure 4-14-a)

For higher values of computational intensity (such as N = 100000) and with $O > 25\%$, a significant improvement is observed in performance for all the partitioning frameworks in comparison to the no partitioning case. For $O \leq 25\%$, the performance of the backend framework is inferior to that of the no partitioning case. The performance of the backend framework is never close to the performances of the other two partitioning frameworks that demonstrate a significantly higher performance for $O \geq 25\%$. The higher overheads contribute to the inferior performance of the backend framework in which the mobile node has to perform most of the WSEE processing, coordination of the different WS partitions and the exchange of data with remote WS partitions.



(a)



(b)

Figure 4-13: Performance comparison of the three partitioning frameworks when the mobile device is operated at 624 MHz and the sample π Calculator WS (with N = 1000) is invoked by a) six concurrent WS clients b) twelve concurrent WS clients

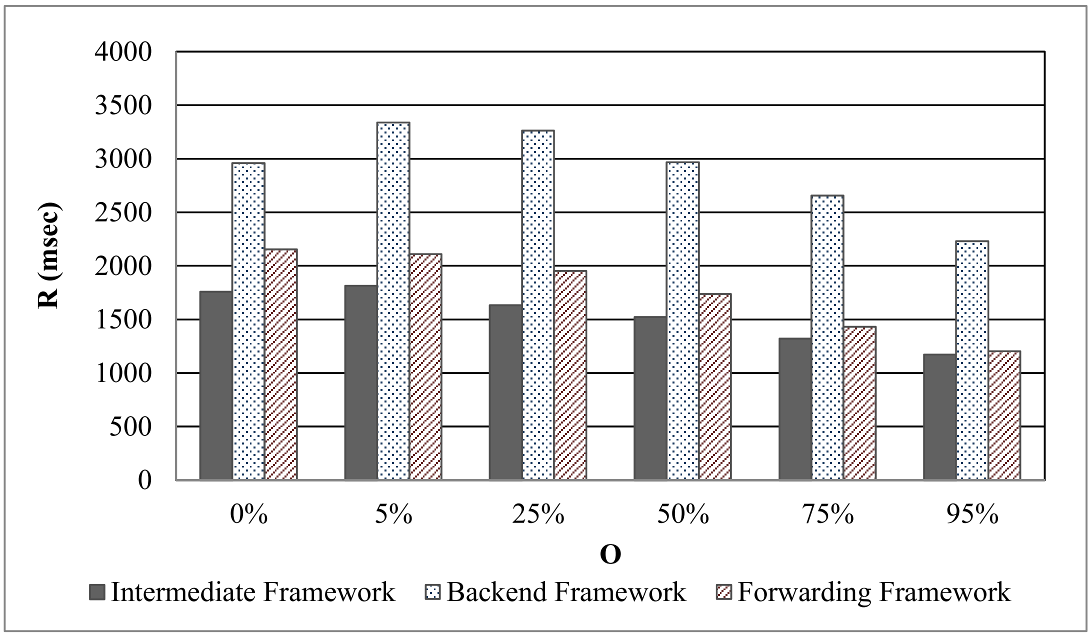
4.6.4.3.4 For N = 100,000 and C = 12 (see Figure 4-14-b)

A similar trend is observed in relative performance of the three partitioning frameworks with twelve WS clients. For O ≥ 75%, the performance of the forwarding framework is observed to be marginally better than that of the intermediate framework. The performance of the backend framework is inferior in comparison to the other two frameworks.

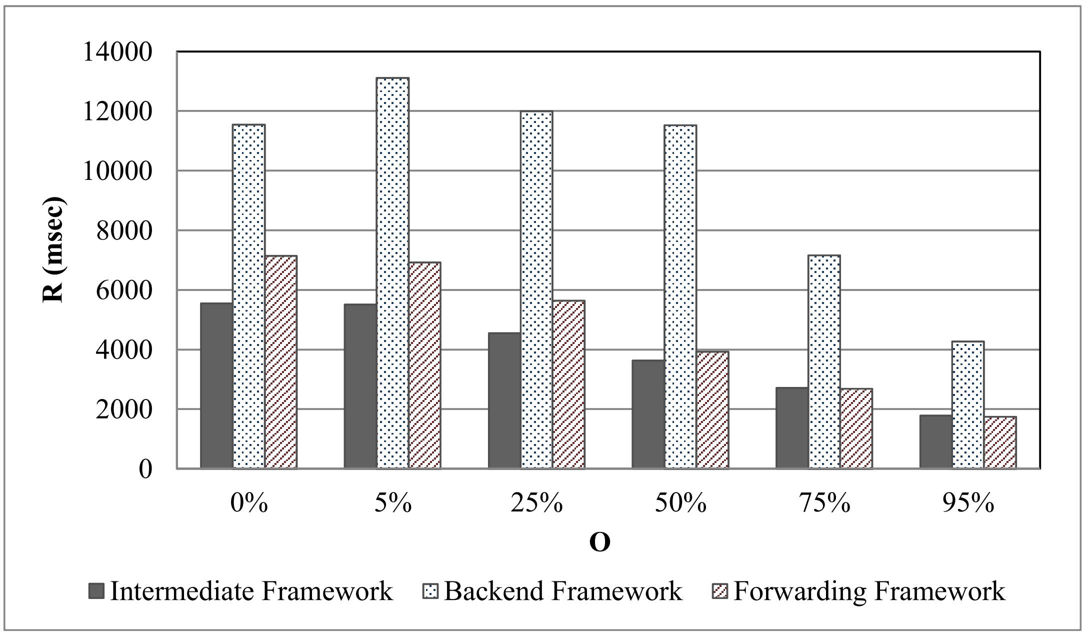
It is interesting to note that the performances of the intermediate framework and the forwarding framework are very close to each other when O ≥ 50% even when multiple concurrent clients are active in the system. The rationale of this behavior is similar to that provided in Section 4.6.4.2.

4.6.4.4 Effect of the Size of Data Exchanged between WS Partitions

The effect of the size of data exchanged on the relative performance of the three partitioning frameworks is investigated by sending dummy data of different sizes between the two partitions. The size of artificial data used in this experiment is varied from 1 Kbyte to 100KBytes (1KBytes, 5KBytes, 10KBytes, 25 Kbyte, 50Kbytes and 100 Kbytes). Such a range of data size is appropriate in the context of mobile web services that this thesis focuses on. For this experiment, two values of computational intensity (N = 1000 and N = 100000) are used. The offloaded partition size (O) of the sample WS is set to 50%. The experiment is carried out using one, six and twelve concurrent WS clients. The experimental results with twelve concurrent WS clients are shown in Figure 4-15 (see Appendix A.1 and Appendix A.2 for experimental results achieved with one and six WS clients respectively).

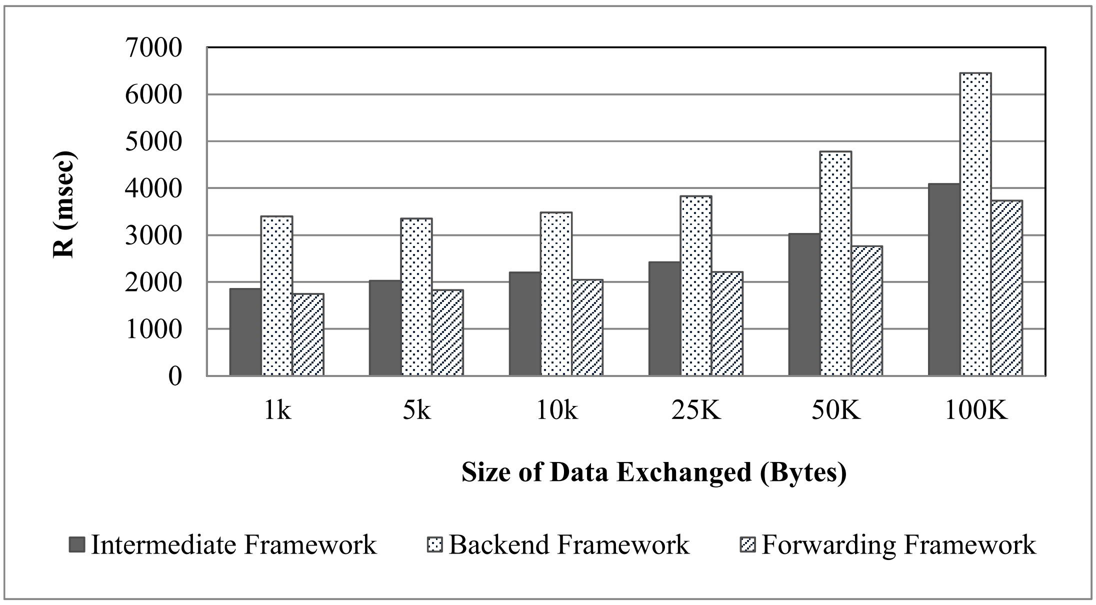


(a)

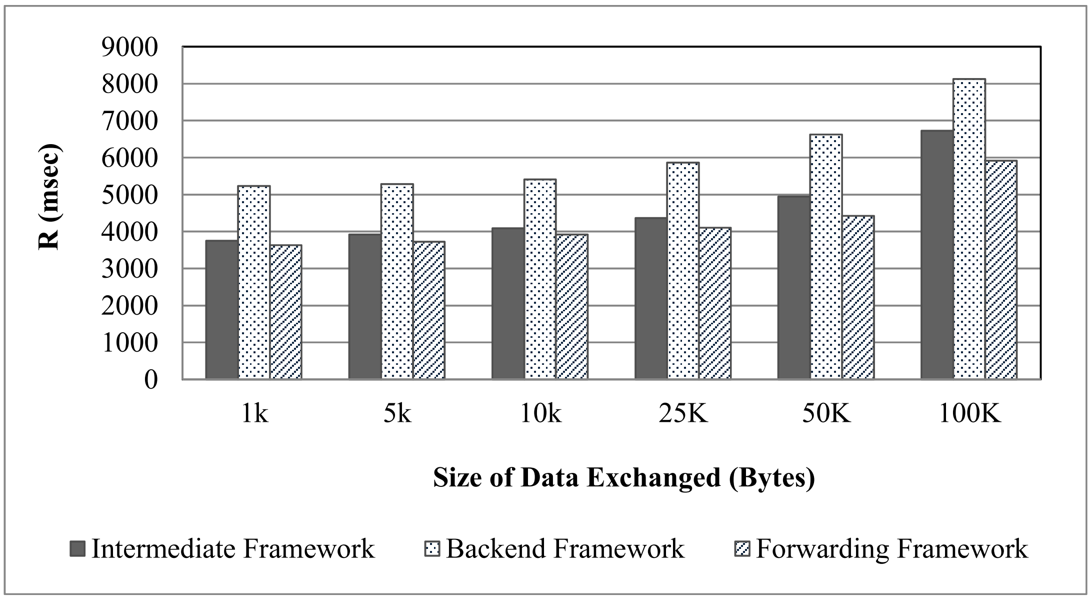


(b)

Figure 4-14: Performance comparison of the three partitioning frameworks when the mobile device is operated at 624 MHz and the sample π WS (with N = 100000) is invoked by a) six concurrent WS clients b) twelve concurrent WS clients



(a)



(b)

Figure 4-15: The effect of the size of data exchanged between WS partitions on the performance of the three frameworks when the sample π Calculator WS with O = 50% is invoked by twelve concurrent WS clients (a) with N = 1000 (b) with N = 100000

The size of the data exchanged between the WS partitions does not seem to affect the relative performance of the three partitioning frameworks (see Figure 4-15). The mean response time for any given framework seems to vary sub-linearly with increase in size of the data exchanged between the WS partitions. The performance of the forwarding framework is superior to the intermediate framework for all the data sizes. It is because of the lower number of messages (three) exchanged between the three nodes. Once again, the backend framework displayed an inferior performance in comparison to the two other partitioning frameworks. A similar relative performance of the three frameworks is observed with one and six WS clients.

4.6.4.5 Effect of Using a WS Security Standard

To investigate the effect of a higher WSEE overhead on the performance of the different partitioning frameworks, an experiment is performed with a WSEE that supports a WS security standard as well. The WS security standard is used to verify the integrity of incoming SOAP messages using XML Signatures and also to sign the final WS response. As a result of these additional requirements, the WSEE overheads increase significantly.

For this experiment two values of computational intensity (N = 1000 and N = 100000) are used. The value of O is set to 50%. The experiment is carried out using one, six and twelve concurrent WS clients, but only the experimental results achieved with twelve concurrent WS clients are shown in Figure 4-16. A similar trend in response time has been observed for one and six concurrent WS clients (see Appendix A.3). In case of the forwarding framework, the integrity of the incoming request message is verified at the mobile node whereas the task of signing the outgoing WS response is performed at the forwarding node. It has been observed that the performance of the backend framework (second set of bars in Figure 4-16) is significantly inferior to that achieved with the other partitioning frameworks. For N = 100,000, for example, the mean response time of the backend framework is 1.33 times that achieved with the intermediate framework and 1.4 times that achieved with the forwarding framework. It is because the entire code required for handling the WS security standard is executed on the mobile device. Thus, it consumes more resources of the mobile device and leads to a high response time.

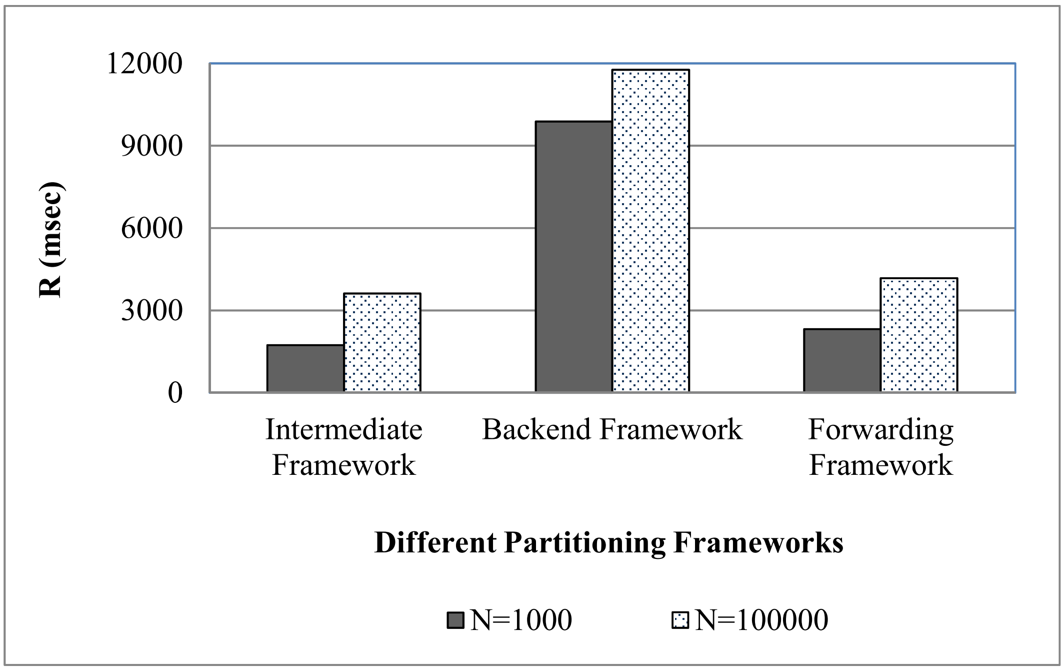
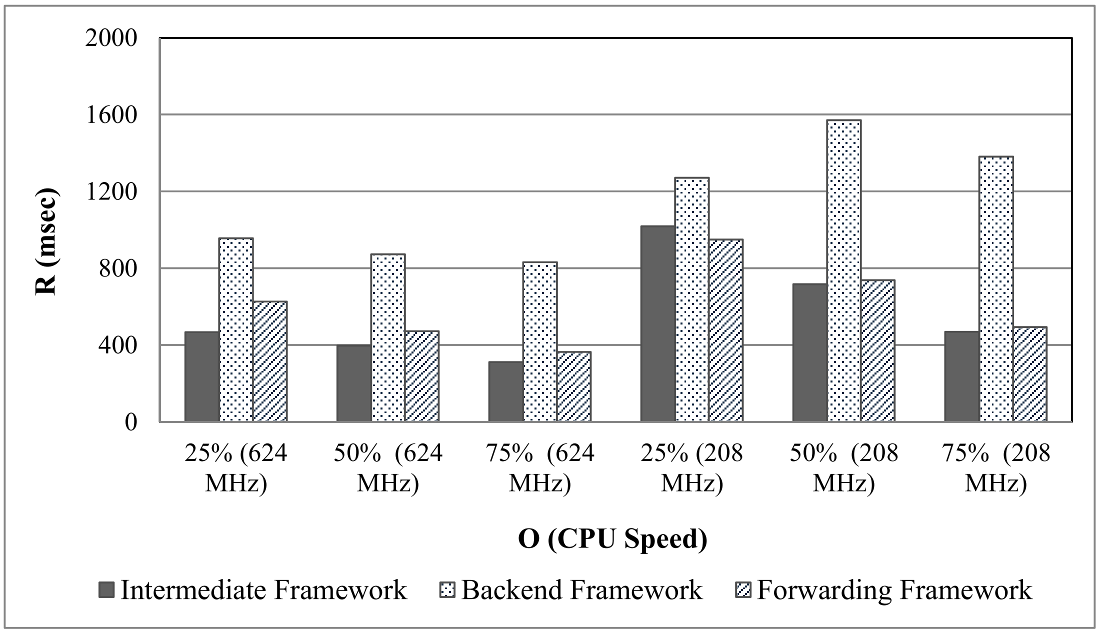


Figure 4-16: Effect of using WS security standard on the relative performance of the three partitioning frameworks when the mobile device is operated at 624 MHz and the sample π Calculator WS with O = 50% is invoked by twelve concurrent WS clients

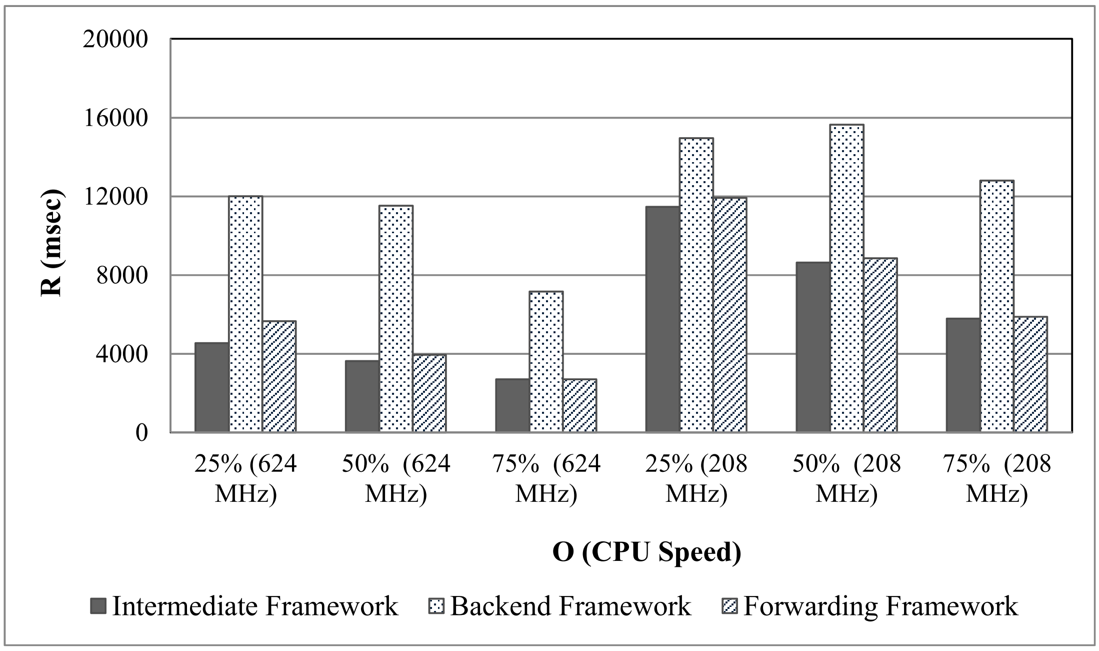
In case of the intermediate framework, the entire code for handling the security WS standard is executed on a powerful intermediate node. The intermediate framework exhibits the best performance because the powerful intermediate node is responsible for executing the entire code for handling the security WS standard, for handling all message exchanges with WS clients and for coordinating WS partitions. The performance of the forwarding framework is slightly inferior to the performance of the intermediate framework because the resource constrained mobile node is responsible for handling incoming messages from WS clients and partial coordination of partitions. Also, note that the forwarding framework executes only 50% of the code on the backend node for handling WS security in comparison to the intermediate framework that executes the entire code on the intermediate node for WS security.

4.6.4.6 Effect of the Speed of the Processing Resource

In this experiment, the Dell Axim PDA (mobile node) is operated at two different processor speeds: 624 MHz and 208 MHz. The experiment is performed with one, six and twelve WS clients and using different values of N (10 to 1000000). The experiment is repeated for different values of O (25%, 50% and 75%). As expected, with a CPU speed of 208 MHz, the mean response times for all the frameworks are increased. A similar trend is observed in the relative performance of the three frameworks as the value of O is increased, but it has been observed that there is very little or no improvement in performance when the sample WS with a lower computational requirements (for $N < 10000$) is invoked. Figure 4-17 displays the results achieved with $N = 100000$ and using one and twelve WS clients. A similar trend in relative performances of the three frameworks has been observed when six WS clients are used (see Appendix A.4).



(a)



(b)

Figure 4-17: Effect of CPU speed of the mobile device on relative performance of the three partitioning frameworks when the π Calculator WS (with N = 1000000) is invoked by a) one WS client b) twelve WS clients

Figure 4-17 captures the mean response time improvement when the offloaded part of the WS is varied from 25% to 75%. Once again, the results presented in Figure 4-17 show that the intermediate framework and the forwarding framework are superior in comparison to the backend framework. In case of slower processor speeds (208 MHz), another important observation is that the performance improvement, when O is increased from 25% to 50% and 50% to 75%, is far better in comparison to the performance improvement that is observed when O is varied from 25% to 50% and 50% to 75% using a processor speed of 624 MHz. This implies that although application partitioning can lead to a significant benefit for all devices, its importance is even higher for devices with lower speed processers.

4.7 Summary

In this chapter a forwarding framework for mobile WS partitioning is proposed and is compared with the two existing application partitioning frameworks. Based on prototyping and measurement, an in-depth analysis is performed to investigate the performance of the three partitioning frameworks for hosting web services on resource constrained mobile devices. Some of the WS partitions are run on the mobile device while the others are run on a more powerful computing node. The insights gained in to system behavior and performance are summarized.

- The intermediate and the forwarding frameworks lead to a significant performance improvement over an un-partitioned system. The performance improvement produced for the two frameworks is significant when
    - Used for compute intensive (resource demanding) applications

    - A reasonable size of the application (O = 25% or higher in case of experiments with π Calculator WS) is offloaded to a remote computing node.

- In most cases when one WS client is used, the performance of the backend framework is observed to be either very close to (for Image WS and Tracking WS) or inferior (for π Calculator WS) to that of an un-partitioned system for most values of O. It seems that the overheads of this framework often offset the performance gain achieved by offloading a part of the application to a backend node. This framework demonstrates a better performance than the no partitioning case when a very high value of O is used ($O \geq 75$).
- Both the forwarding and the intermediate frameworks have shown performance close to each other. The intermediate framework seems to be a better choice for web services that require the WSEE to support extra WS standards such as the security standard. The rationale behind such a behavior is the execution of a larger component of the WSEE on the intermediate node.
- The forwarding framework also showed a good performance in most cases. The experimental results presented in Section 4.6.1 and Section 4.6.4.4 show that for high sizes of data exchanged, this framework performs the best producing a lower response time in comparison to the intermediate framework. This seems to be an effect of the lower number of messages exchanged in the forwarding framework, the impact of which becomes significant for higher sizes of data exchanged. The forwarding framework achieved a comparable performance to the intermediate framework without delegating the application control to the intermediate node.

- The performance improvement that accrues from WS partitioning is observed to be higher for devices with lower speed CPUs. This implies that WS partitioning is expected to be even more important with resource constrained inexpensive mobile devices with slower CPUs.

Chapter 5: Design Time WS Partitioning

Different techniques for hosting web services on a mobile device were discussed in Chapter 3 and Chapter 4. The main focus of these earlier chapters was to facilitate hosting of mobile web services and devising distributed web service execution environments. The remainder of this thesis focuses on devising algorithms for partitioning WS applications. This chapter presents an analysis of graph-partitioning based design time WS partitioning algorithms.

5.1 Overview

Hosting web services on wireless mobile devices can be simple or complex depending on the goals of a WS application. Some applications such as checking the availability of resources in a resource pool require only a few lines of programming. However in certain applications, such as image format conversion used in image processing applications (discussed in Section 4.4.1.1), a WS can be a complex process that can involve a number of software components and execute complex algorithms to achieve its goals. Invoking such a service for a number of times by multiple clients can lead to temporarily stopping the mobile device from performing its core functionalities such as voice services. The repeated invocation of such a resource demanding (WS) application increases the probability of the device going out of battery power more quickly.

To facilitate execution of such resource demanding applications on mobile devices, this thesis proposes to apply application partitioning for hosting WS applications effectively on mobile devices. WS application partitioning is performed to divide a WS application into multiple components so that computationally complex components can be offloaded to remote computing nodes. As already mentioned in Chapter 2, the existing application partitioning algorithms available in the literature are designed to partition large scientific applications into multiple partitions of similar size so that the partitions can be run in parallel. Little work has been done for partitioning of WS applications that have different constraints and objectives. For example, achieving partitions of the same size is not a key requirement in case of mobile WS applications. The objectives for WS partitioning include achieving a considerable size for the offloaded partitions with smaller communication costs.

For mobile WS application partitioning, two types of approaches have been investigated in the past. Some researchers have suggested the offloading of the entire application code ([Hem05], [Kim07], [Riv07]) to a remote computing node in a fixed infrastructure. Another approach available in the literature proposes to use the minimum cut (also known as MinCut) algorithm for partitioning the application if the application can be modelled as a graph ([Hun99], [Mes02]). A key concern in applying the MinCut algorithm for application partitioning is that it tends to partition the application in such a way that the communication costs between the application components are minimized. This may not always lead to an improvement in overall performance in a mobile device based system where reducing the resource demands on the mobile device that includes the lowering of its CPU load is a key concern.

In this thesis, two new design time graph-based partitioning algorithms are proposed: Maximum Offloading Minimum Cost (MOMC) and Cluster based Application Partitioning (CAP). The proposed algorithms can be used for mobile WS applications as well as for traditional mobile applications. The MOMC algorithm uses a global maxima approach for achieving the partitions for offloading while the CAP algorithm uses a local maxima approach for obtaining the partitions for offloading. The effectiveness of the proposed algorithms is analysed by comparing the performance of the partitioned systems achieved with the two proposed algorithms with the un-partitioned system and the partitioned systems achieved with the MinCut algorithm and the offloading entire application approach.

The intermediate framework and the forwarding framework exhibits the best performance among the three partitioning frameworks discussed in Chapter 4. For the WS partitioning analysis, however the backend framework is used because of the limitations of the intermediate framework and the backward framework. The intermediate framework is virtually same as hosting web service on an intermediate node. The forwarding framework is useful only when the mobile node completes its execution first and then forward execution of the rest of the application to the forwarding node. Moreover, the relative performance of the proposed WS partitioning techniques is not expected to be affected by the partitioning framework used.

5.2 Design Time WS Application Partitioning Guidelines

Before presenting algorithms for WS partitioning, it is important to describe the different factors that should be considered in devising these algorithms. These factors are presented next:

- Number of partitions a WS application can be divided into (n).

- Dependency of the WS application on the local resources.
- Frequency of communication between different components of a WS application.
- Resource limitations of handheld devices.
- Resource capabilities of remote computing node (S).

Based on these factors, a set of guidelines are proposed for partitioning of WS applications and are devised from basic principles underlying distributed computing. These guidelines are described next.

Web services hosted on handheld devices are expected to use local resources. Services can make local systems calls to achieve their goals. The first guideline suggests identifying the components of the application that make local system calls. The identified components can be put in a partition that is to be deployed on the handheld device. The part of the application that does not have dependency on local resources can be put in one or more partitions that can be deployed on one or more remote computing nodes.

For improving system performance, the communication between different partitions of a WS application should be minimized. The different partitions of a WS must be designed to reduce inter-partition communication. The second guideline suggests moving a component of a WS to a partition that is to be deployed on a remote computing node only if it does not demand a large amount of data transfer between the remote computing node and the mobile node.

5.3 Graph Terminologies

To understand the algorithms presented in Section 5.4 and Section 5.5, the necessary graph related terminology is introduced.

5.3.1 Graph

A graph is characterized by two finite sets V and E.

$G = (V, E)$

Where

$V = \{V_1, V_2, V_3, \ldots V_i\}$

$E = \{E_1, E_2, E_3, \ldots E_j\}$

Note that *i* and *j* represents the total number of vertices and edges respectively.

The elements of V are called *Vertices* (or nodes) and the elements of E are called *Edges* [Gro06]. For this research, a WS application is assumed to be modeled as a directed graph G = (V, E) in which the vertices correspond to the different components of the WS application and the edges represent communication between the different WS application components. A number of techniques for constructing a graph model of an application is discussed in [Ou07] and [Wan08]. Further discussion of constructing a graph model of an application is beyond the scope of this thesis.

5.3.2 Edge Weight (W_E):

The edge weight (W_E) represents the communication cost between the two components (vertices connected by the edge) when they are executed on different computing nodes. In the context of the research presented in this thesis, the communication cost is the time (in time units) consumed to exchange data between one component of the application and another if they are being executed on different computing nodes. The communication cost between the two components deployed on the same computing node is negligible and is assumed to be zero in our analysis because the data is exchanged through a shared memory. When the components of an application run on different computing nodes, a cost due to network delays is incurred during communication across two partitions running on different nodes.

5.3.3 Vertex Weight (W_V):

Weight of a vertex can be used to represent requirement of resources such as CPU time and memory by an application component. The prototype WS applications used for the experimental analysis do not require a great deal of memory. Thus, the memory requirements are not considered as a critical constraint in this thesis. For the analysis presented in this thesis, the weight of a vertex represents only the CPU time (in time units) required for execution of an application component. The effect of memory constraint on partitioning of large applications forms an interesting direction for future research.

5.3.4 Source Vertex

A source vertex $s \in V$ represents the entry point for an application. The weight of the source vertex (*s*) is set to zero and is not considered for offloading while applying the partitioning algorithms.

5.3.5 Vertex Distance

A vertex distance (D) is the least distance of a vertex $v \in V$ from the source vertex *s*. D is measured as the least number of edges that need to be traversed to reach a vertex v from the source vertex.

5.3.6 Graph Size (D_{max})

Graph size is the maximum value of D for any vertex, $v \in V$.

5.3.7 Edge Cut

An edge cut is defined as a set of edges the removal of which leads to two disjoint graphs. In the context of this research, the "edge cut" represents partitioning of a graph into two sub-graphs. An edge cut weight is defined as the sum of the weights of all the edges in the edge cut.

5.3.8 Degree of Benefit (β)

The degree of benefit is a term introduced in this thesis. It is defined as the difference of the sum of the weights of vertices in set X and the sum of weights of edges that separate vertices in set X from rest of the graph. For vertices in set X, the degree of benefit can be computed using the following relation:

$$\beta\,(X) = \sum W_{VX} - \sum W_{EX}$$

where $\sum W_{VX}$ represents the sum of the weights of vertices in set X and $\sum W_{EX}$ represents the sum of weights of edges that separate vertices in set X from rest of the graph.

5.3.9 Beneficial Cut

This is another term introduced in this thesis. It is defined as the edge cut for which the degree of benefit is the maximum.

5.4 Maximum Offloading Minimum Cost (MOMC) Algorithm

The MOMC algorithm uses a heuristic approach for selecting a set of vertices from the graph for being offloaded for execution on a remote computing node. The selected set of vertices attempts to maximize the difference between the sum of the weights of vertices in the set and the weight of the edge cut that separates the set from rest of the graph. This approach is used to achieve a maximum degree of benefit. The algorithm focuses on computing the beneficial cut using a global maxima approach.

The MOMC algorithm selects a starting vertex and iterates through the graph to find the beneficial cut. Once a starting vertex is selected, the next vertex is selected from a set of neighbor vertices. The vertex selected produces the maximum degree of benefit in comparison to other neighbor vertices.

5.4.1 Assumptions

Input for the algorithm is an application modeled as a directed graph G = (V, E) where V is a set of vertices representing the application components and E is a set of edges. An edge is placed between a vertex A and a vertex B when component B is invoked by component A. The weights of vertices and edges are input parameters for the algorithm. It is assumed that resource contention on the remote computing node is not a concern and delay caused by other applications running on that node is negligibly small.

5.4.2 Objective Function

The objective function of MOMC is to divide a graph, G = (V, E), into two partitions, P_1 and P_2, such that

Condition 1: $P_1 \cap P_2 = \Phi$ P_1 and P_2 are two disjoint sets of vertices

Condition 2: $P_1 \cup P_2 = V$

Condition 3: $\beta_{max}(P_2) = \text{Max}\left(\sum_{i=1}^{n} W_{Vi} - \sum_{j=1}^{m} W_{Ej}\right)$

Where

- P_1 and P_2 are disjoint sets of vertices and $P_1 \in V$ and $P_2 \in V$.
- $\sum_{i=1}^{n} W_{Vi}$ is the sum of weights of vertices in partition P_2.

- $\sum_{j=1}^{m} W_{Ej}$ is the sum of weights of the edges that separate the vertices of set P_2 from set P_1.

Condition 1 and *Condition 2* define the basic constraints. Condition 1 states that the two sets of vertices P_1 and P_2 are disjointed sets. Condition 2 requires the union of the sets (P_1 and P_2) equal to all vertices in graph G, which is set V.

Condition 3 defines a beneficial cut for partition P_2. Partition P_2 is the one that is assumed to be offloaded to a remote computing node. This condition ensures computing of the edge cut (beneficial cut) in such a way that the degree of benefit for partition P_2 is maximized.

5.4.3 Algorithm Steps

The algorithm uses temporary variables that include different sets of vertices (*B, X, Q, N* and *Y*), variables (*w, ε and β)* for calculation of the degree of benefit and a table *T* for storing candidate partitions. *w* and, *ε* are used for storing the sum of the weights of vertices of X (W_{VX}) and the weight of the edge cut that separates vertices in set X from rest of the graph (W_{EX}) *β* is used for storing the degree of benefit for set X. The set *Q* contains all vertices of set *V* except *s* (initialized in step 1 of the algorithm). Note that '\' is a set operation of taking relative complement. If *A* and *B* are two sets, then the relative complement of B in A, also known as the difference of *A* and *B*, is the set of elements that are in *A*, but not in *B*. Rest of the variables are explained in line with explanation of individual steps of the algorithm presented in Figure 5-1.

The main steps of the algorithm (the pseudo code is presented in Figure 5-1) are explained next. The basic steps such as steps 1 and step 5 of Figure 5-1 that involve initializations of different variables are not included in this discussion.

```
//V is a set of vertices of graph G and Q is a set of vertices that are considered for
//partitioning. s is the source vertex and i is an index variable for the number of
//partitions computed. X and Y are temporary sets of vertices. β is the degree of
//benefit. T[β,Y] is a table to hold partitions and D is the vertex distance.

   01:  Q = V \ s, i = 0, T [β, Y] = Φ;
   02:  B = Set of vertices in Q with maximum D;           (Boundary Vertices)
   03:  ñ = Number of vertices in B;
   04:  do {                                                //Outer Loop Start
   05:          X = Φ, Y = Φ, β = 0;
   06:          ν = B[i];       // ν is a variable to hold a selected vertex from B
   07:          do {                                        //Inner Loop Start
   08:            X = X U ν;
   //w and ε are temporary variables storing sum of the weights of vertices and the
   //edge cut weight respectively
   09:            w = Sum of weights of vertices in X;
   10:            ε = Edge cut weight of set X;
   11:            If (w - ε) > β
   12:                 Y = X, β = w – ε;
   13:            N = Set of vertices in Q that are directly connected to vertices of X;
   14:            ν = Vertex in N with maximum degree of benefit;
   15:          } while (Q \ X != Φ)                        //Inner Loop End
   16:          i++;
   17:          Add (β, Y) to T;
   18:  } while (i != ñ)                                    //Outer Loop End
   19:  P_MOMC = select Y from T with maximum β;    (P_MOMC is a final partition)


  Note: A\B = {x ∈ A, x ∉ B}          (relative complement)
```

Figure 5-1: The MOMC algorithm for mobile WS partitioning

- *Find boundary vertices (B) from set Q (line 2 in Figure 5-1)*
    - In context of the algorithms presented in this thesis, the boundary vertices are those with maximum value of *D*. It has been observed during preliminary research that by using different vertices of set *Q*, when one of the vertices that exist far away from the source vertex '*s*' is used as a starting vertex, more beneficial partitions are achieved in comparison to partitions that are achieved by selecting a starting vertex randomly or by selecting vertices close to '*s*'.
- *Select one boundary vertex (v)* at a time as a starting vertex (line 6 in Figure 5-1)
    - This step is only executed when a new vertex is to be selected from *B*.
- *Add selected vertex 'v' to a temporary set (X) (line 8 in Figure 5-1)*
    - *X* is a set of vertices that represents a candidate partition for offloading.
- *Calculate w and ε* (*line 9* and *line 10* in Figure 5-1).
    - *w* is the sum of the weight of the vertices in *X* and *ε* is the weight of the edge cut that separates the vertices in set *X* from the rest of the graph.
- *Update β* (lines 11-12 in Figure 5-1).
    - *β* is used to hold the maximum value of the difference between *w* and *ε* for any iteration in Inner Loop. It is initialized to zero at the start of every iteration of Outer Loop. The value of *β* is the degree of benefit in offloading the partition that is comprised of vertices in set *X*. The objective of this algorithm is to find set *X* with the maximum value of *β* (degree of benefit). If the difference between *w* and *ε* is more than the current value of *β*, the vertices in *X* are assigned to *Y* and the value of *β* is updated to the current difference between *w* and *ε.* The set *Y* is a set used for temporarily

storing the vertices of X for which the difference between w and ε is the maximum.

- *Select an appropriate neighbor vertex* (lines 13-14 in Figure 5-1).
    - For the rest of the iterations of Inner Loop, vertex 'v' is selected from a set of neighbor vertices of X. The set of neighbor vertices, N, (where N is a subset of $\{Q \setminus X\}$) represents those vertices of $\{Q \setminus X\}$ that are connected to any vertex of X. A vertex in N that results in the maximum degree of benefit is selected.
- *Repeat steps in lines 8-14 for all vertices of set Q.*
    - Once all vertices of set Q are considered, the current value of set Y will contain a set of vertices that gives rise to the maximum value of β (β_{max}).
- *Add current value of β and Y to table T* (line 17 in Figure 5-1) that holds the maximum value of β and the corresponding set of Y.
- *Select a new boundary vertex* for the next iteration of Outer Loop (line 6 in Figure 5-1) and repeat step 7-18.
    - The steps of finding β_{max} are repeated by using one of the boundary vertices at a time. The value of β_{max} and set Y achieved using one boundary vertex are stored in table T as a one row.
- *Select final partition (as P_{MOMC}) from table T* (step 19 in Figure 5-1) for which the value of β_{max} is the largest.
    - This step is performed when all vertices of boundary set B are evaluated.

The set of vertices stored in P_{MOMC} represents the partition for offloading to a remote computing node. Rest of the vertices ($\{Q \setminus P_{MOMC}\}$) belongs to a partition to be executed locally on the mobile device.

Table T used in the algorithm can be replaced with two variables to hold the maximum value of β and the corresponding set Y that is achieved through different iterations of Outer Loop. The advantage of using a table is to provide an opportunity to apply additional criteria other than the maximum value of β for selecting Y. For example, if multiple iterations achieve the same value of β, then the set Y can be selected based on the overall weight of vertices of set Y.

To obtain more partitions, the algorithm can be applied iteratively on the partitions P_{MOMC} and $Q \setminus P_{MOMC}$ separately. For example, P_{MOMC} can be partitioned further into two partitions if P_{MOMC} is too large for a single remote computing node. By dividing P_{MOMC} into two partitions, each partition can be executed on two remote computing nodes. Note that the MOMC algorithm focuses on achieving two partitions in a single iteration. The algorithm that focuses on computing multiple partitions in a single iteration is discussed next.

5.5 Clustering based Application Partitioning (CAP)

This algorithm uses a clustering approach for selecting a set of vertices from a graph to form a partition. The CAP algorithm is different from the MOMC algorithm in selecting a new vertex from a set of neighbor vertices. In the CAP algorithm, a new vertex is selected only if its inclusion results in an improvement in the degree of benefit of the selected set of vertices. If no such vertex is found from a set of neighbor vertices, the algorithm marks the already selected vertices as one partition and starts looking for another partition using a different starting vertex from the rest of the vertices of the application graph. In the MOMC algorithm, a new vertex is always selected from a set of neighbor vertices.

5.5.1 Objective Function

The objective of this algorithm is for partitioning a WS application into ‘n’ number of partitions (P_1, P_2 …. P_n) of a graph G = (V, E) such that

Condition 1: $\cup\,(P_1\,, P_2, P_3 \ldots P_n) = V$

Condition 2: $\cap\,(P_1\,, P_2, P_3 \ldots P_n) = \Phi$

Condition 3: For any partition P_k,

$$B_{max}\,(P_k) = \mathrm{Max}\,(\textstyle\sum_{i=1}^{nk} W_{Vi} - \sum_{j=1}^{mk} W_{Ej}\,)$$

Where

- $\sum_{i=1}^{nk} W_{Vi}$ is the sum of weights of vertices in partition P_k.
- $\sum_{j=1}^{mk} W_{Ej}$ is the sum of weights of edges that separate the vertices of set P_k from rest of the graph.
- $P_1\,, P_2, P_3\ to\ P_n$ are disjoint sets of vertices of set V.

Condition 1 and *Condition 2* define the basic constraints. Condition 1 states that all partitions (sets of vertices) are mutually exclusive. Condition 2 requires the union of the partition sets equal to all vertices in graph G, which is set V.

Condition 3 defines a beneficial cut for any partition ‘k’. The partition P_k can be any partition that is a candidate partition for offloading to a remote computing node. This condition requires computing the beneficial cut to be complied in such a way that the degree of benefit for partition P_k is maximized.

5.5.2 Assumptions

Input for the algorithm is an application modeled as a directed graph G = (V, E) and the weights of all edges and vertices are provided as an input to the algorithm. The required number of partitions can be provided as ‘n’. The algorithm will stop when the required number of partitions (*n*) is achieved. The algorithm steps are presented in Figure 5-2 and are explained next.

5.5.3 Algorithm Steps

To explain the working of the algorithm steps, temporary variables are introduced. These temporary variables include sets of vertices (*B*, *X, N and Q*), numeric variables (*w, ε and β)* for calculation of the degree of benefit and a table *T* for storing partitions achieved by the algorithm. The set *Q* contains all vertices of set *V* except *s*. *w* and*, ε* are used for storing the sum of the weights of vertices of X (W_{VX}) and the weight of the edge cut that separates vertices in set X from rest of the graph (W_{EX}) *β* is used for storing the degree of benefit for set X. The remainders of the variables are explained in line with explanation of individual steps of the algorithm (see in Figure 5-2). The main steps of the CAP algorithm (presented in Figure 5-2) are explained next. Note that the steps involving initialization of different variables are not included in this discussion.

- *Find set of neighbor vertices 'N'* (line 4 and line 15 in Figure 5-2).
    - In the first iteration when set '*X*' is empty, a set of boundary vertices is determined and assigned to set variable *N* (line 4 in Figure 5-2). In following steps, a set of neighbor vertices of set X are determined and assigned to set N (line 14 in Figure 5-2). The boundary vertices for set *Q* are those with maximum value of *D*. The neighbor vertices (a subset of *{Q \ X}*) are those vertices of *{Q \ X}* that are connected to any vertex of set *X*.

```
//V is a set of vertices of graph G and Q is a set of vertices that are considered for
//partitioning. s is the source vertex and i is a variable for keeping track of the
//number of partitions computed. X and Y are temporary sets of vertices. β is the
//degree of benefit. C is a loop control variable, T[β, X] is a table to hold partitions.
//D is the vertex distance and n is the required number of partitions.
01:   $Q = V \setminus s$; T [β, X] = Φ; C = false; i = 0;
02:   do {                                        //Outer Loop Start
03:     X= Φ;
04:     N = Set of vertices in Q with maximum D;   (Boundary Vertices)
05:     β = 0, ν = null;        // ν is a temporary variable for a selected vertex
06:     do {                                      //Inner Loop Start
07:         ν = Vertex in N with maximum degree of benefit;
                                                  //see algorithm in Figure 5-3
08:         If ν exists {
09:           $X = X \cup ν$;
   //w and ε are temporary variables storing sum of the weights of vertices and the edge
   //cut weight respectively
10:           w = Sum of weights of vertices in X;
11:           ε = Edge cut weight of set X;
12:           If $(w - ε) > β$ {
13:             $β = w – ε$; C = true;
14:             N = Set of vertices in Q that are directly connected to vertices of X;
15:           }
16:           else { 17:                X = X – ν; C = false; }
17:         }
18:     } while (C == true)                       //Inner Loop End
19:     if $β > 0$ {
20:         add (β, X) as one candidate partition into table T;
21:         i++;  Q = Q – X;
22:     }
23:   } while (i != n  and β > 0 and Q != Φ)       //Outer Loop End
24:   Table T contains partitions for graph G
```

Figure 5-2: The CAP algorithm for mobile application partitioning

```
// X and N are sets of vertices that are described in Figure 5-2. u is a temporary
//variable for storing the vertex with the maximum degree of benefit. Z is a
//temporary variable for holding vertices of set X. κ is a temporary variable for
//holding a selected vertex from set N and γ is the degree of benefit for set Z.
u = null; γ = 0;
Z = X;
for (κ in N) {                       // select one vertex at a time
        Z = Z U κ;
   // w' and ε' are temporary variables storing sum of the weights of vertices and
   //the edge cut weight respectively
        w' = Sum of weights of vertices in set Z;
        ε' = Edge cut weight of set Z;
        If (w' – ε') > γ {
                u = κ;
                γ = w' – ε';
        }
        Z = Z – κ;
}
return u;
```

Figure 5-3: Algorithm for computing a vertex with the maximum degree of benefit from a set of vertices

- *Select a starting vertex (v) from set N* that produces the maximum degree of benefit (line 7 in Figure 5-2).
    - For this selection, all vertices in set N are analyzed and the vertex which produces a maximum degree of benefit when added to set X is selected. The details of the steps used for determining this vertex that produces the maximum degree of benefit are captured in the algorithm shown in Figure 5-3. Note that u, the value returned by this algorithm, is assigned to v (line 7 in Figure 5-2).

- *If a vertex v is found, evaluate the selected vertex* (lines 9-18 in Figure 5-2) by performing the following actions:
    - If vertex *v* is not found based on the evaluation performed in lines 9-18 of Figure 5-2, skip all the actions and go to line 21 directly.
    - *Add selected vertex 'v' to a set X* (line 9 in Figure 5-2). *X* is a set of vertices that represents a candidate partition for offloading.
    - *Calculate w and ε* (line 10 and line 11 in Figure 5-2). As already discussed, *w* is the sum of the weight of vertices in *X* and *ε* is the weight of the edge cut that separates the vertices in set *X* from the rest of the graph *{Q\X}*.
    - *Compare current value of β with the difference between w and ε* (lines 12-22 in Figure 5-2). Note that *β* is initialized to zero in line 5 of Figure 5-2.
        - If the difference between *w* and *ε* is greater than or equal to the current value *β,* then update the current value of *β* (to the difference of *w* and *ε*), set C equal to true and select a new set of neighbor vertices (line 14 in Figure 5-2). Note that the value of *β* represents the degree of benefit in offloading the partition represented by vertices of set *X.*
        - If the difference between *w* and *ε* is less than the current value *β*, the selected vertex is removed from set X as it is not improving the overall degree of benefit *(β)* for set X. As already explained, the CAP algorithm only selects a new vertex when the selected vertex results in an improvement in the degree of benefit of the selected partition (X).

- *Repeat steps in lines 7-16* as long as it is possible to select a new vertex that results in improving degree of benefit *(β)* of set X.
- *Store partition X in Table T* (lines 20-21 in Figure 5-2) if the degree of benefit for set X is greater than zero (line 19 in Figure 5-2). When it is not possible to find a vertex in Q that can improve the degree of benefit *(β)* of set X (as evaluated in line 12 of Figure 5-2), then store the current set X as a partition in table T if $\beta_{max}(X)$ is greater than zero (line 19 of Figure 5-2). Note that table T contains the partition set and the value of the maximum degree of benefit associated with it. For the next iteration, set Q is updated by removing vertices that are selected as a partition in set X (line 20 of Figure 5-2). This updated set Q is used to find the next partition.
    - Once a partition is stored in table T, the partition count variable (*i*) is incremented (line 21 of Figure 5-2) and set Q is updated by removing the vertices of set X from it (line 21 of Figure 5-2).
- *Repeat steps in lines 2-23 of Figure 5-2* to find another partition using an updated set Q. The steps in lines 2-23 are repeated as long as following conditions are true.
    - required number of partitions is not achieved (i != n condition in line 23 of Figure 5-2)
    - set Q is not empty (Q != Φ condition in line 23 of Figure 5-2)
    - it is possible to find a vertex '*v*' in set Q that can give rise a positive value of the degree of benefit ($\beta > 0$ condition in line 23 of Figure 5-2).

Once the algorithm completes its execution, rows in table T represent partitions with corresponding degrees of benefit.

5.6 Using Remote Computing Node Information

The proposed (MOMC and CAP) algorithms can also use information of remote computing nodes that includes their processing speed while determining partitions of an application. For example, the processing speed up factor (S) for the remote computing node can be used to estimate a more accurate degree of benefit ($w - \varepsilon$). The speed up factor is the ratio of the CPU speed of a remote computing node and the CPU speed of the mobile device. When such information is available and is required to be used, the operation used for the computation of ω in both algorithms (line 9 in MOMC and line 16 in CAP) will be replaced by the following operation:

$$w = \sum (W_{V_j}) / S \qquad \text{where } V_j \in X \qquad (1)$$

Dividing $\sum (W_{V_j})$ by S results in reducing the degree of benefit ($w - \varepsilon$). Thus, it is expected that both algorithms may offload partitions of smaller sizes when remote node information is used. But the partitions achieved with MOMC by using the remote node information are expected to exhibit better performance than no partitioning.

The use of the MOMC and the CAP algorithms without using remote information is also useful because both algorithms attempt to offload a substantial part of an application from a mobile device to a remote computing node leading to an improvement in system performance. This can minimize the use of the resources on the handheld mobile device (such as the power, CPU and memory). The improvement in overall response time of an application using the partitioned systems achieved with MOMC and CAP is occurring because the remote computing node is more powerful than the handheld mobile device.

A detailed analysis of the proposed algorithms is performed by using a system prototype and a simulator. These are discussed in the next two sections.

5.7 Experimental Analysis using a System Prototype

System performances achieved by the MOMC algorithm and the CAP algorithm are compared with the system performance achieved with other partitioning techniques that are discussed in the literature. The other partitioning techniques are described next.

5.7.1 Partitioning Techniques Used

Effectiveness of the two proposed algorithms is compared with a no partitioning technique, an offloading entire application technique and the technique that uses the MinCut algorithm.

5.7.1.1 No Partitioning Technique

In this case, no partitioning is performed and the entire WS application is run on a resource constrained mobile device. The results of executing the WS application entirely on a mobile device helps in understanding the performance difference resulting from the partitioned systems achieved by the various partitioning techniques experimented with.

5.7.1.2 Offloading Entire Application (OEA) Technique

Previous techniques ([Hem05], [Kim07] and [Riv07]) suggest that the execution of the entire WS application on a more powerful computing node in the fixed infrastructure. These techniques are based on migrating the entire application from one node to the other without considering any dependency on local resources.

5.7.1.3 Partitioning using MinCut Algorithm (MinCut)

There is a number of algorithms proposed in the literature for determining the minimum cut in a graph. For the performance comparison of the partitioned systems, a tool available online, Internet Accessible Program Packet for Graph Algorithms (IAPPGA), [Wu05] is used. The tool provides different graph based algorithms including the Edmonds-Karp Max-Flow MinCut algorithm [Edm72]. There may be other algorithms available for achieving MinCut of a graph such as the simple MinCut algorithm [Sto97]. Note that the execution time or complexity of the MinCut algorithm does not concern the relative performance of the partitioned systems presented in this chapter because the partitioning is not performed at runtime.

Algorithms based on recursive bisection are not considered because of their complexity. Such algorithms are suitable for large scale scientific applications and are not considered in the context of mobile web services that are expected to be of much lower complexity.

5.7.2 Sample WS Application – Tracking WS

For this analysis, three different versions of a Tracking WS application are used. These versions are of different complexity. Prototypes of the Tracking WS application are invoked in a wireless environment using a mobile device. The details of the experimental environment are discussed in Section 5.7.4. The Tracking WS hosted on a mobile device can be invoked to get the location of a person carrying the mobile device. Such an application can be used in medical emergency situation, for example, for locating a doctor [Mes02]. The Tracking WS application can also be used to locate the current position of a shipment in a moving truck, the driver of which is carrying the mobile device [Asi07-1]. The Tracking WS is chosen because it has one or more components that need local resources of the mobile device and it is easy to vary its complexity (by adding more components), which is important for the performance analysis presented in this section.

The prototype of Tracking WS1 is implemented in Java ME. Three different versions of Tracking WS are considered: Tracking WS1, Tracking WS2 and Tracking WS3. Tracking WS1 provides basic location information such as the closest city name, population etc. The Tracking WS2 and WS3 applications also provide a rescaled and transcoded image of the location in addition to the basic location information respectively. For Tracking WS2 and Tracking WS3, instead of rescaling and transcoding the retrieved image, the CPU execution time is emulated on the wireless mobile device by burning CPU cycles for processing costs of image processing related component using a while loop. The communication costs are only emulated if the components are to be executed on two different nodes.

The processing costs of the different components used in the three versions of Tracking WS are estimated by executing the application on the mobile device (Dell Axim PDA) a number of times (in the range of 10,000) and then taking the average of the execution times for each component. The communication cost is estimated based on the number of bytes transferred when the components are executed on different computing nodes. Note that the number of bytes transferred includes the parameters and the SOAP header.

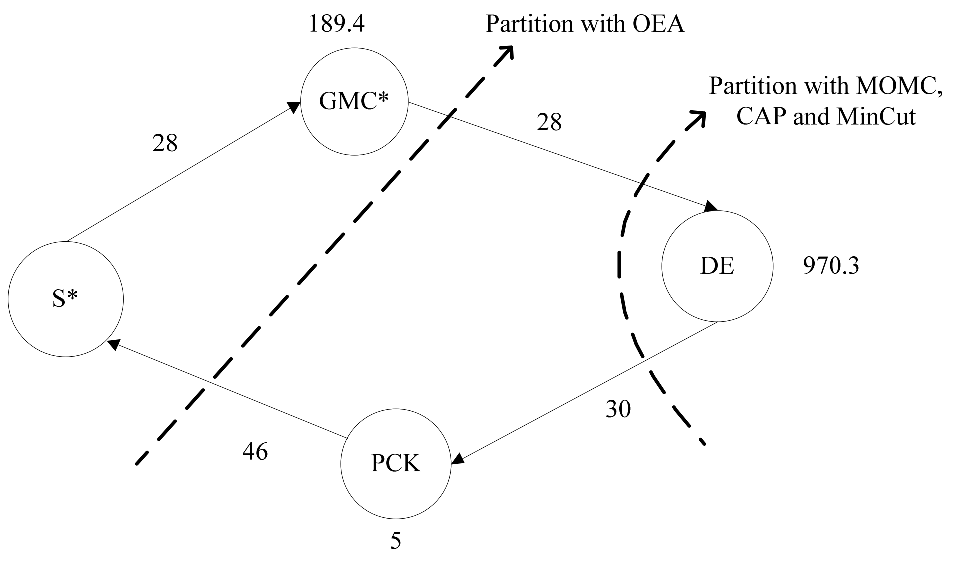
The different versions of the Tracking WS application are described next.

5.7.2.1 Tracking WS 1

This is the simplest version that uses only three components to achieve its goal. The components of Tracking WS1 include *GetMyCoordinates (GMC), DetailExtractor*

(DE) and *Packing (PCK)*. On receiving a WS request, GMC uses a Global Positioning System (GPS) receiver to get the GPS coordinates of the location. Since GMC uses a local resource (GPS receiver emulator) it is not to be offloaded to a remote computing node. Note that for this version of Tracking WS, the step of getting the GPS coordinates is emulated by fetching a random set of coordinates from a local file containing more than a thousand locations. This step is emulated because it was not possible to use a real GPS receiver inside a lab environment. The GPS receivers work when there is a direct line of sight with satellites. DE queries a database of locations to find the details of a location that is closest to the GPS coordinates. The details of the location include the closest city name, time zone, elevation and population. PCK serializes the location information as a response message and sends it back to the WS requester. The interaction of these three components is captured in a graph G (V, E) presented in Figure 5-4.

Note that vertex 'S' represents the starting component of the WS application and is tagged as '*' in all versions of Tracking WS. A component is tagged with '*' when it must be executed locally on the mobile device because of its dependence on a local resource (e.g. the local file on the mobile device in the Tracking WS). The solid directed lines represent the edges between different components (vertices) of Tracking WS1. The labels on every edge show communication cost (in msec) and the label on every vertex indicates the processing cost (in msec). The edge cuts computed with different partitioning techniques are shown in the table of Figure 5-4 (also shown by two broken lines). For this version of Tracking WS, the partition {DE} is selected for offloading by all the three algorithms: MinCut, MOMC and CAP (see Figure 5-4). For the OEA technique, the partition {DE, PCK} is offloaded to the remote node.

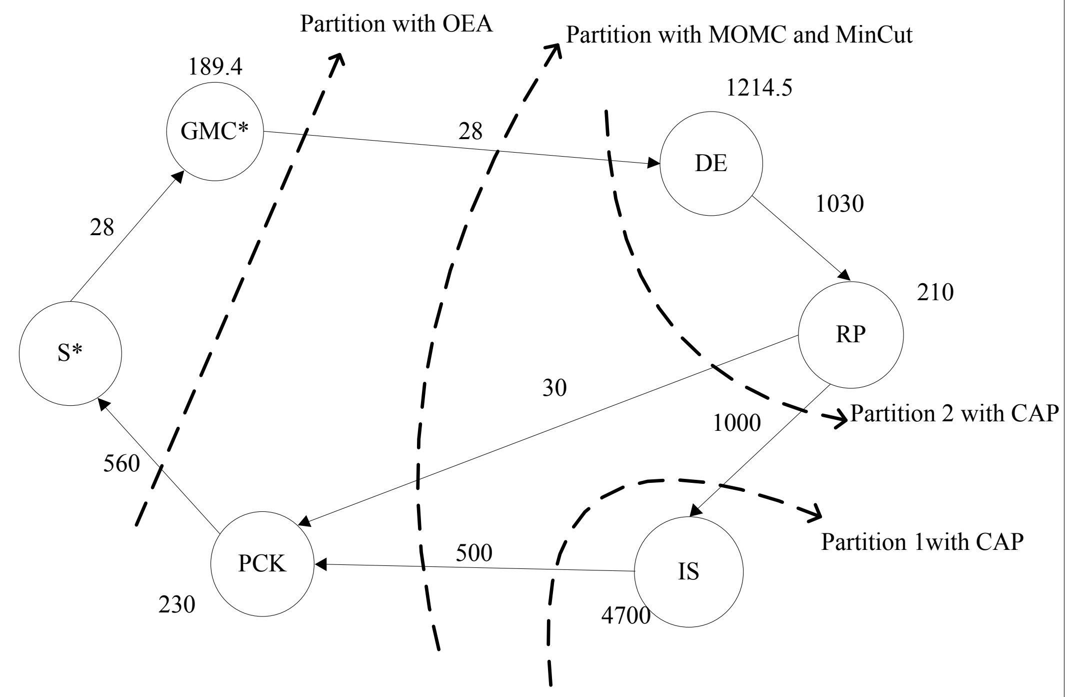


	Partition for mobile device	Partition for offloading
OEA	{S, GMC}	{DE, PCK}
MOMC/CAP/MinCut	{S,GMC, PCK}	{DE}

Figure 5-4: Partitioning of Tracking WS1 achieved with the three techniques: OEA, MinCut, MOMC and CAP

5.7.2.2 Tracking WS 2

Tracking WS2 has two additional components. The components used in this version are: *GetMyCoordinates (GMC), DetailExtractor (DE), ResponseParser (RP), ImageScaler (IS)* and *Packing (PCK)*. The functionalities of GMC and DE are the same as described in the last subsection except that DE fetches the location information from an external service such as Google Map Service [Goo08]. The CPU cycles spent in fetching the location information from the external WS are emulated. For this version of Tracking WS, it is assumed that location information also includes the image of the location. The component RP parses the location information and passes non-image data to the PCK component and the image data to the IS component. The IS component resizes the image according to the requirements of the WS requester. The PCK component performs the serialization of the location information and the resized image data into a response message and sends it back to the WS requester. Tracking WS2 is captured as a graph G (V, E) that is shown in Figure 5-5. Note that the labels on every edge show communication cost (in msec) and the label on every vertex indicates the processing cost (in msec).



	Partition for mobile device	Partition for offloading
OEA	{S, GMC}	{DE, RP, IS, PCK}
MOMC/MinCut	{S, GMC, PCK}	{DE, RP, IS}
CAP	{S, GMC, PCK}	{DE, RP},{ IS}

Figure 5-5: Partitioning of Tracking WS2 achieved with the three techniques: OEA, MinCut, MOMC and CAP

For this WS application, the typical sizes of the images (40-60 KB) available from Google Map Services are estimated and then an average size of such images is used for the estimation of communication costs among DE, RP, IS, PCK components. The IS component is assumed to resize the retrieved image to half in size. The processing cost of IS is estimated by executing a prototype of the Bicubic Interpolation algorithm [Rus02] that is used for rescaling of the retrieved image on the mobile device.

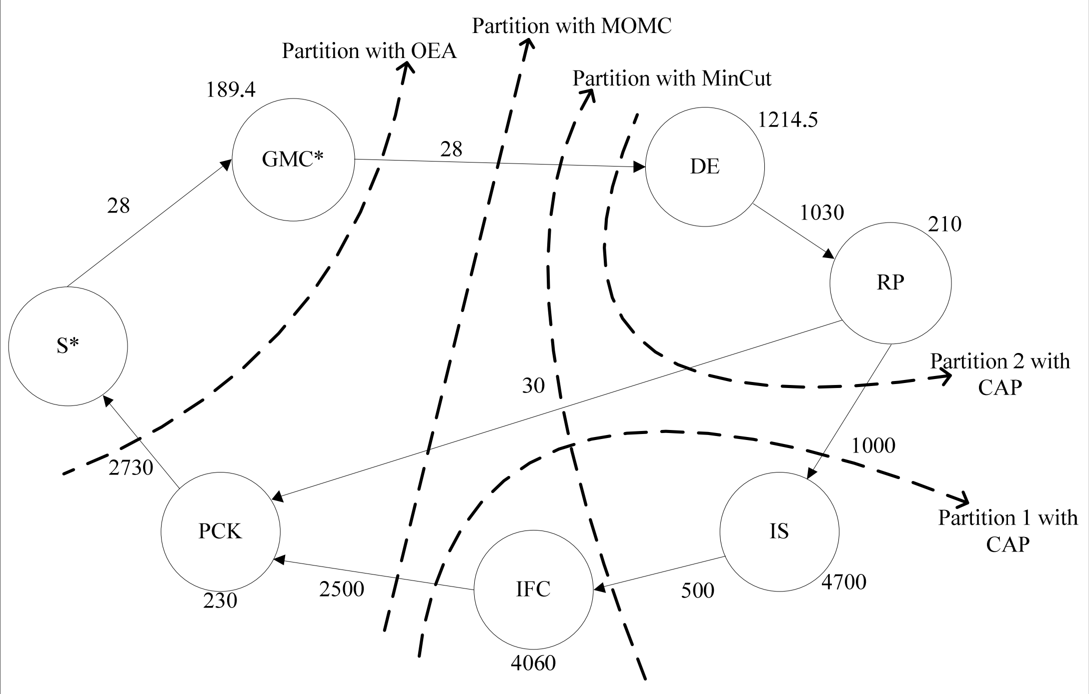
The table in Figure 5-5 shows the two partitions (also shown by broken lines on graph) that are selected by different partitioning techniques for offloading. The partition {DE, RP, IS} is selected for offloading by MinCut and MOMC (see Figure 5-5). Two partitions {IS} and {DE, RP} are achieved for offloading with the CAP algorithm based technique. For the OEA technique, the partition {DE, RP, IS, PCK} is executed on the remote node.

5.7.2.3 Tracking WS 3

This version of Tracking WS has one additional component in comparison to Tracking WS2. The components used in this version are: GetMyCoordinates (GMC), DetailExtractor (DE), ResponseParser (RP), ImageScaler (IS), Image Format Changer (IFC) and Packing (PCK).

For Tracking WS3, it is assumed that the retrieved image needs to be transcoded before it can be displayed on the WS requester's node (mobile device). The WS requester may require the image in a particular encoding. For example, the retrieved image may be in lossy Joint Photographic Experts Group (JPEG) format but the WS requester is expecting the image in a lossless Portable Network Graphics (PNG) format. So the role of IFC is to change the format of the fetched image in accordance with the requirement of the WS requester. The processing cost of IFC is estimated by executing the prototype of the Java ImageIO library [Jav04] API for transcoding the JPG image into a PNG image. The functionalities of the rest of the components are the same. The Tracking WS3 is captured as a graph G (V, E) is shown in Figure 5-6.

Note that the labels on every edge show communication cost (in msec) and the label on every vertex indicates the processing cost (in msec).



	Partition for mobile device	Partition for offloading
OEA	{S, GMC}	{DE, RP, IS, IFC, PCK}
MinCut	{S,GMC, IFC, PCK}	{DE, RP, IS}
MOMC	{S,GMC, PCK}	{DE, RP, IS, IFC}
CAP	{S, GMC, PCK}	{IS, IFC}, {DE, RP}

Figure 5-6: Partitioning of Tracking WS3 achieved with the four techniques: OEA, MinCut, MOMC and CAP

The table in Figure 5-6 shows the different candidate partitions (indicated by broken lines on graph as well) that are selected for offloading by using the different partitioning techniques. The partition {DE, RP, IS} is selected for offloading by the MinCut algorithm, the partition {DE, RP, IS, IFC} by the MOMC algorithm, the partitions {IS, IFC} and {DE, RP} by the CAP algorithm and the partition {DE, RP, IS, IFC, PCK} by the OEA technique. Note that for this WS application, the partitions achieved with MOMC and MinCut are different. Workload and System Parameters

The following workload and system parameters are varied during the experimentation.

5.7.2.4 WS Complexity:

The proposed algorithms are analyzed by varying three different versions of Tracking WS. The three Tracking WS versions vary in complexity as discussed in Section 5.7.2.

5.7.2.5 Number of WS Clients (C):

The number of WS clients invoking one of the three versions of Tracking WS at a time is varied to investigate the scalability of the system.

5.7.2.6 Mobile Device Speed (ω):

The experiments are run on a real handheld device using different processing speeds configurations. The Dell Axim PDA device used in the experiments can be run with different processing speed. The option of running a mobile device at different processing speeds is provided to manage the battery power efficiently.

5.7.3 Performance Metrics

The performance of the system is analyzed by measuring the end to end ***response time (R)***. Response time is defined as the difference between the time when a WS response is received by the WS client and the time when the WS client sends the corresponding SOAP message request to a WS provider.

5.7.4 Experimental Setup

For the system prototype based performance analysis, the WS client program that is used to access different versions of Tracking WS is written using standard Java. The WS clients are run on a laptop equipped with 2 GB of RAM and an AMD Turion 64

X2 processor with speed of 1.9 GHz. The remote computing node is a desktop computer equipped with Intel Quad Core 2 processor that is running under Ubuntu Linux operating system. Its CPU speed is 2.4 GHz and 3 GB of RAM is available on this node. Various versions of the tracking WS application are deployed on a Dell Axim x51v PDA. The PDA used as the mobile device has an Intel XScale ARM processor (PXA270) that can be run at multiple speeds (208 MHz, 520 MHz and 624 MHz). The Dell Axim PDA used is equipped with a 64 MB of RAM and runs the Windows Mobile 5.0 operating system. The Java ME environment (J9) available on the PDA is a JVM (J9) provided by IBM [Ren09]. The installed JVM is based on the specification of Connected Device Configuration (CDC) 1.1 [CDC05]. The client machine and the remote computing node are on a fixed network and communicate with the PDA using an IEEE 802.11compliant wireless local area network.

The partitions of sample WSs are hosted on the mobile device and the remote computing node using a lightweight web service execution environment (WSEE) described in Section 3.2. Note that the backend framework is used for investigation of WS partitioning techniques presented in this thesis.

5.7.5 Performance Results

The effectiveness of the proposed algorithms is analyzed by comparing the performance of partitioned Tracking WS applications achieved with MOMC, CAP and the other partitioning techniques. In all the experiments, the mean response times of the three versions of Tracking WS are measured when

- the application is deployed entirely on mobile device (no partitioning case – NPC),
- the application is partitioned based on the OEA technique,

- the application is partitioned using the MinCut algorithm (MinCut),
- the application is partitioned using the proposed MOMC algorithm,
- The application is partitioned using the proposed CAP algorithm.

Only one version of Tracking WS is invoked at a time. In the first set of experiments, the performance of partitioned systems for all three versions of Tracking WS is investigated for the situation in which only one WS client is used. In the second set of experiments, the sample Tracking WS applications are accessed using multiple WS clients for accessing the Tracking WS applications. The third set of experiments analyzes the effect of device speed. For this set of experiments, Tracking WS1 is invoked by 1, 2, 3, 6 and 12 WS client at a time and the mobile device is operated at two different CPU speeds.

For all the experiments, a closed system model is used for experimentation. Each client (a Java thread) operates cyclically and sends one request at a time. As soon as the response is received, the client repeats the cycle. The system is stressed by increasing the number of concurrent WS clients. For a single experiment, each client generates 10000 requests. So for an experiment with 10 WS clients, for example, the response time is calculated by taking the average of response times of 100,000 requests. Each experiment is repeated 10-30 times to obtain sufficiently small confidence intervals for the average values. For the experiments presented next, confidence intervals of ±5% (or less) for mean response time were obtained at a confidence level of 95%.

A Java system API that provides a milliseconds level accuracy was used in measurement of time (System.currentTimeMillis()). Such a number of requests and resolution of measurement for time were found to provide adequate measurement accuracy required for analysing the relative performances of the partitioned Tracking WS based on the system prototype.

5.7.5.1 Performance of the Tracking WS When Accessed by Single WS Client

In this experiment, the mean response times of all the three sample WS applications are measured when invoked by a single WS client for both CPU speeds of the mobile device using all partitioning techniques (NPC, OEA, MinCut, MOMC and CAP). The mean response times observed for all three versions of Tracking WS when accessed by a single WS client are shown in Figure 5-7, Figure 5-8 and Figure 5-9 respectively. It is interesting to observe that the the performance of the Tracking WS2 application and the Tracking WS 3 application, when partitioned using any of the partitioning techniques, is superior to the no partitioning case. The results of individual version of Tracking WS are discussed next.

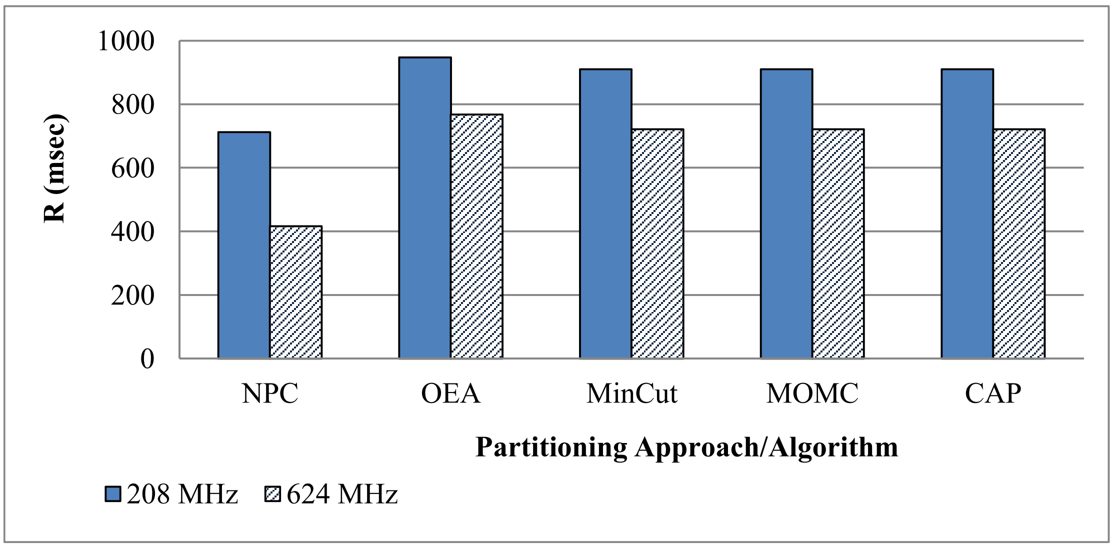


Figure 5-7: Performance of the partitioned systems achieved using different techniques (NPC, OEA, MinCut, MOMC and CAP) for Tracking WS1

The mean response times observed for Tracking WS1 when accessed by a single WS client are shown in Figure 5-7. As discussed in Section 5.7.2.1, the same partitioned system is achieved with MinCut, MOMC and CAP techniques. The performance of the partitioned system achieved with these three techniques and the OEA approach is observed to be inferior in comparison to the one achieved with the NPC technique. This is because Tracking WS1 has a small processing demand and the communication overheads offset the benefit of offloading partitions.

Note that Tracking WS2 performs image processing, so it has significantly more computations to perform in comparison to Tracking WS1. For Tracking WS2, MinCut and MOMC gives rise to the same partition for offloading (see Figure 5-5). The CAP algorithm achieves two partitions although when the two partitions are combined they result in same partition as achieved with MinCut and MOMC. The partitioned application achieved with MinCut and MOMC performs better than the partition applications achieved with the OEA technique and the CAP algorithm for both CPU speeds of the mobile device (see Figure 5-8). The partitioned application achieved with CAP performs better than the one achieved with OEA. Although the OEA technique results in offloading majority of the application components to a remote computing node, it results in introducing significant communication overheads and thus results in degraded performance. The performance of the partitioned application achieved with the CAP algorithm based technique is marginally inferior to the performance achieved with MinCut and MOMC because of the additional overheads that accrue as a result of the coordination of multiple partitions achieved with the CAP algorithm.

For Tracking WS3, the partitioned application obtained using MOMC gives rise to the best performance. Surprisingly, the performance achieved with the MinCut algorithm is inferior to that achieved with the OEA technique. This trend is observed for both CPU speeds of the mobile device (see Figure 5-9). The performance of the partitioned system achieved with the CAP algorithm is better than that for OEA and the MinCut techniques but inferior to the performance of MOMC.

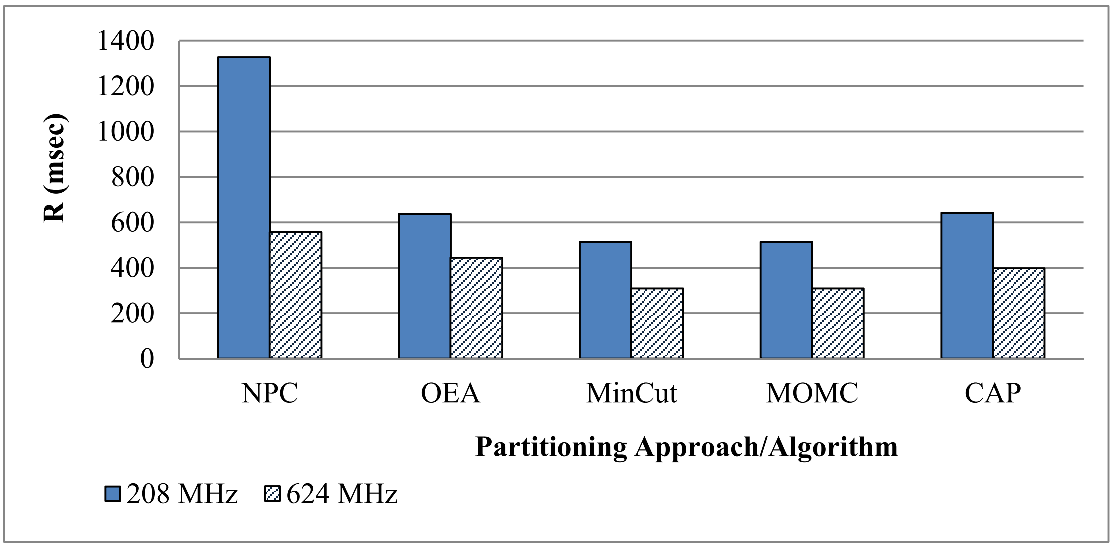


Figure 5-8: Performance of the partitioned systems using different techniques (NPC, OEA, MinCut, MOMC and CAP) for Tracking WS2

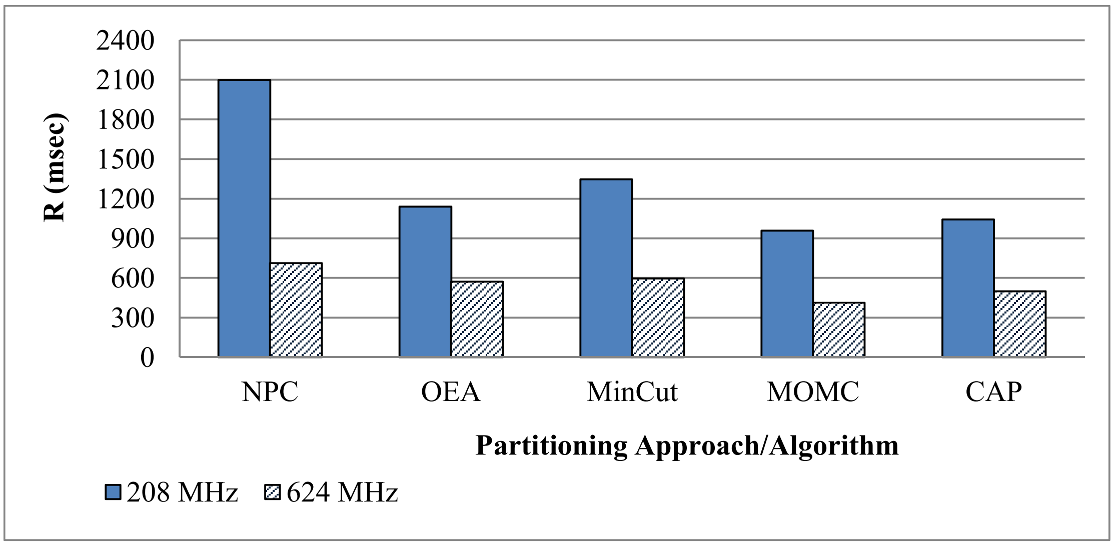


Figure 5-9: Performance of the partitioned systems achieved using different techniques (NPC, OEA, MinCut, MOMC and CAP) Tracking WS3

5.7.5.2 Performance of Tracking WS When Accessed by Multiple WS Clients

In this experiment, all three versions of Tracking WS are invoked by multiple (1, 2, 3 and 6) WS clients. The experiments are performed with a mobile device at the CPU speed of 624 MHz. It was not possible to run the experiments with more than six concurrent WS clients and for the CPU speed of 208 MHZ. For both cases, the time required for executing the entire application on the mobile device is inordinately large. The large response times resulted in triggering timeouts for the transport protocol that is used to exchange data between WS clients and the WS application. The mean response times observed when the sample web services are accessed by multiple WS client are shown in Figure 5-10 (for Tracking WS1), Figure 5-11 (for Tracking WS2) and Figure 5-12 (for Tracking WS3). Note that these graphs presented in Figure 5-10, Figure 5-11 and Figure 5-12 are plotted using a $\log_{10}$ scale for the response time (Y-axis).

The relative performance achieved with the partitioning algorithms does not change significantly as the number of WS clients is increased (see Figure 5-10, Figure 5-11 and Figure 5-12). However, the performance improvement produced by a partitioned system over an un-partitioned system is observed to increase with multiple clients. As the number of concurrent clients is increased, the mean response times of the Tracking WS applications also increase linearly when one of the partitioning techniques is used. For Tracking WS2 and Tracking WS3 (see Figure 5-11 and Figure 5-12), the applications take inordinately large amounts of time when they are executed entirely on the mobile device and invoked by more than 1 WS clients at a time. Such performances are not acceptable and illustrate the importance of WS partitioning.

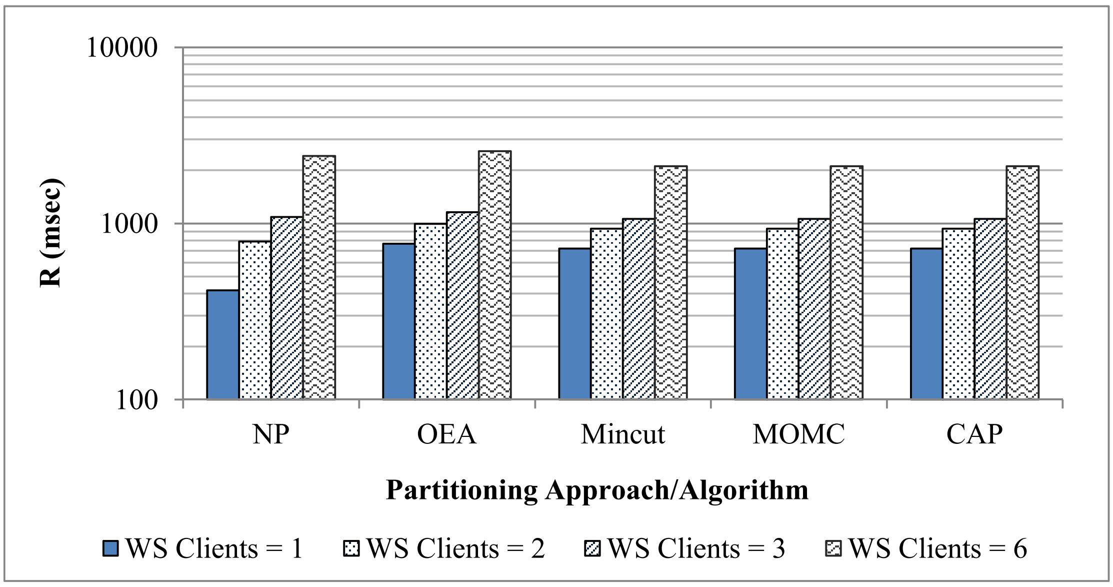


Figure 5-10: Performance comparison of the partitioned systems achieved with NPC, OEA, MinCut and MOMC and invoked by multiple WS clients for Tracking WS1

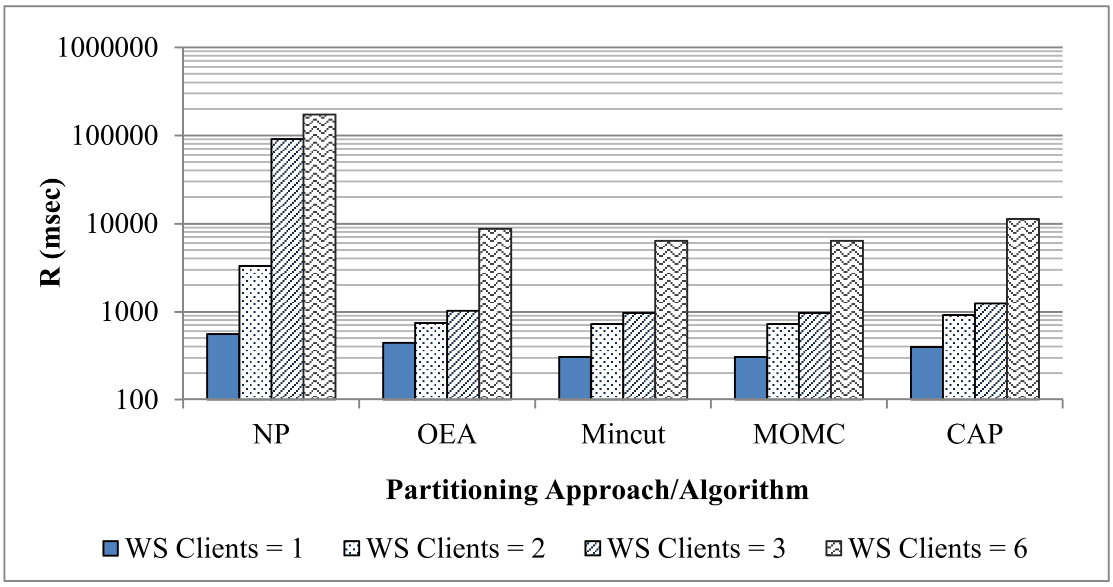


Figure 5-11: Performance comparison of the partitioned systems achieved with NPC, OEA, MinCut and MOMC and invoked by multiple WS clients for Tracking WS2

For all three versions of Tracking WS, the partitioned system achieved with the MOMC technique exhibit the best performance. The performance improvement observed for the partitioned system using MOMC over other partitioning algorithms is higher when:

- a higher number of WS clients are active (see bar 3 and 4 in Figure 5-10, Figure 5-11 and Figure 5-12).
- a complex WS application is accessed (see Figure 5-12).

The performance of the partitioned system using the CAP algorithm is observed to be marginally better than the system using OEA. The additional overheads that accrue as a result of the coordination of multiple partitions achieved with the CAP algorithm are the main cause of its inferior performance in comparison to the OEA and the MOMC based techniques.

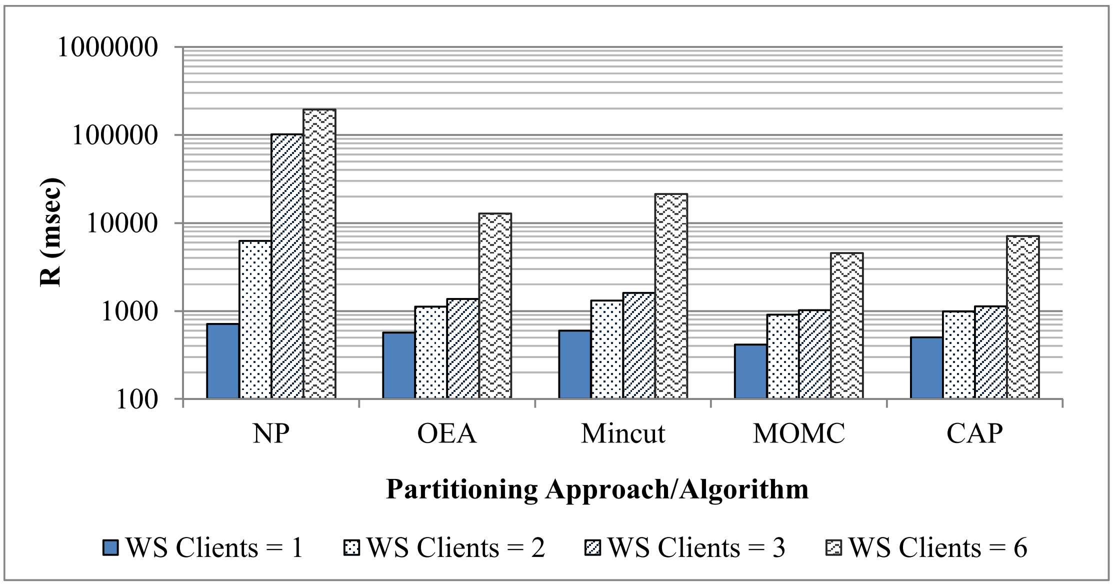


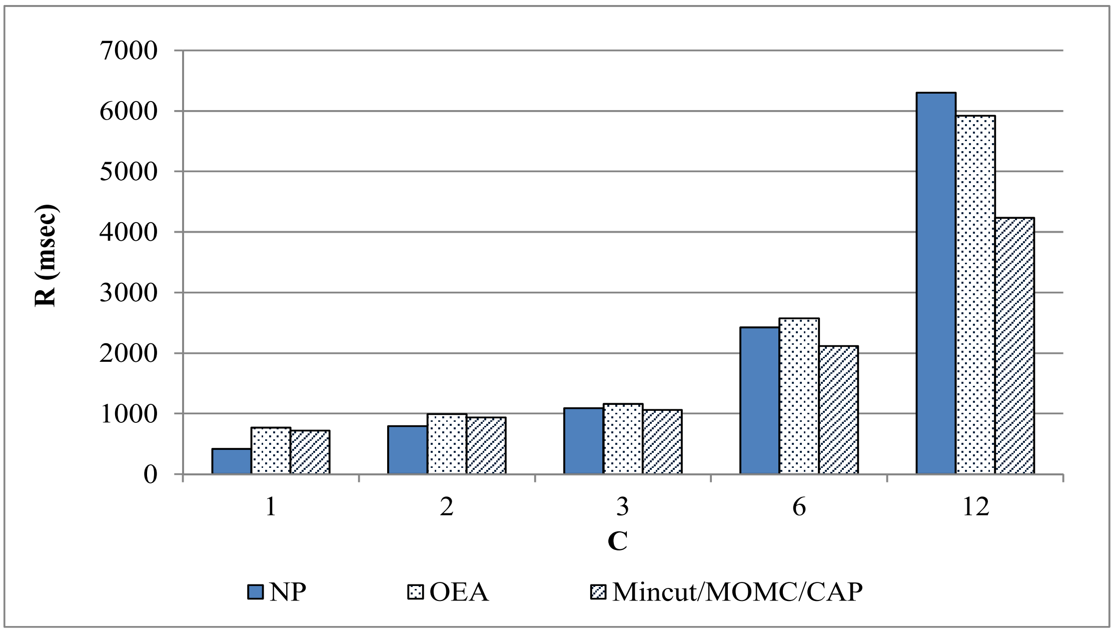
Figure 5-12: Performance comparison of the partitioned systems achieved with NPC, OEA, MinCut and MOMC and invoked by multiple WS clients for Tracking WS3

The performance achieved with the MinCut algorithm is again inferior to that achieved with the OEA technique. The results of experiments presented in this section shows that the partitioning based on the knowledge of edge weights only (MinCut) can lead to an inferior performance for mobile web services in comparison to the other partitioning techniques.

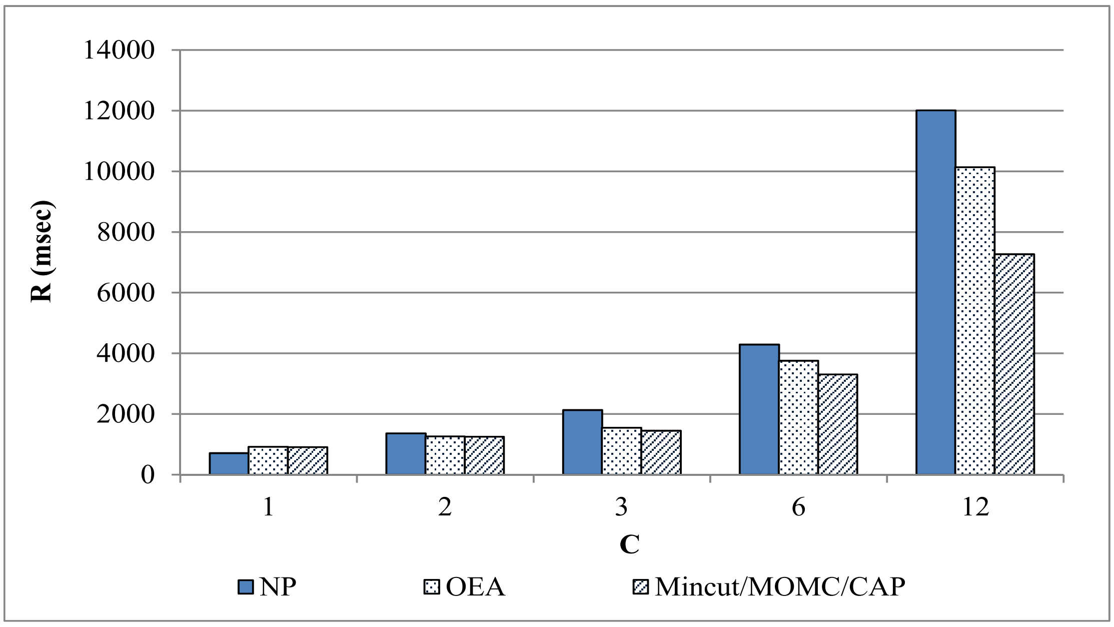
5.7.5.3 Effect of the Speed of the Processing Resources (ω)

In this set of experiments, a detailed analysis of the partitioned system for Tracking WS1 is performed using single and multiple WS clients and two different processing speeds (208MHz and 624 MHz) for the mobile device. Operating the mobile device at two different CPU speeds enables the investigation of the impact of applying application partitioning on different types of mobile devices. The number of WS clients is varied from 1 to 12. The performance results are captured in Figure 5-13.

Figure 5-13 shows the mean response time achieved with Tracking WS1 when multiple WS clients are active. When invoked by one WS client for both CPU speeds of the mobile device, the performance of a partitioned system achieved by using any of the partitioning techniques is inferior to that of an un-partitioned system. This trend is also observed when the service is invoked by 2 WS clients for 624 MHz CPU speed of the mobile device. This is because Tracking WS1 has a small processing demand and the communication overheads offset the benefit of offloading partitions. Note that for this application, MinCut, MOMC and CAP give rise to the same partitions for offloading; that is why their mean response times are the same in Figure 5-13.



(a)



(b)

Figure 5-13: Performance of the partitioned systems achieved with NPC, OEA, MinCut and MOMC for Tracking WS1 when mobile device is operated at CPU speed of (a) 624 MHz (b) 208 MHz

As soon the number of concurrent WS clients is increased, the performance of partitioned Tracking WS1 achieved by using any of the partitioning techniques (especially with MOMC/MinCut/CAP) is superior to that of an un-partitioned system. This trend has been observed for both CPU speeds of the mobile device (see Figure 5-13-a and Figure 5-13-b). However, the performance improvement produced by a partitioned system with MOMC/MinCut/CAP over the OEA technique is observed to increase more significantly. Also, the MOMC/MinCut/CAP based partitioned systems showed a significantly better performance in comparison to the un-partitioned system and the partitioned system achieved with OEA for slower CPU speed of the mobile device (see Figure 5-13) when it is invoked by more than 2 WS clients.

For the CPU speed of 624 MHz, the OEA technique is observed to show inferior performance (see Figure 5-13-a) in comparison to the un-partitioned system for any number of WS clients except 12. For 12 WS clients, the communication overheads for a system achieved with the OEA technique appear to be lower than the benefit of offloading partitions and thus results in marginally better performance in comparison to the no offloading case.

5.8 Experimental Analysis using a simulator

The experiments performed with the prototype produces limited insights into the behavior of the system. As discussed in the last section, it was not possible to experiment with a very large number of WS clients. The complexity of sample web services is also limited. For a detailed investigation, simulation based experimentation is performed. The WS partitioning techniques based on the proposed algorithms are compared with no partitioning, the offloading entire application based technique and the MinCut algorithm based technique. These techniques are already described in Section 5.7.1.

Before discussing the experimental setup and the results of the simulation based experiments, it is important to discuss the tool for generating random graphs and the simulator used for experimentation. To apply the proposed algorithms to a large variety of graphs, a random graph generating tool, Random Graph Generator (RGG), is devised. The RGG tool is described in Section 5.8.1. The performance of the resulting partitioned systems is measured by using a simulator that is developed for this purpose. The simulator is described in Section 5.8.2. The RGG tool and the simulator are implemented using the standard Java edition. The simulator and the RGG tool used for this thesis are designed for the requirements of this thesis analysis.

5.8.1 Random Graph Generator (RGG)

Generating a random graph requires a careful consideration. The graphs generated without following the WS applications requirements can result in misleading analysis and conclusions. Before discussing the internal details of the graph generating tool, a brief discussion of the templates supported by the tool is presented.

5.8.1.1 Templates

To study the nature of graphs for different applications, the eclipse TPTP profiling tool [Pop10] is used. The profiling tool is used to study Call Tree graphs of a few sample applications (see Appendix B). The results of the profiling tool have been used to determine the templates used in the simulation analysis. Based on this analysis, three types of templates are selected to generate graphs: Linear, Binary and Binary Meshed. These templates are shown in Figure 5-14. The linear template is observed for very simple applications in which a component A invokes another component B that in turn invokes another component C (see Figure 5-14-a). Note that a component can be an application module, a library, a method of a class or even a block of statements.

The binary template is applicable to applications in which every component is invoking two more components to achieve its goals. From the analysis performed with the profiling tool on a few applications as discussed earlier, only small parts of applications are observed to be using this template. A sample graph based on this template is shown in Figure 5-14-b. This is because the template does not represent reusable components and the applications use reusable components very often in modern programming languages. The reusable components are the ones that are being invoked multiple times by different components.

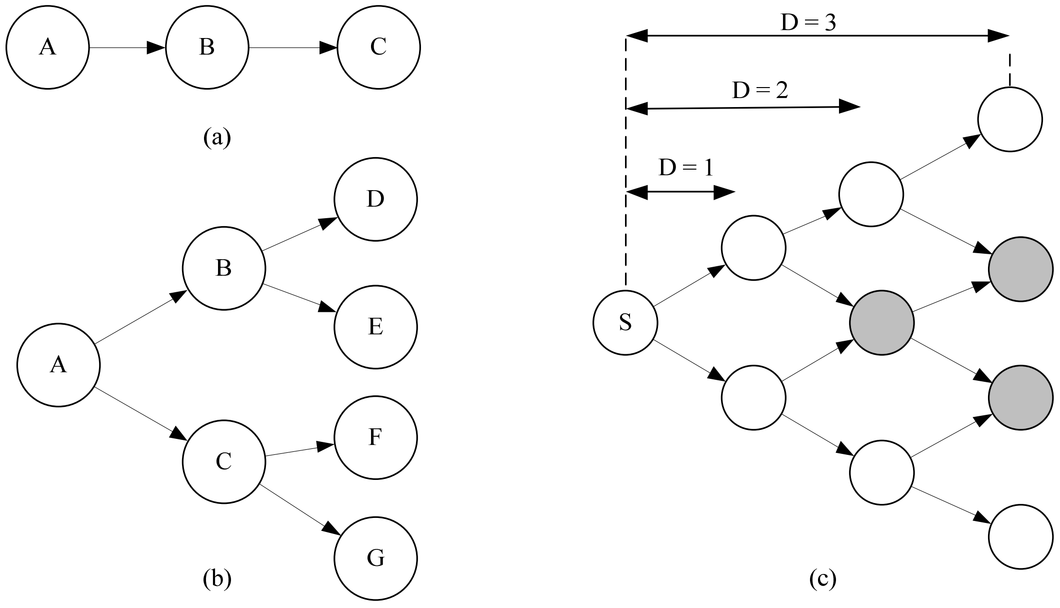


Figure 5-14: Sample directed graphs based on (a) Linear Template (b) Binary Template (c) Binary Meshed Template

To represent reusable components on a graph, a binary meshed template is introduced. A number of components of applications that are analyzed with the profiling tool are observed to be characterized by the mesh template. In the binary meshed template, every component uses one private component and one shared component representing a reusable component such as a third party library component. A sample graph based on this template with a graph size equal to 3 is presented in Figure 5-14-c. The shared components are shown as shaded vertices in Figure 5-14-c. Note that the binary meshed template converges to the binary template if there is no reusable component.

The graph generating tool uses a number of input parameters which are described next.

5.8.1.2 Input Parameters

The input parameters for the RGG tool are presented next.

Graph Template:

The type of graph template to use for generating graphs needs to be provided as an input to the tool. Currently, the tool accepts one of the three types of *templates* described in the last subsection.

Graph size (D_{max}):

Larger the value of this parameter, larger is the graph. Its values can vary from 2 to any finite number.

Mean Vertex Weight (W_{MV}):

It is the mean weight of the vertices in a graph. It is generated using the uniform distribution that ranges from L_{VW} to U_{VW}.

Mean Edge Weight (W_{ME}):

It is the mean weight of the edges in a graph. It is generated using the uniform distribution that ranges from L_{EW} to U_{EW}.

Variability Factor (Δ):

The variability factor indicates the breadth or spread of the uniform distributions:

$\Delta = W_{MV} - L_{VW}$;

In the simulation results presented in Section 5.8, the same value of Δ is used for generating the vertex and edge weights.

Certain edges are assigned high edge weights in the graph. Such edges (called salted edges) represent dependency of a component on local resources and offloading them to a remote node can lead to additional communication costs. For a binary meshed template, from a set of vertices leading to the same value of D, one edge is randomly selected as a salted edge. The edge weight of a salted edge (W_{SE}) is computed based on how far it is from the source vertex. The salted edges close to the source vertex are assumed to have high communication costs in comparison to the ones that are away from the source vertex. This assumption is based on the fact that the vertex representing the local dependencies are more likely to be close to the source vertex as local information is collected first and then actions are performed to accomplish objectives of a WS application. An empirical formula (decaying exponential function) is introduced to compute varying weights for salted edges. The formula also uses the graph size (D_{max}) and the mean edge weight as input parameters:

$$W_{SE} = W_{ME} * e^{(Dmax-D)}$$

Note that D is a vertex distance for the target vertex of the edge and it represents how far the edge is from the source vertex. Also, in all simulation based experiments discussed in this thesis, the time is measured in simulation time units.

5.8.1.3 Components

Different components of the graph generating tool are presented in Figure 5-15 and are explained next.

Graph Controller is a key component of the tool that takes input parameters and interacts with other components (described next) to generate a graph. *Init* takes input parameters and initializes the tool for generating a graph. *Edge Factory* is used to generate an edge. This component is repeatedly used by the graph controller whenever a new edge is to be created. *Vertex Factory* either provides an already created vertex for sharing or creates a new vertex. *RN Generator* is a random number (RN) generator which is used to compute randomly distributed values for edge weights and vertex weights.

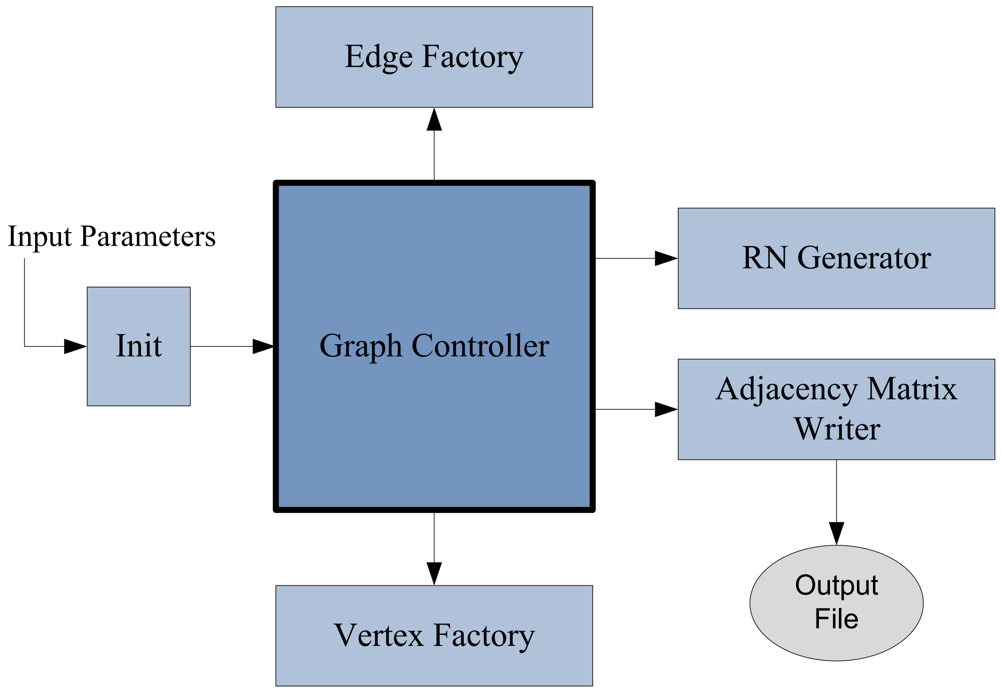


Figure 5-15: Internal details of graph generating tool

Graphs are normally represented by an adjacency matrix. To save a generated graph in an output file, *Adjacency Matrix Writer* is used to generate the adjacency matrix along with a list of vertices and their weights. The output file can be used to analyze the graph using third party tools. In this analysis, the output is used for computing the MinCut of the generated graph.

RGG works as an independent tool as well as a plug-in. As an independent tool, it generates a graph in an adjacency matrix format and saves it in an output file. As a plug-in tool, it outputs the graph as a Java object which can be directly provided to a partitioning tool (for MOMC and CAP algorithms) and the simulator.

5.8.2 Simulator

The simulator can work as an independent tool and as a plug-in. The simulator takes a graph and the value of the speed up factor (S) as input parameters. As an independent application, the simulator takes an input graph in an adjacency matrix format, builds an in-memory graph object from the adjacency matrix and then simulates execution of the application represented by the graph. When it is used as a plug-in with the RGG tool, it takes an in-memory graph as a Java object from the RGG tool and then simulates the execution of the application (graph).

The speed up factor is provided as an input as well. The speed up factor is used to compute the execution time of an application component when it is to be executed on a remote computing node (see Section 5.6). The default value of the speed up factor is 4 but an analysis is performed to investigate its impact on overall performance of partitions when it is varied from 2 to 8 which is close to the speed up of the real devices available in the market.

Working of the simulator is explained with the help of two sample graphs which are presented in Figure 5-16. Note that the simulator can simulate an application with a graph of any size. The technique used for computing the execution time of different components using the graph model of an application is explained next.

Figure 5-16-a shows the execution of a sample application when it is not partitioned and is executed on a single computing node. Different components of this sample application are shown as A, B, C, D, E and Vs vertices. The numbers in the shaded balloons attached with the vertices shown in Figure 5-16-a indicate the order of execution of application components when simulated by the simulator. The order of execution is determined by the simulator as it traverses through the application graph. The execution starts with vertex 'Vs'. Since there are two outgoing edges from 'Vs', the control moves to a vertex connected to one of the outgoing edges of vertex Vs. Let us assume the simulator picks vertex A (as it connects to the edge that comes first in the list of outgoing edges). From vertex A, there are two edges going out to vertex B and vertex D. If the simulator picks the edge going towards vertex B, then control moves to vertex B. Since there is no outgoing edge from vertex B, the weight of vertex B is added to the execution time. Once the weight of vertex B is added to the execution time, control comes back to vertex A. From vertex A, control moves to vertex D through the second outgoing edge. Since there is no outgoing edge from vertex D, the weight of vertex D is added to the execution time and the control comes back again to vertex A. After navigating vertices B and D, vertex A has no more outgoing edge left, the weight of vertex A is now added to the execution time. Since all vertices accessible from vertex A are navigated, control comes back to vertex Vs. From vertex 'Vs', control goes to C, D and E to add their weights to the execution time in a similar fashion as described for A, B and D. Because vertex D is connected to vertex A and vertex C (an example of reusable component), its weight is added twice to the execution time. Since the application graph shown in Figure 5-16-a is not partitioned (running on a single machine), weights of edges are not considered. This is because all components are assumed to be interacting through a shared memory and the communications costs (edge weights) are negligible.

Figure 5-16-b shows an application graph which is partitioned across two computing nodes. In the sample graph, the vertices that are to be executed on a remote computing node with a speed up factor of ‘S’ are marked by a flag (for simulator engine) and are shown with horizontal lines in Figure 5-16-b. Note that the weight of the edges linked to such vertices will be used in computing the execution time and that is why they are indicated in Figure 5-16-b. As an example consider the edge between vertex A and vertex D. The component represented by vertex D is marked (by horizontal lines) as part of the remote partition and the component represented by vertex A is to run locally on a mobile node. Since two vertices are running on different nodes, the commination costs (edge weight) between them need to be considered. Thus, weights of such edges are shown in Figure 5-16-b. The simulator adds weights of such edges to the execution time.

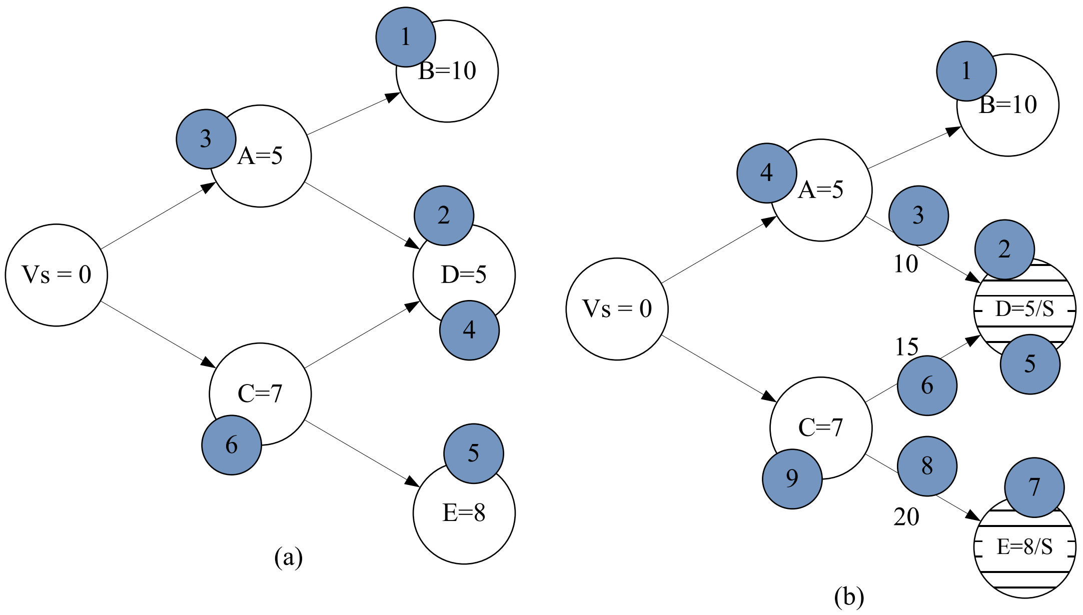


Figure 5-16: Execution of application (a) on a single machine (b) on two machines

As discussed for Figure 5-16-a, the execution of an application starts from vertex 'Vs' and it walks to vertex A and then to vertex B. Simulator adds the weight of vertex B to the execution time as there is no edge going out from vertex B. When the control goes to vertex D as par the order of execution, the simulator identifies vertex D as a component that is to be executed on a remote computing node. For such a vertex (vertex D), the simulator performs following actions:

- Add the weight of the edge between vertex A and vertex D (communication cost) to the execution time. Whenever there is a communication cost involved, the simulator also adds a randomly generated network delay to the execution time. Network delays are modeled by using an exponential distribution [Suk10]. In all the experiments reported in this section, the mean value of the network delay used is 10 time units.
- Add the result of the weight of vertex D divided by the speed up factor to the execution time. The speed up factor represents the ratio of the processing speed of the remote computing node and the processing speed of the mobile node. Note that the weights assigned to each vertex (by the RGG tool) are based on a uniform distribution.

The simulator adds weights of the rest of the vertices and the edges (if they are connecting vertices in different partitions) to the execution time in a same manner as explained for vertex B and vertex D.

5.8.3 Workload and system Parameters

In all the experiments, performance of partitioned systems is measured by varying various system and workload parameters. The workload and system parameters that are varied are presented in Table 5-1. These parameters were defined in Section 5.8.1.

The majority of the parameters are for varying the characteristics of randomly generated graphs. In one experiment, only one of the parameters is varied at one time. All other parameters used in the experiment are set to their default values.

5.8.4 Performance Metrics

The key performance metric is the *mean execution time (τ)*. The execution time (in time units) is the difference between the time when the simulator has finished execution of an application and the time when the simulator receives a request for execution of that application. The mean execution time is computed by taking an average of the execution times measured for all randomly generated applications (graphs) for each experiment. As mentioned in Section 5.8.1.2, the execution time is measured in simulation time units.

Table 5-1: Mean values/Value Range and default mean values used for workload and system parameters

Parameter	***Mean Values / Value Range***	***Default Value***
Graph Template	Linear, Binary, Binary Mesh	Binary Mesh
Graph Size (D_{max})	2 to 24	8
Mean Vertex Weight (time units)	20 to 180	100
Mean Edge Weight (time units)	20 to 180	100
Variability Factor	0.05 to 0.2	0.1
Speed Up Factor	2 to 8	4

Another performance metric used is *percentage improvement (Ƥ)*. Percentage improvement is the proportional reduction in the mean execution time of a partitioned system achieved with one of the partitioning techniques from the mean execution time of an un-partitioned system.

5.8.5 Experimental Results

Four different sets of experiments are performed. In the first set of experiments, the performance of the partitioned systems that are achieved using the proposed algorithms (discussed in Section 5.4 and Section 5.5) for different graph sizes are compared with the performance of an un-partitioned system. In the second set of experiments, the performance of the partitioned systems achieved with the proposed algorithms is compared with the performance of partitioned systems achieved with other partitioning techniques that are discussed in Section 5.7.1. In the third set of experiments, the partitioned systems achieved with the proposed algorithms are analyzed by varying different input parameters described in the last subsection. In the fourth set of experiments, the effect of using the remote node information discussed in Section 5.6 is investigated.

Experimental results reported in this section are obtained with 10,000 graphs. An analysis is performed by experimenting with various numbers of graphs and the accuracy resulting from the experiments with 10,000 graphs is found to be adequate for analyzing the relative performance of the partitioned systems achieved with the various partitioning algorithms discussed in this chapter. Each experiment is repeated 15-30 times to obtain sufficiently small confidence intervals for the average values. For the experiments presented next, confidence intervals of ±5% (or less) for mean response time were obtained at a confidence level of 95%.

The execution time for each non-partitioned application modeled by a graph is determined using the simulator. Next, the graphs are partitioned using one of the partitioning algorithms and execution times of the partitioned systems are determined by the simulator as explained in Section 5.8.2. Results reported in the following subsections are mean execution times and are presented in the form of bar graphs.

5.8.5.1 Analyzing MOMC and CAP Algorithms for Different Graph Sizes

In this set of experiments, performance of the partitioned systems achieved with the MOMC and the CAP algorithms is compared with the performance of the un-partitioned systems. The performance comparison is performed for different graph sizes. The graph size (D_{max}) is varied from 2 to 24. The results presented in Figure 5-17 show the percentage improvement observed in the mean execution time when the MOMC and the CAP algorithms are used. Due to large execution times observed for graphs with graph sizes greater than 10, it is not possible to show the actual mean execution times for all graph sizes using a single scale. That is why Figure 5-17 shows percentage improvement (defined in Section 5.8.4) for MOMC and CAP.

Figure 5-17 presents an interesting pattern in the results. For graph sizes less than 10, the MOMC algorithm is observed to produce partitions that exhibit better performance than the one achieved with the CAP algorithm. For large graph sizes (graph size > 10), however, the percentage improvement of the partitioned systems achieved with the CAP algorithms are observed to be higher than the percentage improvement of the partitioned systems that are achieved with the MOMC algorithm. This is because of the difference in objectives of the two algorithms. MOMC performs a complete walk through over the application graph and it attempts only once to find the best suitable cut in the graph to obtain the two candidate partitions.

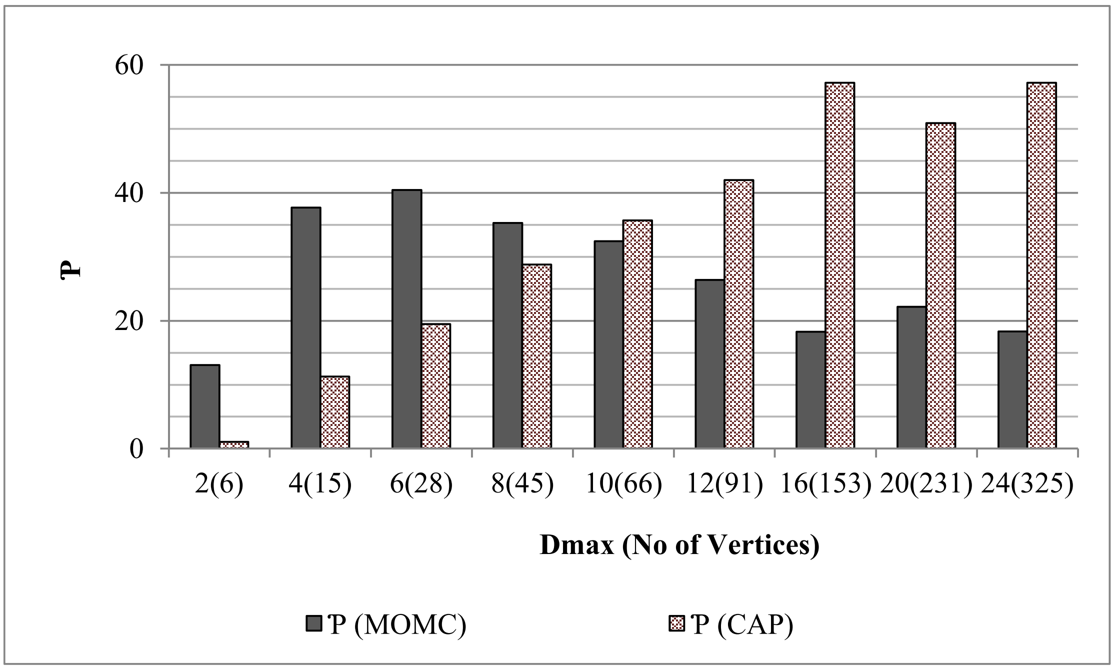


Figure 5-17: Comparison of effectiveness of MOMC and CAP for different graph sizes

For small graphs, the algorithm has a high probability of finding a suitable cut in the graph where one of the disjoint sub-graphs has the maximum difference between the sum of the weights of its vertices and the edge cut weight that separates it from the other sub-graph. For large graphs, the performance of the partitioned systems achieved with MOMC is still significantly better than the performance of the un-partitioned systems. For large graphs, the effectiveness of the CAP algorithm is more significant. It is because it attempts multiple times to get different sets of vertices from the graph to form the partitions. The CAP algorithm first selects a single set of vertices based on the local maxima approach. Once a set of vertices is selected, the algorithm walks through the rest of the graph to identify another set of vertices for making another partition. This technique of selecting sets of vertices in multiple attempts is observed to be effective in case of large graphs (see Figure 5-17).

5.8.5.2 Comparison with Other Partitioning Techniques

In this set of experiments, effectiveness of the proposed algorithms is analyzed by comparing the performance of the partitioned systems achieved with the proposed algorithms (MOMC and CAP) with the ones achieved with the OEA technique and the MinCut algorithm based technique.

In the first experiment, analysis is performed using different graph sizes ($2 \leq D_{max} \leq 10$). The graphs are partitioned using the offloading entire application technique and the two proposed algorithms (MOMC and CAP). The results are presented in Figure 5-18. The sequence of bars in Figure 5-18 is the same as the legends shown at the bottom of the figure. Note that the graph presented in Figure 5-18 is plotted using a $\log_{10}$ scale for the execution time τ (Y-axis).

Performance of partitioned systems achieved with the offloading entire application technique is observed to be poor for majority of the cases (especially when the graph size is greater than 2). MOMC is observed to exhibit the best efficacy because the mean execution time of partitioned systems achieved with it is the lowest in comparison to all other techniques.

For comparison with the MinCut algorithm, 10,000 graphs for each graph size of 4, 6 and 8 as maximum D are generated. To apply the MinCut algorithm, it is also required to identify a source vertex and a sink vertex. A source vertex is a vertex which has no incoming edges (start point) while vertices with no outgoing edges are candidates for the sink vertex. Note that the proposed algorithms do not require a sink vertex to be identified. The proposed algorithms select the sink vertex using a heuristic as discussed in Section 5.4 and Section 5.5. For a comparison of MinCut with MOMC and CAP, a list of boundary vertices for each graph along with their adjacency matrix is generated. The boundary vertices used to select a sink vertex for the MOMC and the CAP algorithms are also used for selecting a sink vertex for the MinCut algorithm. One vertex from the set of boundary vertices is selected at a time and is used as a sink vertex for achieving a partitioned system with the MinCut algorithm. This is repeated for all boundary vertices. The execution times of all the partitioned systems (achieved using the MinCut algorithm for every boundary vertex) are determined using the simulator. The partitioned system with the minimum execution time is used in this comparison.

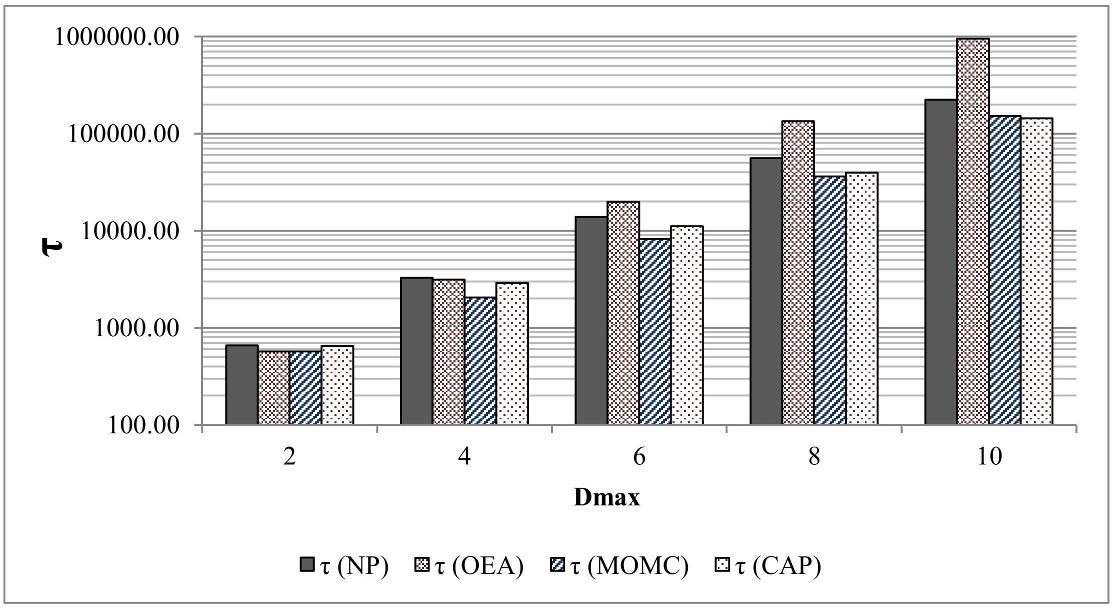


Figure 5-18: Comparison of MOMC and CAP with OEA and NP (no partitioning)

For the same graphs, the execution times of the partitioned systems achieved with MOMC, CAP and the offloading entire application technique are also determined. A comparison of the execution times of the partitioned systems is presented in Figure 5-19 and the execution time (Y-axis) is plotted using a $\log_{10}$ scale.

For small graph sizes (graph size = 4 in Figure 5-19), the performances of the partitioned systems achieved with all partitioning techniques are observed to be comparable although the partitioned system achieved with MOMC demonstrates the best performance. For graph sizes larger than 4, the performance of the partitioned systems with OEA and MinCut is observed to be significantly inferior in comparison to the performance of the partitioned systems achieved with MOMC and CAP. The results of experiments presented in Figure 5-19 show that the partitioning based on using no knowledge (OEA) or using the knowledge of edge weights only (MinCut) can lead to an inferior performance especially in comparison to the MOMC and the CAP algorithms.

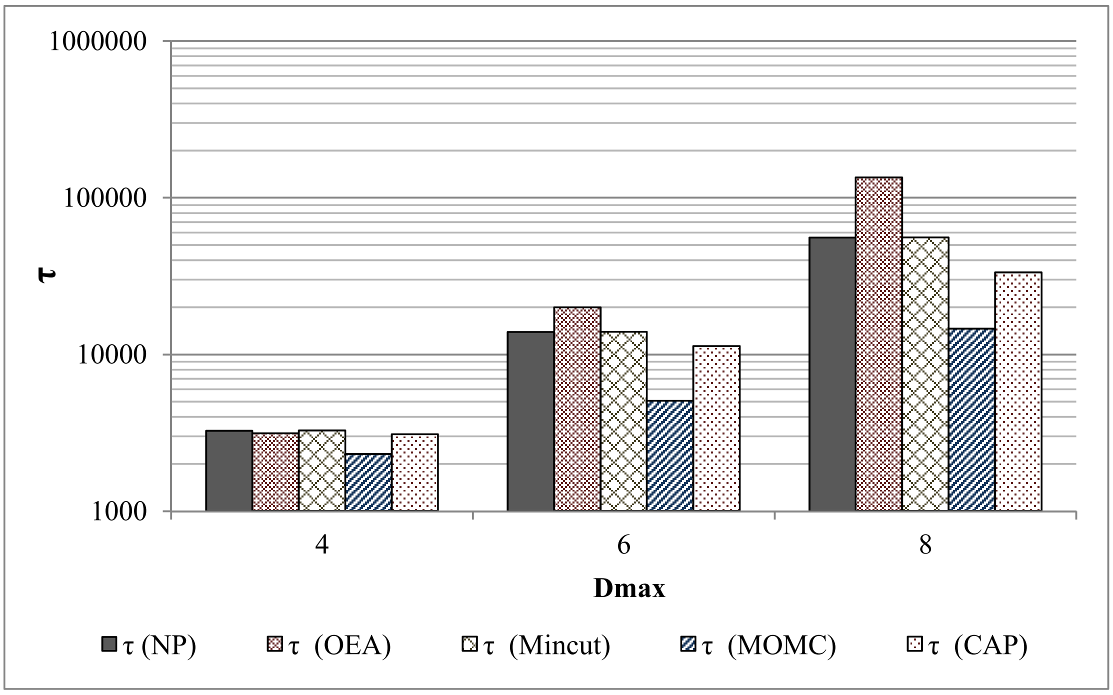


Figure 5-19: Comparison of efficacy of different partitioning techniques with the MinCut algorithm

5.8.5.3 Effect of Varying Mean Vertex Weight

In this set of experiments, the effect of varying the mean vertex weight is investigated. Note that a uniform distribution is used to generate a value for vertex weights. The results reported in this section use a mean vertex weight that is varied from 20 time units to 180 time units. The values used for the mean vertex weights are not affecting the relative performance of the partitioning techniques. One value of the mean vertex weight is used in one experiment and rests of the parameters are set to their default values as listed in Table 5-1. The mean execution time of the un-partitioned system is compared with that of the partitioned systems achieved with the MOMC and the CAP algorithms. The results are presented in Figure 5-20

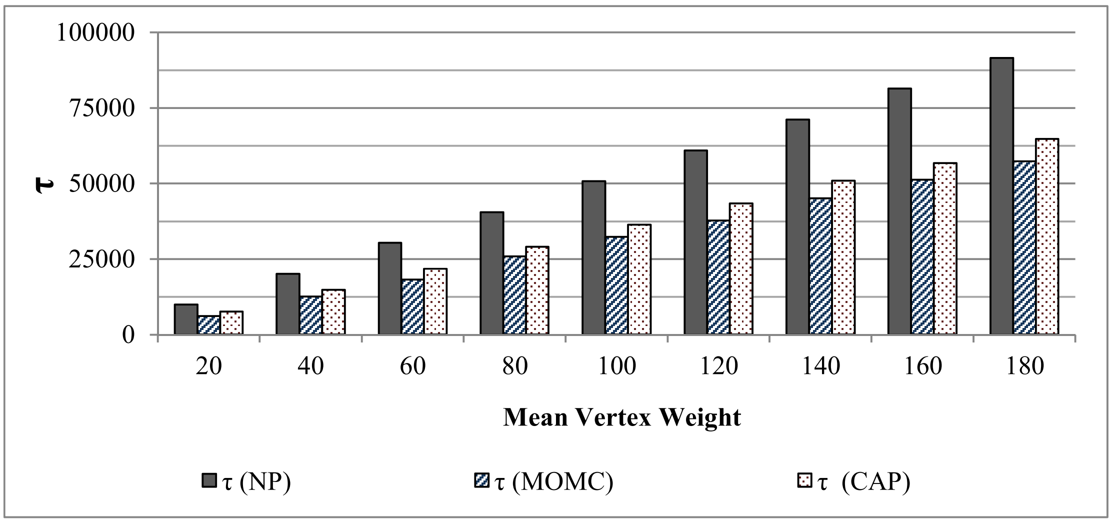


Figure 5-20: Effect of the mean vertex weight on the mean execution time of the un-partitioned system and partitioned systems using the MOMC algorithm and the CAP algorithm

Relative performance of un-partitioned and partitioned systems is observed to be not affected significantly with variation of the mean vertex weight. The mean execution time τ seems to increase linearly with the mean vertex weight. This is expected behavior. As the mean vertex weight is increased, the mean execution time of a partitioned or un-partitioned system is also expected to increase as the vertices represent execution components of an application.

5.8.5.4 Effect of Varying Mean Edge Weight

In this set of experiments, the effect of varying the mean edge weight is analyzed by varying the mean edge weight. Note that a uniform distribution is used to generate a value for weight of edges. The mean value for edge weight is varied from 20 time units to 180 time units. A specific mean value is used for one experiment. The default values are used for rest of the parameters as mentioned in Table 5-1. The mean execution times of the un-partitioned system and the partitioned systems achieved with the MOMC and the CAP algorithms are measured. The results are presented in Figure 5-21. Note that the execution time τ (Y-axis) is plotted using a $\log_{10}$ scale.

The mean execution times of un-partitioned and partitioned systems are observed to be not affected much by the variation of the mean edge weight. The variation in the mean execution times achieved with different mean edge weights for a given partitioning technique presented in Figure 5-21 are observed to be within 1%. The reason for this behavior is that the edge weight contributes to the execution time only when the end point vertices of that edge are to be executed on different nodes (e.g. one on the mobile node and the other on a remote computing node). Thus the effect of varying the mean edge weight is not affecting the overall execution time of the application.

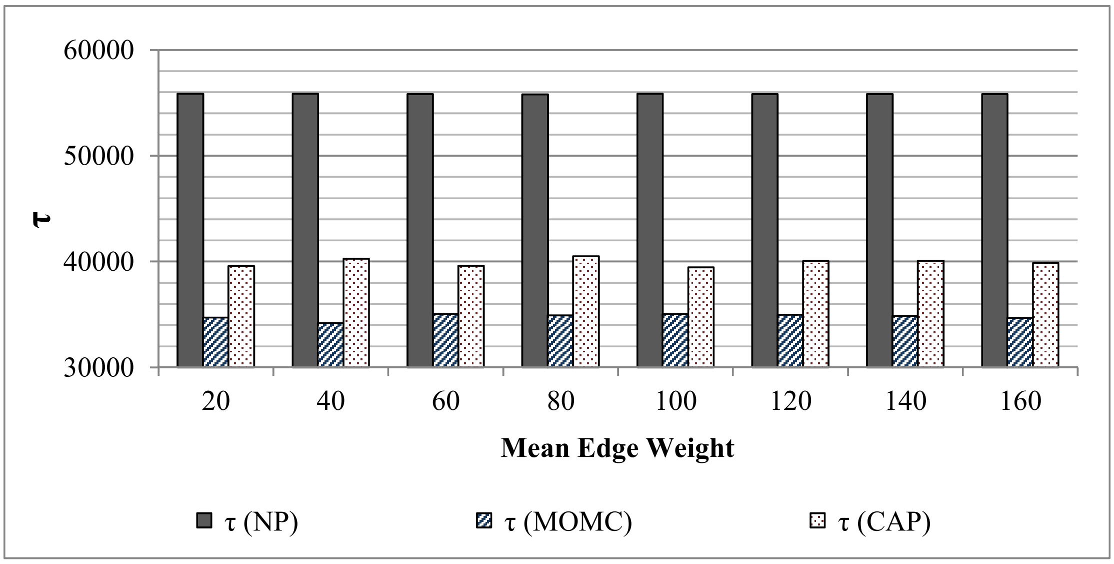


Figure 5-21: Effect of the mean edge weight on the mean execution time of the un-partitioned system and partitioned systems using the MOMC algorithm and the CAP algorithm

5.8.5.5 Effect of Varying Variability Factor

In this set of experiments, un-partitioned system and the partitioned systems achieved with the MOMC and the CAP algorithms are analyzed by varying the variability factor (Δ). Note that a uniform distribution is used to generate a value for the variable factor for each graph. The experiments are repeated using four different values Δ (0.05, 0.1, 0.15 and 0.2). Values of rest of the parameters (listed in Table 4-2) are held at their default values. The mean execution times achieved for the un-partitioned system and the partitioned systems are presented in Figure 5-22.

The mean execution times of un-partitioned and partitioned systems are observed to be not affected significantly by the variation in the variability factor. The variability factor is used in calculation of random weights of vertices and edges. The variation of the variability factor seems to contribute evenly for weights of vertices and edges and thus the overall performance of un-partitioned and partitioned systems is not affected significantly.

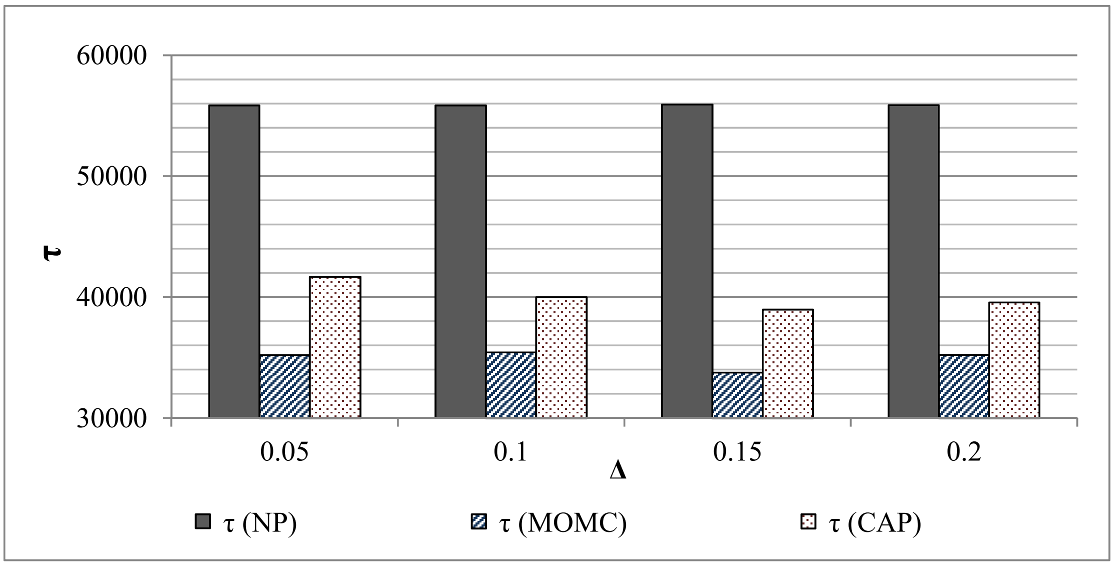


Figure 5-22: Effect of the mean variability factor on the mean execution time of the un-partitioned system and partitioned systems using the MOMC algorithm and the CAP algorithm

5.8.5.6 Effect of Varying Speed Up Factor

In this set of experiments, the effect of the speed up factor on the performance of the un-partitioned system and the partitioned systems achieved with the MOMC and the CAP algorithms is investigated. The speed up factor represents the ratio of the processing speed of the remote computing node and the processing speed of the mobile device. Four different values (2, 4, 6 and 8) of the speed up factor are used. Values of rest of the parameters (listed in Table 5-1) are held at their default values. The mean execution times achieved for the un-partitioned system and the partitioned systems are presented in Figure 5-23.

The relative performance of the un-partitioned and partitioned systems is observed to remain the same for the variation of the speed up factors. Note that the execution time for the un-partitioned system is unchanged for all values of the speed up factor because the speed up factor is only applied when one or more partitions are run on one or more remote computing nodes. The partitioned system achieved with MOMC is observed to exhibit the best performance for all values of the speed up factor. With increase in the speed up factor, the mean execution times for the partitioned system achieved with the MOMC algorithm and the CAP algorithm are reduced. This is expected because the execution time of the partition when it is run on a more powerful remote computing node is reduced with increase in the speed up factor. Note that the speed up factor is the ratio of the processing speed of the remote computing node and the processing speed of the mobile device.

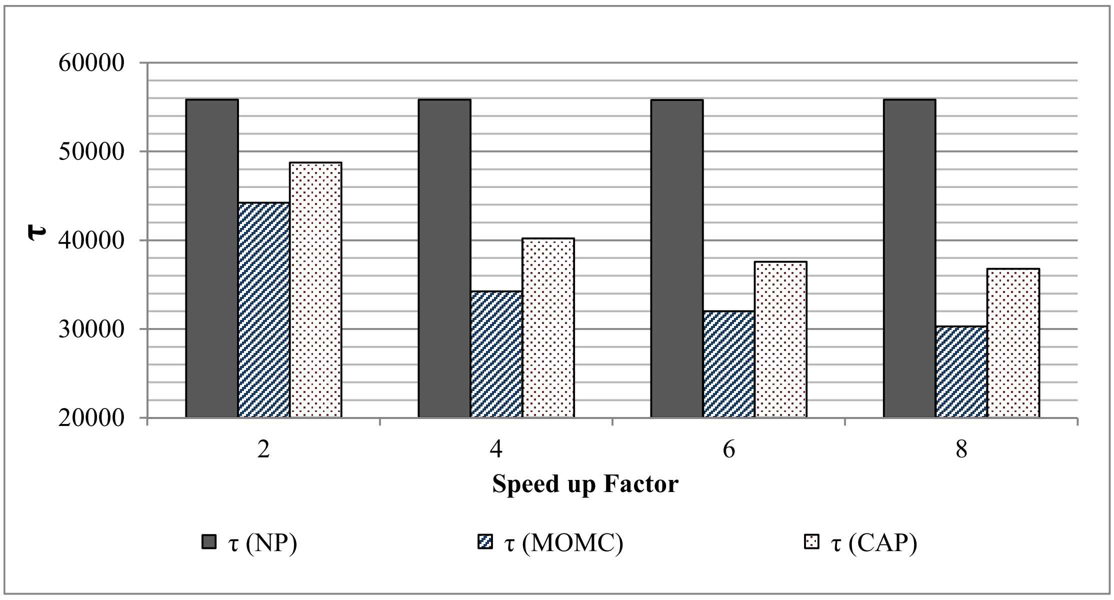


Figure 5-23: Effect of the speed up factor on the mean execution time of the un-partitioned system and partitioned systems using the MOMC algorithm and the CAP algorithm

5.8.5.7 Effect of Using Remote Node Information (RNI)

In this set of experiments, the effectiveness of using the knowledge of the processing characteristics of the remote computing node at the time of computing the partitions is analyzed. As discussed in Section 5.6, the MOMC and the CAP algorithm can use information of the remote node while computing the partitions. The partitioned systems are achieved with the MOMC algorithm and the CAP algorithm using the remote node information. Performances of these partitioned systems that use the remote node information are compared with the performance of the un-partitioned system. The results are captured in Figure 5-24.

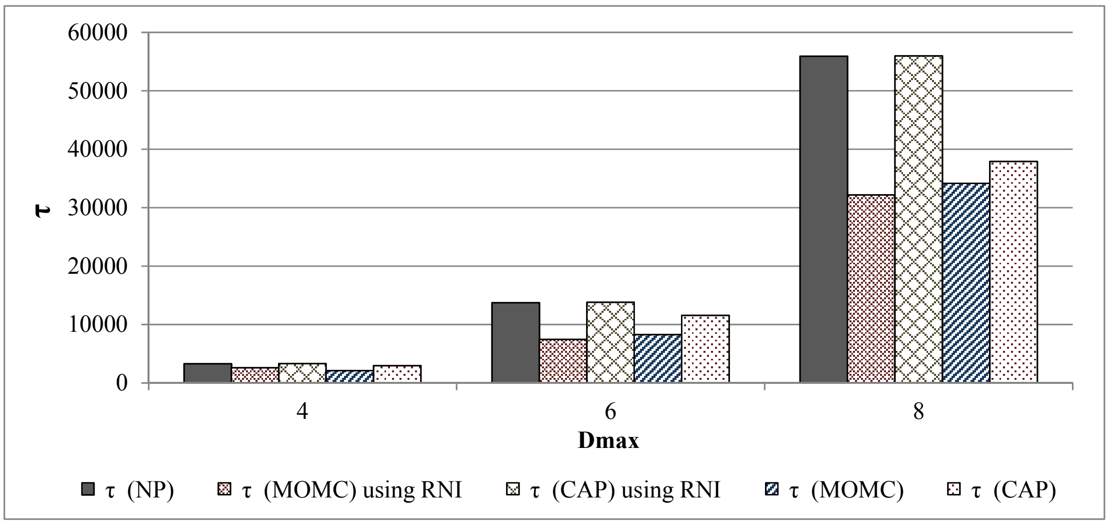


Figure 5-24: Effect of using remote node information in computing partitions with the MOMC algorithm and the CAP algorithm

The experiments show only a marginal improvement in the system performance when a partitioned system achieved with the MOMC algorithm that uses the remote node information in comparison to the partitioned system achieved with the MOMC algorithm without using any remote node information (see 2nd and 4th bars in Figure 5-24).

Since the CAP algorithm already uses a strict policy for selecting vertices, the use of additional remote node information results in either producing no partitioning or partition with very small size and thus resulting in degraded performance in comparison to the system achieved with the CAP algorithm that uses no remote node information (see bar 3rd and 5th in Figure 5-24).

5.9 Summary

In this chapter, two algorithms for mobile WS partitioning are proposed: Maximum Offloading Minimum Cost and Cluster based Application Partitioning. The effectively of the proposed algorithms is investigated through a system prototype as well as simulation. For prototyping, real world sample web services are used. The sample web services are partitioned using MOMC, CAP, MinCut and OEA. The un-partitioned and partitioned systems achieved are hosted on a mobile device and are accessed through a local area wireless network. For simulation, a graph generating tool is devised for generation of random graphs of different sizes and different characteristics. The randomly generated graphs are partitioned using MOMC, CAP and the other partitioning techniques (MinCut and OEA). The performances of the resulting partitioned systems are analyzed through a simulator.

The results achieved with the prototype and the simulations demonstrate that the proposed algorithms outperform the existing algorithm for WS partitioning. The experimental analysis shows that:

- The MOMC partitioning algorithm performs the best for small to medium size applications.
- The effectiveness of the CAP algorithm is observed to be better for large scale applications in comparison to MOMC and other partitioning techniques. For smaller applications, the algorithm is inferior in comparison to MOMC but it always performs better than the no partitioning case, the offloading entire application technique and the MinCut algorithm based technique.

- The partitioned systems achieved with offloading entire application and MinCut exhibit poor performances in general.
- The graphs generated by the RGG tool have randomly distributed edges with significantly high weights. Such edges have more probability to exist closer to the source vertex of the graph than to the sink vertex. For such graphs, OEA and MinCut are observed to produce poor partitions. These two techniques (OEA and MinCut) can however be expected to produce good partitions only when the edges closer to the source vertex have significantly lower weights in comparison to the rest of the edges. For such graphs, the proposed MOMC algorithm also produce partitions similar to the ones achieved with the OEA and the MinCut techniques. This is observed in Figure 5-4, Figure 5-5 and Figure 5-6.

The OEA and the MinCut based WS partitioning techniques produce effective partitions only for specific types of graphs whereas the MOMC algorithm and the CAP algorithm based WS partitioning techniques are effective on a wide variety of graphs. The type of knowledge to be used in devising a partitioning algorithm is an important issue. MinCut that uses only the communication cost is found to lead to an inferior performance of the partitioned system in comparison to that achieved with MOMC that uses both communication and processing costs.

Chapter 6: Run Time WS Partitioning

Chapter 5 discussed design time WS partitioning algorithms to achieve one or more partitions that can be offloaded to a remote computing node. This chapter proposes a runtime WS partitioning technique with the objective of improving the overall system performance. The proposed WS partitioning technique is devised to offload different sizes of partition on a remote computing node based on the system load. Performance of the proposed WS partitioning technique is analyzed by performing experiments on a simulator.

6.1 Overview

As already discussed, WS application partitioning is a process of dividing a WS application into multiple components so that computationally complex components can be executed on powerful remote computing nodes. The objectives of WS partitioning considered in this thesis include achieving a considerable size of the offloaded partitions with least possible communication costs. In case of change in the device system load or available battery power, a dynamic partitioning technique can help to offload more considerable size of an application for execution on a remote computing node.

The results presented in Chapter 5 show the two main drawbacks of design time partitioning:

1. The partitioned system achieved with it may not be a best solution for various devices on which the system may get deployed.
2. The partitioned system is generally insensitive to the variation in the system load.

Performing an application partitioning at runtime can use system load information and device characteristics for achieving an effective partitioned system. Achieving a partitioned system with runtime partitioning approach has a few limitations as well. For example, deciding when to run the application partitioning algorithm is an important question. Should it be run for every request arrival or run only when the system load changes significantly? In both cases, there will be a substantial overhead affecting the application response time. The benefit that accrues from running the application on a partitioned system must be able to offset the overheads. But there are many situations when it makes sense to use a runtime partitioning approach. For example, the device may be running multiple applications at a time sharing the device resources such as CPU and memory or a large number of WS clients are sending requests for hosted web services or the device has a limited battery power.

In such situations, using a runtime application partitioning is expected to be effective because the size of the partition to be executed on a remote computing node can be varied based on the situation. For example, in a situation when a device is running out of battery power, then a substantial amount of application components can be offloaded to preserve device's battery power. If a large number of WS requests are waiting to be executed, running large part of an application on a remote computing node can improve system response time. If a device is not experiencing any of the situations as mentioned earlier, it may be beneficial to execute a little part of an application (or even no part of an application) on a remote computing node.

In this chapter, a hybrid technique for runtime application partitioning that combines advantages of both the design time application partitioning and the runtime application partitioning is proposed. The proposed hybrid technique for runtime application partitioning first applies a graph based algorithm for achieving multiple execution plans (discussed in the next section) for a WS application at design time. The technique uses a runtime middleware system which selects an appropriate execution plan based on the system load information and then uses that execution plan for executing the application partition.

6.2 Generating Execution Plans for a WS

An execution plan is a scheme that determines what part or parts of an application is to be executed on a local mobile node and what part or parts to be run on a remote computing node (see Figure 6-1). A part of an application can be a method of a class or a collection of methods of one or more classes. An execution plan also defines a sequence of execution of different methods within a partition. For the work presented in this thesis for runtime WS partitioning, the execution plan is a set of a maximum of two partitions (P_L and P_R), each partition comprises of one or more methods. The two partitions are shown as vertical grey line boxes (P_L – to be executed on a local mobile node) and horizontal grey line boxes (P_R – to be run on a remote computing node) in Figure 6-1.

The WS partitioning used for run time WS partitioning technique computes different sized partitions for execution on a remote computing node for different range of system load. For achieving this, a graph based algorithm is proposed for generating multiple execution plans that are ordered based on the size of P_R. A set of a few sample execution plans is shown in Figure 6-1. Execution plan #1shown in Figure 6-1 suggests executing the whole application on a local mobile node. Execution plan #2 represents a plan that suggests running a small part (~25%) of the application on a remote computing node. Execution plan #3 and Execution plan #4 suggest running a major part of the application (~50% and ~75% respectively) on a remote computing node. Execution plan #5 suggests executing the entire application on a remote computing node.

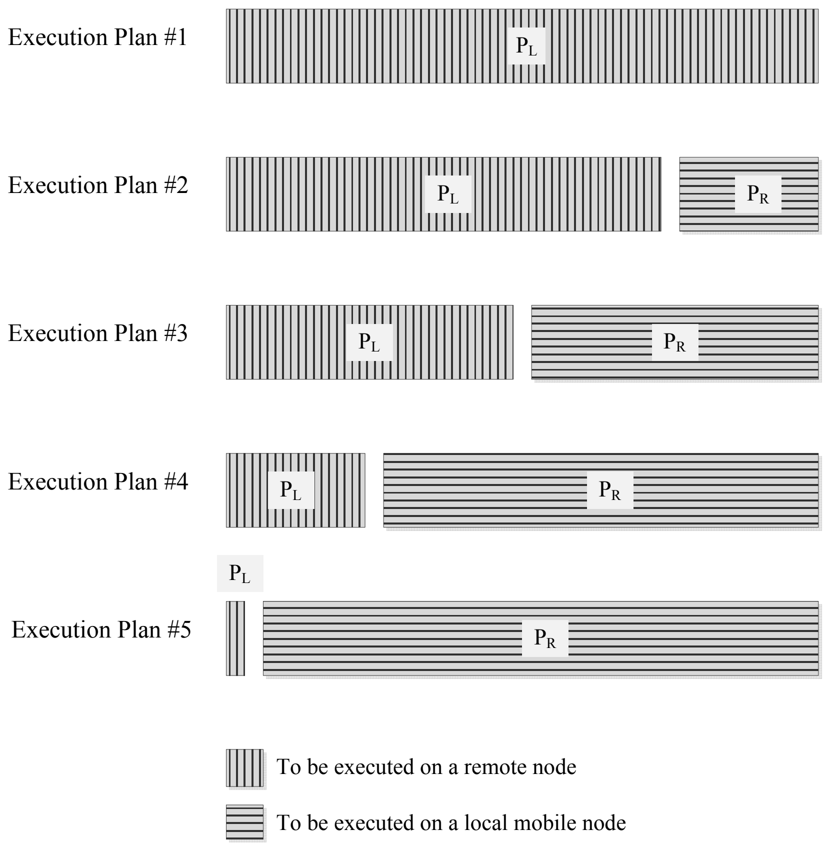


Figure 6-1: A set of sample execution plans

The proposed algorithm for achieving multiple execution plans is based on navigating a graph modeling the application to be partitioned iteratively.

6.2.1 Input Parameters

The graph based WS partitioning algorithm requires the following input parameters.

6.2.1.1 Application Graph

A graph model (G (V, E)) of a WS application is a key input parameter.

6.2.1.2 Number or Execution Plans (N_E)

This is the number of execution plans to achieve. The value of N_E can be between 1 and the number of vertices in a graph.

6.2.2 Upper Bound for Remote Partition of an Execution Plan

As already mentioned, an execution plan includes two partitions: one for running locally (P_L) and the other for running on a remote computing node (P_R). The two partitions are computed using a graph based algorithm proposed in the following subsections. The size of the remote partition, P_R, is the sum of the weights of vertices (representing processing costs) in P_R. Each execution plan is associated with an upper bound on the size of P_R. The approach used for the calculation of upper bound on the size of P_R for each execution plan is discussed next.

The upper bounds on the size of remote partitions for any two consecutive execution plans are separated by a fixed size step. The fixed size step is the same for all execution plans. The fixed size step is determined from the number of execution plans (N_E) and the total processing cost of the application which is a sum of the weights of all vertices of an application graph. If the number of executions plans is N_E, then the fixed step size (P_{SS}) between the upper bounds on the size of remote partitions for any two consecutive executions plans is calculated by dividing the sum of the weights of all vertices in the application graph by the number of executions plans

$P_{SS} = (\sum W_{Vi}) / N_E$ where $v_i \in V$ 6-1

Now, the upper bound on the size of P_R for any execution plan can be computed by adding the fixed step size to the upper bound on the size of P_R of the previous execution plan. For example, for the first execution plan, the upper bound on the size of the remote partition will be

$L_{U-1} = 0 + P_{SS} = P_{SS}$

Note that there is no previous execution plan for the first execution plan, thus zero is added to P_{SS}. Similarly for the second and third execution plans, the upper bounds on the size of remote partitions will be

$L_{U-2} = L_{U-1} + P_{SS} = P_{SS} + P_{SS} = 2*P_{SS}$

$L_{U-3} = L_{U-2} + P_{SS} = 2*P_{SS} + P_{SS} = 3*P_{SS}$

Thus, an upper bound on the size of a remote partition for any k^{th} execution plan will be

$L_{U-k} = k * P_{SS}$ 6-2

where k = 1, 2, 3 …. N_E

6.2.3 Objective Function

The proposed technique for run time WS partitioning is based on computing multiple execution plans of different sizes in advance and then selecting an execution plan based on the system load. The computation of 'N' execution plans is performed in advance in order to eliminate the computation overheads during runtime. Each execution plan is computed with an objective of selecting a beneficial partition such that the difference between the processing cost of that partition and the communication cost incurred by offloading the partition is the maximum. Each computed execution plan is also characterized by the size of the partition that is marked to be executed on a remote computing node. The run time component of the proposed technique selects the execution plan based on the system load and executes the WS application according to the execution plan.

The objective function for the graph based algorithm for generating execution plans is to achieve a partitioned system of a graph G with N_E execution plans such that for every k^{th} execution plan, the following two conditions are satisfied

Condition 1: $\beta (P_k) = \text{Max } [W_V (P_k) - W_{EC} (P_k)]$

Condition 2: $W_V (P_k) < L_{U-k}$

Where

- $W_V (P_k)$ is the sum of the weights of vertices of a partition 'P_k' to be executed on a remote computing node (processing cost).

- $W_{EC} (P_k)$ is the sum of the weights of edges that separates the vertices of a partition 'P_k' (communication cost).

- $\beta (P_k)$ is a maximum difference of $W_V (P_k) - W_{EC} (P_k)$

The rationale behind condition 1 and 2 is presented next.

Condition 1 defines a 'Beneficial Cut', a term introduced in the last chapter (see Section 5.3.9), that renders a graph into two disjoints sets such that the difference of sum of the weights of vertices in a partition ($W_V (P_k)$) and the sum of the weights of edges ($W_{EC}(P_k)$) that separates the graph is maximized.

Condition 2 concerns the size of a partition that can be executed on a remote computing node. According to this condition, the maximum size of a partition ($W_V (P_k)$) for an execution plan cannot exceed the upper bound on the size of P_R computed for that execution plan.

6.2.4 Algorithm Steps

To explain the working of the algorithm steps, temporary variables are introduced. These temporary variables include sets of vertices (*B*, *X, N and Q*), numeric variables (*w, ε and β)* for calculation of the degree of benefit a table *T* for storing candidate execution plans. The set *Q* contains all vertices of set *V* except. *w* and, *ε* are used for storing the sum of the weights of vertices of X (W_{VX}) and the weight of the edge cut that separates vertices in set X from rest of the graph (W_{EX}) *β* is used for storing the degree of benefit for set X. Note that '\' is a set operation of taking relative complement. The rest of the variables are explained during the discussion of the individual steps of the algorithm.

The main steps of the algorithm, the pseudo code for which is presented in Figure 6-2, are explained next. The steps that involve initializations of different variables (such as lines 1-6 and lines 5-6 of Figure 6-2) are not included in this discussion.

- *Find boundary vertices (B)* from set *Q* (line 3 in Figure 6-2). As mentioned for MOMC and CAP algorithms, the boundary vertices are those with the maximum value of vertex distance (*D)*. It has been observed that choosing the starting vertex which is located far away from the source vertex '*s*' results in more effective partitioned systems in comparison to the partition systems that are achieved by selecting a starting vertex randomly or by selecting a starting vertex closer to source vertex '*s*'.
- Se*lect a boundary vertex (v) with maximum weight* as a starting vertex (line 5 in Figure 6-2).

```
//$V$ is a set of vertices of graph G, $Q$ is a set of vertices that are considered for
//partitioning and $s$ is the source vertex. $X$ is a temporary set variable for vertices.
//$\beta$ is the degree of benefit. $T[X, \beta, w]$ is a table to hold partitions and $N_E$ is the
//required number of execution plans.
01:  Q = V \ s;
02:  T [set, number, number] = Φ;
03:  B = Set of vertices in Q with maximum D;            (Boundary Vertices)
04:  X = Φ, β = 0;
05:  ν = Vertex with maximum weight in set B;
06:  do {
07:          X = X U ν;
  //w and ε are temporary variables storing sum of the weights of vertices and
  //the edge cut weight respectively
08:          w = Sum of weights of vertices in X;
09:          ε = Edge cut weight of set X;
10:          β = w – ε;
11:          Add X, β and w to table T;
12:          N = Set of vertices that are directly connected to vertices of X;
13:          ν = Vertex in N with maximum degree of benefit;
14:  } while (ν != null)
15:  for (k = 1 … $N_E$) {     //k is an index variable, $L_{U-k}$ is an upper bound on
                             //the size of the remote partition for the kth
                             // execution plan
16:      Select a set of rows ($T_1$) from table T for which $w < L_{U-k}$;
17:      Select a row from $T_1$ rows for which β is the maximum;
18:      $P_R$ = X of the selected row;        // $P_R$ is a remote partition
19:      $P_L$ = V – $P_R$;                    // $P_L$ is a local partition
20:  }
```

Figure 6-2: The runtime partitioning algorithm for achieving multiple partition plans

- *Add selected vertex 'v' to a temporary set X* (line 7 in Figure 6-2). *X* is a set of vertices that represents a candidate partition to be executed on a remote computing node.
- *Calculate β as follows (see line 8-10 in* Figure 6-2).

$$\beta = w - \varepsilon \qquad 6\text{-}3$$

where w is the sum of weights of vertices in set *X* (W_V (*X*)) and ε is the sum of weights of edges that separates vertices in set *X* from rest of the graph (W_{EC} (*X*)).

Store values of X, *β* (*X*) and W_V (*X*) in the partition table *T* (line 11 in Figure 6-2).

- *Select vertex 'v' for the next iteration* (lines 12-13 in Figure 6-2)
- The set of neighbor vertices, *N*, (where *N* is a subset of *{Q \ X}*) represents those vertices of *{Q \ X}* that are connected to any vertex of *X*. Note that this step was not executed in the first iteration when a new vertex was selected from the set of boundary vertices. In this step, a vertex '*v*' that has the maximum degree of benefit is selected from a set of neighbor vertices of *X*.
- *Repeat steps in lines 7-13 for all vertices of set Q*. Once all vertices of set *Q* are evaluated, the table T will contain a list of records that can be used to compute execution plans.
- *Compute execution plans* by walking through all rows of table T. For the kth execution plan with upper bound $L_{U\text{-}k}$ on the size of a remote partition, the execution plan can be computed by using following steps.
    - Select all rows (T_1) from table T for which value of W_V (X) is less than or equal to $L_{U\text{-}k}$ (condition 2 of the objective function).
    - Select a row from T_1 rows for which value of β (X) is the maximum (Condition 1 of the objective function).

6.3 Runtime Middleware for WS Partitioning

In this section, a middleware for runtime partitioning of WS applications hosted on mobile devices is proposed. The implementation of the proposed middleware uses only one remote computing node for executing partition P_R. Techniques for the selection of a remote computing node from a set of available nodes or a cloud for execution of a given partition can form an important direction for future research. Various characteristics such as the system load can be used in mapping a partition to a remote computing node. The proposed middleware will be an enhancement of the WS execution environment proposed in Chapter 3. The middleware with WSEE is based on three main components: a web service execution environment (WSEE), a runtime WS partitioning engine and executions plans for web services applications that are to be hosted on a mobile device. A high level architecture of the proposed middleware is presented in Figure 6-3. The details of each of these components are described next.

6.3.1 Web Service Execution Environment

A WSEE is a platform that facilitates the execution of web services. In the context of the proposed middleware, WSEE uses two key components: Transport Listener and WS Manager. A brief overview of these two components is presented next. A more detailed discussion is available in Section 3.2.

6.3.1.1 Transport Listener

Transport Listener is responsible for exchanging SOAP messages. SOAP messages can be exchanged between nodes using any transport mechanism. Note that there is no specific transport protocol associated with the exchange of SOAP messages. The proposed Transport Listener uses standard transport protocols such as HTTP or TCP for exchange of SOAP messages. On receiving a WS request, Transport Listener extracts the SOAP message from the incoming request and puts in a queue 'Q_{srv_in}'. Transport listener is also responsible for sending a response back to a WS client. For this, Transport Listener gets the response message (a SOAP message put by WS Manager in 'Q_{srv_out}') from 'Q_{srv_out}' and sends the message to the WS client.

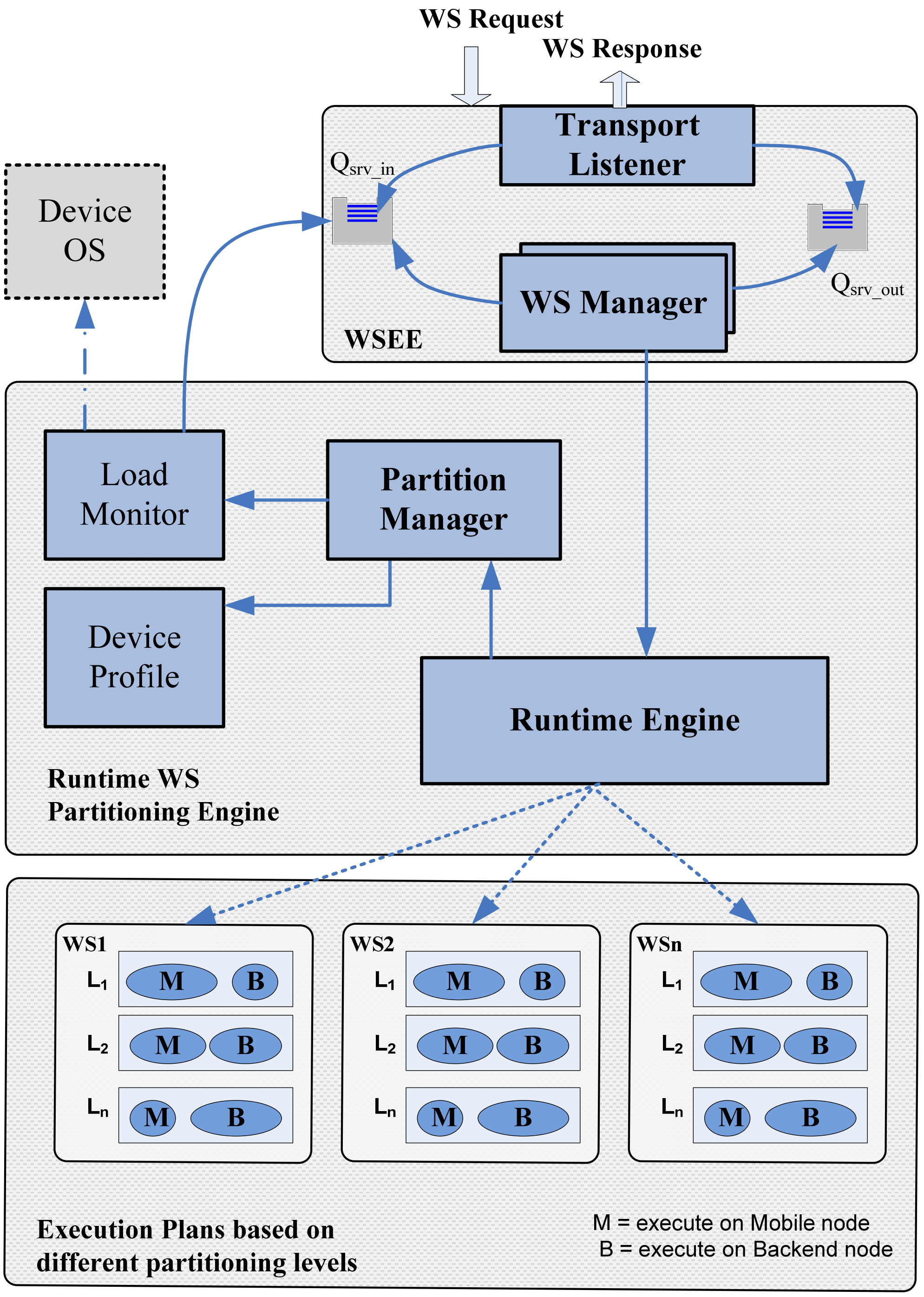


Figure 6-3: Architecture of the proposed middleware for runtime WS partitioning

6.3.1.1 WS Manager

The primary responsibilities of WS Manager are the parsing of the incoming SOAP messages, executing the WS application, and then wrapping the results of the WS application into a response SOAP message. Multiple threads of WS Manager are available in a thread pool. These threads are used to process WS requests waiting in 'Q_{srv_in}'. On receiving a new SOAP message, WS Manager parses the SOAP message using standard SOAP/XML libraries available for mobile devices (such as KSOAP [Kso03] and KXML [Kxm03]). The implementation discussed in this thesis uses both KSOAP and KXML libraries. After parsing the request, WS Manager looks up the appropriate WS application based on the WS request parameters. Once a WS application is selected, WS Manager hands over the parsed input parameters and name of the WS application to Runtime Engine for execution.

6.3.2 Runtime WS Partitioning Engine (Runtime Engine)

The execution plans for a WS application are computed using the graph based WS partitioning algorithm described in Section 6.2. In the proposed runtime partitioning technique, all partitions of web services are available on a remote computing node. No code migration is performed at runtime; only the appropriate WS partition identified by the execution plan is executed at runtime. Application partitioning can be performed either at a class level (object oriented Java class) or at a class method level. For the proposed runtime WS partitioning, a method level granularity is used. This means each component partition would comprise one or more class methods.

Runtime Engine is responsible for selecting an appropriate plan based on the system load. To establish a relationship between the system (load) information and available execution plans for a WS application, a mechanism based on the real time profiling is devised. This is discussed further in Section 6.3.4.

6.3.3 Relationship between System Load Information and Execution Plans

The system load is typically estimated by measuring CPU utilization due to other applications or by measuring the number of requests waiting to be executed by a system. This research focuses only on the system load based on the number of WS requests arriving for a hosted WS. Therefore, number of WS requests waiting to be processed (size of the queue) is used as an indicator for the system load.

The following terms are introduced before discussing the relationship: degree of system load, device profile and device profile index. These are described next.

6.3.3.1 Degree of System Load (ρ)

The degree of system load can be different for different scenarios. For the work presented in this thesis, the focus is to improve the overall response time. That is why the degree of system load is the number of requests waiting to be executed. The degree of system load is determined by computing an exponential moving average of the number of WS requests waiting in Q_{srv_in}. The size of the window used for computing the exponential moving average is 5. The exponential moving average (EMA) is computed by using a simple moving average model described by Brown [Bro04]:

$$EMA = \alpha_n * e + (\alpha_{n-1} * (1 - e))$$

Where α_n is a current value, α_{n-1} is a previously calculated value and e is an exponential factor (2 / size of window)

6.3.3.2 Device Profile/ Device Profile Index (χ)

The device profile includes device processing power (for example CPU speed), total memory, available memory for execution and number of applications running on the device. Factors such as processing speed and the total available memory are available from the device specifications. Factors such as memory in use and number of applications running on a device depend on system state and are available at runtime through an operating system's application interfaces.

Device profile index is an indicator of device ability to accept new tasks for execution. One or more factors (such as CPU speed, available memory, number of applications running) can be used to determine the device profile index. The technique used in this thesis for measurement of the device profile index is based on real time profiling and it does not consider the device processing speed and its memory. The objective of this technique is to use the device profile index as an indicator of the number of WS requests that can be processed without degrading the overall response time significantly. The real time profiling is performed by running an experiment for a different number of WS clients for a graph application of sizes of four, eight and twelve. The number of concurrent WS clients is varied from 1 to 100. The results of this experiment are presented in Appendix C.

6.3.4 Selecting an Execution Plan

As already mentioned, the runtime partitioning engine is responsible for selecting an appropriate execution plan for a partitioned system based on the system load. The proposed algorithm for selecting an execution plan is presented next. The input to the proposed algorithm is the set of execution plans for a partitioned system and the device profile index (*χ*).

Step 1: Compute the degree of system load (ρ) using the number of waiting requests.

Step 2: Compute k (execution plan #) using the following relation:

$$k = 1 + f(\rho, \chi / N_E) \qquad 6\text{-}4$$

where

- ρ is the current system load (moving exponent average), χ is a device profile index and N_E is the number of execution plans
- Dividing the device profile index by the number of execution plans (χ / N_E) gives the range of the system load for each execution plan. For example, if the device profile index is 20 and the number of execution plans is 4, then the range of system load for each execution plan is 5 (20 / 4).
- $f(\rho, \chi / N_E)$ is an integer division function of ρ and χ / N_E. This function divides the system load (ρ) by the range of the system load for an execution plan which is given by χ / N_E. The result of this integer division identifies the execution plan # corresponding to the current system load. For the example shown earlier, if the current system load is 12, then the result of $f(\rho, \chi / N_E)$ will be 2 (12 / 5). Note that the remainder is discarded in integer division. '1' is added to the result of $f(\rho, \chi / N_E)$ to make the execution plan (k) index starts from '1'instead of '0'.

Step 3: Select k^{th} execution plan from the available execution plans for a partitioned system.

In the following subsection, different subcomponents of the runtime partitioning WS engine are introduced.

6.3.5 Components of Runtime Partitioning Engine

The components of the proposed implementation of the runtime partitioning engine are discussed next.

6.3.6 Runtime Engine

Runtime Engine is the main controller for executing a WS execution plan based on the system load. On receiving a request for executing a WS application from WS Manager, Runtime Engine communicates with Partition Manager (see Figure 6-3) to determine a particular execution plan.

6.3.7 Partition Manager

Partition Manager is responsible for determining an execution plan based on the system load and the device profile index. The degree of system load is computed by Load Monitor (see Figure 6-3). Once the degree of system load is known, Partition Manager selects an appropriate execution plan using the algorithm presented in the previous subsection.

6.3.8 Load Monitor

For computation of the system load, Load Monitor interacts both with the operating system of the mobile device and the input queue (see Q_{srv_in} Figure 6-3). Since memory is not considered in the determination of the system load (as discussed in Section 6.3.3.1), Load Monitor only measures the number of requests waiting to be executed in Q_{srv_in} to compute an exponential moving average (EMA) of the number of waiting requests using a window size of 5.

6.3.9 Device Profile

For the research presented in this chapter, the device profile indicates the number of requests that can be run on the device simultaneously without affecting device performance significantly as discussed in 6.3.1.1.

6.4 Approach used for Performance analysis

The approach used for analyzing the effectiveness of the proposed runtime WS partitioning technique is based on analyzing system performance by using a variety of application graphs. For this, a tool Random Graph Generator (RGG) described in Section 5.8.1 is used. The performance of the proposed middleware system for runtime WS partitioning is evaluated by using a simulator that is developed for this purpose. The simulator is described briefly in the next subsection. Note that the RGG tool and the simulator are implemented using the standard Java edition.

6.4.1 Simulator

The simulator used for performance analysis of the proposed middleware system is an enhanced version of the simulator that is described in Section 5.8.2. For a given size of the graph and the template, the simulator is provided a partitioned system with multiple execution plans. The execution plans are achieved using the graph based WS partitioning algorithm proposed in Section 6.2. The other input parameters are speed up factor(S), number of WS request to simulate, the device profile index to use and the number of execution plans. The speed up factor represents the ratio of the processing speed of the remote computing node and the processing speed of the mobile device. The default value of the speed up factor used is 4. The simulator incorporates network delays when a remote computing node is involved. Network delay is a component of W_{EC} in equation 6-3. A more detailed discussion of generating a random network delay is presented in Sukhov *et al.* [Suk10]. Network delays are modeled by using an exponential distribution [Suk10]. In all the experiments reported in this chapter, the mean value of the network delay used is 10 time units. The core of the simulator is the same as discussed in Section 5.8.2.

6.5 Performance Analysis

System performance achieved by using the proposed runtime WS partitioning technique is analyzed by using the simulator described earlier. The different input parameters that are varied to get insights into system behavior and performance are described next.

6.5.1 Input Parameters

In all the experiments, the mean response times of application graphs (partitioned or un-partitioned) are measured by varying a number of input parameters. In one experiment, only one input parameter is varied while all other parameters are held at their default values. The input parameters and their default values are presented in Table 6-1. Note that the client think time used in experiments is generated by using an exponential distribution. Similar distributions for clients think times have been used by Rosen [Ros87].

Graph size is equal to the maximum value of the vertex distance (defined in Section 5.3.5). For graph generation, default values are used for the mean vertex weight, the mean edge weight and the variability factor as discussed in Table 5-1. The number of Execution Plans (N_E) is the number of execution plans to be used for the runtime WS partitioning. In a preliminary analysis, the number of execution plans equal to four is observed to be appropriate for the application graphs used in the experimentation. Number of WS clients (C) is the number of applications that are accessing the system simultaneously. Each client sends a request, waits for the response, receives the response, waits for a time interval equal to the mean think time and then sends the request again. The mean think time is the mean time a client waits before sending the next request.

Table 6-1: Input Parameters and their default value

Parameter	***Value Range***	***Default Value***
Graph Size	4, 8 and 12	8
Number of Execution Plans (N_E):	4	4
Number of Clients (C)	Varied from 1 to 40.	12
Mean Think Time (time units)	Varied from 500 to 5000	1000

6.5.2 Performance Metrics

The key performance metric is the *mean execution time (τ)*. The execution time (in time units) is the difference between the time when the application has finished its execution and the time when the application request is received by the simulator for execution. The mean execution time is computed by taking an average of the execution times measured for all randomly generated applications (graphs) for each experiment. Energy of the mobile device consumed before and after web service partitioning is an interesting performance parameter. Investigation of this performance parameter forms an important direction for future work.

6.5.3 Experimental Results

Two different sets of experiments are performed. In the first set of experiments, the mean execution time of a partitioned system using one of the execution plans at a time is measured. The experiment is repeated for graph sizes of 4, 8 and 12. In the second set of experiments, the performance of the partitioned system achieved with the runtime WS partitioning technique is compared with a 'no partitioning' case and the partitioned system achieved with the design time WS partitioning technique (described in Chapter 5). The comparison is performed by varying number of clients (*C*).

A closed system model is used for all the experiments. Each client operates cyclically and sends one request at a time. As soon as the request is processed, the client sends another request after a delay equal to the client think time. Each experiment is performed by using a randomly generated graph. Random graphs are generated using a Random Graph Generator tool discussed in Section 5.8.1. The result of each experiment is obtained by using 10,000 randomly generated graphs. Each experiment is repeated 15-30 times to obtain sufficiently small confidence intervals for the average values of the execution times. For the experiments presented in this chapter, confidence intervals of ±5% (or less) for mean response time were obtained at a confidence level of 95%.

6.5.3.1 Performance Analysis using One Execution Plan at a Time

In this set of experiments, the effectiveness of the proposed runtime WS partitioning technique is analyzed by measuring the mean execution time achieved with the partitioned systems using one of the execution plans (preselected) at a time. The number of WS clients is varied from 1 to 20. The performance comparison is presented for different graph sizes (see Figure 6-4 for graph size = 4, Figure 6-5 for graph size = 8 and Figure 6-6 for graph size = 12). The results observed for different graph sizes appear to exhibit a similar trend in the performance. Each line shown in the graphs of Figure 6-4, Figure 6-5 and Figure 6-6 displays the mean execution time of the partitioned system when a preselected execution plan is used. The number of execution plans used for this set of experiment is 4. Execution plan #1 represents an un-partitioned system. Execution plan #2, #3 and #4 are achieved with the graph based WS partitioning algorithm. Higher the execution plan number, higher is the size of P_R. Note that no runtime selection of an execution plan is performed for this experiment.

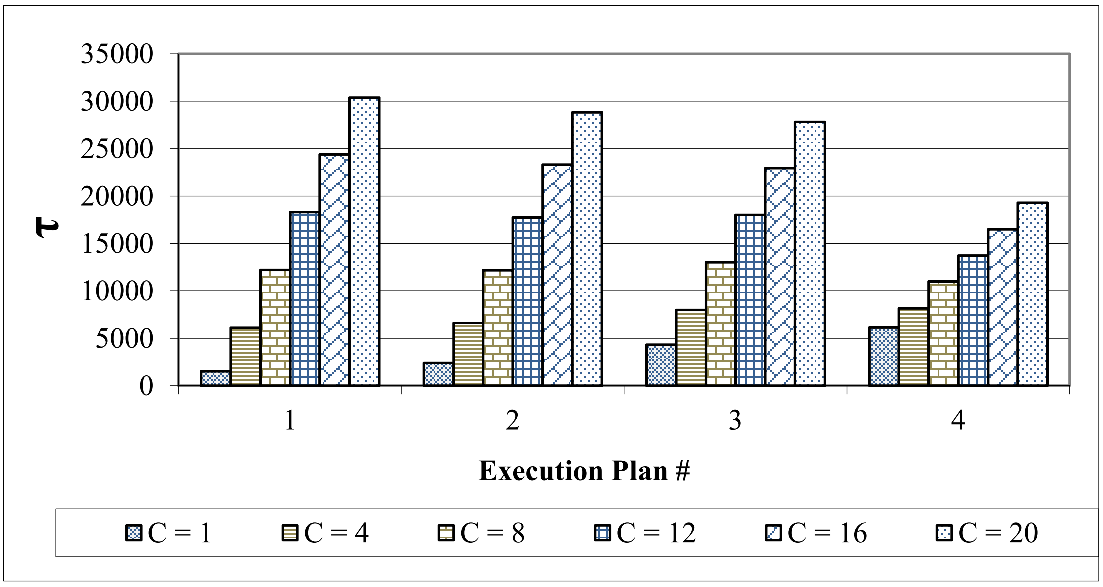


Figure 6-4: Comparison of performance of partitioned systems for various execution plans, C and for a graph size = 4

The results presented in Figure 6-4, Figure 6-5 and Figure 6-6 exhibit an interesting pattern. For all graphs shown in Figure 6-4, Figure 6-5 and Figure 6-6, for a lower number of clients, using an execution plan that runs a larger part of an application on a remote computing node (such as execution plan #3 or #4) is resulting in a degradation in the system performance. For such scenarios, using no partitioning (execution plan #1) appears to be the most effective. This is because the overheads of executing a partition on a remote computing node in comparison to executing it locally are significant and thus results in a degradation of overall system performance of the partitioned system (see bars for C = 1 and C = 4 in Figure 6-4, Figure 6-5 and Figure 6-6).

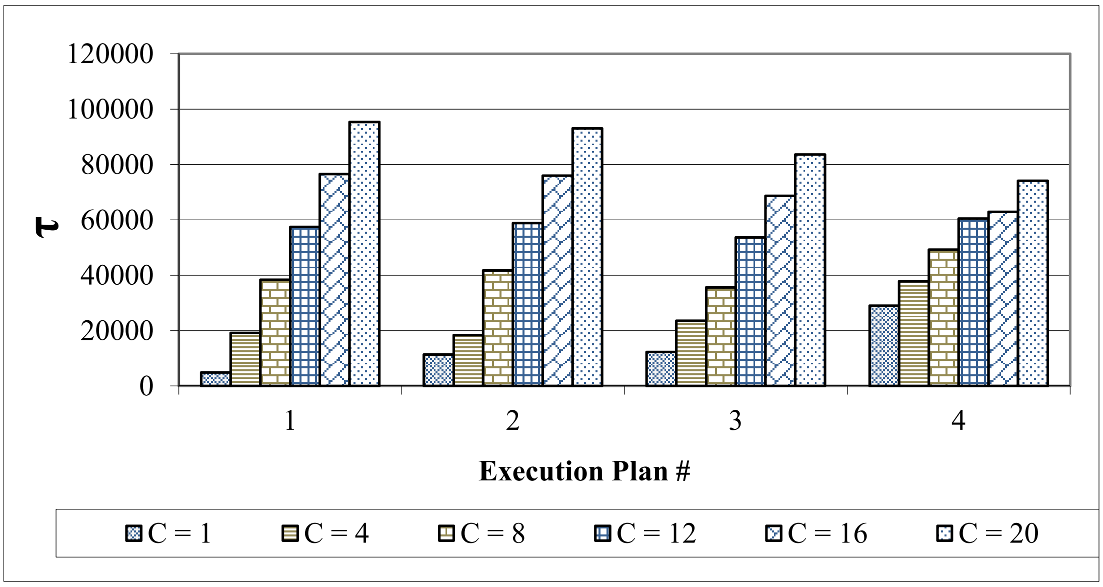


Figure 6-5: Comparison of performance of partitioned system when one execution plan is used at a time for graph size = 8

For C = 8, overall system performance is observed to be the best with execution plan #3 for graphs of size 8 and 12. For a higher number of clients (e.g. 12, 16 and 20 in Figure 6-4, Figure 6-5 and Figure 6-6), using execution plan #4 is observed to result in the best performance. This set of experiments demonstrates that using an appropriate execution plan based on the system load information (number of clients in these experiments) is important. Using a different execution plan for different number of clients accessing the system can result in achieving a high system performance.

This set of experiments provides a motivation for devising a runtime technique of selecting an execution plan based on the system load.

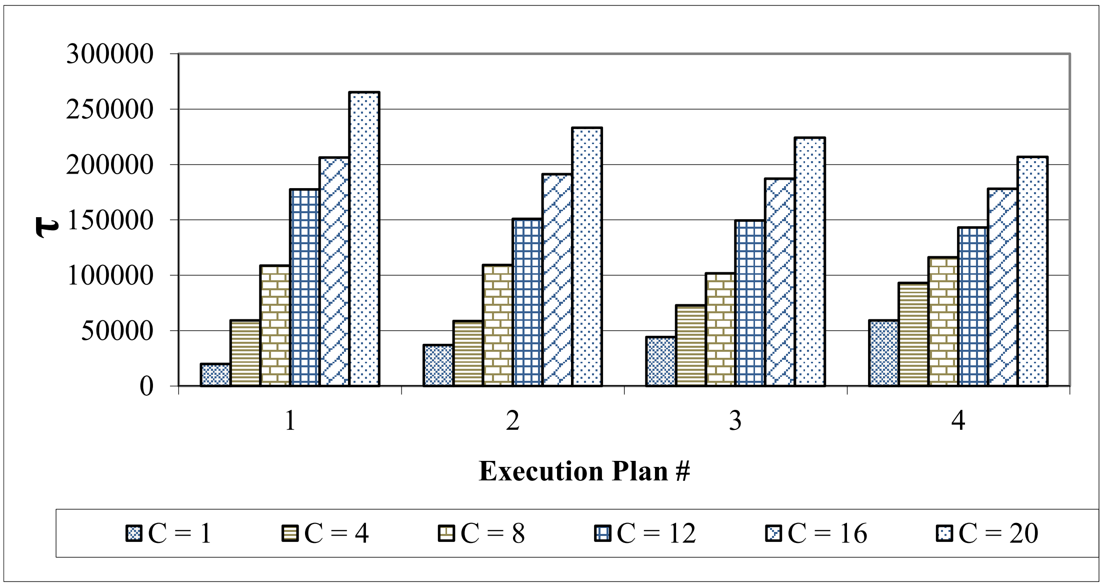


Figure 6-6: Comparison of performance of partitioned system when one execution plan is used at a time for graph size = 12

6.5.3.2 Performance Comparison with No Partitioning Case and the Design Time WS Partitioning Technique

For this set of experiments, the mean execution times are measured for an un-partitioned system, a partitioned system achieved with a design time partitioning technique and the partitioned system achieved with the proposed runtime WS partitioning technique. For the design time partitioning technique, the MOMC algorithm is used. The experimental analysis presented in Section 5.8 has demonstrated that the MOMC algorithm gave the best performance in comparison to the other design time partitioning techniques for small to medium sized application graphs. For the simulator used for experimentation, the value of the profile index measured is 20. As discussed in Section 6.3.3.2, the device profile index represents the maximum number of requests handled by a device. The device profile index is measured by experimenting with different number of clients. The performance comparison of the three techniques is done using three different ways of varying the number of clients. These are presented in the following subsections.

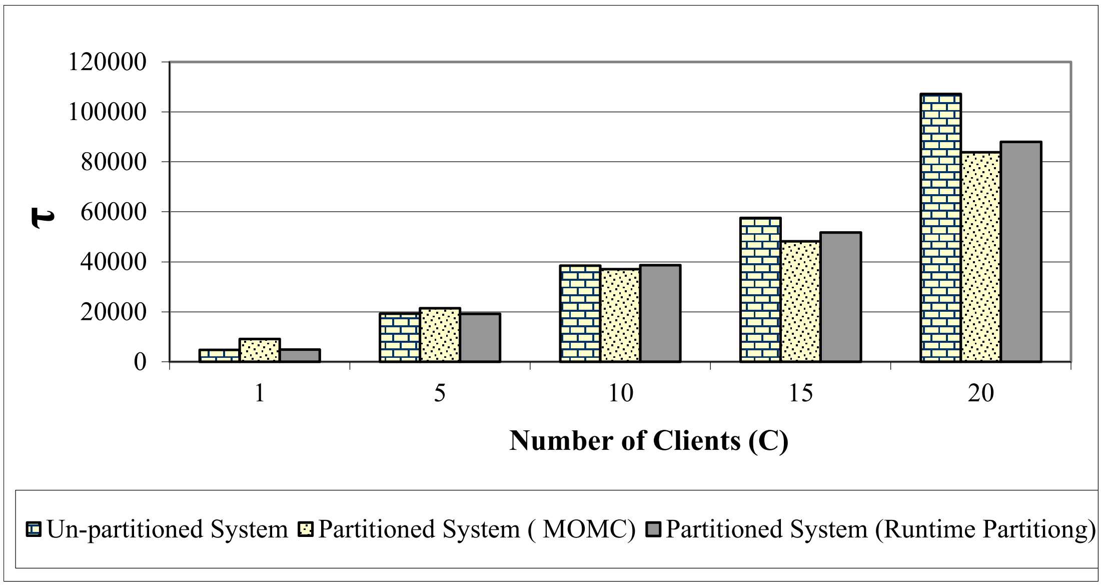


Figure 6-7: Performance comparison of the partitioned systems achieved with the three techniques using a fixed number of clients with a zero think time

For each set of bars shown in Figure 6-7, Figure 6-8 and Figure 6-9, the first bar represents the mean execution time achieved for an un-partitioned system. The second bar is for the mean execution time of a partitioned system achieved with the MOMC algorithm based on the design time WS partitioning technique. The third bar represents the mean execution time that is observed for a partitioned system achieved with the proposed runtime WS partitioning technique.

6.5.3.3 A System with a Fixed Number of Clients with No Client Think Time

In this set of experiments, the results are achieved on a system hosting a fixed number of clients with a zero think time. The values of the number of clients used are 1, 5, 10, 15 and 20. The results are presented in Figure 6-7.

The graph presented in Figure 6-7 shows that the design time partitioning technique seems to demonstrate the best performance for various number of clients (see the third, fourth and fifth sets of bars in Figure 6-7). For a lower number of clients, an un-partitioned system and the runtime partitioning technique seem to exhibit marginally better performance (see the first and second sets of bars in Figure 6-7) in comparison to the design time partitioning technique. The benefit of the design time WS partitioning technique is significant only when the partitioned system is actively accessed by a large number of clients. For a lower number of clients, the overheads of executing a part of an application on a remote computing node seem to be offsetting its benefits. The performance achieved with the runtime WS partitioning technique is always superior to that achieved on a non-partitioned system because it uses a different execution plan based on the number of clients. For example, for a lower number of clients, the runtime WS partitioning technique uses an execution plan that may demand to run the whole application locally or execute a small part of the application on a remote computing node. For a higher number of clients, the runtime WS partitioning technique is expected to use execution plans that require execution of a major part of an application on a remote computing node. The performance improvement over an un-partitioned system achieved by the runtime WS partitioning technique is observed to increase with an increase in the number of clients.

6.5.3.4 A system with a Fixed Number of Clients with Non-Zero Think Times

For this set of experiments, the number of clients used is 10, 20 and 40. Experiments are repeated with different values of the mean think time (500, 1000, 2000, 3000, 4000 and 5000 time units). The results presented in Figure 6-8 are obtained when with mean think time equal to 1000 time units. When the number of clients used is 10, the partitioned system achieved with the design time partitioning exhibits an inferior performance in comparison to cases when no partitioning or the runtime WS partitioning technique is used.

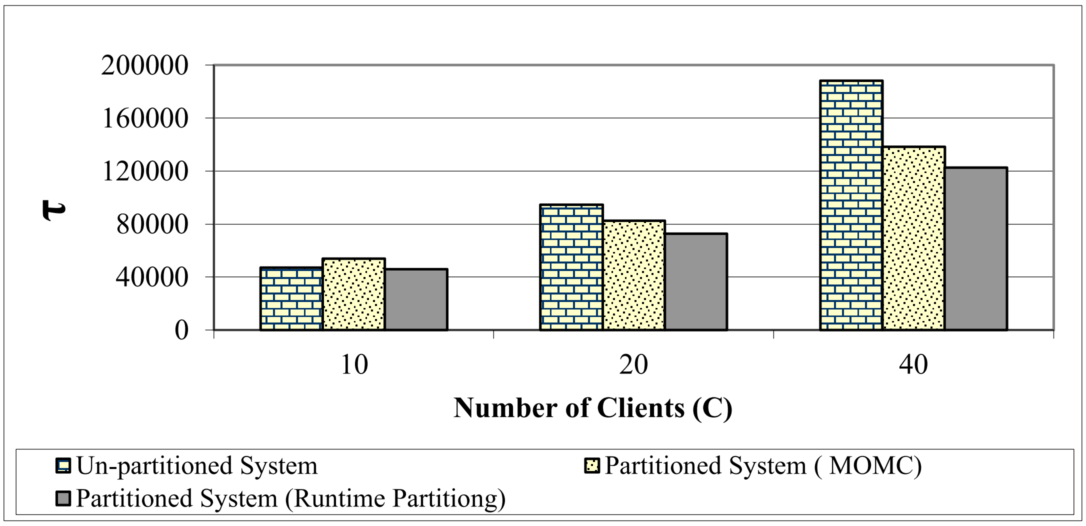
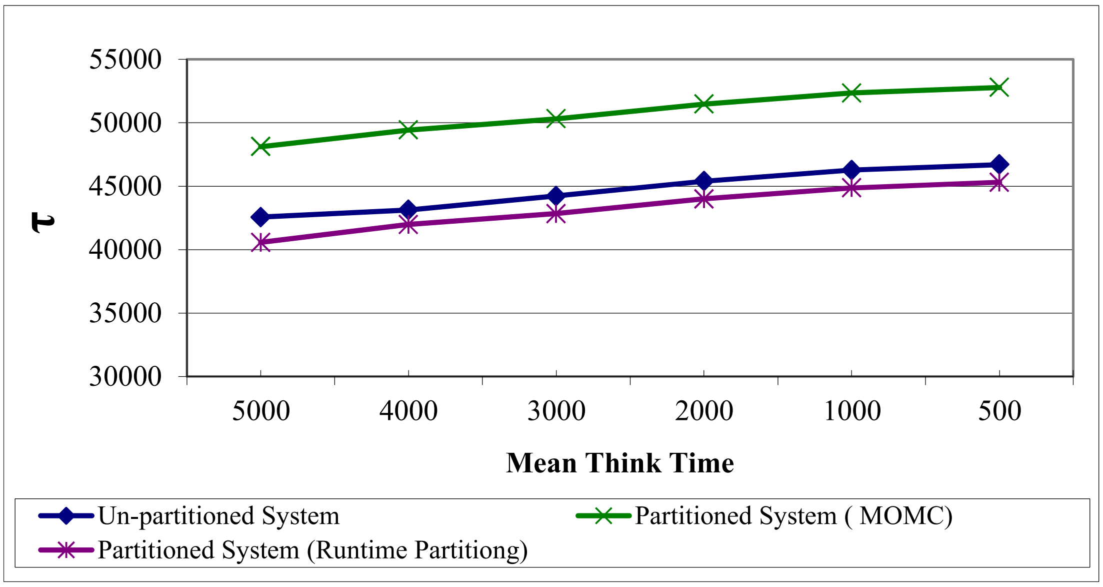
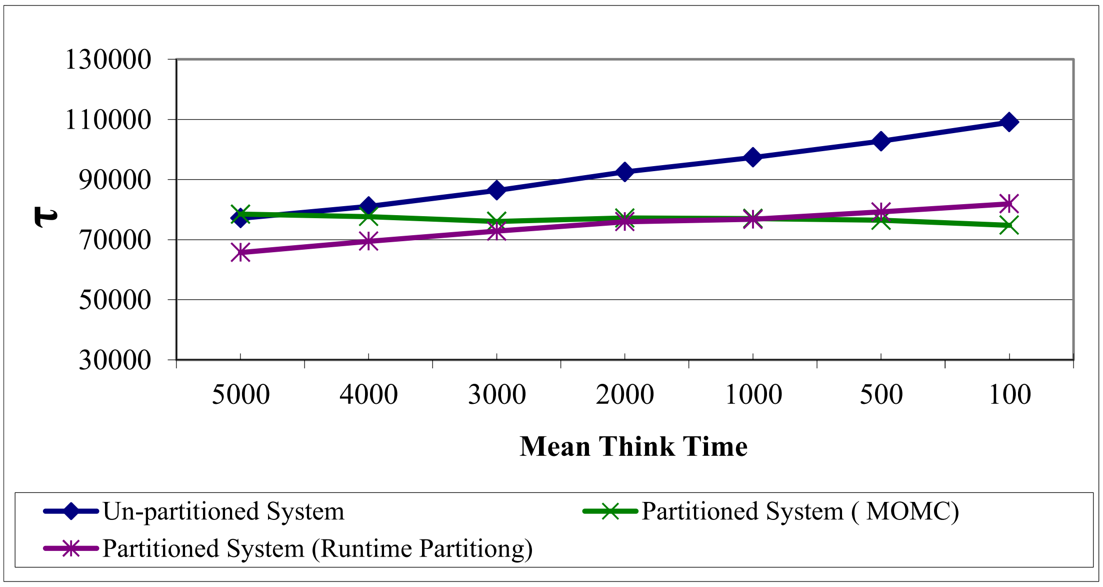


Figure 6-8: Performance comparison of the partitioned systems achieved with the three techniques using fixed number of clients with mean think time = 1000 time units

When a large number of clients are used (see the second and third sets of bars in Figure 6-8), the partitioned systems achieved with the design time partitioning and the runtime partitioning are showing better performance in comparison to the case when no partitioning is used. The performance of the partitioned system achieved with the runtime WS partitioning technique is observed to be the best among the three techniques. This experiment confirms the effectiveness of the runtime WS partitioning especially when the system is experiencing a variation in the number of active clients accessing the partitioned system because of the different values of think time.



(a)



(b)

Figure 6-9: Performance comparison of the three WS partitioning techniques with varying mean client think time and a) C = 10 and b) C = 20

The graphs presented in Figure 6-9 show the mean response time for the three WS partitioning techniques for different values of the mean think time. When a lower number of clients such as 10 is used, the proposed runtime WS partitioning technique exhibits the best performance (see Figure 6-9-a) for all values of the mean think time. But for a large number of clients such as 20, an interesting pattern is observed. The design time WS partitioning technique is observed to be the winner for lower values of the mean think time (see Figure 6-9-b) whereas the proposed runtime WS partitioning technique displays the best performance for higher values of the mean think time (see Figure 6-9-b).

6.5.3.5 Using Variable Number of Clients

In this set of experiments, the number of clients is varied at runtime between 1 and the maximum number of clients (C_{max}) which is an input for each experiment. Each experiment starts with 1 client. A set of clients are added to the system after processing a fixed number of requests. The value used for the fixed number of requests for this experiment is 500. Once the maximum number of clients (C_{max}) is reached, a number of clients is set to 1 client again and then incremented after processing every 500 requests and so on. The experiment is repeated using different values of C_{max}. The values used for C_{max} are 5, 10, 15, 20 and 40. The results observed for the partitioned systems achieved with the three WS partitioning techniques are presented in Figure 6-10.

For all values of the maximum number of clients (C_{max}), the partitioned system achieved with the runtime partitioning techniques exhibit the best performance. The performance improvement of this partitioned system over the other two techniques (no partitioning case and the design time WS partitioning technique (MOMC)) is more significant for various values of maximum number of clients (C_{max}). This means that for systems with large variations in the system load, the runtime WS partitioning techniques is expected to show a significant performance improvement

The partitioned system achieved with the design time WS partitioning technique (MOMC) shows an inferior performance in comparison to the no partitioning case when a lower value is used for C_{max}. The partitioned system achieved with MOMC shows an improvement in the mean execution time in comparison to the no partitioning case only when C_{max} is greater than 15. The design time WS partitioning algorithm (MOMC) tries to compute a significant part of a WS application for offloading and can result in increased overheads. When lower values are used for C_{max}, the overheads that accrue for the partitioned system achieved with MOMC offset the advantages of executing a part of application on the remote computing node.

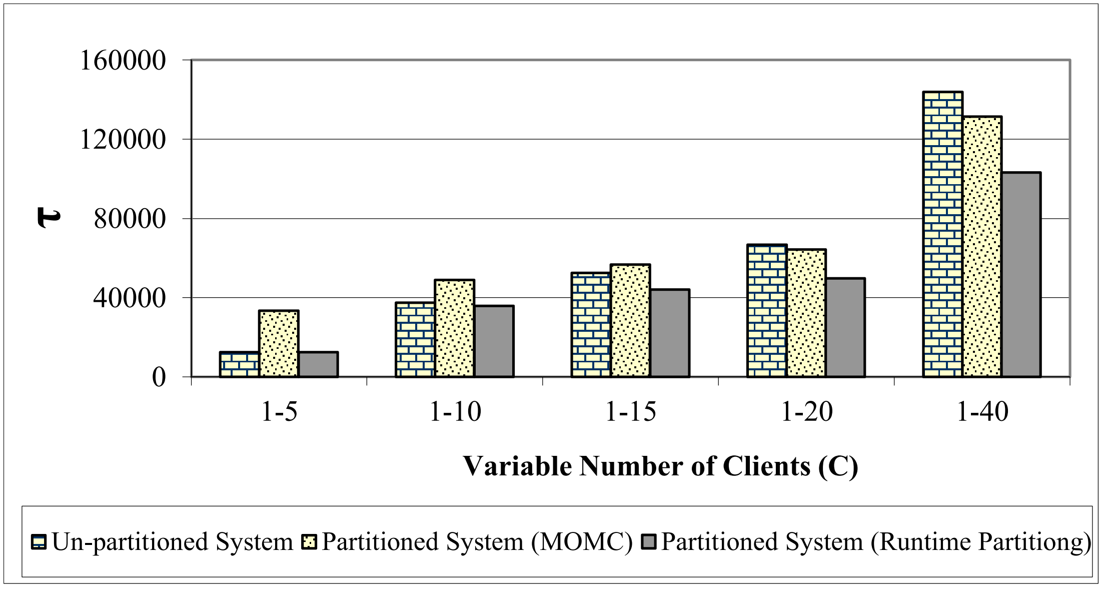


Figure 6-10: Performance comparison of the three techniques using variable number of clients.

With the variable number of clients, the partitioned system achieved with the design time partitioning technique (MOMC) seems to exhibit an inferior performance in comparison to the un-partitioned system and the partitioned system achieved with the runtime WS partitioning technique. The partitioned system achieved with the runtime WS partitioning technique proposed in this chapter is observed to outperform the other two techniques for all scenarios experimented with. The performance improvement observed on the partitioned system achieved with the runtime WS partitioning technique is higher when the variation in the number of clients is large.

6.6 Summary

This chapter introduced a runtime WS partitioning technique for mobile web services. WS application partitioning is expected to improve system performance on resource constrained mobile devices by executing a part of the application on a remote computing node. The partitioned system with multiple execution plans is determined first using the proposed graph based WS partitioning algorithm. The runtime middleware is then used for selecting an appropriate execution plan based on the system load information. The experimental analysis shows that

- For different system load scenarios, offloading a different size of partitions to be executed on a remote computing node is important. For a lower system load, executing a small part of an application on a remote computing node is beneficial. For heavier loads, the system performance is observed to be better when a large part of the application is executed on a remote computing node.
- For systems with a fixed number of clients and zero client think times the relative performance of the three techniques, no partitioning, design time partitioning and run time partitioning depends on the number of clients. If the number of clients is

lower than 10 then it is better to use run time partitioning (see graphs for C < 10 in Figure 6-7 for example). Using design time partitioning for a higher number of clients gives rise to the best system performance (see graph for C >10 in Figure 6-7 for example).

- For systems with a fixed number of clients but non-zero client think times the relative performances of the three techniques depends on the number of clients. For a smaller number of clients, the run time partitioning technique gives rise to the best performance for all the think times experimented with (see Figure 6-9 (a) for example). For a higher number of clients, 20 for example, the winner depends on the value of the mean think time. For higher values of the mean think time, run time partitioning displays the best performance (see Figure 6-9 (b)). As shown in this figure, when the mean think time is lower than 2000, the deign time partitioning technique exhibits the best performance but performance achieved with the runtime WS partitioning technique is close to the design time partitioning technique.
- For systems with a variable number of clients discussed in Section 6.5.3, the run time partitioning technique always gives rise to the best performance for the parameter values experimented with (see Figure 6-10).

The experimental results indicate that for low to medium load when the number of clients is fixed, run time partitioning is to be used. For high load, design time WS partitioning is a better choice. For systems characterized by a large variability in the number of clients running at a time on the system the run time partitioning technique gives rise to the best performance.

Chapter 7: Conclusions

Hosting web services on mobile devices is challenging because of the resource constraints associated with such devices. A number of different techniques for addressing these challenges are presented in this thesis. System performance is investigated in detail in the context of each of these techniques. This chapter summarizes the thesis contributions, conclusions and directions for future research.

7.1 Summary

Hosting web services on mobile devices is investigated in detail. The results of these investigations are summarized next.

7.1.1 Lightweight and Partitioned Web Service Execution Environments

For hosting web services on mobile devices, two web service execution environments (WSEE) are proposed: a lightweight WSEE (see Section 3.2) and a partitioned WSEE (see Section 3.3). The lightweight WSEE is proposed for resource constrained mobile devices to support a set of basic WS standards for hosting web services. To conform to the data and the resource intensive WS standards such as WS-Security and WS-AtomicTransaction, a configurable partitioned WSEE is proposed. For a lower number of WS clients and a basic set of WS standards, the lightweight WSEE demonstrates better performance over the partitioned WSEE. For a larger number of WS clients or when additional resource demanding WS standards are required to be supported, the partitioned WSEE exhibits a superior performance. For mobile devices with very limited resources such as pagers and conventional mobile phones, the performance of the partitioned WSEE is observed to be less sensitive to the increase in the number of WS clients in comparison to the lightweight WSEE. Also, the partitioned WSEE is observed to be scalable with the number of WS clients.

7.1.2 WS Partitioning Frameworks

The thesis also analyzes performance of three application partitioning frameworks (Intermediate framework, Backend framework and Forwarding framework) for hosting of web services on mobile devices. The intermediate and backend frameworks were proposed in the literature for both mobile and conventional applications. This thesis analyzes their feasibility for hosting web services on mobile devices. Note that these two frameworks were never been analyzed for WS hosting. This thesis also proposes a new framework called the forwarding framework (see Section 4.2.3). Based on prototyping and measurement, an in-depth analysis is performed to investigate the performance of the three partitioning frameworks for hosting web services on resource constrained mobile devices. The analysis shows that the intermediate and the forwarding frameworks lead to a significant performance improvement over an un-partitioned system but the performance of the backend framework is observed to be superior to an un-partitioned system only when a large number of WS clients are active or a significantly large portion of an application is executed on a backend computing node. For the experimental data presented in Section 4.6, both the forwarding and the intermediate frameworks have shown a superior performance in comparison to the backend framework. The intermediate framework seems to be a better choice over the backend framework and the forwarding framework for web services that require the WSEE to support extra WS standards such as the security standard. The rationale behind such a behavior is the execution of a larger component of the WSEE on the intermediate node. The forwarding framework is observed to show a slightly better performance over the intermediate framework when large volumes of data are exchanged among different WS partitions. This seems to be an effect of the lower number of messages exchanged in the forwarding framework, the impact of which becomes significant for higher volumes of data exchanged. For most of other scenarios, the forwarding framework achieved a comparable performance as the intermediate framework.

7.1.3 Design Time WS Partitioning Techniques

The thesis proposes two design time graph based algorithms for WS partitioning. The proposed design time algorithms are Maximum Offloading Minimum Cost (see Section 5.4) and Cluster based Application Partitioning (see Section 5.5). The effectiveness of these proposed algorithms is investigated through a system prototype (using sample web services) as well as simulation (using randomly generated application graphs). The results achieved with the prototype and the simulations demonstrate that the proposed algorithms outperform the existing techniques (OEA and MinCut) for WS partitioning. The MOMC partitioning algorithm is observed to perform the best for small to medium size applications whereas the CAP algorithm exhibits its effectiveness for large applications. The OEA and the MinCut based WS partitioning techniques produce effective partitions only for specific types of graphs whereas the MOMC algorithm and the CAP algorithm based WS partitioning techniques are effective on a wider variety of graphs. The type of knowledge to be used in devising a partitioning algorithm is investigated. MinCut that uses only the communication cost is found to lead to an inferior performance of the partitioned system in comparison to those achieved with MOMC and CAP that use both communication and processing costs.

7.1.4 Runtime WS Partitioning Technique

The thesis proposes a runtime WS partitioning technique for mobile web services (see Section 6.2). For the runtime WS partitioning technique, the partitioned system with multiple execution plans is determined first using the proposed graph based WS partitioning algorithm. The runtime middleware is then used at runtime for selecting an appropriate execution plan based on system load information. For different system load scenarios, offloading a different size of partitions to be executed on a remote computing node is observed to be important. For a lower system load, executing the whole application locally or executing only a small part of the application on a remote computing node is beneficial. For heavier loads, the system performance is observed to give rise to a significant performance improvement over an un-partitioned system when a large part of the application is executed on a remote computing node.

An analysis is performed to compare the performance of design time WS partitioning techniques and the runtime WS partitioning technique. This comparison produces interesting results (see Section 6.5.3.3, Section 6.5.3.4 and Section 6.5.3.5). For low to medium load and when the number of clients is fixed, run time partitioning seems to be a better choice. For high load, the design time WS partitioning is a better choice.

A number of important insights into the system behavior and performance gained through the performance analysis of the WS partitioning techniques is presented next. For WS applications with a small number of components that are invoked by a single

WS client, using no WS partitioning is a better choice. For systems characterized by a large variability in the number of clients running at a time on the system, the run time partitioning technique gives rise to the best performance. For systems with a large number of WS clients and characterized by a low variability in the number of clients running at a time, the design time WS partitioning is a better choice. Among the design time partitioning techniques, the MOMC based technique exhibits the best performance for low to medium sized WS applications whereas the CAP based technique is suitable for WS applications comprising a large number of components.

7.2 Directions for Future Research

A few directions for future research are presented next.

- The use of information on the battery life of a mobile device and the available memory on a mobile device by the WS partitioning algorithms forms another important direction of future work.
- For runtime WS partitioning techniques, the use of remaining battery life of the mobile device for selection of an execution plan forms another direction for future research.
- Generating partitions such that the bandwidth used is lower than a predefined bandwidth budget warrants further investigation.
- The three nodes of the proposed WS partitioning frameworks can be connected via switching nodes. Analyzing the effect of the number of such switching nodes on the relative performance of the frameworks can be another interesting direction for future research. The impact of the location of WS clients and mobile devices on the relative performance of the proposed frameworks warrants investigation. For example, if a WS client

and a WS provider (mobile device) exist in the same network, a system that does not involve an intermediate, backend or forwarding node may result in a better performance in comparison to the frameworks involving such nodes.

- Investigation of a WS partitioning framework that uses caching techniques for caching data from a mobile device on an intermediate node and access the mobile device only whenever it is required forms another direction for future research.
- Techniques for the selection of one or more remote computing nodes from a set of available computing nodes for execution of one or more partitions require further investigation.
- Getting one more remote computing nodes from a cloud on demand and downloading the appropriate partitions for execution forms another interesting direction for future research.

References

[Ada06] M. Adaçal and A. Bener, “Mobile Web Services: A New Agent-Based Framework”, *IEEE Internet Computing*, Vol. 10, No. 3, pp. 58-65, June 2006.

[Aij08] F. Aijaz, S. Adeli and B. Walke, “Middleware for Communication and Deployment of Time Independent Mobile Web Services”, *In the Proceedings of the IEEE International Conference on Web Services (ICWS’08),* pp. 797-800, Hawaii, HI, U.S.A., July 2008.

[Aij10] F. Aijaz, M. Chaudhry and B. Walke, "Mobile Web Services in Health Care and Sensor Networks", *In the Proceedings of the Second International Conference on Communication Software and Networks (ICCSN’10),* pp. 254-259, Singapore, February 2010.

[Alc07] Alchemy APIs, “AlchemySOAP 1.0.0: C++ Open Source SOAP-based Web Services Framework”, 2007, *available at http://sourceforge.net/ projects/alchemysoap/* [Accessed: March 03, 2012].

[Als11-1] F. Alshahwan, K. Moessner and F. Carrez, “Providing Light Weight Distributed Web Services from Mobile Hosts”, *In the Proceedings of the 2011 IEEE International Conference on Web Services (ICWS’11)*, pp. 652–659, Washington, DC, U.S.A., July 2011.

[Als11-2] F. Alshahwan, K. Moessner and F. Carrez, “Evaluation of Distributed SOAP and RESTful Mobile Web Services”, *International Journal on Advances in Internet Technology*, Vol. 3, No. 3 & 4, pp. 447 – 461, April 2011.

[Apa08] Apache Software Foundation, "Apache Axis 2 User's Guide", 2008, *available at http://ws.apache.org/axis2/1_4/userguide.html* [Accessed: March 03, 2012].

[Asi07-1] M. Asif, S. Majumdar and R. Dragnea, "Hosting Web Services on Resource Constrained Devices", *In the Proceedings of the 2007 IEEE International Conference on Web Services (ICWS'07)*, pp. 583-590, Salt Lake City, UT, U.S.A., July 2007.

[Asi07-2] M. Asif, S. Majumdar and R. Dragnea, "Application Partitioning for Enhancing System Performance for Services Hosted On Wireless Devices", *In the Proceedings of the First International Workshop on Service Oriented Engineering and Optimization (SENOPT'07)*, pp. 1-15, Goa, India, December 2007.

[Asi-08-1] M. Asif, S. Majumdar and R. Dragnea, "Partitioning the WS Execution Environment for Hosting Mobile Web Services", *In the Proceedings of the 2008 IEEE International Conference on Services Computing (SCC'08)*, pp. 315-322, Honolulu, HI, U.S.A., July 2008.

[Asi-08-2] M. Asif and S. Majumdar, "Performance Analysis of Mobile Web Service Partitioning Frameworks", *In the Proceedings of the Sixteenth International Conference on Advanced Computing and Communication*, pp. 190-197, Chennai, India, December 2008.

[Ava02] S. Avancha, V. Korolev, A. Joshi and T. Finin, "On Experiments with a Transport Protocol for Pervasive Computing Environments", *Journal of Computer Networks*, Vol. 4, No. 40, pp. 515-535, November 2002.

[Bar93] S. Barnard and H. Simon, "A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems", *In the Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pp. 711–718, Norfolk, VA, U.S.A., March 1993.

[BPE03] IBM, BEA Systems, Microsoft, SAP AG, Siebel Systems, "Business Process Execution Language for Web Services version 1.1", 2003, *available at http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf* [Accessed: March 29, 2012].

[Bra05] P. Braun and W. Rossak, “Mobile Agents - Basic Concepts, Mobility Models and the Tracy Toolkit”, *Morgan Kaufmann Publishers Inc.*, San Francisco, CA, U.S.A., 2005.

[Bro04] R. Brown, “Smoothing Forecasting and Prediction of Discrete Time Series”, *Dover Publications*, Mineola, NY, U.S.A., 2004.

[CDC05] Oracle Corporation, “Connected Device Configuration, JSR 218”, 2005, *available at http://java.sun.com/products/cdc/* [Accessed: March 24, 2012].

[Cha02] D. Chandra, C. Fensch, W. Hong, L. Wang, E. Yardımci and M. Franz, “Code Generation at the Proxy: An Infrastructure-based Approach to Ubiquitous Mobile Code”, *In the Proceedings of the Fifth ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS’02)*, pp. 123-130, Malaga, Spain, June 2002.

[Che04] G. Chen, B. Kang and M. Kandemir, “Studying Energy Trade-offs in Offloading Computation/Compilation in Java-Enabled Mobile Devices”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, No. 9, pp. 795-809, September 2004.

[Che02] S. Cheng, J. Liu, J. Kao and C. Chen, “A New Framework for Mobile Web Services”, *In the Proceedings of the 2002 Symposium on Applications and the Internet Workshops (SAINT’02)*, pp. 28-34, Nara City, Japan, January- February 2002.

[Chu04] H. Chu, C. You and C. Teng, “Challenges: Wireless Web Services”, *In the Proceedings of the Tenth International Conference on Parallel and Distributed Systems (ICPADS’04)*, pp. 657-662, Newport Beach, CA, U.S.A., July 2004.

[CLD05] Oracle Corporation, “Connected Limited Device Configuration, JSR 139”, 2005, *available at http://java.sun.com/products/cldc/* [Accessed: May 2, 2012]

[Edm72] J. Edmonds and R. Karp, “Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems”, *Journal of the ACM (JACM)*, Vol. 19, No. 2, pp.248-264, April 1972.

[Eng03] R. Engelen, “gSOAP Project”, 2003, *available at http://gsoap2. sourceforge.net/* [Accessed: March 24, 2012].

[Eso04] Ultimodule Inc., “eSOAP Toolkit”, 2004, *available at http://esoap.ulti module.com/bin/esoap/templates/splash.asp?NC=1881X* [Accessed: June 24, 2011].

[Fel04] S. Fell, “PocketSOAP online documentation, Version 1.5”, 2004, *available at http://www.pocketsoap.com/pocketsoap/docs/default.htm* [Accessed: March 04, 2012].

[Fie00] R. Fielding, “Architectural Styles and the Design of Network-based Software Architectures”, *Ph.D. Thesis,* University of California, Irvine, CA, U.S.A., 2000.

[Ful56] D. R. Fulkerson, "Maximal Flow through a Network", *Canadian Journal of Mathematics,* Vol. 8, No. 3, pp. 399-404, 1956.

[Gam95] E. Gamma, R. Helm, R. Johnson and J. Vlissides**, “**Design Patterns: Elements of Reusable Object-Oriented Software**”,** *Addison-Wesley Professional Computing Series*, First Edition, Boston, MA, U.S.A., January 1995.

[Geh05] G. Gehlen and L. Pham, “Mobile Web Services for Peer-to-Peer Applications”, *In the Proceedings of the Second IEEE Consumer Communications and Networking Conference (CCNC’05)*, pp. 427-433, Las Vegas, NV, U.S.A., January 2005.

[Geo08] GeoNames, “Geographical Database System”, *available at http:// www.geonames.org* [Accessed: May 2, 2012].

[Goo08] Google Inc., “Google Maps Services”, 2008, *available at http://code. google.com/apis/maps/documentation/services.html* [Accessed: April 08, 2011].

[Gro06] G. Gross and J. Yellen, “Graph Theory and its Applications”, *Chapman and Hall*, Second Edition, New York, NY, U.S.A., 2006.

[Han05] R. Handorean, R. Sen, G. Hackmann and G. Roman, "Context Aware Session Management for Services in Ad Hoc Networks", *In the Proceedings of the 2005 IEEE International Conference on Services Computing (SCC'05)*, pp. 113-120, Orlando, FL, U.S.A., July 2005.

[Hem05] H. Hemmati, A. Ranjbar, M. Niamanesh, and R. Jalili, "A Model to Support Context-Aware Service Migration in Pervasive Computing Environments", *In the Proceedings of the Ninth World Multi-Conference on Systemics, Cybernetics and Informatics*, pp. 223-225, Orlando, FL, U.S.A., July 2005.

[Hen93] B. Hendrickson and R. Leland, "A Multilevel Algorithm for Partitioning Graphs", *Technical Report, SAND93-1301*, Sandia National Laboratories, 1993.

[Hun99] G. Hunt and M. Scott, "The Coign Automatic Distributed Partitioning System", *In the Proceedings of the Third USENIX Symposium on OS Design and Implementation* (*OSDI'99)*, pp. 187-200, New Orleans, LA, U.S.A., February 1999.

[Jam05] V. Jamwal and S. Iyer, "Automated Refactoring of Objects for Application Partitioning", *In the Proceedings of the Twelfth Asia-Pacific Software Engineering Conference (APSEC'05)*, pp. 671-678 Taipei, Taiwan, December 2005.

[Jav04] Oracle Corporations, "Java Image I/O API Specifications, JSP-015", *available at http://java.sun.com/j2se/1.4.2/docs/guide/imageio/index.html* [Accessed: March 08, 2011].

[Jme06] Oracle Corporations, "Java Platform, Micro Edition (J2ME)", 2006, *available at http://java.sun.com/javame/* [Accessed: March 24, 2012]

[Kar99] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs", *SIAM Journal on Scientific Computing,* Vol. 20, No. 1, pp. 359-392, 1999.

[Kim07] Y. Kim and K. Lee, "A Light-weight Framework for Hosting Web Services on Mobile Devices", *In the Proceedings of* the *Fifth*

European Conference on Web Services (ECOWS'07), pp. 255 – 263, Halle, Germany, November 2007.

[Kso03] kSOAP Project, 2003, *available at http://ksoap.objectweb.org/* [Accessed: May 4, 2012]

[Kxm03] KXML2 Project, 2003, *available at http://kxml.sourceforge.net/kxml2/* [Accessed April 21, 2012].

[Lee06] W. Lee, K. Lee and S. Lee, "Intermediary based Architecture for Mobile Web Services", *In the Proceedings of the Eighth International Conference on Advanced Communication Technology (ICACT'06)*, pp. 1973-1978, Phoenix Park, Korea, February 2006.

[Li01] Z. Li, C. Wang and R. Xu, "Computation Offloading to Save Energy on Handheld Devices: A Partition Scheme", *In the Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 238-246, Atlanta, GA, U.S.A., November 2001.

[Lio04] N. Liogkas, B. MacIntyre, E. Mynatt, Y. Smaragdakis, E. Tilevich and S. Voida, "Automatic Partitioning: A Promising Approach to Prototyping Ubiquitous Computing Applications", *IEEE Pervasive Computing, Special Issue on Building and Evaluating Ubiquitous System Software*, Vol. 3, No. 3, pp. 40-47, July 2004.

[Luq08] L. Luqun, "An Integrated Web Service Framework for Mobile Device Hosted Web Service and Its Performance Analysis", *In the Proceedings of the Tenth IEEE International Conference on High Performance Computing and Communications, (HPCC'08)*, pp. 659-664, Dalian, China, September 2008.

[Mac05] J. MacDonald and C. Mitchell, "Using the GSM/UMTS SIM to Secure Web Services", *In the Proceedings of the Second IEEE International Workshop on Mobile Commerce and Services (WMCS'05)*, pp. 70-78, Munchen, Germany, July 2005.

[Mcf03] S. McFaddin, C. Narayanaswami and M. Raghunath, "Web Services on Mobile Devices – Implementation and Experience", *In the*

Proceedings of the Fifth IEEE Workshop on Mobile Computing Systems & Applications (WMCSA'03), pp. 100-109, Monterey, CA, U.S.A., October 2003.

[Mes02] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. Giuli and X. Gu, "Towards a Distributed Platform for Resource-Constrained Devices", *In the Proceedings of the International Conference on Distributed Computing Systems (ICDCS'02)*, pp. 43-56, Vienna, Austria, July 2002.

[Mil93] G. Miller, S. Teng, W. Thurston, and S. Vavasis, "Automatic Mesh Partitioning", *Graph Theory and Sparse Matrix Computation*, pp. 57-84, Springer-Verlag, New York, NY, U.S.A., 1993.

[Nou86] B. Nour-Omid, A. Raefsky, and G. Lyzenga, "Solving Finite Element Equations on Concurrent Computers", *In the Proceedings of the Symposium on Parallel Computations and their Impact on Mechanics*, pp. 291-307, Boston, MA, U.S.A., December 1986.

[Net05] Microsoft Corporation, ".NET Compact Framework 2.0", *available at http://msdn.microsoft.com/en-us/netframework/aa731542* [Accessed: March 24, 2011].

[Ou07] S. Ou , K. Yang and J. Zhang, "An Effective Offloading Middleware for Pervasive Services on Mobile Devices", *Elsevier Journal of Pervasive and Mobile Computing*, Vol. 3, No. 4, pp. 362-385, August 2007.

[Par06] G. Park, S. Kim, G. Bae, Y. Kim and B. Kang, "An Automated WSDL Generation and Enhanced SOAP Message Processing System for Mobile Web Services", *In the Proceedings of the Third International Conference on Information Technology: New Generations (ITNG'06)*, pp. 382-387, Las Vegas, NV, U.S.A., April 2006.

[Pha05] L. Pham and G. Gehlen, "Realization and Performance Analysis of a SOAP Server for Mobile Devices," *In the Proceedings of the Eleventh European Wireless Conference,* Vol. 2, pp. 20-27, Nicosia, Cyprus, April 2005.

[Pil03] T. Pilioura, A. Tsalgatidou and S. Hadjiefthymiades, “Scenarios of using Web Services in M-Commerce”, *ACM SIGecom Exchanges*, Vol. 3, No. 4, pp. 28-36, January 2003.

[Pop10] V. Popescu, “Java Application Profiling using TPTP”, 2010, *available at http://www.eclipse.org/articles/Article-TPTP-Profiling-Tool/tptp ProfilingArticle.html* [Accessed: April 23 2012].

[Ren05] O. Rendon, F.Pabon, M. Gomez, Vargas and J. Guaca, “Architectures for Web Services Access from Mobile Devices”, *In the Proceedings of the Third Latin American Web Congress (LA-WEB’05)*, pp.93-97, Buenos Aires, Argentina, October-November 2005.

[Ren09] C. Renouf, “Pro (IBM) WebSphere Application Server 7 Internals”, *Apress*, First Edition, New York, NY, U.S.A., 2009

[Riv07] O. Riva, T. Nadeem, C. Borcea, and L. Iftode, “Mobile Services: Context-Aware Migratory Services in Ad Hoc Networks”, *IEEE Transactions on Mobile Computing*, Vol. 6, No. 12, pp. 1313-1328, December 2007.

[Ros02] J. Rosenberg, “SIP: Session Initiation Protocol. Request for Comments (Standards Track) 3261”, *Internet Engineering Task Force (http://www.ietf.org/)*, June 2002.

[Ros87] S. Rosen, “Simulation*”, Lectures on the Measurement and Evaluation of the Performance of Computing Systems, Society for Industrial Mathematics*, February 1987.

[Rus02] J. Russ, “The Image Processing Handbook”, Fifth Edition, *CRS Press*, Boca Raton, FL, U.S.A., 2002.

[Sch99] A. Schill, A. Held, T. Ziegert and T. Springer, “A Partitioning Model for Applications in Mobile Environments”, *In the Proceedings of the Mobile Agents in the Context of Competition and Cooperation (MAC3) Workshop at Autonomous Agents*, pp. 34-41, Seattle, WA, U.S.A., May 1999.

[Sch06] D. Schall, M. Aiello and S. Dustdar, “Web Services on Embedded Devices”, *International Journal of Web Information Systems,* Vol. 2 No. 1, pp.45-50, February 2006.

[She04] J. Shen, B. Han, M. Yuen and W. Jia, “End-to-End Wireless Multimedia Transmission System”, *In the Proceedings of the IEEE Vehicular Technology Conference (VTC'04)*, pp. 2616-2620, Milan, Italy, May 2004.

[Sil04] A. Silberschatz, G. Gagne and P. Galvin, “Operating Systems Concepts”, Seventh Edition, *John Wiley & Sons Inc.*, December 2004.

[Sri06] S. Srirama, M. Jarke and W. Prinz, “Mobile Web Service Provisioning”, *In the Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW 2006)*, pp. 120-128, Guadeloupe, French Caribbean, February 2006.

[Sri07-1] S. Srirama and A. Naumenko, “Secure Communication and Access Control for Mobile Web Service Provisioning”, *In the Proceedings of the International Conference on Security of Information and Networks (SIN’07)*, pp. 64-70, North Cyprus, Turkey, May 2007.

[Sri07-2] S. Srirama, M. Jarke and W. Prinz, “A Performance Evaluation of Mobile Web Services Security”, *In the Proceedings of the Third International Conference on Web Information Systems and Technologies (WEBIST’07),* pp. 386-392, Barcelona, Spain, March 2007.

[Sri10] S. Srirama, M. Jarke, E. Vainikko and V. Sor, “Scalable Mobile Web Services Mediation Framework”, *In the Proceedings of the Fifth International Conference on Internet and Web Applications and Services*, pp. 315–320, Barcelona, Spain, May 2010.

[Sri11] S. Srirama, M. Jarke, E. Vainikko and V. Sor, “Supporting Mobile Web Service Provisioning with Cloud Computing”, *International Journal on Advances in Internet Technology*, Vol. 3, No. 3 & 4, pp. 261-273, April 2011.

[Ste05] R. Steele, K. Khankan and T. Dillon, "Mobile Web Services Discovery and Invocation through Auto-Generation of Abstract Multimodal Interface", *In the Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)*, pp. 35-41, Las Vegas, NV, U.S.A., April 2005.

[Sto97] M. Stoer and F. Wagner, "A Simple Min-Cut Algorithm," *Journal of the ACM*, Vol. 44, No. 4, pp. 585-591, July 1997.

[Suk10] A. Sukhov, N. Kuznetsova, A. Pervitsky and A. Galtsev, "Generating Function for Network Delay", *CoRR abs/1003.0190*, February 2010.

[Til02] E. Tilevich and Y. Smaragdakis, "J-Orchestra: Automatic Java Application Partitioning", *In the Proceedings of the Sixteenth European Conference on Object-Oriented Programming*, pp. 178-204, Malaga, Spain, June 2002.

[Udd05] Universal Description, Discovery and Integration (UDDI) Version 3.0, 2005, *available at http://www.uddi.org* [Accessed: March 24, 2012]

[W3c02-1] World Wide Web Consortium (W3C), "XML Encryption: Syntax and Processing", 2002, *available at http://www.w3.org/TR/xmlenc-core/* [Accessed: June 24, 2011]

[W3c02-2] World Wide Web Consortium (W3C), "XML Signature: Syntax and Processing", 2002, *available at http://www.w3.org/TR/xmldsig-core/* [Accessed: April 24, 2012]

[Wan08] L. Wang and M. Franz, "Automatic Partitioning of Object-Oriented Programs for Resource-Constrained Mobile Devices with Multiple Distribution Objectives", *In the Proceedings of the Fourteenth IEEE International Conference on Parallel and Distributed Systems (ICPADS'08)*, pp. 369-376, Melbourne, Australia, December 2008.

[Wat95] T. Watson, "Effective Wireless Communication through Application Partitioning", *In the Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, pp. 24-27, Orcas Island, WA, U.S.A., May 1995.

[Wel86] D. Wells, "The Penguin Dictionary of Curious and Interesting Numbers", *Penguin Books*, Middlesex, England, 1986.

[Wsa04] Oracle Corporations, "J2ME Web Services APIs (WSA), JSR 172", 2004, *available at http://java.sun.com/products/wsa/* [Accessed: June 24, 2011]

[Wsd01] Web Service Description Language (WSDL), 2001, *available at http://www.w3.org/tr/wsdl* [Accessed: June 24, 2011].

[Wu05] M. Wu, "Teaching Graph Algorithms using Online Java Package, IAPPGA", *ACM SIGCSE Bulletin,* Vol. 37, No. 4, pp. 64-68, December 2005.

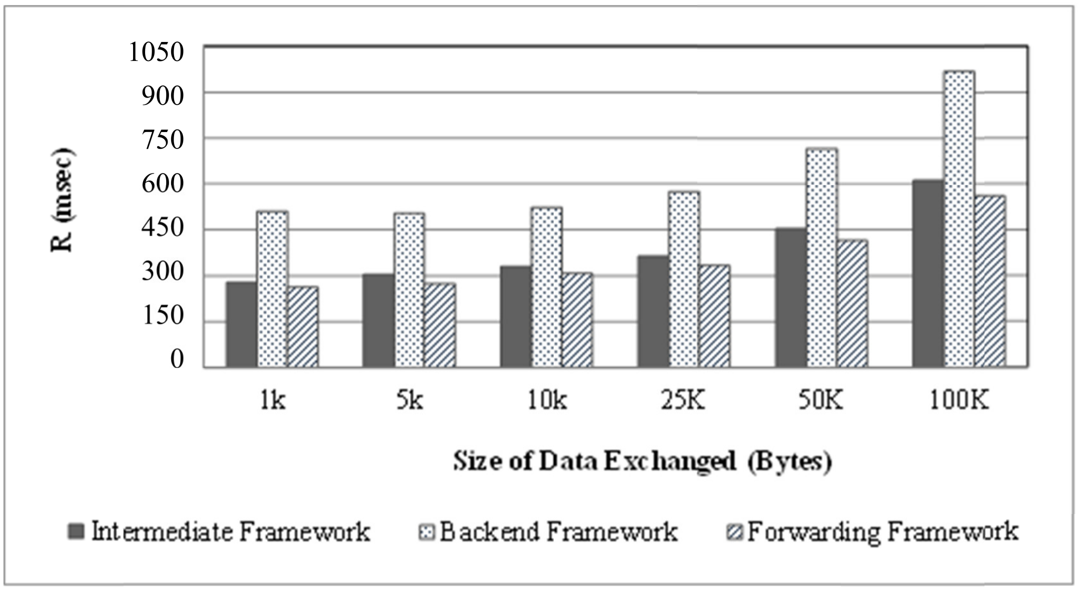
[Xml02] XML PULL Parser API, *available at http://www.xmlpull.org/* [Accessed: March 12, 2011].

[You01] C. Young, Y. Lakshman, T. Szymanski, J. Reppy, D. Presotto, R. Pike, G. Narlikar, S. Mullender and F. Grosse, "Protium, an Infrastructure for Partitioned Applications", *In the Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, pp.47–52, Elmau/Oberbayern, Germany, May 2001.

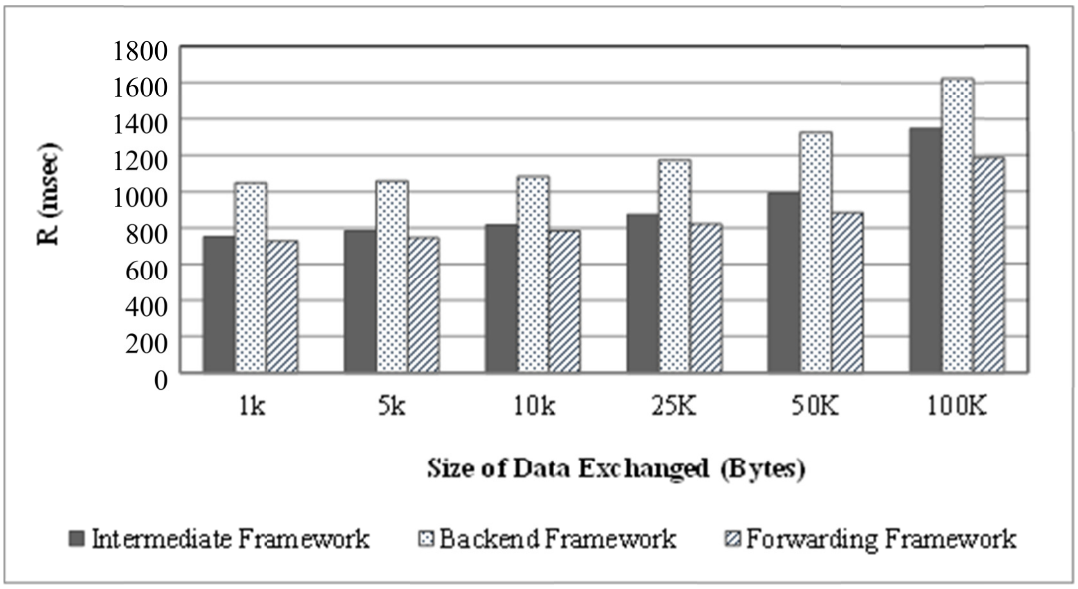
[Zha11] J. Zhang, S. Chen, Y. Lu and D. Levy, "A Mobile Web Service Middleware and its Performance Study", *Transactions on Large-Scale Data and Knowledge-Centered Systems (TLDKS)*, Springer LNCS, Vol. 6790, pp. 185-207, 2011.

Appendix A: WS Partitioning Frameworks

A.1 Effect of the Size of Data Exchanged between WS Partitions using one WS client



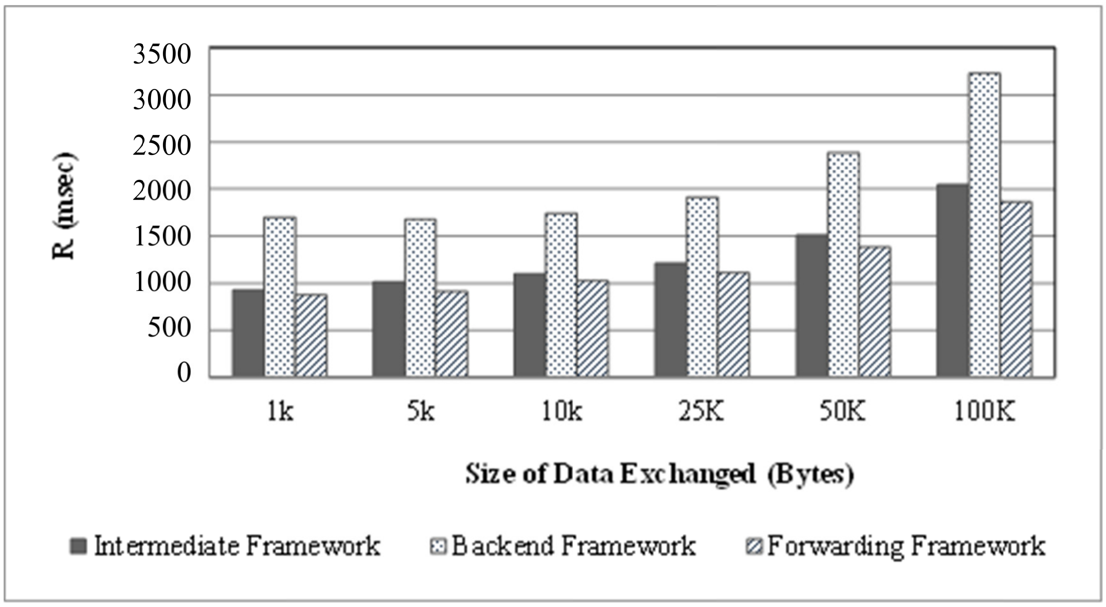
(a)



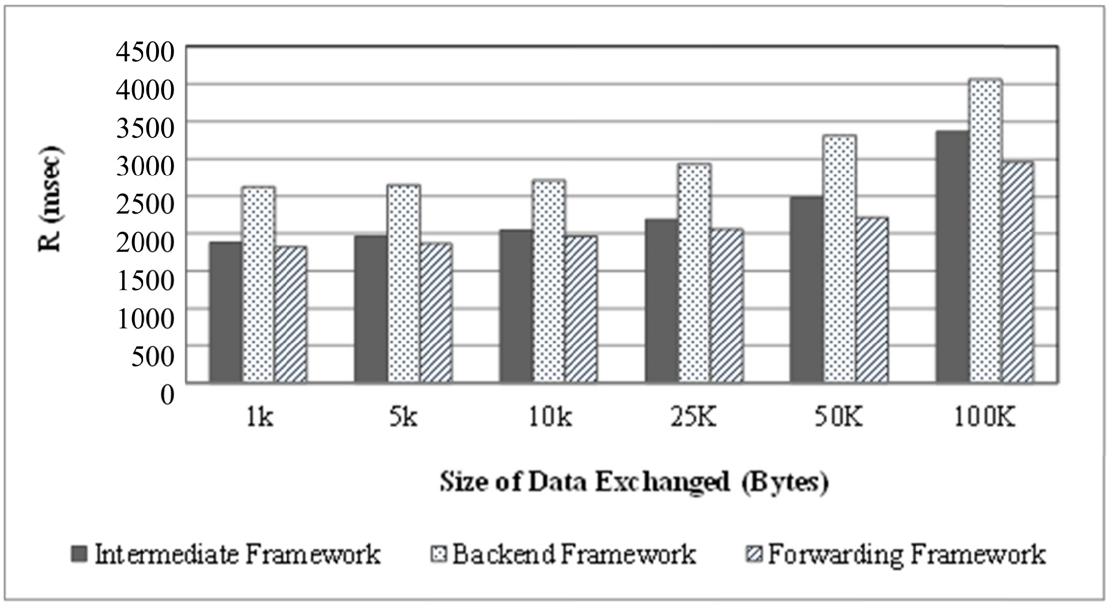
(b)

Figure A-1: The effect of the size of data exchanged between WS partitions on the performance of the three frameworks when the sample π Calculator WS with O = 50% is invoked by one WS client (a) with N = 1000 (b) with N = 100000

A.2 Effect of the Size of Data Exchanged between WS Partitions using six WS clients



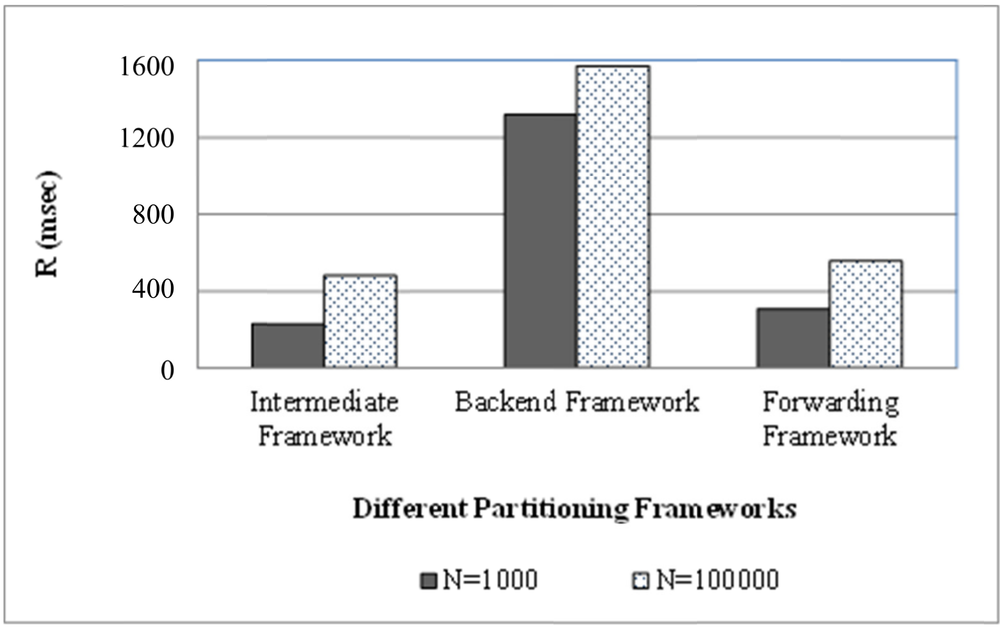
(a)



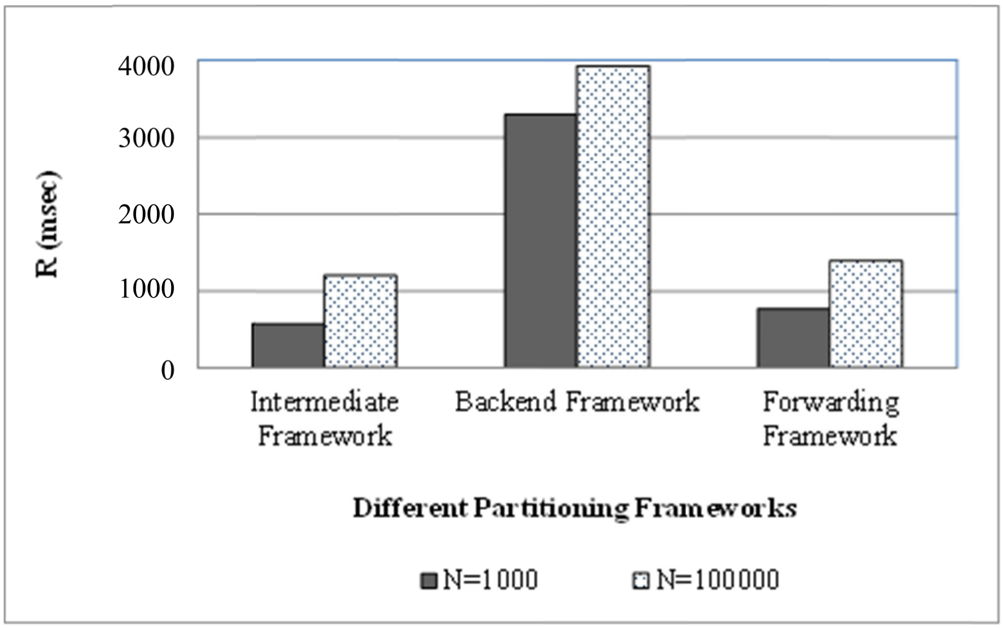
(b)

Figure A-2: The effect of the size of data exchanged between WS partitions on the performance of the three frameworks when the sample π Calculator WS with O = 50% is invoked by six concurrent WS clients (a) with N = 1000 (b) with N = 100000

A.3 Effect of Using a WS Security Standard with One and Six WS client



(a)



(b)

Figure A-3: Effect of using WS security standard on the relative performance of the three partitioning frameworks when the mobile device is operated at 624 MHz and the sample π Calculator WS with O = 50% is invoked by (a) one (b) six WS clients

A.4 Effect of the Speed of the Processing Resource using six WS clients

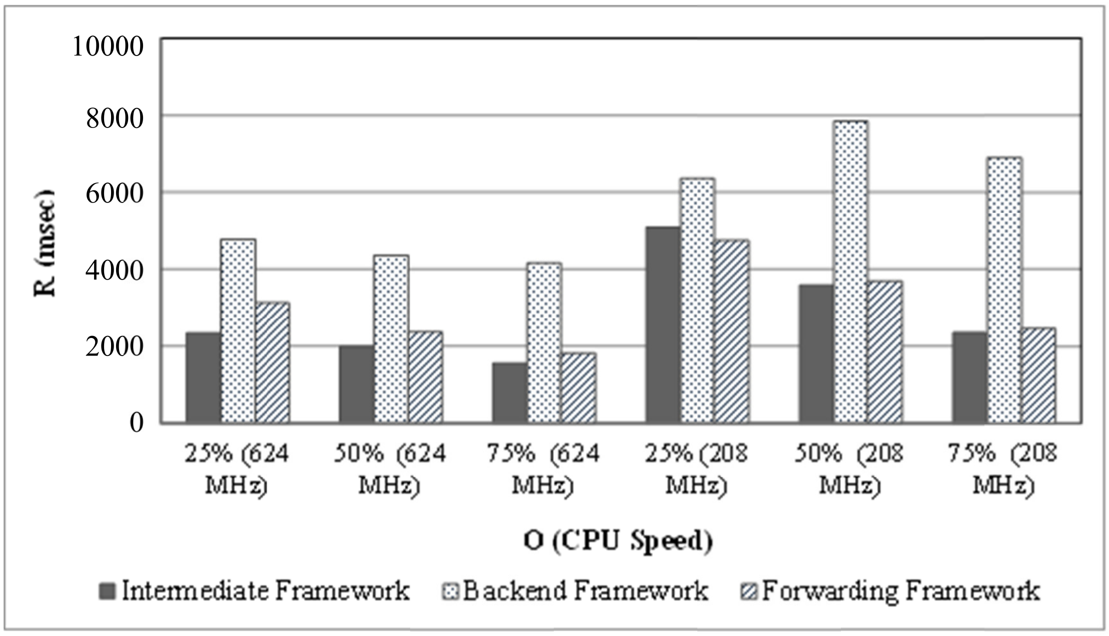
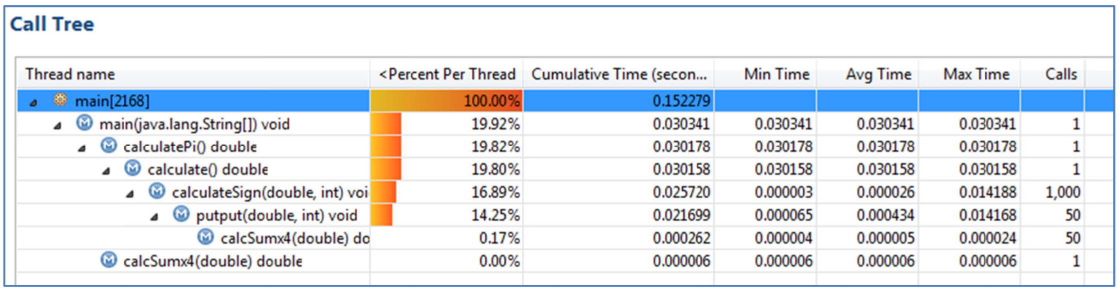


Figure A-4: Effect of CPU speed of the mobile device on relative performance of the three partitioning frameworks when the sample WS (with N = 1000000) is invoked by six concurrent WS clients

Appendix B: Profiling Output of a Few Sample Applications

The profiling outputs of the sample applications presented in this Appendix are a few of the applications which are investigated to study the call graph.

B.1 Call Tree of a π Calculator Application

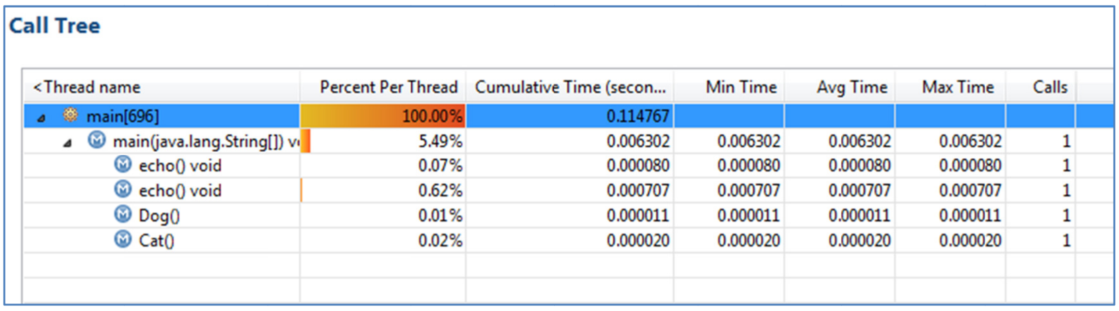


Call Tree

Thread name	<Percent Per Thread	Cumulative Time (secon...	Min Time	Avg Time	Max Time	Calls
main[2168]	100.00%	0.152279				
main(java.lang.String[]) void	19.92%	0.030341	0.030341	0.030341	0.030341	1
calculatePi() double	19.82%	0.030178	0.030178	0.030178	0.030178	1
calculate() double	19.80%	0.030158	0.030158	0.030158	0.030158	1
calculateSign(double, int) voi	16.89%	0.025720	0.000003	0.000026	0.014188	1,000
putput(double, int) void	14.25%	0.021699	0.000065	0.000434	0.014168	50
calcSumx4(double) do	0.17%	0.000262	0.000004	0.000005	0.000024	50
calcSumx4(double) double	0.00%	0.000006	0.000006	0.000006	0.000006	1

Figure B-1: Call tree achieved using eclipse TPTP profiling tool for π Calculator Application

B.2 Call Tree of an Echo Application using Two Classes (Cat and Dog)

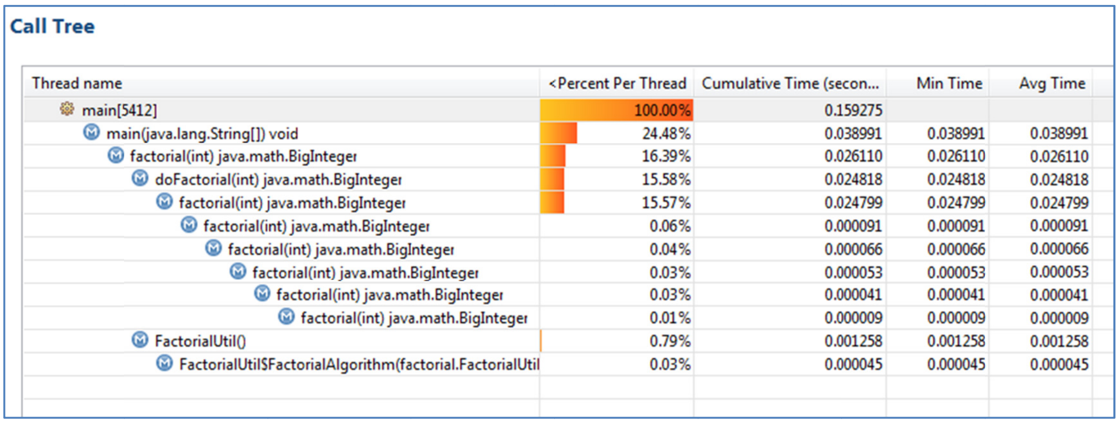


Call Tree

<Thread name	Percent Per Thread	Cumulative Time (secon...	Min Time	Avg Time	Max Time	Calls
main[696]	100.00%	0.114767				
main(java.lang.String[]) v	5.49%	0.006302	0.006302	0.006302	0.006302	1
echo() void	0.07%	0.000080	0.000080	0.000080	0.000080	1
echo() void	0.62%	0.000707	0.000707	0.000707	0.000707	1
Dog()	0.01%	0.000011	0.000011	0.000011	0.000011	1
Cat()	0.02%	0.000020	0.000020	0.000020	0.000020	1

Figure B-2: Call tree achieved using eclipse TPTP profiling tool for of an Echo application using two Classes

B.3 Call Tree for a Factorial Calculator Application

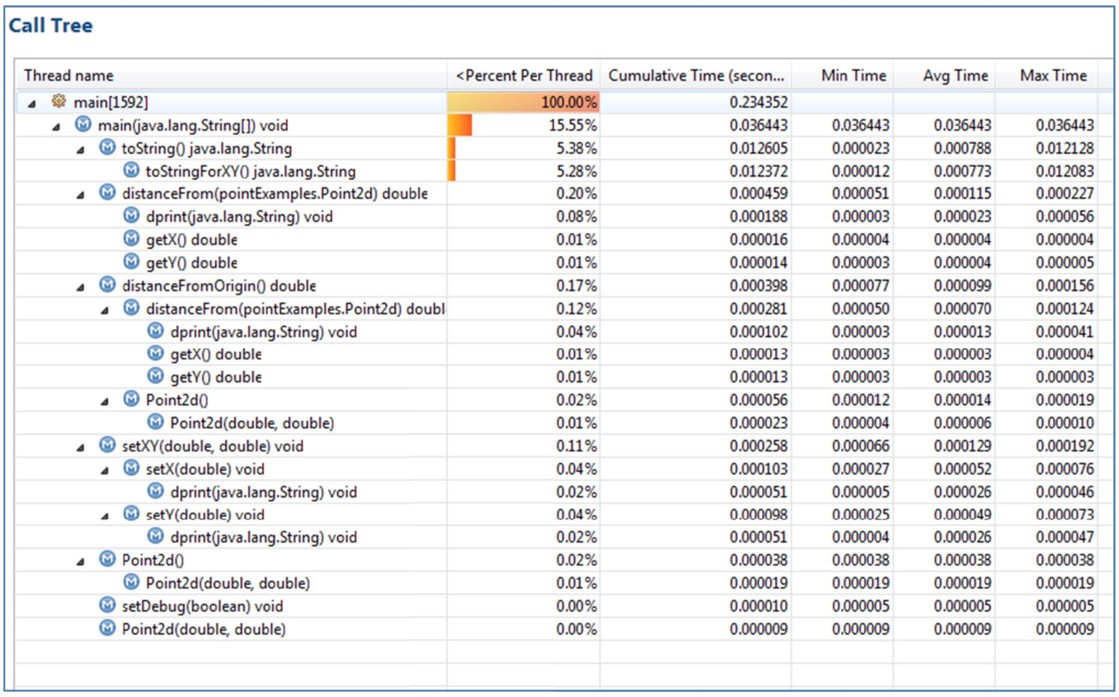


Call Tree

Thread name	<Percent Per Thread	Cumulative Time (secon...	Min Time	Avg Time
main[5412]	100.00%	0.159275		
main(java.lang.String[]) void	24.48%	0.038991	0.038991	0.038991
factorial(int) java.math.BigInteger	16.39%	0.026110	0.026110	0.026110
doFactorial(int) java.math.BigInteger	15.58%	0.024818	0.024818	0.024818
factorial(int) java.math.BigInteger	15.57%	0.024799	0.024799	0.024799
factorial(int) java.math.BigInteger	0.06%	0.000091	0.000091	0.000091
factorial(int) java.math.BigInteger	0.04%	0.000066	0.000066	0.000066
factorial(int) java.math.BigInteger	0.03%	0.000053	0.000053	0.000053
factorial(int) java.math.BigInteger	0.03%	0.000041	0.000041	0.000041
factorial(int) java.math.BigInteger	0.01%	0.000009	0.000009	0.000009
FactorialUtil()	0.79%	0.001258	0.001258	0.001258
FactorialUtil$FactorialAlgorithm(factorial.FactorialUtil	0.03%	0.000045	0.000045	0.000045

Figure B-3: Call tree achieved using eclipse TPTP profiling tool for a Factorial Calculator Application

B.4 Call Tree for a 2D Points Arithmetic Application



Call Tree

Thread name	<Percent Per Thread	Cumulative Time (secon...	Min Time	Avg Time	Max Time
main[1592]	100.00%	0.234352			
main(java.lang.String[]) void	15.55%	0.036443	0.036443	0.036443	0.036443
toString() java.lang.String	5.38%	0.012605	0.000023	0.000788	0.012128
toStringForXY() java.lang.String	5.28%	0.012372	0.000012	0.000773	0.012083
distanceFrom(pointExamples.Point2d) double	0.20%	0.000459	0.000051	0.000115	0.000227
dprint(java.lang.String) void	0.08%	0.000188	0.000003	0.000023	0.000056
getX() double	0.01%	0.000016	0.000004	0.000004	0.000004
getY() double	0.01%	0.000014	0.000003	0.000004	0.000005
distanceFromOrigin() double	0.17%	0.000398	0.000077	0.000099	0.000156
distanceFrom(pointExamples.Point2d) doubl	0.12%	0.000281	0.000050	0.000070	0.000124
dprint(java.lang.String) void	0.04%	0.000102	0.000003	0.000013	0.000041
getX() double	0.01%	0.000013	0.000003	0.000003	0.000004
getY() double	0.01%	0.000013	0.000003	0.000003	0.000003
Point2d()	0.02%	0.000056	0.000012	0.000014	0.000019
Point2d(double, double)	0.01%	0.000023	0.000004	0.000006	0.000010
setXY(double, double) void	0.11%	0.000258	0.000066	0.000129	0.000192
setX(double) void	0.04%	0.000103	0.000027	0.000052	0.000076
dprint(java.lang.String) void	0.02%	0.000051	0.000005	0.000026	0.000046
setY(double) void	0.04%	0.000098	0.000025	0.000049	0.000073
dprint(java.lang.String) void	0.02%	0.000051	0.000004	0.000026	0.000047
Point2d()	0.02%	0.000038	0.000038	0.000038	0.000038
Point2d(double, double)	0.01%	0.000019	0.000019	0.000019	0.000019
setDebug(boolean) void	0.00%	0.000010	0.000005	0.000005	0.000005
Point2d(double, double)	0.00%	0.000009	0.000009	0.000009	0.000009

Figure B-4: Call tree achieved using eclipse TPTP profiling tool for 2D Points Arithmetic Application

Appendix C: Real Time Profiling for Device Profile Index

C.1 Real Time Profiling for Device Profile Index with Graph Size = 4

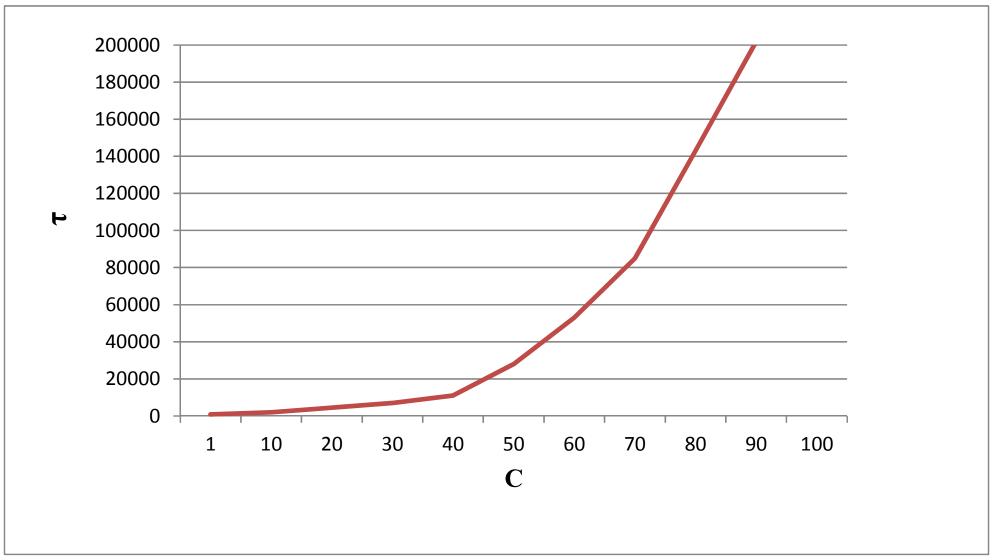


Figure C-1: Mean execution time measure by varying number of WS clients for a graph size = 4

C.2 Real Time Profiling for Device Profile Index with Graph Size = 8

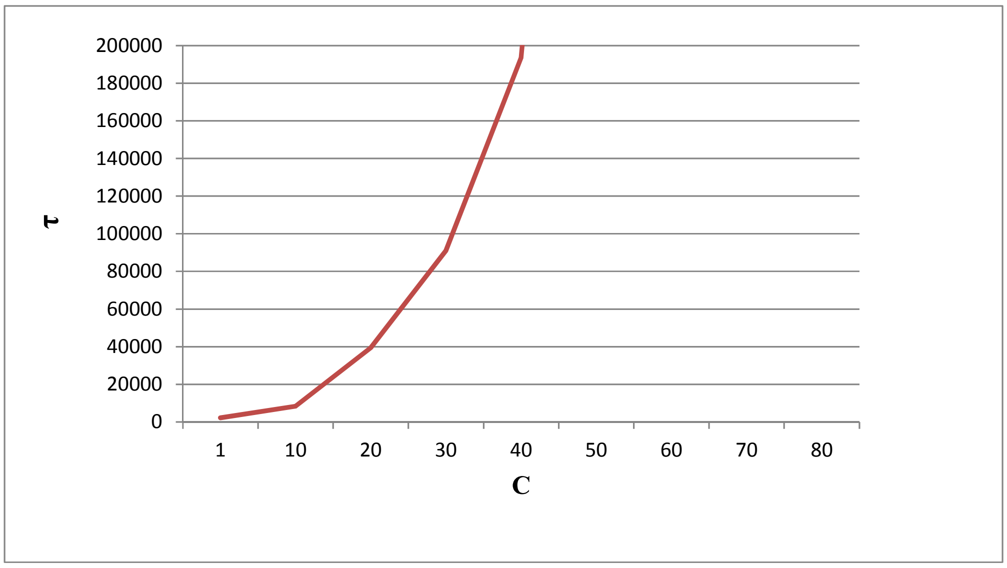


Figure C-2: Mean execution time measure by varying number of WS clients for a graph size = 8

C.3 Real Time Profiling for Device Profile Index with Graph Size = 12

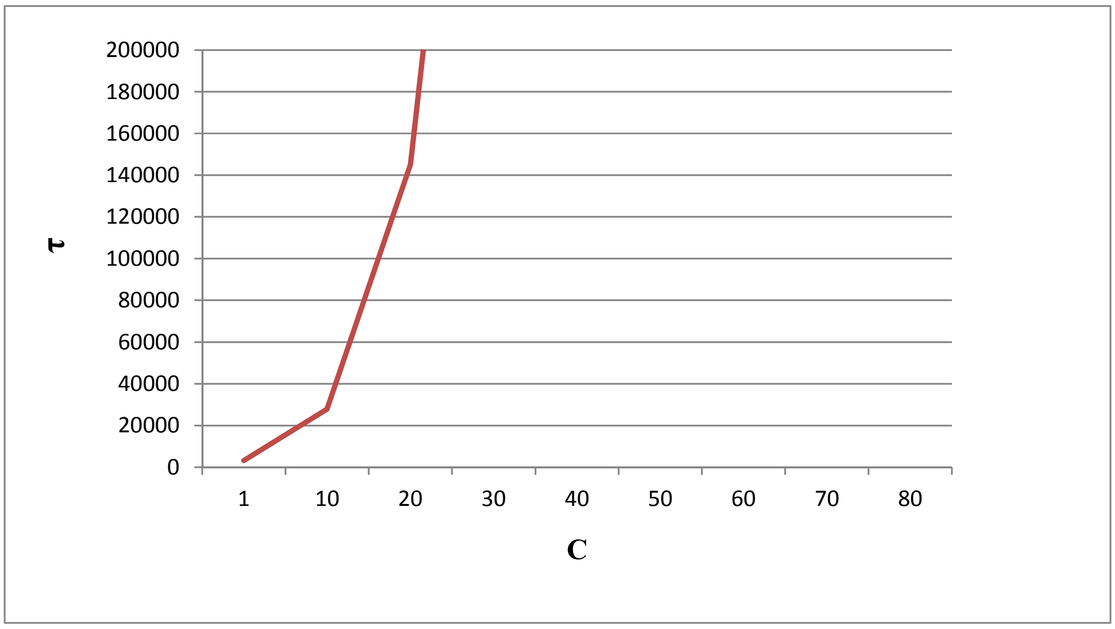


Figure C-3: Mean execution time measure by varying number of WS clients for a graph size = 12