Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

To request permissions please use the Feedback form on our webpage. http://researchspace.auckland.ac.nz/feedback

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form and Deposit Licence.
A Holistic Approach to Mobile Service Provisioning

Author: Andrew Meads
Supervisor: Dr. Ian Warren

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy
in the
Software Engineering Research Group (SERG)
Department of Computer Science

August 2015
In recent years, we have seen an explosion in the capabilities of smartphones, and an increase in the availability of wireless connectivity solutions. With this, smartphones have become capable of acting as providers of mobile services. A mobile service is similar to a service in the well-known Service-Oriented Architecture paradigm, but hosted on a roaming device as opposed to a machine within a fixed network.

Even with the large leaps in mobile technology in recent years, developers have traditionally been required to account for several challenges when creating mobile services. Such challenges include the reachability of services, the mobility and limited resources of devices, and the desire to support multiple mobile platforms. This thesis explores possible solutions to these and other challenges.

This thesis describes three main contributions. Firstly, we contribute Odin2, a middleware solution which allows developers to create mobile services in a way that addresses the challenges above, while being based on familiar technologies to ease the learning curve of both service and client developers. Odin2’s performance has been evaluated and has been shown to provide comparable performance to leading industry solutions. Furthermore, Odin2’s usefulness as a middleware has been shown through the implementation of several types of mobile service applications, along with a large real-world case study, REMOTE-CR. Secondly, we contribute OdinTools, a toolkit that incorporates Model-Driven Engineering techniques to allow developers to partially generate cross-platform service implementations. Finally, we contribute the Odin Test Harness. Originally developed to aid in the testing of Odin2 and REMOTE-CR, the harness is suitable for simulating networked mobile applications running on many concurrent devices, and, crucially, testing the behavior of those applications in dynamic operating environments.
Acknowledgements

Firstly, I would like to acknowledge my supervisor, Dr. Ian Warren, for his invaluable advice and guidance throughout the course of this thesis. His expertise has really helped me to successfully complete this work.

I would also like to thank my colleagues who have worked with me during the thesis: Thiranjith Weerasinghe for showing me the ropes and the initial collaborative effort on Odin, Adam Roughton and Kumar Akshay for their valuable work as summer students, Habib Naderi for his work on the test harness, Jonathan Rawstorn for his support during the REMOTE-CR project, and Alexandr Shirokov for his work on the REMOTE-CR web application. Each of them has been a pleasure to work with, and I would happily take the opportunity to work with them again.

Finally, a sincere thank you to my wonderful friends and family who have provided tremendous support and have helped to keep me sane over the years. Without their support this work would not have been possible.
Co-Authorship Form

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. Please include one copy of this form for each co-authored work. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

REMOTE-CR (Chapter 7)

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Everything except web application (plus some changes to web application)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>85</td>
</tr>
</tbody>
</table>

**CO-AUTHORS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alexandr Shirokov</td>
<td>Most of the web application</td>
</tr>
</tbody>
</table>

**Certification by Co-Authors**

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrew Meads</td>
<td></td>
<td>23/03/2015</td>
</tr>
<tr>
<td>Alexandr Shirokov</td>
<td></td>
<td>23/03/2015</td>
</tr>
</tbody>
</table>

Last updated: 25 March 2013
Co-Authorship Form

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. Please include one copy of this form for each co-authored work. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Odin Test Harness (Chapter 6)

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Requirements, smartphone implementation, evaluation, tool usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>70</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Habib Naderi</td>
<td>Implementation of Linux box</td>
</tr>
</tbody>
</table>

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrew Meads</td>
<td></td>
<td>23/03/2015</td>
</tr>
<tr>
<td>Habib Naderi</td>
<td></td>
<td>23/03/2015</td>
</tr>
</tbody>
</table>

Last updated: 25 March 2013
This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. Please include one copy of this form for each co-authored work. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Communication abstraction, interconnects, vertical handover, data integrity, some evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>50</td>
</tr>
</tbody>
</table>

**CO-AUTHORS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thiranjith Weerasinghe</td>
<td>Surrogate migration, context awareness, some evaluation</td>
</tr>
</tbody>
</table>

**Certification by Co-Authors**

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrew Meads</td>
<td></td>
<td>24/03/2015</td>
</tr>
<tr>
<td>Thiranjith Weerasinghe</td>
<td></td>
<td>24/03/2015</td>
</tr>
</tbody>
</table>
# Contents

Abstract iii

Acknowledgements v

List of Figures xix

List of Tables xxiii

1 Introduction 1
  1.1 Example Mobile Services 3
  1.2 Research Questions 5
  1.3 Research Contributions 7
  1.4 Thesis Outline 8

2 Literature Review 9
  2.1 Challenges with Mobile Service Provisioning 9
    2.1.1 Reachability 11
    2.1.2 Availability 13
    2.1.3 Scalability 13
    2.1.4 Testability 14
    2.1.5 Heterogeneity 14
  2.2 Mobile Service Provisioning Middleware 15
    2.2.1 Rover Toolkit 15
    2.2.2 Ice-E 16
    2.2.3 Soap ME 16
    2.2.4 Web Services with IP Multimedia Subsystem 16
    2.2.5 Mobile Host 17
    2.2.6 Nokia Mobile Web Server 17
    2.2.7 Jini-Based Solutions 18
  2.3 Addressing Reachability 19
    2.3.1 Mobile IP 19
    2.3.2 GTP 21
    2.3.3 Push Notification Services 21
    2.3.4 Discussion 23
  2.4 Addressing Availability 25
    2.4.1 Discussion 25
2.5 Addressing Scalability ........................................ 28
  2.5.1 Code Offload ........................................ 28
  2.5.2 Discussion ........................................ 29
2.6 Addressing Testability .................................. 31
  2.6.1 Discussion ........................................ 32
2.7 Addressing Heterogeneity ................................. 34
  2.7.1 Model-Driven Engineering ............................ 34
  2.7.2 Web-Based and Hybrid Mobile Development .......... 36
  2.7.3 Discussion ........................................ 38
2.8 Summary .................................................. 39

3 Odin: A Mobile Service Provisioning Middleware 43
  3.1 Requirements ........................................... 44
    3.1.1 Functional Requirements ............................ 44
    3.1.2 Non-Functional Requirements .................... 45
  3.2 Supporting Technologies ................................ 45
    3.2.1 Jini ................................................ 45
    3.2.2 Jini Surrogate Architecture ...................... 47
    3.2.3 Jini Extensible Remote Invocation (JERI) ........ 48
  3.3 Technological Overview ................................ 48
    3.3.1 Communication Protocol and Vertical Handover ... 50
    3.3.2 Surrogate Migration ................................ 53
    3.3.3 Context Awareness ................................ 56
  3.4 Evaluation .............................................. 57
    3.4.1 Initial Performance Evaluation .................... 58
      3.4.1.1 Interconnect Overheads ........................ 58
      3.4.1.2 Vertical Handover Delays ..................... 59
    3.4.2 Odin Applications ................................ 61
  3.5 Summary ................................................ 62

4 Odin2: A Robust, Cross-Platform Mobile Services Middleware 63
  4.1 Requirements and Features ............................. 64
    4.1.1 Functional Requirements ............................ 64
    4.1.2 Non-Functional Requirements .................... 65
    4.1.3 Features .......................................... 66
  4.2 Supporting Technologies ................................ 66
  4.3 Architectural Overview ................................ 68
    4.3.1 Shared Components ................................ 71
    4.3.2 Device Components ................................ 71
    4.3.3 Surrogate Host / Surrogate Components ........... 74
  4.4 Connection and Device Registration ..................... 78
    4.4.1 Initial Handshake ................................ 78
    4.4.2 Sequence Numbers and Synchronization ............ 79
  4.5 Messaging ............................................... 80
    4.5.1 Components Involved in Messaging ............... 83
  4.6 Error Handling .......................................... 88
  4.7 Cross-Platform Support ................................ 89
5 OdinTools: A Model-Driven Tool for Cross-Platform Development of Odin2-based Applications
  5.1 Design
    5.1.1 Model Design
      5.1.1.1 Overview
      5.1.1.2 Odin Application Model
      5.1.1.3 Platform-Independent Model
      5.1.1.4 Platform-Specific Models
      5.1.1.5 Manual Completion
    5.1.2 Tool Design
  5.2 Service Development with OdinTools
    5.2.1 Use of OdinTools
    5.2.2 Generated Code
    5.2.3 Completing the Application
  5.3 Summary

6 Odin Test Harness
  6.1 Requirements
  6.2 Design
    6.2.1 Test Harness Software
    6.2.2 Test Harness VM Software
    6.2.3 App Software
  6.3 Test Harness API
    6.3.1 Host Script Library
    6.3.2 VM Scripts
    6.3.3 Test Harness Application Library
  6.4 Example Test Scripts
  6.5 Test Harness Evaluation
  6.6 Summary

7 REMOTE-CR: An Odin2-enabled Telemed System for Cardiac Rehabilitation
  7.1 Background
  7.2 Requirements
    7.2.1 Functional Requirements
    7.2.2 Non-Functional Requirements
  7.3 Design Overview
    7.3.1 Mobile Application Design
    7.3.2 Mobile Application User Interface
    7.3.3 Web App and Surrogate Design
    7.3.4 Web App User Interface
  7.4 Leveraging the Odin2 Middleware
  7.5 Summary

8 Evaluation
  8.1 Performance Evaluation
8.1.1 Experimental Setup ............................................ 158
8.1.2 Results: Bandwidth Consumption and Responsiveness .... 161
8.1.3 Results: Throughput ........................................... 164
8.1.4 Results: Reliability ............................................ 165
8.1.5 Results: Battery Consumption ................................. 167
8.2 Odin2 Expressiveness .............................................. 169
  8.2.1 Mobile Media Service ........................................ 170
  8.2.2 Patient Monitoring Service ................................. 171
  8.2.3 Social Networking Service ................................. 172
  8.2.4 Discussion .................................................. 172
8.3 Productivity with OdinTools ...................................... 172
  8.3.1 Mobile Media Service Definition ....................... 172
  8.3.2 Patient Monitoring Service Definition ................... 174
  8.3.3 Social Networking Service Definition ................... 175
  8.3.4 Generated Code Evaluation ................................ 176
  8.3.5 Discussion .................................................. 178
8.4 Test Harness Evaluation ........................................... 178
  8.4.1 Odin2 Tests .................................................. 179
  8.4.2 REMOTE-CR Tests ........................................... 181
  8.4.3 Discussion .................................................. 184
8.5 Real-World Usage ................................................ 185
  8.5.1 REMOTE-CR Trial Setup ................................ 185
  8.5.2 Bug Reports ................................................ 186
  8.5.3 REMOTE-CR Feedback ...................................... 187
  8.5.4 Discussion .................................................. 188
8.6 Summary ......................................................... 189

9 Conclusions and Future Work .................................. 191
  9.1 Research Contributions ...................................... 191
    9.1.1 Literature review of mobile service provisioning, cross-platform
development, and mobile application testing techniques .......... 192
    9.1.2 Odin2 Mobile Service Provisioning Middleware .......... 193
    9.1.3 Model-Driven Service Development with OdinTools .... 195
    9.1.4 Odin Test Harness ......................................... 196
    9.1.5 REMOTE-CR Case Study .................................. 196
  9.2 Answers to Research Questions ................................ 197
    9.2.1 RQ #1: What are the challenges inherent in mobile service provisioning? To what degree does the existing body of work address these challenges? .......... 198
    9.2.2 RQ #2: What technologies and techniques are used commercially for mobile service development, if any? ........................................... 198
    9.2.3 RQ #3: What are the existing approaches to cross-platform mobile development, and cross-platform development in general? .... 198
    9.2.4 RQ #4: How can we support more rapid application development of mobile services? ........................................... 199
    9.2.5 RQ #5: How can we sufficiently test our mobile service provisioning solution and associated apps to ensure they meet their requirements? 199
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.2.6</td>
<td>RQ #6: Is our solution dependable enough to be used in a real-world scenario?</td>
<td>199</td>
</tr>
<tr>
<td>9.3</td>
<td>Future Work</td>
<td>200</td>
</tr>
<tr>
<td>9.3.1</td>
<td>With Odin2</td>
<td>200</td>
</tr>
<tr>
<td>9.3.2</td>
<td>With OdinTools</td>
<td>200</td>
</tr>
<tr>
<td>9.3.3</td>
<td>With the Test Harness</td>
<td>201</td>
</tr>
<tr>
<td>9.3.4</td>
<td>In Industry</td>
<td>201</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Service Development with Odin2</td>
<td>203</td>
</tr>
<tr>
<td>A.1</td>
<td>Surrogate</td>
<td>204</td>
</tr>
<tr>
<td>A.2</td>
<td>Android</td>
<td>208</td>
</tr>
<tr>
<td>A.3</td>
<td>Windows Phone</td>
<td>213</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Detailed Requirements and Metrics Explanation for REMOTE-CR</td>
<td>217</td>
</tr>
<tr>
<td>B.1</td>
<td>Requirements</td>
<td>217</td>
</tr>
<tr>
<td>B.1.1</td>
<td>Requirements: Profile</td>
<td>217</td>
</tr>
<tr>
<td>B.1.2</td>
<td>Requirements: Monitoring</td>
<td>218</td>
</tr>
<tr>
<td>B.1.3</td>
<td>Requirements: Routes</td>
<td>221</td>
</tr>
<tr>
<td>B.1.4</td>
<td>Requirements: Goals</td>
<td>221</td>
</tr>
<tr>
<td>B.1.5</td>
<td>Requirements: Review</td>
<td>222</td>
</tr>
<tr>
<td>B.2</td>
<td>Calculated Performance Metrics in REMOTE-CR</td>
<td>224</td>
</tr>
<tr>
<td>B.2.1</td>
<td>Percent of Heart Rate Reserve</td>
<td>224</td>
</tr>
<tr>
<td>B.2.2</td>
<td>Training Load</td>
<td>224</td>
</tr>
</tbody>
</table>

Bibliography                                                                 | 227  |
List of Figures

1.1 High-level overview of a mobile service. Clients must receive the illusion of a static, well-known service endpoint while a host is moving amongst heterogeneous networks. .................................................. 2
1.2 A mobile media service, demonstrating the need for two-way communication, publish-subscribe semantics, caching, and high bandwidth in a solution. ................................................................. 3
1.3 A patient monitoring system, demonstrating the need for streaming communication and external sensor support in a solution. ......................... 4
1.4 A social networking service, demonstrating the need for peer-to-peer communication support in a solution. .............................. 5

2.1 General Service-Oriented Architecture (SOA) paradigm (adapted from [1]) 10
2.2 Interruption of communication pathways due to device mobility in traditional wireless networks ............................................................... 12
2.3 Interruption of communication pathways due to MNO firewalls in traditional 3G networks ................................................................. 12
2.4 Classification of middleware-based mobile service provisioning solutions ................................................. 15
2.5 Overview of Mobile Host in a MNO-supported JXTA environment. Adapted from [2] .......................... 17
2.6 Overview of Mobile Web Server ......................................................... 18
2.7 Overview of Mobile IP Protocol .......................................................... 20
2.8 Essential Architecture for Push Notifications ........................................ 21
2.9 Overview of different communication paradigms using Push Notifications 22
2.10 High-level overview of the Model Driven Architecture (MDA) specification 35
2.11 High-level overview PhoneGap architecture ........................................... 37
2.12 High-level overview of Titanium architecture ......................................... 38

3.1 Overview of the Jini Service Oriented Architecture .......................... 46
3.2 Overview of Jini Surrogate Architecture (JSA) .................................. 47
3.3 High-level overview of Odin’s architecture ......................................... 49
3.4 Classes involved in Odin’s interconnect layer, important for its vertical handover mechanism ...................................................... 51
3.5 Components involved in Odin’s vertical handover mechanism ............... 52
3.6 Sequence diagram showing a reactive vertical handover in response to a network failure .............................................................. 54
3.7 Steps involved with surrogate migration in Odin. Adapted from [3] ............ 55
3.8 Overheads with Odin interconnect channels with respect to direct communication over the same network type ................................. 60
3.9 Overheads with Odin interconnect channels with respect to direct com-
munication over the same network type .......................... 60

4.1 High-level overview of communication between different clients and Odin2
service hosts .............................................................. 69
4.2 Overview of Odin2 middleware layers within devices and surrogate hosts .... 70
4.3 Key components shared between the Device and Surrogate Host ................. 72
4.4 Key components located on the device .................................. 73
4.5 Key components located on the Surrogate Host involved with surrogate-
device communication ................................................. 75
4.6 Key components located on the Surrogate Host involved with surrogate-
client communication ................................................. 77
4.7 Sequence diagram showing a device connecting to the surrogate host for
the first time ............................................................... 78
4.8 Sequence diagram showing a high-level view of request-response messaging
between device and surrogate under ideal circumstances ..................... 81
4.9 Overview of Odin2 Messaging and Interconnect layers ....................... 83

5.1 Overview of OdinTools Models and Transformations. Bold writing to the
right-hand side indicates the corresponding MDA model levels. ............... 95
5.2 Overview of OdinTools’ Odin Application Model (OAM) ....................... 96
5.3 Overview of OdinTools PIM. Compared with the OAM, red indicates
removals, while green indicates changes or additions ........................ 99
5.4 A partial screenshot of OdinTools’ Odin Application Model (OAM), as
implemented with the Visual Studio Modeling SDK 2013 edition ............ 105
5.5 A view of OdinTools running as a plugin to Visual Studio 2013 .............. 106
5.6 Adding a CustomType shape to an OdinTools diagram ....................... 106
5.7 Adding a Database entry to an OdinTools diagram .......................... 107
5.8 Adding AsynchronousMessage entries to an OdinTools diagram ........... 108

6.1 Setup of a single Odin Test Harness machine ............................... 113
6.2 Software components within a test harness machine ....................... 114
6.3 Software components within each VM on the test harness machine .......... 115
6.4 Overview of key classes in the Test Harness Application Library .......... 123

7.1 Overview of REMOTE-CR Architecture .................................. 136
7.2 Components and interactions within the REMOTE-CR mobile application
(does not show interactions between components within the UI layer) .... 138
7.3 Overview of REMOTE-CR mobile application flow. The arrows represent
possible navigation options, which are explained in section 7.3.2 ............ 140
7.4 Screenshots for the REMOTE-CR mobile application (1 of 3) ............. 141
7.5 Screenshots for the REMOTE-CR mobile application (2 of 3) ............. 142
7.6 Screenshots for the REMOTE-CR mobile application (3 of 3) ............. 144
7.7 Overview of components withing the REMOTE-CR web app and surrogate,
showing the connections between them and the utilization of the Odin2
middleware ............................................................... 148
7.8 REMOTE-CR web app login panel ...................................... 149
7.9 REMOTE-CR web app participant list page .............................. 149
7.10 REMOTE-CR web app page allowing clinicians to add new participants . 149
<table>
<thead>
<tr>
<th>List of Figures</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.11 REMOTE-CR web app summary page for a particular participant</td>
</tr>
<tr>
<td>7.12 REMOTE-CR web app participant detail pages</td>
</tr>
<tr>
<td>7.13 REMOTE-CR web app live monitoring session in-progress with four active participants</td>
</tr>
<tr>
<td>7.14 REMOTE-CR web app additional windows shown during live monitoring sessions</td>
</tr>
<tr>
<td>8.1 Average protocol bandwidth usage per 256-byte message sent</td>
</tr>
<tr>
<td>8.2 Average protocol bandwidth efficiency</td>
</tr>
<tr>
<td>8.3 Average protocol responsiveness over Wi-Fi</td>
</tr>
<tr>
<td>8.4 Average protocol responsiveness over 3G</td>
</tr>
<tr>
<td>8.5 Average reliable message throughput for 5,000 messages over Wi-Fi</td>
</tr>
<tr>
<td>8.6 Percentage of successfully delivered messages during induced network disruption over Wi-Fi</td>
</tr>
<tr>
<td>8.7 Vertical handover delays for Odin2</td>
</tr>
<tr>
<td>8.8 Remaining battery percentage after 24 hours over Wi-Fi</td>
</tr>
<tr>
<td>8.9 Energy efficiency index for each protocol over 3G</td>
</tr>
<tr>
<td>8.10 Screenshots for each Odin2 example service</td>
</tr>
<tr>
<td>8.11 Mobile media service client, accessed from Internet Explorer</td>
</tr>
<tr>
<td>8.12 OdinTools OAM definition for the mobile media service</td>
</tr>
<tr>
<td>8.13 OdinTools OAM definition for the patient monitoring service</td>
</tr>
<tr>
<td>8.14 OdinTools OAM definition for the social network service</td>
</tr>
<tr>
<td>8.15 Breakdown of generated and manual code using OdinTools</td>
</tr>
<tr>
<td>A.1 Overview of example messenger service</td>
</tr>
<tr>
<td>A.2 User interface for simple messenger app</td>
</tr>
</tbody>
</table>
# List of Tables

1.1 Summary of the research contributions arising from this thesis. . . . . . . 7  
2.1 Summary of how the identified works address reachability . . . . . . . . . 23  
2.2 Summary of how the identified works address availability . . . . . . . . . 25  
2.3 Summary of how the identified works address scalability . . . . . . . . . 30  
3.1 Summary of contributors to Odin features . . . . . . . . . . . . . . . . . . 44  
3.2 Context information used by Odin . . . . . . . . . . . . . . . . . . . . . . . 57  
3.3 Overheads introduced by Odin over HTTP . . . . . . . . . . . . . . . . . . . 59  
3.4 Overheads introduced by Odin over Bluetooth . . . . . . . . . . . . . . . . . 59  
4.1 Summary of how Odin2’s design meets its functional requirements . . . . 91  
4.2 Summary of how Odin2’s design meets its non-functional requirements . . 92  
5.1 Comparison of generated components with those that must be manually  
    implemented on each platform . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104  
5.2 Summary of generated code for the simple social network service . . . . . 109  
6.1 Summary of how the Odin Test Harness’s design satisfies its functional  
    requirements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128  
8.1 Threats to validity arising from heterogeneity, and their countermeasures. 159  
8.2 Threats to validity arising from experimental procedure, and their counter- 
    measures. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 160  
8.3 Summary of Odin2 sample apps and how Odin2 enables them. . . . . . . . 173
Chapter 1

Introduction

In recent years, we have seen an explosion in the capabilities of smartphones, and an increase in the availability of wireless connectivity solutions such as 3G, 4G, and Wi-Fi. With this, smartphones have become capable of running applications comparable in functionality to their traditional fixed-network counterparts.

A particular class of application, that smartphones, until recent years, have not had the computing power to run is that of services. As well as acting as capable consumers of the large numbers of web services and other available online resources, smartphones today, with their improved CPU, memory, and storage capacities, are more than capable of acting as providers for clients anywhere on the web. In other words, today’s smartphones can take on the role of hosts in the traditional Service-Oriented Architecture (SOA) paradigm [1]. In this thesis, we dub applications of this class, mobile services.

Figure 1.1 shows a high-level overview of a mobile service. A client, anywhere on the internet (or other network), may initiate communication with a mobile service via some form of network protocol, using a static address. This address could be well-known by the client, or could be obtained by communicating with a service registry, itself with a well-known address.

Even with today’s high-powered smartphones and the availability of convenience APIs from companies such as Microsoft and Google, there are key challenges which developers must consider when building mobile services. These challenges can be classified into one of the following categories:

1. Reachability refers to the problem of allowing clients to discover and consume services in a mobile networking environment, regardless of their location. Compared with traditional fixed networks and powerful machines, service developers must deal
with mobile endpoints in networks run by Mobile Network Operators (MNOs) that may place restrictions on the type of traffic allowed within those networks.

2. **Availability** refers to the problem of allowing services to remain available for consumption by clients *whenever necessary* (within reason). Device and network resource constraints, such as limited battery life, low bandwidth, and intermittent connectivity inherent in some mobile networks, make this a non-trivial matter for mobile service developers.

3. **Scalability** refers to the ability of a service to gracefully handle increases in traffic. Mobile device and network resources, despite being exponentially more powerful than in the past, still must be carefully managed in order to facilitate this.

4. **Heterogeneity** refers to the requirement of some mobile applications to run in a variety of different network types and on multiple mobile platforms. Typically this results in developers having to write a lot of extra code.

While developers can manually implement solutions to these concerns within their application, there is limited support today in terms of a holistic solution to mobile service provisioning which relieves developers from these concerns as much as possible. In this thesis, we explore the research, development, and testing performed in order to provide such a solution, in the hope that future mobile service developers will be able to write novel, cross-platform mobile service applications as easily as any other kind of mobile app.
The remainder of this chapter is organized as follows. Section 1.1 introduces distinct mobile service applications, with varying requirements and feature sets, which are important to consider when developing a mobile service provisioning solution. In section 1.2 we formulate the research questions which have guided this research. We then list our contributions in section 1.3, prior to outlining the remainder of the thesis in section 1.4.

### 1.1 Example Mobile Services

When considering a potential mobile service provisioning solution, there are several classes of applications that should be considered. Certainly, a traditional SOA-style application in which a client makes requests of a mobile host should be encompassed. However there are other communication paradigms and features specific to mobile devices which should be supported. In this section we introduce three examples of mobile services, each one necessitating certain features. Using an evolving frameworks approach \[4\], we can then draw the requirements for a solution which supports a diverse set of applications. The requirements elicitation and resulting middleware is discussed in chapters 3, 4, and 5.

Figure 1.2 shows a mobile media service, where clients may request the latest photos or videos from a mobile host, and / or take photos or videos in real-time. While this is the closest service we consider to the traditional client - server architecture, there are still things to consider. Firstly, a useful feature for this kind of service would be for clients to subscribe such that they have certain media delivered to them as it is created. This necessitates that, not only must clients be able to locate mobile services, but these services must be able to call back to clients - even if those clients themselves are mobile. Furthermore, media - especially video - traditionally requires high bandwidth compared to other forms of data. Any solution claiming to support such services should offer a method to avoid inundating a host’s mobile network connection. For example, a cache sitting between a client and host could host some media itself, reducing the need for hosts to send popular items multiple times.
Figure 1.3: A patient monitoring system, demonstrating the need for streaming communication and external sensor support in a solution.

Figure 1.3 shows a healthcare service, intended to reduce the need for patients with particular medical conditions to frequently visit hospital. Using the service, patients wear a Bluetooth-enabled vital sign sensor or sensors, which communicates with the patient’s smartphone. The information acquired by the sensors, along with the patient’s location and other important information, is streamed to a central location accessible via doctors in real-time. Any anomalies in the sensor data detected by the sensor itself, the smartphone, or the doctor can be relayed to ambulance crews who can be guided to the patient’s location using the patient’s location information. With this service, we can see the need for streaming support in a mobile service provisioning solution. Necessarily, the solution must prevent data loss or duplication when the stream is interrupted, and allow it to be restored automatically without further developer intervention. This is of particular importance when designing medical applications with potential lives at stake such as this example, where every piece of data sent could be of vital importance. Furthermore, we can see that any solution must not interfere with a device’s ability to communicate with external sensors and other devices over Bluetooth or other local network schemes.

Figure 1.4 shows a social networking service, in which users can broadcast their location and other desired status updates to all, or a subset of, other users of the service. In addition, two-way communication between any two users is required for features such as messaging. Here we can see that peer-to-peer support is required of a mobile service provisioning solution, in addition to traditional client-server communication.

To be widely adoptable by a range of mobile service scenarios, we believe a solution must allow the development of applications with characteristics represented by the above three applications. Additionally, the solution should assist developers with the cross-platform development of these applications.
Chapter 1. *Introduction*

**Figure 1.4:** A social networking service, demonstrating the need for peer-to-peer communication support in a solution.

### 1.2 Research Questions

The aim of the research conducted in this thesis has been to research, develop, implement, and evaluate a mobile service provisioning solution which allows developers to create novel applications including but not limited to those described in section 1.1. The solution, *Odin*, and its associated cross-platform tool, *OdinTools*, aims to allow developers to create these types of applications without concern for the challenges presented in this chapter (and expanded upon in Chapter 2).

Specifically, we have formulated the following set of research questions, which have guided and focused the direction of research over the course of the thesis:

**RQ #1:** What are the challenges inherent in mobile service provisioning? To what degree does the existing body of work address these challenges?

**RQ #2:** What technologies and techniques are used *commercially* for mobile service-type development, if any?

**RQ #3:** What are the existing approaches to cross-platform mobile development, and cross-platform development in general?

**RQ #4:** How can we support more rapid application development of mobile services?

**RQ #5:** How can we sufficiently test our mobile service provisioning solution and associated apps to ensure they meet their requirements?

**RQ #6:** Is our solution dependable enough to be used in real-world scenarios?
The justification for this set of research questions is as follows:

1. It is important to understand the challenges one is trying to address, as missing any important challenges will lead to a solution which is less widely applicable than it could have been. A good understanding of the existing body of work is useful so we may learn by example, and avoid duplication of work where possible.

2. We are interested in the commercial solutions specifically because these are the ones which have been demonstrated to be useful. Evaluating our solution against widely adopted solutions will show the potential for widespread use.

3. With the diverse nature of today’s smartphones, a widely adopted solution will ideally provide some degree of cross-platform support. We will investigate existing techniques with the aim of further developing them in the context of heterogeneous mobile services.

4. Having a functional solution will be of little use if service development has a high learning curve and / or takes a significant proportion of developer effort over and above that of traditional service development.

5. It is necessary to understand how mobile services behave in the presence of a dynamic operating environment. This is especially true of critical services such as healthcare applications.

6. We wish to provide a solution that is useful outside a laboratory context. To this end we wish to determine our solution’s applicability to developing real applications.
1.3 Research Contributions

To answer the research questions in section 1.2, we have contributed to the body of research as shown in table 1.1. Some of this work has been included in publications, while other aspects are as yet to-be-published.

<table>
<thead>
<tr>
<th>Contribution</th>
<th>Publications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completed a literature review into the extent of current mobile service provisioning, cross-platform development, and relevant mobile application testing techniques.</td>
<td>[5], [6], [7], [8]</td>
</tr>
<tr>
<td>In collaboration with Thiranjith Weerasinghe, designed, developed, and evaluated an initial prototype, mobile service provisioning <em>middleware</em>, dubbed <em>Odin</em>. Odin provides a solution to many of the challenges inherent in mobile service provisioning.</td>
<td></td>
</tr>
<tr>
<td>Redesigned Odin, incorporating modern frameworks and paradigms such as Spring, Hibernate and Web Services to create Odin’s second prototype, dubbed <em>Odin2</em>. Compared with Odin, Odin2 offers significantly increased performance, allows for more rapid application development, and addresses heterogeneity.</td>
<td>[9], [10], [11]</td>
</tr>
<tr>
<td>Investigated the use of <em>model-driven engineering</em> to promote the rapid, cross-platform development of mobile services. Created and evaluated an exploratory prototype to assist service developers with this, dubbed <em>OdinTools</em>.</td>
<td></td>
</tr>
<tr>
<td>Investigated several commercially available providers of <em>push notification communication services</em> - which aim to address mobility and reachability concerns - and evaluated their performance with respect to each other and Odin2.</td>
<td>[12]</td>
</tr>
<tr>
<td>Developed a tool to aid in the automated testing of Odin2-based applications - though the tool could potentially be used to test any networked mobile application. The tool, dubbed the <em>Odin Test Harness</em>, allows developers to simulate multiple devices and changes in network state in a controlled, repeatable manner.</td>
<td></td>
</tr>
<tr>
<td>In collaboration with the University of Auckland’s <em>School of Population Health</em>, developed and tested a cardiac patient monitoring and exercise service, named <em>REMOTE-CR</em>. This shows the potential for Odin2 to be adopted and used successfully for dependable real-world applications.</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1: Summary of the research contributions arising from this thesis.
Chapter 1. Introduction

1.4 Thesis Outline

This thesis is organized as follows. In Chapter 2, we present a literature review into the extent of current mobile service provisioning, cross-platform development, and relevant software testing techniques. This chapter contains all literature review work completed at all stages of the thesis.

Chapter 3 presents our initial work on Odin, our mobile service provisioning middleware. An overview of the entire solution is provided for reference, but we focus on work undertaken as part of this thesis (many parts of this work were completed as part of the Masters thesis of Thiranjith Weerasinghe [3]). In Chapter 4 we discuss some of Odin’s shortcomings. Following this, we present the design and implementation of our current mobile service middleware implementation, Odin2, which offers increased performance, more rapid service development, and cross-platform API support, when compared with Odin.

Chapter 5 presents OdinTools, which aims to assist developers in creating cross-platform Odin-based applications. In this chapter we show how the tool was designed, is used, and can assist in developing one of our example services (see section 1.1).

Chapter 6 presents the Odin Test Harness, which was designed to enable us to more accurately test a mobile application’s response to changes in network connectivity. We then present our primary case study in Chapter 7. This chapter details the design and operation of REMOTE-CR - a real-world service providing a patient monitoring and exercise service for patients undergoing cardiac rehabilitation.

Chapter 8 presents our evaluation of Odin2 and OdinTools. Here, we show, drawing upon data from chapters 4 through 7, that Odin2 serves as a sufficient mobile service provisioning framework in terms of performance, reliability, and features, while OdinTools provides support to developers wishing to create cross-platform mobile services.

Finally, Chapter 9 concludes the thesis. Here, we draw upon the evaluation to show how our research questions have been answered, and provide suggestions for future research work, both with mobile service provisioning and cross-platform support.
Chapter 2

Literature Review

There are several challenges inherent in any mobile service provisioning solution. It is important to understand those challenges so that they may be appropriately addressed. This chapter presents a literature review of many relevant tools and techniques for addressing, to various extents, the challenges inherent in mobile service provisioning.

The remainder of the chapter is organized as follows. Initially in section 2.1 we identify the key mobile service provisioning challenges, and explore why these are important to address. Section 2.2 then introduces some of the existing mobile service provisioning solutions in the literature. Sections 2.3 through 2.7 then identify tools and techniques from the existing body of work which seek to address the identified challenges, and we explore the extent to which these solutions address the challenges. Finally we conclude in section 2.8 with remarks on how the literature review has guided our own development.

2.1 Challenges with Mobile Service Provisioning

Service provisioning involves the discovery and consumption of services by clients over a distributed network, as shown in figure 2.1 [1]. Traditionally, these services are hosted on machines in fixed networks, with abundant resources in terms of processing power, memory, and bandwidth [13]. Despite numerous advances in mobile computing technology in recent years, the computing resources of today’s most powerful smartphone pale in comparison to a modest desktop PC. Furthermore, they are often connected to networks offering significantly less bandwidth, and with a higher degree of failure compared with today’s high-speed, always-on broadband networks. These limitations mean that the resources of a mobile device wishing to host a service (a host device) should be carefully managed in order to provide consistently high Quality of Service (QoS) levels. Specifically,
any mobile service provisioning solution must account for the two characteristics of mobile devices that differ from traditional PCs: comparatively limited resources and mobility. These resource and mobility related issues have an effect on the following key areas associated with the delivery of mobile services:

1. **Reachability** refers to the problem of allowing clients to discover and consume services in a mobile networking environment, regardless of their location. Compared with traditional fixed networks and powerful machines, service developers must deal with mobile endpoints in networks run by Mobile Network Operators (MNOs) that may place restrictions on the type of traffic allowed within those networks.

2. **Availability** refers to the problem of allowing services to remain available for consumption by clients *whenever necessary* (within reason). Device and network resource constraints make this a non-trivial matter for mobile service developers.

3. **Scalability** refers to the ability of a system to gracefully handle increases in traffic. The limited resources of mobile devices must be carefully managed in order to facilitate this.

Reachability and availability issues are often caused by a device’s mobility, while the limited resources available to a host device affect the scalability of any hosted services. Sections 2.1.1 - 2.1.3 describe these challenges in detail.

In addition to these issues, two further challenges have been identified. The first is that of **testability**. In general, this refers to a software system’s ability to be sufficiently tested,
to determine whether it reliably meets its functional and non-functional requirements. The second is that of **heterogeneity**. This refers to the wide variety of mobile devices in use today. A developer wishing to widely support these devices will need to consider differences in device capabilities and alternative programming models. Sections 2.1.4 and 2.1.5 further detail these challenges.

### 2.1.1 Reachability

In order to consume services, clients must be able to discover and connect to service providers [1, 14]. This means that service providers (in this case, host devices) must be discoverable, and clients must be able to send requests and receive responses from those providers. Within fixed networks, this is often a trivial problem. However, the mobile networks in which host devices often operate introduce a number of issues which must be overcome. This is known as the **reachability problem** [15]. The reachability problem comprises two sub-problems: **addressability**, which focuses on the ability for clients to discover mobile services; and **accessibility**, which focuses on the ability for service connections to be established and for data to be sent bi-directionally across those connections [16].

**Addressability**

As with traditional networks, each host device in a mobile network is assigned an *Internet Protocol* (IP) address by the MNO. Today, this unique IP address within a network is usually assigned by IPv4 [17], a widely used transport protocol. However, this protocol lacks support for mobility. As shown in figure 2.2, a device’s IP address often changes as it moves from one point of attachment (e.g. a base station or wireless router) to another [15, 16, 18]. When this occurs, any connections between clients and a service over that IP address will break.

Within mobile networks, any changes to mobile IP addresses are applied by MNOs and generally not published to other devices within the network [19, 20]. Most MNOs, including Vodafone NZ, do not make mobile device IP address public unless a user pays for a premium service [17, 20, 21]. Given a small number of host devices, this may not be economically viable [21]. Furthermore, mobile networks sit behind *Network Access Translators* (NATs), further hiding the addresses of devices within those networks [15, 16]. Therefore, clients lack a destination address to which to send service requests.

**Accessibility**
Assuming the addressability problem is solved, clients must still be able to establish a bi-directional communication link with mobile services, in order to send requests and receive responses [16].

When operating within mobile networks such as 3G, host devices rely on MNOs to expose their service endpoints to outside clients. Even when the endpoint address is exposed, incoming requests from clients are often blocked by MNO firewalls, as shown in figure 2.3. Usually this is due to security concerns or the desire to charge a premium for public access [15, 16].

Sometimes, non-standard protocols are used within mobile networks, usually to conserve valuable bandwidth. Such protocols include custom TCP protocols that may only be
initiated from within networks and are disconnected due to inactivity after an arbitrary MNO-specified period of time [16].

Due to these restrictions, clients are often either completely unable to access mobile services within a network, or are required to pay subscription fees to consume those services [16], thus reducing interoperability and ubiquity, two key aspects of SOA [1].

2.1.2 Availability

It is expected that a service will be available to a client whenever that client requires it, within reason. However, mobile networks are often subject to unreliable, low-bandwidth connectivity, and device resources such as CPU and battery life are limited. Ensuring the availability of a mobile service involves addressing these issues via preventative measures and / or appropriate recovery mechanisms [1, 22].

Mobile devices operating in wireless networks are often subject to fluctuations in the connectivity offered by those networks. Often these fluctuations are due to outside influences such as weather conditions and the limited range of wireless receivers [23]. Furthermore, devices may experience momentary loss of connectivity when moving between different base stations on Wi-Fi and 3G networks [18, 20, 24]. As these changes in connectivity occur, any clients consuming services hosted on these devices will experience service disruptions, including high latencies and complete loss of service. Any mobile services middleware should attempt to mitigate this as much as possible [1, 14].

In addition to wireless network issues, mobile devices themselves are subject to limited resources. This includes CPU and memory resources, but also battery life. The middleware could provide battery life optimizations to increase service lifetime, such as switching to low-power Bluetooth networks [23, 25] when available.

2.1.3 Scalability

Scalability in service-oriented architecture is concerned with the ability for a service to handle increases in clients, and to make use of any extra available resources [1]. With mobile services this involves the non-trivial handling of concurrent requests from multiple clients on resource-constrained devices, in a manner that does not affect the normal operation of a device (for example, a mobile service host may be used as a regular smartphone at the same time).

Compared with traditional servers, even today’s comparatively powerful mobile devices are significantly resource constrained. Furthermore, the supporting infrastructure of
traditional servers is much more capable, often including such features as replication and distributed servers to spread load [1, 26], and always-on network connections in excess of 1 Gbps [16] as opposed to more limited wireless networks. Therefore, in comparison with traditional servers, unaided mobile devices can process fewer client requests in a given amount of time, while additionally being unable to receive those requests or send responses as quickly [16].

2.1.4 Testability

Any potential mobile service provisioning solution should address the reachability, availability and scalability challenges described above. They should allow the creation of performant and dependable mobile services. To test whether a particular solution addresses these challenges and offers sufficient performance and dependability, there is a need for a testing environment which can simulate a wide variety of network conditions, similar to those experienced by mobile devices. Furthermore, the environment needs to be capable of simulating multiple devices, to speed up the testing process and allow testers to have greater confidence in any reported results. Finally, the testing environment should be automated as much as possible - requiring little input from testers other than starting tests and gathering results. In the case of mobile applications this necessitates a method for interacting with a device’s user interface.

In the absence of such a test environment, the alternative would be to obtain multiple physical devices, and seek out many real-world network conditions under which to perform manual black-box testing. While testing with real devices should definitely form part of the testing process, relying on this method exclusively is infeasible. Firstly, obtaining multiple real devices - and humans to operate those devices - may not be practical at all times. Secondly, it is very hard, if not impossible, to provide test repeatability in this case. For example, the mobile network conditions at any given time are subject to fluctuations. In such an environment it would be difficult to track down bugs that are triggered by certain network conditions - especially if such conditions only sometimes trigger these bugs. We need to remove this as a variable from the testing environment.

2.1.5 Heterogeneity

Service developers may wish to deploy their mobile services on a wide range of physical devices. In order to do this, developers will, at the very least, need to consider the differences in available computational resources on these devices. Developers may additionally wish to support devices with different underlying operating systems. In this case they may need to consider the differences in features offered by these platforms,
or even entirely different programming models. Ideally, a comprehensive mobile service provisioning solution would provide some measure of cross-platform support to service developers.

### 2.2 Mobile Service Provisioning Middleware

The literature describes several systems with the goal of enabling mobile services. Typically, these solutions take the form of either *embedded* or *intermediary-based* middleware. A classification of these types is given in figure 2.4. Embedded middleware is deployed directly on host devices. Generally, these solutions are lightweight versions of existing fixed-network middleware solutions. Intermediary-based middleware consists of some components deployed directly on host devices, with other components deployed within the fixed network. Clients communicate with services via these fixed-network components.

![Classification of middleware-based mobile service provisioning solutions](image)

**Figure 2.4:** Classification of middleware-based mobile service provisioning solutions

#### 2.2.1 Rover Toolkit

As shown in figure 2.4 (1), embedded middleware solutions are deployed entirely within the host-device. The Rover toolkit [27] is an example of this approach and is designed to provide a client/server based distributed object model to shield mobile application developers from limited mobile device and network resources. Rover handles limited resources and intermittent connectivity using two techniques: relocatable dynamic objects
(RDOs) and queued remote procedure calls (QRPC) [27]. RDOs represent an object that can be dynamically moved between a client and a server or vice versa. The QRPC mechanism enables client requests to be automatically queued if connectivity with a server is lost, to be delivered at a later time by the Rover middleware when connectivity is restored.

### 2.2.2 Ice-E

ZeroC’s embedded Ice (Ice-E) [28] is an ultra-lightweight Ice implementation designed for devices with limited CPU and memory capacity. Ice is itself an alternative to the CORBA middleware platform which shares similar objectives and offers similar programming abstractions [29]. Ice provides services for locating remote objects, known as the IceGrid [29]. Ice-E shares a common protocol with Ice, thus enabling both Ice and Ice-E devices to discover and consume services hosted by other such devices.

### 2.2.3 Soap ME

Soap ME [30] is a lightweight web service container that enables mobile service provisioning for Java ME CLDC [31] devices. It enables clients to directly consume web services hosted within the mobile device. Using SoapME, devices may host multiple SOAP or HTTP-based web services. Clients communicate with a Web Service Manager installed on the device, which routes requests to the intended service at runtime. The manager supports runtime addition and removal of web services.

### 2.2.4 Web Services with IP Multimedia Subsystem

Using intermediary-based middleware, clients communicate with host-devices through an intermediary, or proxy machine in a fixed network (see figure 2.4 (2)). These intermediary machines aim to shield clients from the mobility related issues discussed in section 2.1. In the literature, the use of an intermediary is seen as a valuable technique as it can potentially help address reachability, availability, and scalability. Hence, there are a significantly larger number of solutions which use this technique.

IP Multimedia Subsystem (IMS) [32] is a framework that provides IP media management and session control for mobile network operators. The work in [32] demonstrates how IMS can be extended to enable participant devices to expose Web Services. IMS framework routes client requests to the correct device with the help of the MNO.
2.2.5 Mobile Host

Mobile Host \cite{2,33} is a lightweight framework that enables mobile phones to expose web services. Initially it allowed mobile services to be exposed as SOAP-based services; more recently it added support for REST services \cite{34}. The Mobile Web Services Mediation Framework (MWSMF) is designed to allow Mobile Hosts to be discovered by clients. It uses peer-to-peer (P2P) based technologies coupled with the JXTA protocol \cite{35} to overcome mobility issues \cite{36}. Figure 2.5 shows how, in a mobile network supporting Mobile Host, the JXTA protocol and infrastructure is implemented by the MNO. Mobile services use JXTA to allow clients to discover and consume them, and to improve quality of service (QoS) \cite{2,37}. Alternatively, clients may communicate directly via an IP link to mobile hosts if their IP address is public, or registered with a web service directory (UDDI) \cite{34}.

2.2.6 Nokia Mobile Web Server

Nokia Mobile Web Server (Nokia WS) \cite{16} enables mobile service provisioning using an intermediary that acts as a HTTP gateway to mask device mobility, as shown in figure 2.6. The device and the gateway communicate using a TCP based proprietary protocol. Clients communicate with the gateway, which in turn uses this protocol to relay any
HTTP client requests to the device. The device handles these requests using a standard HTTP server [16].

2.2.7 Jini-Based Solutions

Jini [14, 38] is a middleware designed to enable Java-based Service Oriented Architecture. The Jini Surrogate Architecture (JSA) [39] extends Jini with support for devices unable to run Java, by allowing a Surrogate Host machine (SH) to expose a Surrogate - a software component which participates in the Jini network on behalf of the device. A detailed overview of Jini and the JSA are given in Chapter 3 section 3.2. Neither Jini nor JSA were designed for the purpose of enabling mobile service provisioning. However, their rich feature set serves as a useful base which can be extended to facilitate this purpose.

Mobile Service Platform (MSP) [15] and UORB [40] are two frameworks that are developed based on JSA. Both MSP and UORB have extended the JSA specification to enable host-devices to expose mobile services using the SH, which is responsible for exposing and managing services on behalf of the host-device. Clients first connect to SH to consume the service, and the application developer can utilize the APIs provided by MSP and UORB to delegate the request back to the device if required.

MSP supports Java and J2ME-based devices and extends JSA by providing an HTTP-based interconnect to handle communication between devices and surrogates [15]. It
has support for context-aware vertical-handover [41] between different TCP/IP based network connections such as Wi-Fi and 3G [42, 43]. UORB exposes services using CORBA middleware and provides neither context awareness nor vertical handover support [40]. UORB and MSP are not interoperable.

2.3 Addressing Reachability

As outlined in section 2.1.1, reachability needs to be addressed to allow clients to discover mobile services. The majority of existing solutions introduced in section 2.2 rely on mobile network operators or other existing infrastructure to solve this issue. However, several of these solutions - particularly those that use an intermediary - offer some level of reachability support. Furthermore we have identified other techniques from the literature which, while not specifically aimed at mobile service provisioning, can be used to address reachability.

2.3.1 Mobile IP

As explained in section 2.1.1, host-devices connected to mobile operator networks do not have a fixed end-point due to mobility. Mobile IP [20] is a protocol that aims to address this problem by ensuring that a mobile device is reachable using a single IP address even after its actual point of attachment to the network has changed. Figure 2.7 presents an overview of the Mobile IP protocol. The mobile network in which each mobile device usually resides is referred to as that device’s Home Network. Each network has a Home Agent (HA) which assigns mobile devices belonging to that network a Home Address (IP_{HA}). All IP packets sent to IP_{HA} will be directly forwarded to the device by the HA (steps 1, 2). When the device moves to a different Mobile-IP-supported network, it will contact that network’s Foreign Agent (FA) which will then assign the device a Care-Of Address (IP_{CoA}) (steps 3, 4). The Mobile IP infrastructure will take care of informing the HA of this new IP_{CoA} (step 5). Subsequently, any packets the client sends to IP_{HA} will be routed to the device through the FA, via IP_{CoA} (steps 6 - 8). The protocol will also update the client’s endpoint to point directly to the device in the foreign network after the first request, to enable direct communication [20]. Implementing the Mobile IP protocol requires setting up the necessary network infrastructure and requires all associated networks to support the protocol as demonstrated by MRPC [44].
1. Client sends IP packet to device’s home address
2. IP packet received by home agent and forwarded to device
3. Device moves to foreign network
4. Device notifies foreign agent of arrival and is given a care-of address
5. Foreign agent notifies device’s home agent of the care-of address
6. Client sends IP packet to device’s home address
7. Home agent opens IP tunnel to the care-of address and forwards the packet
8. Foreign agent forwards the packet to device

Figure 2.7: Overview of Mobile IP Protocol
2.3.2 GTP

The GPRS tunneling protocol (GTP) [18, 45] is an alternative to Mobile IP that consists of a collection of IP based communication protocols. It is used by some mobile networks. It allows MNOs to provide a single IP address for a given device for a given session [45]. It is used commonly by mobile network operators to ensure link layer mobility when a mobile device moves from one point of attachment to another (such as from one base station to another).

2.3.3 Push Notification Services

In recent years, Push Notification technology has emerged as a way to deliver time-sensitive, location-aware information to mobile devices [46]. Push services provide infrastructure which frees application developers from managing asynchronous notification channels over mobile networks.

![Figure 2.8: Essential Architecture for Push Notifications](image)

Figure 2.8 shows the fundamental components of push architecture, as proposed in a reference model by the Open Mobile Alliance [47]. Client Devices, which wish to receive push notifications, register with a Push Proxy Gateway (PPG). Devices communicate with the PPG using an Over the Air (OTA) protocol, which manages network resources and keeps the connection alive in the presence of intermittent connectivity and varying endpoints. When devices register, they identify one or more Content Providers from which they would like to receive notifications. When a content provider sends a notification, it communicates with the PPG via a Push Access Protocol (PAP). Both the OTA and PAP protocols are implementation-specific, offer varying levels of security and QoS guarantees, and are not interoperable with different push solutions.

Different push implementations support different interaction paradigms [48]. An overview of different supported paradigms is given in Figure 2.9. Google’s Google Cloud Messaging (GCM), RIM’s Blackberry Push Service (BBPS), Microsoft’s Microsoft Push Notification Service (MPNS), and Apple’s Apple Push Notification Service (APNS) all offer the point-to-point model shown in Figure 2.9 (a), where each content provider may send
a single notification to exactly one device. GCM and BBPS additionally support the multicast model (Figure 2.9 (b)), in which a single notification may be sent to multiple, known devices. BBPS also offers broadcast communication (Figure 2.9 (c)), in which a notification may be sent to all devices which have registered interest in a particular content provider. Figure 2.9 (d) demonstrates the publish/subscribe paradigm offered by IBM’s MQTT Telemetry Transport (MQTT) service. Devices may subscribe to receive notifications on various topics, which are hierarchically ordered. Content providers publish notifications about particular topics, which are received by devices subscribed to those topics.
2.3.4 Discussion

Table 2.1 presents a summary of reachability solutions identified in the literature. Solutions generally fall into two categories: either network-oriented solutions which give devices the illusion of static IP addresses, or intermediary-based middleware acting as a gateway between clients and host devices. Clients communicate with these intermediaries (with known, static, public addresses) rather than directly with devices, freeing them from the requirement that the device’s IP address be known. Additionally, in all of these cases, the device is required to initiate communication with the intermediary, preventing the intermediary from requiring knowledge of, and access to, the device’s IP address.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Technique(s) used to address Reachability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile IP</td>
<td><strong>Addressability</strong>: assigns static IP addresses to devices. <strong>Accessibility</strong>: IP addresses are public.</td>
</tr>
<tr>
<td>GTP</td>
<td><strong>Addressability</strong>: assigns static IP addresses to devices. <strong>Accessibility</strong>: public IP addresses, when offered by MNO.</td>
</tr>
<tr>
<td>Mobile Host</td>
<td>Limited - MNO must offer either public IP addresses or implement the JXTA infrastructure.</td>
</tr>
<tr>
<td>Nokia WS</td>
<td>Device opens proprietary TCP connection to gateway. Client needs only to connect to gateway.</td>
</tr>
<tr>
<td>MSP</td>
<td>Device opens HTTP connection to SH on fixed network. Client needs only to locate SH.</td>
</tr>
<tr>
<td>Push Architecture</td>
<td>Device opens proprietary connection to push gateway, which content providers may access with well-known URL.</td>
</tr>
</tbody>
</table>

Table 2.1: Summary of how the identified works address reachability

Some MNOs such as Vodafone NZ use the GTP protocol [45] to address reachability. Use of GTP allows host-devices connected to the Vodafone network to use a single IP address when switching between different base stations. In addition, MNOs often provide services such as Vodafone’s Corporate Connect [21] to provide devices with static IP addresses. However, these solutions are usually intended for large-scale use [49], making them economically unsuitable for mobile service provisioning unless large numbers of services are deployed across a large number of devices. When not subscribed to these expensive plans, it is common for MNOs to restrict access to their network from entities outside that network [21, 49], and to regularly change device IP address whenever the device moves between different networks.

Any solution relying on the MNO to solve the reachability problem will be unable to offer services to clients outside the mobile network unless those clients are subscribed to these expensive large-scale solutions [21, 49] or if Mobile IP is available on the network.
This includes IMS, ESB, Mobile Host, UORB, and all embedded middleware solutions [28, 30, 32, 33, 37, 50–52].

Mobile Host provides an alternative means of consumption if IP reachability is not possible: the ability to expose services as JXTA nodes in a P2P network [2]. However, this alternative requires the JXTA infrastructure to be implemented by the MNO, again hindering pervasiveness.

Jini and JSA also assume the host device to be reachable, and so provide no solution to this problem by themselves [14, 39]. MSP, which is based on JSA, requires that the host device always initiates the connection to the intermediary [15], and notifies it whenever its network address changes. Furthermore, MSP uses an HTTP-based interconnect, allowing it to make use of HTTP tunneling [53] to overcome accessibility issues. This allows the intermediary to be deployed outside the mobile network and communicate without being blocked by MNO firewalls. For communication between the intermediary and the client, MSP’s use of the Jini architecture [15, 42] makes addressability and accessibility a non-issue. However, host devices are required to have prior knowledge of the locations of these intermediaries, and a mechanism to update this knowledge upon the addition of new intermediaries is not specified.

Nokia WS similarly requires its host devices to initiate connections with the HTTP gateway to overcome accessibility [16]. The device establishes a TCP connection to allow bi-directional exchange of information with the gateway. However, MNOs make TCP connections stale after an arbitrary period of inactivity to conserve network resources [16]. This requires the device to periodically send keep-alive messages or establish new TCP connections to maintain connectivity. Communication between clients and devices is achieved by assigning devices a URL, which is intercepted by the gateway and routed to the correct device [16], allowing clients to consume mobile services without prior knowledge of the device’s IP address.

When registering to receive notifications using a push architecture, devices register with the Push Proxy Gateway and open a connection to that gateway [47]. The protocol implemented by this connection is proprietary and specific to a given push technology. However, in all cases, the protocol keeps the connection alive and re-establishes it if the device’s endpoint changes [48]. Furthermore, the gateways are accessible to content providers via a well-known URL. This addresses the reachability problem. While push services were not designed to offer mobile service provisioning, their solutions to communication in mobile networks can be adopted to assist in offering such a service.

For our solution, we do not wish to rely on MNOs to provide any guarantees regarding the reachability of devices within their networks. Furthermore, we wish to allow mobile
services to utilize any available wireless network. Our review shows that the use of an intermediary-based architecture is a promising way to achieve these goals.

2.4 Addressing Availability

As discussed in section 2.1.2, the availability of a mobile service may be hindered by low-bandwidth, intermittent connectivity and / or limited mobile device computational power and battery life. Existing mobile service provisioning solutions may attempt to address availability by addressing one or more of these issues.

2.4.1 Discussion

Table 2.2 presents a summary of how the identified techniques address availability.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Technique(s) used to address Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rover</td>
<td>QRPC buffers calls to disconnected devices, to be forwarded once device reconnects.</td>
</tr>
<tr>
<td>Ice-E</td>
<td>Clients repeatedly retry failed requests.</td>
</tr>
<tr>
<td>Nokia WS</td>
<td>Heartbeat mechanism detects disconnected devices, but the response to clients upon making a request of disconnected devices is not specified.</td>
</tr>
<tr>
<td>MSP</td>
<td>Allows handover to “fastest” or “least power-consuming” IP-based network. Allows computational offload to SH to reduce CPU power usage (left up to service developers). Uses heartbeat to detect disconnected devices (client interaction with disconnected devices is left up to service developers).</td>
</tr>
<tr>
<td>UORB</td>
<td>Allows computational offload to SH to reduce CPU power usage (left up to service developers). Uses heartbeat to detect disconnected devices (client interaction with disconnected devices is left up to service developers).</td>
</tr>
<tr>
<td>Push Architecture</td>
<td>Push requests to disconnected devices are buffered by Push Gateway. The exact level of delivery guarantee in this case is implementation-specific. Many requests to a single device are batched and sent at once, to conserve power usage from continual wireless radio activation (GCM and MPNS only).</td>
</tr>
</tbody>
</table>

Table 2.2: Summary of how the identified works address availability
Mobile network operators such as Vodafone NZ tend to offer session management abilities for devices while switching between different base stations within a network [21, 49]. However, as discussed in section 2.3, they tend to charge a premium for such a service. Furthermore, the session state is not maintained when switching between different mobile networks (e.g. 3G to 4G), or to external networks (e.g. Wi-Fi) [21]. Therefore this provides only a limited and cost-ineffective solution to intermittent connectivity.

Among the embedded middleware solutions, Rover and Ice-E attempt to provide solutions to intermittent connectivity. Rover uses QRPC to buffer client requests to unreachable devices [27]. When the device reconnects, it will notify its clients, causing unsent client requests to be resent. Ice-E uses a retry mechanism, where several attempts are made to deliver unsuccessful requests [28]. Comparatively, Rover’s technique favors bandwidth conservation whereas Ice-E frees services from having to explicitly manage their clients. Other embedded middleware technologies do not provide solutions to intermittent connectivity.

Intermediary-based solutions hide device connectivity issues from clients by using the intermediary as a proxy. Nokia WS, UORB, and MSP monitor connectivity using a keep-alive mechanism where the device periodically sends a heartbeat message to the intermediary (MSP) [15] or vice versa (Nokia WS, UORB) [16, 40]. If the intermediary and the device cannot communicate within a specified period of time (known as the keep-alive period), the connection between the two is closed. If a client makes a request of the mobile service while the device is disconnected, UORB and MSP leave the details on what should happen up to developers [15, 40]. Typically this could involve sending a response to the client containing either cached response data or a message informing the client to try again later. Nokia WS does not mention how client requests are handled in this case [16].

MSP additionally reasons on currently available context information, such as connectedness and signal strength of various wireless connections, to perform vertical handover between 3G and Wi-Fi networks [42, 43]. This allows the service to overcome intermittent connectivity in one network to be mitigated by availability of another. However, the addition non-IP networks, such as Bluetooth, is not supported.

Each push notification technology examined offers some level of ability to deal with intermittent device connectivity. In all cases, the push gateway buffers some number of push requests from content providers, and replies to push requests with a response that contains information regarding whether (i) the notification will be immediately forwarded to the device, (ii) the notification will be buffered and delivered at a later date, or (iii) the notification will not be delivered [48]. Upon reconnecting, a device will receive all buffered notifications from BBPS, MQTT, and GCM. MPNS buffers only the
first 32 notifications before discarding the rest (though the content provider is notified of
this occurrence), while APNS keeps only the most recent notification. In addition, BPSS
and MQTT provide the ability for content providers to be explicitly notified by the push
gateway of the delivery status of individual notifications.

MSP’s context-awareness framework allows device battery life to be monitored [42]. Using
this information, if battery life decreases beyond a certain threshold, MSP can select an
alternative IP-based network interface that offers lower power consumption [23]. However,
this context processing occurs on the device [43] which uses precious limited device CPU
and memory resources, in turn increasing power consumption and potentially negating
any benefit.

UORB, MSP, and Rover provide the ability to offload computational tasks to other
entities [15, 27, 40]. UORB and MSP allow developers to offload tasks to the intermediary,
whereas Rover transfers the tasks to clients as part of its RDO architecture. This can help
conserve CPU and memory of host devices and therefore reduce power usage. However,
MSP and UORB provide a very low level of abstraction for this purpose - developers
must acquire access to the raw byte stream and manually implement any marshaling
or unmarshaling of data. Furthermore, Rover may require computationally intensive
synchronization of different copies of remote objects [27], making it feasible only with
small numbers of clients.

Within examined push technologies, GCM and MPNS use a notification buffering scheme
to conserve power. When receiving many push requests destined for a single device over a
short time period, a push gateway may batch all of these requests and send them at once,
thus reducing the number of times the device’s radio needs to be activated to receive
those requests.

By examining the related work, we can see that intermediary-based solutions can be
used to manage intermittent connectivity by shielding clients from the need to connect
directly to devices. However, in all cases except the push services, the details on what
happens when clients attempt to connect to temporarily-offline services is not specified.
Furthermore, the reliability of protocols between devices and intermediaries is not spec-
ified in the literature - it is not clear whether these protocols offer any kind of data
integrity guarantees. In fact, several of the push technologies explicitly allow data loss
as part of their protocol. Ideally, we would be able to leverage the intermediary to in-
crease availability as shown in the literature while additionally providing such reliability
guarantees.
2.5 Addressing Scalability

As discussed in section 2.1.3, developers must address the comparatively limited processing power, memory, and bandwidth of mobile devices if they wish to offer scalable mobile services. In all examined cases where a solution claims to offer improved scalability, it does so via the use of an intermediary or other fixed-network machine.

2.5.1 Code Offload

In an effort to decrease the burden on mobile devices when performing computationally intensive tasks, there are several works which investigate the offload of code from devices to more computationally-able infrastructure such as the cloud. In order to be effective, the time taken to transfer the data used by a computation should be less than the difference in execution times between the device and the fixed-network machine [7].

MAUI [54] allows developers of Windows Mobile applications to specify via annotations whether a certain method is allowed to execute remotely. At runtime, it solves a linear algebra equation to optimize the predicted energy use, in millijoules, of all remoteable methods by selecting some to execute on the device, and others to execute remotely. Estimates for energy consumption of a method are based on actual energy consumed while invoking previous iterations of that method. However, energy consumption measurements were carried out only for a specific device, and were obtained using external hardware, making this solution not portable.

CloneCloud [55] is an alternative approach which allows any mobile application which runs on a Dalvik virtual machine (such as most Android applications) to benefit from code offload. Prior to execution, a static analyzer identifies legal points at which execution could branch between local and remote machines, making sure not to allow execution of mobile-specific code (e.g. code that uses the GPS or Bluetooth) on the cloud. An application to use CloneCloud is then run multiple times locally and remotely in order to build a profile of execution times and energy consumption for each possible partition. Finally, the byte code of the application is modified by the profiler to add migration points (to or from the cloud) at various locations in order to minimize the total energy use or execution time of the app.

Work in [56] attempts to further optimize such code offloading middleware with evidence-based learning techniques. In this work, many examples of code offload traces from executing applications are analyzed, and the results are used to inform future decisions on whether to offload code.
Work in [57] investigates how code offload can be used to assist with mobile service provisioning, and considers how such a system interacts with data sources external to the smartphone or offloading infrastructure (e.g. a particular service might require access to files on Dropbox\(^1\)). Typically, the bandwidth is much higher - and latency much lower - between the offloading infrastructure and the external resource, compared with the smartphone. When deciding whether to offload, their system, the Follow-Me Provider, takes this into account. Furthermore, if execution of a service request is offloaded to the cloud, the result of the request may be sent directly from the cloud back to the client, rather than returning to the mobile device first. As their infrastructure provides REST services, this is presumably accomplished through URL redirection, though not explicitly mentioned in the paper.

### 2.5.2 Discussion

Table 2.3 presents a summary of how the identified software solutions address scalability. As mentioned in Section 2.4, MSP, UORB and Rover provide the ability to offload tasks to either intermediaries or clients \([15, 27, 40]\). As well as conserving battery life, these solutions can leverage this extra computational power to improve scalability. However, each of these solutions must perform the computational offload manually - there is no mechanism to determine whether, at a given point in time, it would be beneficial to do so. This may lead to scalability problems of its own if, for example, many MSP host devices offload to the same intermediary, thus overloading that intermediary. Furthermore, there is no support for any abstraction over and above access to a simple byte stream - developers must manually implement their own marshaling and unmarshaling logic if required.

MAUI, CloneCloud, and other work \([54–57]\) all provide more comprehensive ability to offload code compared with the above solutions, though they do not attempt to address any other mobile service provisioning challenges and thus cannot be used on their own for this purpose.

Other intermediary-based solutions (Nokia WS \([16]\), the IMS-based solution in \([32]\), and Mobile Host \([33]\)) do not allow their intermediaries to be leveraged to increase the computational power available to the service, and hence do not improve scalability in this manner. However, Mobile Host’s intermediary compresses communication between host devices and intermediaries to conserve bandwidth, thus increasing the scalability of the network, if not the service itself. Furthermore, in recent work \([34]\), the authors

\(^1\)https://www.dropbox.com/
Chapter 2. Literature Review

Solution | Technique(s) used to address Scalability
---|---
Rover | RDOs can be offloaded to clients for processing, increasing overall compute power of the system.
Ice-E | Services could potentially be replicated using IceGrid, though implementation specifics are left to service developers.
Mobile Host | Compresses communication between devices and intermediaries to conserve bandwidth. Cloud-based intermediary.
MSP | Can increase overall compute power by leveraging SH machine (left to service developers).
UORB | Can increase overall compute power by leveraging SH machine (left to service developers).
Push Architecture | Notification overwriting to avoid sending unwanted notifications, thus conserving CPU bandwidth. Available with GCM (left to service developers) and APNS (automatic, but aggressive and may cull important notifications).
Computational Offload Solutions | Allows devices to leverage external computing resources in a more automated manner, but does not offer solutions to any other mobile service provisioning challenge.

Table 2.3: Summary of how the identified works address scalability

have moved their intermediary framework, MWSMF, into the cloud to allow it to take advantage of the cloud’s inherent scalability.

Of the various push technologies examined, GCM and APNS provide a *notification overwriting* feature in an attempt to increase scalability. Using GCM’s notification overwriting feature, developers may give a notification a certain *collapse-key*. Whenever notifications are to be sent from the push gateway to the device, only the most recent notification with a given collapse-key will be sent, thus conserving bandwidth. APNS provides the same ability but the overwriting is instead mandatory - only the most recent notification bound for a given device will be sent.

Devices using Ice-E could potentially use the IceGrid service [29] provided by the Ice middleware platform to address scalability. Using this, a mobile service could potentially be replicated amongst many host devices to distribute client load. However, the exact implementation details, both in terms of accessing IceGrid and synchronizing between different host devices, are left up to the developer.

To the best of our knowledge, any solution which attempts to increase scalability does so through the use of infrastructure external to the mobile device - e.g. an intermediary or other fixed-network or cloud-based machine. However, many of the identified solutions either do not sufficiently leverage the intermediary for scalability, or do not address any other mobile service provisioning challenge, rendering them unsuitable as stand-alone
solutions. Of the identified solutions, MSP, based on the Jini Surrogate Architecture, claims to offer the best balance of reachability, availability, and scalability, though the reliability of its protocol is unclear, it cannot support non-IP-based networks, and it provides a very low level of abstraction.

2.6 Addressing Testability

We have identified in section 2.1.4 that a testing framework for mobile service provisioning should, at minimum, be able to:

- Simulate multiple devices;
- Simulate user input on devices; and
- Simulate changes in network conditions.

Testing environments for mobile platforms such as Android allow testers to control a tested app’s user interface. For example, Google\(^2\), Apple\(^3\) and Microsoft\(^4\) each allow developers to write scripts to simulate user input through a platform-specific UI automation library. Each platform also provides tools to automatically simulate user input without the use of tests scripts, though the user input events generated are essentially random. Platform simulators - such as the Simulation Dashboard for Windows Phone\(^5\) - additionally allow the rudimentary simulation of various network conditions. However, as these tools are not integrated, it is difficult to write test scripts to simulate precise combinations of user input and network changes in a repeatable manner. Furthermore these tools each function on one device at a time.

PUMA \[^{58}\] is a testing system which aims to integrate UI automation testing with the ability to simulate changes in device operating conditions, such as differences in network connectivity. It introduces a Monkey, which is essentially a program which trawls through an application’s user interface. Instead of than essentially random navigation, PUMA’s monkey allows testers to customize its behavior. Possible customizations include \(i\) determining which widget to click next, \(ii\), determining which text to enter into a text field, \(iii\) adding custom event handlers in response to page navigation, \(iv\) determining whether to navigate to a particular page, and \(v\) determining when to stop testing. Custom event handlers allow testers to execute arbitrary code whenever a particular

\(^4\)https://msdn.microsoft.com/en-us/library/ms747327%28v=vs.110%29.aspx
\(^5\)https://msdn.microsoft.com/library/windows/apps/jj206953
point is reached during the navigation of an app. Here, testers could, for example, invoke a network tool to alter the network conditions for the device. PUMA also includes a powerful instrumentation feature to allow testers to inject test code to be executed at various navigation points.

Caiipa [59] aims to allow testers to explore how an app behaves under many different conditions by leveraging cloud-based infrastructure. Primarily, Caiipa maintains a database of a large range of possible device contexts (such as differences in available CPU, memory, bandwidth, GPS locations, etc.) and the ability to create a Windows Phone virtual machine simulating a particular combination of these context values. Network simulation is achieved using the Network Emulator for Windows Toolkit, while available CPU is controlled by Windows Phone virtual machine manager. The tool will then place an application under test in many of these virtual machines deployed within the cloud.

In order to provide relevant results to testers in a timely fashion, Caiipa uses a technique called contextual fuzzing [60] to decide which contexts should get priority in the testing order. When testing a new app, the tool’s ContextPrioritizer determines an initial set of test cases to run by using results from past test runs on other “similar” apps. The tool decides whether an app is similar to a previous app by performing statistical analysis (Kolmogorov-Smirnov [61]), though the exact details of the procedure - such as how the system was initially populated with comparison apps - is unclear from the paper. Once several test cases have been run on the new app, Caiipa feeds these results back into its database to assist with selecting new test cases for the same or subsequent apps.

### 2.6.1 Discussion

Both PUMA and Caiipa represent the state-of-the-art in the mobile testing field and initially appear to meet all three of our primary requirements for a testing framework - that is, the ability to simulate multiple devices, user input, and network changes. However, both approaches suffer from shortcomings when testing a mobile service or mobile service provisioning middleware.

Ideally, a mobile service provisioning framework would allow us fine-grained control of network conditions at precise points within an application’s execution. PUMA allows testers, via event handlers, to alter a virtual machine’s environment (including network conditions) in response to certain user navigation generated by its monkey. While useful, it is unclear from the paper whether we would be able to schedule a network change at every desirable point. For example, consider a patient monitoring service. We want to begin a monitoring session using the service, then test how it responds to a network

---

[6] Part of Microsoft Visual Studio
outage after streaming data for ten minutes. As there will have been no user input events generated for ten minutes at the time we wish to simulate the network outage, it is difficult to see how this test would be possible using PUMA.

Assuming the above challenge is solved, then the use of a monkey for navigation - even a highly configurable one such as PUMA’s - would prove problematic in terms of test repeatability. Let us again consider the patient monitoring service, where we have just simulated a network outage. During the simulation we experience an app crash. Ideally we would confirm whether the app crash was caused by the network outage by scheduling another test to run. With PUMA, this may result in many other UI interactions being called before getting to the point we want to test. Assuming that the monkey is determinate (i.e. will always perform the same user interactions in the same order - whether PUMA’s monkey is determinate is unclear from the paper), this still unnecessarily increases test length and introduces many other potential points of failure, lessening our confidence in the cause of the failed test. If the monkey is not determinate then the test is not repeatable at all.

Caiipa allows multiple apps to be tested simultaneously across multiple virtual devices by leveraging cloud infrastructure. However, again we cannot be assured of test repeatability. Furthermore, both PUMA and Caiipa are designed primarily to test an app’s user interface under a range of environmental conditions. With mobile services, a large proportion of the app’s functionality is not UI-centric. For example, in the patient monitoring scenario, a common use case would be to begin a monitoring session, then move the app to the background, disregarding the user interface entirely. Moreover, the Caiipa paper specifically mentions that test results could be generated “in a matter of minutes” \[59\]. It is unclear whether the tools account for the long-running network tests which would be required to verify that a mobile service provisioning solution meets the required reliability guarantees.

Initially we will create a testing framework similar to Caiipa, in that it will allow the simulation of multiple devices and network conditions. However, rather than attempting to fully automate the user interface testing, we will instead allow testers to write tests using a scripting language. The test script language will allow testers to precisely interleave UI automation, networking, and other commands in a fine-grained manner, while allowing test repeatability by simply re-running a given test script.
2.7 Addressing Heterogeneity

With the myriad of different mobile device platforms in use today, service developers wishing to achieve widespread adoption will need to consider this platform heterogeneity. Often this requires catering for a variety of different hardware and software characteristics, such as differences in the amount of available CPU and memory, or entirely different programming models. Generally, Model-Driven Engineering (MDE) has been shown to be a promising approach when addressing heterogeneity in other areas [62, 63]. Here we investigate this, and other techniques currently used for cross-platform development.

2.7.1 Model-Driven Engineering

Model-Driven Engineering (MDE) has emerged as a way to help manage the ever-increasing complexity of today’s software systems [62]. Generally, model-driven engineering solutions combine the following:

- **Domain-Specific Modeling Languages** (DSMLs / DSLs) allow developers to specify solutions using concepts specific to a particular application domain (e.g. mobile services).

- **Transformation Engines and Generators** take the models prepared by developers using DSLs and transform them into artefacts such as code, data, deployment scripts, or other model representations.

In addition to the productivity gains allowed by code generation, MDE has other benefits. A recent study [64] shows that MDE is in widespread use today, primarily due to its models’ ability to document a good software architecture. Furthermore, it is most likely to be used to model - and generate - certain parts of a system, rather than a whole system.

One prominent example of MDE in the literature is the Model-Driven Architecture (MDA). MDA is a framework of MDE standards released by the Object Management Group (OMG) [63], and includes concepts related to platform-independence. It outlines a process by which a high-level model of a system may be (semi-automatically) transformed into one or more working implementations. A high-level overview of this process is shown in figure 2.10.

In MDA, four stages of models, and transformations between them, are defined:
Figure 2.10: High-level overview of the Model Driven Architecture (MDA) specification

- The *Computationally Independent Model (CIM)* is a business-model view of the system which specifies functionality without concern for how such functionality will be implemented. It is intended for use by domain practitioners.

- The *Platform-Independent Model (PIM)* is a view of the system which shows how the functionality represented by the CIM is to be implemented using artefacts at a lower level than the CIM, but still retaining a degree of platform-independence.

- The *Platform-Specific Model (PSM)* shows how a particular PIM is to be implemented on a specific platform.

- The *Implementation Artefacts* refer to the actual generated code and other deployment information necessary to execute a solution on a given platform.

While the MDA outlines such processes, it does not provide a concrete implementation.

We have investigated the current use of MDE in the field of mobile application development. Work in [65] proposes the model-driven engineering of smartphone applications through the use of the XIS language [66]. XIS allows developers to specify PIMs of interactive systems by defining *Entities*, *Use-Cases*, and *User-Interfaces*. XIS-Mobile [65] extends XIS by adding support for mobility-specific features such as location tracking and gesture support. Furthermore, XIS-Mobile allows developers to automatically generate an application’s User-Interface view from the Entity and Use-Case views (and to customize the generated views). XIS-Mobile’s visual designer is implemented using Sparx Enterprise Architect\(^7\), while its code generation is implemented using Acceleo\(^8\).

\(^7\)http://www.sparxsystems.com/

\(^8\)https://eclipse.org/acceleo/
Work in [67] again shows how MDE can be used to generate mobile applications. Rather than a visual-language approach, the authors’ tool, MD$^2$, is built upon the Xtext framework for Eclipse$^9$. Xtext is a text editor and parser which allows developers to create textual languages with full syntax highlighting, content assistance and validation support. The tool combines the app specified by the Xtext model with a platform-specific API written for each target platform (currently Android and iOS). Furthermore, the tool allows developers to specify that a particular app requires access to REST services. In addition to generated mobile app code, MD$^2$ also generates a Java EE 6 web application skeleton which implements the required REST API.

### 2.7.2 Web-Based and Hybrid Mobile Development

Academic work in recent years has shown the feasibility of model-driven techniques for the purpose of cross-platform mobile development. However, these techniques have not yet seen wide adoption in industry for this purpose [68]. That said, there are a variety of techniques used commonly in industry for cross-platform mobile development. Many developers choose to simply create mobile web applications. These are simply web applications designed to run within a smartphone’s browser. They are created using standard web technologies such as HTML5, CSS, and JavaScript. Such applications get essentially “free” cross-platform support, as all modern smartphone operating systems incorporate a fully-featured web browser. Furthermore, accessing such applications is often possible using the same URL as their traditional desktop counterparts - web servers are notified of the type of web browser making a request, and can serve either the desktop or mobile version as appropriate.

However, there are several downsides to mobile web apps. Primarily, several operating-system-specific features, such as Bluetooth, or the ability to execute background tasks, cannot be accessed from web apps, making certain types of applications impossible. Another concern is speed - even with today’s computationally-able smartphones and optimized browsers, complex web pages can easily overwhelm a smartphone’s CPU [68]. Furthermore, one cannot use a device’s native look-and-feel when creating web apps. Finally, mobile web app developers lose out on the various app stores for each platform as a possible route of deployment.

PhoneGap [69] has been created to address some of these issues above. It is an example of a hybrid solution - essentially a mobile web application in a native wrapper. Figure 2.11 shows an overview of a PhoneGap application. Developers write PhoneGap apps the same way they would write any mobile web app, except that they have access to a

---

$^9$https://eclipse.org/Xtext/
custom JavaScript API. Through this API, developers may call mobile-specific functions that would otherwise be unavailable. When compiling a PhoneGap app, a package is generated for each target platform. These packages may be uploaded to the app store corresponding to that platform. As each PhoneGap app is packaged locally on a device, performance is improved over that of mobile web apps, as download speeds need not be considered. This leads to significantly increased app start and page load times. However, performance of apps with extremely complex user interfaces typically suffers compared with that of native apps [70].

Titanium [71], from Appcelerator, is another solution created to address the challenges in cross-platform mobile development. As with PhoneGap, it can be classified as a hybrid approach, combining web and native technologies. Compared with PhoneGap, Titanium moves further towards the native end of the development spectrum. Figure 2.12 shows an overview of a Titanium application. Developers code a Titanium app - including the definition of the user interface and application logic - exclusively in JavaScript, making use of Titanium APIs for UI development and to access device-specific features. This JavaScript is then packaged along with a platform-specific runtime which includes a JavaScript interpreter along with native APIs. Developers may also include platform-specific code in the deployment if features not provided by Titanium are required - this allows greater flexibility compared with other approaches. As the runtime interprets the developer’s JavaScript code to build the interface using native UI libraries, each application’s look-and-feel much more closely resembles the native look-and-feel of the target platform. Furthermore, the native UI components are more performant than their web-based counterparts [70].
One potential downside of using Titanium is that other approaches - including PhoneGap, web apps, and native apps - each have a better *separation of concerns*. For example, a PhoneGap app’s UI is defined with HTML5 and CSS, while its business logic is defined in JavaScript. Comparatively, all Titanium code is JavaScript, so if separation of concerns is desired, then it must be manually considered and implemented by developers. Another potential downside of Titanium apps compared with fully native apps is that computationally-intensive tasks written in JavaScript will perform slower than native code. This can me minimized by using Titanium APIs to delegate as much work to native libraries as possible, but any custom algorithms will still suffer this issue.

### 2.7.3 Discussion

When considering an approach to cross-platform development, we can compare model-driven engineering with hybrid applications. Compared with the examined MDE approaches, both Titanium and PhoneGap currently offer a greater degree of flexibility in the types of applications they are able to create. In this way, they are similar in expressiveness to the native languages for each platform (though neither Titanium nor PhoneGap offer the complete set of each native language’s features [68, 70]), with the added benefit of cross-platform support.

While the MDE approaches are typically less expressive in general, they are designed to allow the development of specific kinds of mobile apps. For example, XIS-Mobile [65] is designed to allow the specification of business-style applications with standard Create-Read-Update-Delete (CRUD) data operations and relatively simple user interfaces, while MD² supports apps that consume REST services. Neither of these could be used to write
complex algorithms or graphically intensive apps, but within their specific domains they offer more rapid application development compared with alternatives. Furthermore, as they generate entirely native code, any apps created using these approaches will perform better than their counterparts written with PhoneGap or Titanium.

For our exploratory work with cross-platform development of mobile services, we have decided on an MDE approach. The reasons for this are twofold. Firstly, the mobile services domain is subject to more constraints compared with the mobile application domain in general. The restricted design space makes this domain comparatively easier to model. Secondly, the increased performance resulting from fully native implementations is desirable in order to reduce CPU and memory utilization on resource-constrained devices.

We will not consider model-driven UI development throughout the remainder of this thesis, as it is well-covered by the existing body of work. We will focus primarily on the specification of mobile services themselves. This will necessitate the use of a separate tool for UI development or any mobile service developed this way. For this thesis this will simply be the native UI development tools for each platform, though in the future we will investigate the integration of PhoneGap, Titanium, or another cross-platform tool to assist developers with this task.

2.8 Summary

From our investigation of the literature, we have identified reachability, availability, and scalability as challenges which must be addressed by a mobile service provisioning middleware. Furthermore, we have investigated existing mobile service provisioning solutions. From these, we have identified that an intermediary-based architecture allows us to address these challenges.

Using an intermediary, we can delegate the responsibility of locating and communicating with the intermediary to host-devices. This relieves clients from the concern of locating mobile hosts - they may simply communicate with the intermediary themselves using a well-known address. This addresses the reachability problem.

Mobile devices are subject to sporadic changes in points of network attachment - often between different wireless networks - and intermittent connectivity. The use of an intermediary can mask this from clients. This increases a service’s availability. Furthermore, an intermediary can alleviate the burden of a proportion of a mobile service’s computational and network load from devices. This further increases availability by conserving device battery power, in addition to increasing scalability.
Existing solutions have identified the merit of an intermediary-based mobile service provisioning middleware. However, we believe none of the existing solutions leverage an intermediary’s abilities fully. In all cases, the reliability of the communication protocol between the host-device and intermediary is unclear, or data loss is specifically mentioned as being allowed. Furthermore, many intermediary-based solutions do not utilize the intermediary, other than to address reachability and mask periods of disconnectivity. Those solutions which offer the ability for developers to leverage the intermediary’s capabilities, UORB and MSP, do so at a low level of abstraction, unnecessarily increasing the difficulty for developers to utilize such a feature.

In addition to specific mobile service provisioning solutions, we also investigated other techniques to address reachability and scalability in mobile applications more generally. Push notification services are now widely used in industry, as they provide infrastructure for external providers to push information back to mobile devices without concern for the device’s physical location, thus addressing reachability. However, many push protocols are specified to allow data loss, which is unacceptable for mobile service provisioning. Furthermore, push frameworks are platform-dependent and tend not to offer large payload sizes, which may be problematic.

To address scalability, code offload has been shown to improve execution times and reduce battery consumption for certain types of calculations. However, the identified solutions which offer the best support for code offload do not address other mobile service provisioning challenges.

We have also investigated heterogeneity and testability in the literature. To address heterogeneity, two approaches were investigated: Model-Driven Engineering (MDE) and frameworks such as PhoneGap, known as Hybrid Mobile Development solutions. We found that hybrid approaches tend to offer greater flexibility in the types of applications which can be created, particularly in the case of Titanium which allows developers to add platform-specific extensions when necessary. In comparison, MDE approaches are less generally applicable but have been shown to increase productivity for application development in certain domains, and offer increased performance due to native code generation.

In the field of mobile application testing, the majority of the literature focuses on UI automation testing. This refers to running tests which automatically navigate a tested application’s user interface, possibly under different simulated device conditions such as CPU speed and network bandwidth. Caiipa offers a cloud-based framework for UI automation testing which allows the simulation of a large number of test cases at once, but it is difficult to see from the literature how such frameworks will allow us to run
precise, repeatable tests - especially those which do not depend on a particular user interface.

Based on the literature, we have designed our mobile service provisioning middleware, Odin. Our initial prototype is introduced in Chapter 3. Odin leverages an intermediary-based architecture, but provides greater reliability than existing solutions, allows extensibility with respect to supported network types, and provides a higher level of abstraction for its communication protocol, allowing developers to more easily leverage the intermediary’s capabilities. To address heterogeneity we have created a domain-specific language, OdinTools, which allows developers to partially generate mobile services. We have adopted the MDE approach as we consider that the mobile services domain is sufficiently well-defined, and the performance benefits from native code generation will help improve scalability. OdinTools is introduced in Chapter 5. Finally we have developed our own testing framework for mobile services. Like the current state-of-the-art in literature, it allows the simulation of multiple devices and device contexts, and allows testers to simulate user interaction. Whereas the current literature focuses on UI-automation, we instead propose a scripting language to allow for more fine-grained, repeatable tests. Our test harness is introduced in Chapter 6.
Chapter 3

Odin: A Mobile Service Provisioning Middleware

Through our initial investigation into the challenges inherent in mobile service provisioning, and the degree to which current solutions (at the time) met those challenges, we determined that, while each challenge had been addressed to some extent in the literature, there was no holistic approach to mobile service provisioning which addressed all challenges sufficiently while still allowing developers to easily create mobile services. To provide such a solution, we designed, developed and evaluated our own mobile service provisioning middleware with this holistic goal in mind. We dubbed this middleware, Odin.¹

In this chapter, we initially outline the requirements for Odin in section 3.1, as determined by the literature review. In section 3.2, we then provide a brief outline of the key technologies utilized by Odin. Section 3.3 provides an overview of the Odin middleware itself. We then provide a brief evaluation of the prototype in section 3.4, prior to concluding the chapter in section 3.5.

Note that this chapter outlines work carried out as part of this thesis, in conjunction with work carried out by Thiranjith Weerasinghe as part of his Masters thesis [3]. Table 3.1 summarizes each of Odin’s features, and details the contributors to those features.

¹The name, Odin, obviously references the Norse god of gods, Odin. The link between Norse gods and mobile service provisioning is that the Norse were historically a nomadic race and that a common name in the literature for mobile service provisioning is nomadic mobile services.
Table 3.1: Summary of contributors to Odin features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Contributor(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surrogate architecture</td>
<td>Weerasinghe &amp; Meads</td>
</tr>
<tr>
<td>Communication abstraction</td>
<td>Meads</td>
</tr>
<tr>
<td>HTTP interconnect</td>
<td>Meads</td>
</tr>
<tr>
<td>Bluetooth interconnect</td>
<td>Meads</td>
</tr>
<tr>
<td>Vertical handover &amp; data integrity</td>
<td>Meads</td>
</tr>
<tr>
<td>Surrogate migration</td>
<td>Weerasinghe</td>
</tr>
<tr>
<td>Context awareness</td>
<td>Weerasinghe</td>
</tr>
</tbody>
</table>

3.1 Requirements

From the initial investigation into the challenges inherent in mobile service provisioning, we drew the following set of requirements:

3.1.1 Functional Requirements

Odin should:

**FR #1: Expose mobile services to external clients** - Clients should be able to discover and consume services provided by host devices. Host devices should be able to expose services over any wireless network.

**FR #2: Switch between available network connections within a host device** - Hosts should be able to switch between Wi-Fi, 3G, Bluetooth, and / or other network types when hosting a service. Such a switch should be transparent to clients, including those clients with ongoing communication with that service.

**FR #3: Support multiple modes of communication** - Odin must be able to support *request-response, publish-subscribe*, and *streaming* communication between any client and service.

**FR #4: Maintain connectivity between clients and host devices** - Failures in underlying mobile networks should be handled by Odin such that connectivity is maintained, possibly over an alternative network. This process should be transparent to clients.
3.1.2 Non-Functional Requirements

**NFR #1: Performance** - Odin should not impose excessive bandwidth or latency overheads when compared with a direct communication link across the same network interface.

**NFR #2: Scalability** - Odin should detect variations in the number of clients accessing a particular mobile service, and take steps to increase available resources for that service when necessary.

**NFR #3: Availability** - Odin must ensure that mobile services are made available for as long as possible. This includes allowing services to communicate over a variety of networks in case of the failure of a particular network, as well as optimizing battery usage to increase the battery lifespan of a device.

**NFR #4: Simplicity** - Odin should provide a well-defined API, and hide the complexities involved with managing multiple network channels.

**NFR #5: Interoperability** - Clients need not have any knowledge of Odin in order to consume services, other than the protocol chosen as the client-to-middleware communication protocol.

3.2 Supporting Technologies

During the development of Odin, several technologies were investigated and adopted. This section details the most notable of these.

3.2.1 Jini

Jini is a Java-based distributed service oriented middleware platform [14]. It provides a platform to facilitate interaction between different software and hardware entities, irrespective of the differences in their communication protocols, drivers or operating systems [14]. Our primary motivation for the choice of Jini is its support for protocol independence, configurability and fault tolerance. A core objective of Jini is to enable dynamic addition and removal of services from a Jini federation, which consists of clients and services connected together by Jini middleware [14, 38]. Though based on the Java programming language, Jini is language-independent by allowing non-Java objects to be wrapped within Java code [14]. However, it requires both the service consumer (client) and the provider to support Jini middleware [14].

Figure 3.1 presents the key interactions between different parties in a Jini federation. It can be observed that the design closely follows the traditional SOA paradigm. Jini
services, clients and the Lookup Service make up the three core components of the SOA. The *Jini Lookup Service* (LUS) is a service provided as part of the Jini middleware that acts as a distributed service registry to enable clients to discover services. When joining a Jini federation, services first use a *discovery protocol* to find a list of LUSs. The discovery protocol can operate either using *multicast* discovery to locate all LUSs in a specific network, or *unicast* discovery to locate a LUS at a known address. Services then use the *join protocol* to register with each discovered LUS. This involves sending a proxy for the service, along with a description which can be searched by clients to identify the service. Services are also given a *service id*, which uniquely identifies it. Clients then discover LUSs using the same discovery protocol used by services. Once discovered, they search these LUSs for required services based on service name, service id, and/or service metadata.

In addition to decentralized service registries, protocol independence and code mobility, Jini middleware provides various mechanisms to handle common network problems such as latency, low bandwidth and certain classes of failures [14]. This is enabled by various built-in Jini services and a robust infrastructure implementation [14, 38]. The infrastructure provided by Jini helps to seamlessly move objects between clients and services.

Jini addresses failure of clients and services through leasing [14]. For example, services are responsible for periodically notifying LUSs about their availability; if they fail to renew their leases the LUSs will assume that the service is no longer alive and dereference it. This makes the Jini federation self-healing, enables efficient use of resources and makes it possible for clients and services to join/leave a Jini federation at anytime without
interrupting other participants. In addition to leasing, distributed events is another important Jini component. It provides support for remote event registration and reliable delivery of remote events, enabling richer services to be built. For example, an ambulance may subscribe to an event on our patient monitoring service to be notified when a patient’s vital signs fluctuate.

Finally, Jini provides several standard services designed to assist clients and services participating in a Jini federation. The LUS is one example of this, and is compulsory for all Jini federations. Other examples include Javaspaces, which provides data storage; and a transaction service for distributed transactions [14, 38].

3.2.2 Jini Surrogate Architecture

![Figure 3.2: Overview of Jini Surrogate Architecture (JSA)](image)

Jini provides many features essential for a SOA. However, clients and services are constrained to use the Jini middleware, which necessitates the use of a full JVM, which many mobile devices do not offer. The Jini Surrogate Architecture (JSA) [39] was designed to allow devices not running a full JVM to participate in a Jini federation via an intermediary machine known as a Surrogate Host (SH). Figure 3.2 shows how the Jini architecture can be extended to include the JSA. A device wishing to participate in a Jini federation discovers a SH using a protocol similar to, but distinct from, the Jini discovery protocol. The device then sends a Surrogate to the SH, which in turn registers the Surrogate as a Jini service, participating in the federation on behalf of the device. Communication
between the Surrogate and Device occurs via an *Interconnect*, the implementation details of which are left up to service developers.

### 3.2.3 Jini Extensible Remote Invocation (JERI)

JERI is an upgrade over Java’s standard *Remote Method Invocation* (RMI) protocol. Using JERI, the stubs usually created at compile-time for RMI applications are instead created at runtime using Java’s dynamic proxy support. JERI is also extensible, allowing developers to customize the behavior of its transport and invocation layers \[14\]. Odin uses this extensibility support to allow clients to continue to communicate with mobile services during and after *surrogate migration* (see section 3.3).

### 3.3 Technological Overview

Odin’s architecture follows the Jini Surrogate Architecture outlined in section 3.2. Clients simply see Odin services as Jini services, and they discover and interact with them in an identical manner. In this way, the complexities involved with mobile service provisioning are masked from clients.

Figure 3.3 shows a high-level overview of Odin’s layered architecture. Service developers write code for the *Application* layer on both the device and surrogate, which will then be hosted on a surrogate host. Within the application layer, the device’s *Service* module, and the surrogate’s *Surrogate* module, are the primary modules which service developers must implement to provide their service business logic. The *Context Source* module provides information to Odin’s *context-awareness* functionality (see section 3.3.3) to inform to allow Odin to be informed of various changes in state.

Within Odin’s *Management* layer, the *Management* module is responsible for performing *vertical handover* (section 3.3.1) and *surrogate migration* (section 3.3.2). The *Context Acquisition* module is responsible for gathering raw data from context sources and sending it to the surrogate host’s context processor, while the *Context Adaptation* module receives instructions based on changes in context to inform the management module whether vertical handover or migration is required (see section 3.3.3).

The surrogate host’s management layer has further components not located on the host device. The *Context Processor* module interprets the data from the context sources to determine whether any config changes are necessary. This is located on the host only to free the device’s computational resources from this calculation. The *Discovery* module constantly listens over available network channels for new devices that wish to connect.
Figure 3.3: High-level overview of Odin’s architecture
The Export Server module exposes surrogates as Jini services. Finally, the Surrogate Management module stores surrogate instances and manages their lifecycle.

Within the Transport layer, the Reservoir module is responsible for the marshaling and unmarshalling of data using Odin’s data factories, while the Interconnect module is responsible for communication between host devices and surrogates, while managing interruptions in connectivity in association with the reservoir module (see section 3.3.1).

The Data Factory module, located in all layers, allows data to be marshaled and unmarshaled, for communication between a host-device and its surrogate. The serialization process does not depend on the Java serialization mechanism or any other platform-specific protocol - it only requires that a device is capable of writing to / reading from a byte stream. Developers may extend the provided data factories when support for custom data types are required (as is usually the case).

### 3.3.1 Communication Protocol and Vertical Handover

Odin provides a *messaging* abstraction to service developers for communication between the device and the surrogate. Odin supports request-response, asynchronous, and streaming messages, as dictated by its requirements. To add a particular message type to a system, developers must create a class representing that message, and and entry in the data factory on both the surrogate and device which is responsible for marshaling and unmarshalling those messages. Streaming is obtained by chaining messages. Whenever the host device or surrogate wishes to send a message through the interconnect, it must marshal that message and pass it to the interconnect layer for further processing. Similarly, the application layer is notified of any received messages, which must then be unmarshaled.

Figure 3.4 shows the classes within Odin’s interconnect layer which are important for message sending and vertical handover. The InterconnectManager can be accessed by a developer’s code on the device or surrogate. Marshaled messages to send are deposited here, while the manager additionally notifies the application layer of received messages, which can be retrieved here. The InterconnectManager contains one or more InterconnectAdapter instances - one for each supported interconnect type. By default, Odin provides a HTTP, a TCP, and a Bluetooth adapter. Adapters are responsible for letting the manager know whether or not connectivity is currently available over a particular network interface, and, if so, reading from and writing to the data reservoirs holding marshaled data awaiting transmission or retrieval. The OutgoingDataReservoir contains marshaled message data awaiting transmission across the interconnect, while the IncomingDataReservoir
Figure 3.4: Classes involved in Odin’s interconnect layer, important for its vertical handover mechanism

holds received marshaled message data awaiting retrieval an unmarshalling by the application layer. Figure 3.5 shows the flow of data from an OutgoingDataReservoir, across an active interconnect channel, to an IncomingDataReservoir.

Interconnect Types:

Odin’s HTTP interconnect was intended as a solution for the addressability and accessibility issues introduced in Chapter 2. Some mobile network operators do not grant external access to devices within their networks, even if an IP address is known. However, all operators allow HTTP connections to be initiated from within their networks to outside locations. The HTTP interconnect works by having the device initiate an HTTP connection to a surrogate host during registration. Once a connection has been established, the device periodically sends “keep-alive” messages in further HTTP requests. Any pending messages from the surrogate to the device are piggy-backed on the response to the keep-alive ping. In this way, communication is bi-directional. Furthermore, if a
host device changes its point of network attachment between keep-alive requests, any response to subsequent requests will be directed to the device’s new location as per the HTTP protocol.

The use of HTTP in this manner introduces a tradeoff. When the keep-alive frequency is high, the service will be more responsive as data buffered at the surrogate will more quickly be piggy-backed to the device on a keep-alive response. However, this will also lead to bandwidth wastage if a keep-alive request is sent when no data is waiting to be sent. Conversely, a low keep-alive frequency will minimize bandwidth wastage at the expense of service responsiveness. Odin attempts to mitigate this problem by introducing a dynamic keep-alive frequency. Whenever a keep-alive request is sent, if data is waiting to be sent back to the device, the delay between receiving the keep-alive response and sending the next keep-alive request will be decreased by half, to a minimum of 1,000ms. Conversely, if data is unavailable, the delay will be doubled, to a maximum of 20,000ms.

When not communicating via 3G or other mobile network, the TCP interconnect can be used, which supports direct full-duplex data transfer between devices and surrogates. Similarly, the full-duplex Bluetooth interconnect can be used for Bluetooth communication, which may be desirable when low battery consumption is of greater importance than data transfer rate.

Data Transfer:

As mentioned above, messages to be sent are marshaled and passed to the Interconnect-Manager. This in turn writes the serialized data to the OutgoingDataReservoir (ODR). To send data across the network, the active interconnect channel (IC) requests a chunk of data from the ODR. This chunk is assigned a sequence number, and the IC then attempts to transmit the chunk to the receiving end. If this operation was successful (for example: if a HTTP IC receives a 200 OK response), the ODR is notified that it no longer needs to hold onto that chunk, and it is deleted. If not, this is interpreted as a
transmission error and a reactive switching request is made (see below). In the case of
an error, the chunk is not deleted, thus ensuring no data loss.

At the receiving end, the IC attempts to read the chunk. If the entire chunk is read
successfully, it is written to the incoming data reservoir (IDR). If not, the damaged or
incomplete chunk is discarded and unacknowledged. At the IDR, a received chunk is
examined to determine its sequence number. Chunks with duplicate sequence numbers
are discarded. If chunks arrive out of order, the IDR waits until all chunks with lower
sequence numbers have also been received, ensuring the upper layers receive the data
in the same order that it was sent. Finally, the ordered chunk is placed in a buffer for
consumption by Odin’s management and application layers.

**Vertical Handover:**

As mentioned above, Odin’s InterconnectManager contains a number of Interconnect-
Adapter instances - by default, one each for HTTP, TCP, and Bluetooth interconnects.
Odin supports *vertical handover* between these adapters, either

i) at a user’s request,

ii) when instructed by Odin’s context-awareness mechanism (see section 3.3.3), and

iii) in response to failure, such as the scenario shown in figure 3.6. When the Interconnect-
Manager receives a handover request it first deactivates the currently active IC. It then
selects a new IC. By default, the InterconnectManager selects the TCP interconnect
when a Wi-Fi network is available, the HTTP interconnect for 3G, and the Bluetooth
interconnect for Bluetooth. If more than one of these networks is available, Bluetooth
will be preferred, followed by Wi-Fi, then 3G. The type of IC to use over a particular
network type is configurable (within reason - one cannot use anything other than the
Bluetooth IC over a Bluetooth connection), as is the priority. Once selected, a connection
is established to the same surrogate host over the new IC, then communication may
resume. During the handover process, any data to be sent by upper layers is buffered at
the ODR for transmission once handover is complete.

**3.3.2 Surrogate Migration**

Surrogate Migration refers to the ability for Odin’s surrogates to be moved between
different surrogate hosts at runtime. This process is completed transparently to service
developers and clients. The motivation for surrogate migration is twofold:

1. **Optimize use of resources within surrogate hosts** - It may be undesirable
   for there to be many host devices with large client loads connected to a single
   surrogate host. Allowing hosts to migrate their surrogates at runtime allows them
to load balance amongst themselves to reduce the demand on any given host.
2. **Improve service availability** - Some surrogate hosts may only support devices connecting through specific networks. For example, a user may have a personal surrogate host at their home, which supports host devices connecting via Bluetooth only. When the user then leaves the Bluetooth range, they may connect to an alternative 3G-supported surrogate host, and their surrogate could be migrated there.

Figure 3.7 presents a high-level overview of Odin’s surrogate migration process. In the example, initially, a host device initiates migration. This may be because the user initiates the migration process, or because Odin’s context-awareness framework advises that a migration is necessary (see section 3.3.3). The device initiates migration by first discovering the alternative surrogate host, then notifying its current surrogate host to begin the process.

In figure 3.7 step one, we can see that, at the time a migration is scheduled, the service is currently being consumed by client #1. As in step 2, this client is allowed to complete...
Host Device

(a) Discover Host #2

1. Device initiates surrogate migration

(b) Initiate migration

Host #1

Surrogate

Host #2

(a) Discover Host #2

2. Original host (Host #1) performs migration

(b) Migrate surrogate state

Host #1

Host Device

Client #1

(stale connection)

Client #2

(stale connection)

3. Update endpoints at device

(a) Notify of new endpoints

Host #1

Update endpoints at host device

Host #2

Surrogate

Client #1

(stale connection)

Client #2

(stale connection)

4. Update client references

(b) Update client references (JERI)

Host #1

Host #2

Jini

Surrogate

Register migrated service with jini

Surrogate

Surrogate

Surrogate

Figure 3.7: Steps involved with surrogate migration in Odin. Adapted from [3].
any currently pending requests before the migration process begins. However, any new clients, such as #2 in the diagram, will block until the migration process is complete.

Once the migration process is complete, the device will be notified of all possible network endpoints with which the new surrogate host can be reached. It will then contact its new host to resume operation. At the same time, all clients currently connected to the original host will be unblocked and notified that their connection is now stale. This notification will include the information necessary to locate the new host. Finally, the clients resume communication with the new host. This process is encapsulated within the proxies, which are provided by Odin (and indeed, any Jini service) to clients. Odin uses custom JERI proxies for this purpose. Client developers don’t need to write custom code to handle this in any way.

As Odin’s surrogate migration feature forms part of the work for Weerasinghe’s thesis, we refer the reader to that work [3] for more information.

### 3.3.3 Context Awareness

Context Awareness is the ability of a system to be aware of, and respond to, changes in the environment in which that system is running. In Odin’s case, it uses various context information to inform decisions about whether to perform surrogate migration or vertical handover. Table 3.2 presents a summary of the different kinds of context information used by Odin, and how each is used.

The physical location of devices and surrogate hosts is used to inform both migration and handover decisions. In the case of migration, it is assumed that a host physically closer to a device would be subject to less network latency. In the case of vertical handover, a host in, for example, the same building as a device may offer Bluetooth or direct Wi-Fi connectivity from behind a company firewall.

The CPU and memory of a surrogate host is used to inform migration decisions. Hosts may desire to migrate some surrogates away from themselves when subject to heavy load.

The available interconnect channels, and the current status of those channels, on both the device and surrogate host are used to inform vertical handover decisions. Connections with either increased bandwidth or reduced costs and / or battery consumption are preferred (configurable).

To allow for context-awareness, both devices and surrogate hosts collect the context information in 3.2 from various sources, and send it to the surrogate host’s Context
Context Information & Source & Used for \\
Device Location & Device GPS & Migration, Handover \\
Host Location & Hardcoded & Migration, Handover \\
Host Memory & Host diagnostic APIs & Migration \\
Host CPU & Host diagnostic APIs & Migration \\
Device Available Interconnects & Device networking APIs & Vertical handover \\
Host Available Interconnects & Host networking APIs & Vertical handover \\
Device Interconnect Status & Device networking APIs & Vertical handover \\
Host Interconnect Status & Host networking APIs & Vertical handover \\

Table 3.2: Context information used by Odin

Processor module. All context processing is performed on the surrogate host, both to free devices from the computational burden and to free individual surrogates from managing their own context.

The context processor maintains an ontology representing the current state of the system. This is represented using the Web Ontology Language (OWL)\(^2\) - specifically Stanford University’s Protègé implementation\(^3\). Whenever new context information arrives, Odin’s OWL model is updated to reflect the updates. In addition to the OWL model, a set of rules is defined in the Semantic Query-Enhanced Web Rule Language (SQWRL). The Protègé API allows an observer to be notified whenever the current state of the model fails to satisfy the given rules. When this occurs, a proposed change is generated and sent to the Context Adaptation Model to perform the change. In addition to providing adaptation support, Odin’s context model is fully queryable by service developers, who may also add their own context sources.

As Odin’s context-awareness mechanism forms part of the work for Weerasinghe’s thesis, we refer the reader to that work \([3]\) for more information.

### 3.4 Evaluation

Our original Odin prototype was evaluated with respect to performance. In this section we present the performance evaluation for Odin’s interconnect, as well as its vertical handover mechanism. For the evaluation of Odin’s context-awareness and surrogate migration features, we refer the reader to \([3]\).

\(^2\)http://www.w3.org/TR/owl2-overview/  
\(^3\)http://protege.stanford.edu/
3.4.1 Initial Performance Evaluation

**Experimental Setup:** To test Odin, we initially created a basic mobile service specifically for running the test cases in this section, to avoid any possible overheads associated with extra functionality. Our Odin host-device was simulated on a laptop with an Intel Core 2 Duo 6400 processor @ 2.13 GHz, with 1GByte DDR2 RAM, running Windows XP Professional 32-bit. A laptop was chosen for this at the time due to a lack of access to a smartphone capable of running a full JVM, required by our initial prototype. Later on, Odin would be ported to both the Android and iOS platforms.

For the experiments, our surrogate host and test clients were each deployed on machines running Windows Vista, with Intel Core 2 Duo 3.66 GHz processors with 3GBytes DDR2 RAM. To enable the testing of Bluetooth functionality, it was ensured that both the host-device, and surrogate host machines had Bluetooth adapters installed. To test the HTTP interconnect, the university’s secure Wi-Fi network was used, which at the time, had a theoretical maximum bandwidth of 54 Mbps. Communication between surrogate hosts and clients was through a 1 Gbps LAN.

Each experiment was repeated ten times to reduce variance. Average values are presented in this section.

3.4.1.1 Interconnect Overheads

To determine Odin’s overhead when sending data across the interconnect, we sent varying amounts of data across Odin’s HTTP and Bluetooth interconnect channels, as well as a direct HTTP and Bluetooth connection between the host device and surrogate host machines, using the same APIs to provide HTTP and Bluetooth connectivity. Tables 3.3 and 3.4 show these overheads for different amounts of data, for HTTP and Bluetooth connections, respectively, while figure 3.8 summarizes this information.

As we can see from the results, the HTTP interconnect channel delivered poor performance during these experiments, compared with the theoretical maximum speed of the network interface (average observed speed of 7 Mbps compared with a theoretical maximum of 54 Mbps). However, raw HTTP communication also incurred a large speed penalty, which implies that this result is due to the security measures in place over the university’s Wi-Fi network, as well as its utilization by other users during the experiments.

Analysis shows that Odin incurs an average 9% and 35% overhead compared with direct HTTP and Bluetooth communications, respectively. This overhead is created by the
chunking mechanism within Odin’s interconnect layer, and is a necessary trade-off to ensure that network errors and vertical handover do not cause data loss or duplication.

### 3.4.1.2 Vertical Handover Delays

To evaluate Odin’s vertical handover mechanism, we measure the time taken for the host device to switch from one interconnect type to another. We tested both HTTP-to-Bluetooth, and vice versa. Furthermore, for both directions, we tested both reactive and proactive switching. Reactive switching was tested by forcefully severing the communication link over the currently active channel, while proactive switching was achieved via user input.
Figure 3.8: Overheads with Odin interconnect channels with respect to direct communication over the same network type.

Figure 3.9 shows the delays incurred during each switching process. In addition to these delays, we also recorded whether any data was lost or duplicated by logging the data sent and received at both the host device and surrogate host, and comparing these logs. In each test, no data was lost or duplicated.

From the results we can see, from the minimal difference between proactive and reactive switching times, that the time taken to detect a network failure is minimal for both HTTP and Bluetooth. In the case of Bluetooth, during the experiments we observed that an IOException was thrown and reported to the interconnect layer as soon as the Bluetooth...
connection was disrupted. Similarly for HTTP, when trying to establish an HTTP link over a non-existent network channel, an IOException was immediately thrown.

When performing handover, the primary delay is in creating a new channel. From figure 3.9 we can see that creation of the Bluetooth interconnect takes longer, as evidenced by the larger delay when switching from HTTP to Bluetooth. This makes sense, considering that Bluetooth has a much lower bandwidth compared with HTTP over Wi-Fi.

3.4.2 Odin Applications

To verify Odin’s capability to serve as a useful mobile service provisioning middleware, we created several example applications. Initially, small example applications were created as part of the evaluation work in Weerasinghe’s thesis. For these, we refer the reader to [3]. Largely, these simple services, when run under laboratory conditions, along with the experimental results shown above and in [3], showed that Odin was useful and reliable.

Odin was additionally provided to two final year project teams undertaking the Software Engineering specialization of the Bachelor of Engineering degree at the University of Auckland. Using Odin, Daniel Brooker and Thomas Carey created a social networking service which allowed friends to track each other’s location [72], whereas Hussain Al-Saady and Mohammad Abdullatif experimented with the use of Odin as a real-time framework for use as a communications platform for game development. Brooker and Carey were able to create a port of Odin which could be deployed on iOS devices, though it had a cut-down feature set, lacking any context-awareness and migration support. Negatively, both teams reported difficulties with implementation and a large proportion of development effort was spent in understanding Odin’s API. Furthermore, Odin’s HTTP interconnect was shown to be insufficient for real-time applications due to the surrogate needing to wait for a device’s HTTP keep-alive message before sending data.

Finally, we also undertook a collaborative software development project in association with the National Institute of Health Innovation (NIHI). More information regarding NIHI can be found in Chapter 7. Our earliest work with NIHI involved the creation of a mobile service which allowed a patient’s heart rate and respiratory rate to be logged on a remote server. This work can be seen in [73]. This study shows the potential of Odin as a useful middleware for patient monitoring solutions. However, similarly to Al-Saady and Abdullatif’s project above, it was noted that the delay associated with the HTTP interconnect was often problematic. Furthermore, Odin experienced unexpected disruptions in service during the experiment [73]. While some of this can be attributed to undisclosed maintenance events on the server running our surrogate host, it was determined that iterative development was required to further improve Odin’s reliability.
3.5 Summary

This chapter has presented Odin, our initial prototype mobile service provisioning middleware. It allows developers to create mobile services without concern for many of the challenges introduced in Chapter 2. The work presented in this chapter is comprised both of work undertaken as part of this thesis, in conjunction with that of Weerasinghe’s thesis [3].

Under laboratory conditions, and when writing small mobile services, Odin has been shown to be a useful middleware platform in terms of development effort and reliability. However, the feedback given by final year project students, and our initial collaboration with NIHI, has shown that the prototype introduced in this chapter requires additional development in order to be used for real-world applications.

In terms of performance, Odin’s HTTP interconnect introduces unacceptable latency when its keep-alive frequency is set too low, and unnecessary bandwidth consumption when set too high. Odin’s dynamic keep-alive period is insufficient to prevent unacceptably high latency in all cases. Furthermore, its messaging and vertical handover logic needs to be simplified and further tested to ensure its reliability. Finally, Odin’s API needs to be simplified and cross-platform support needs to be added to reduce developer effort. Chapter 4 presents an updated mobile service provisioning middleware, Odin2, which seeks to address these concerns while still acting as a middleware platform which meets the challenges introduced in Chapter 2.
Chapter 4

Odin2: A Robust, Cross-Platform Mobile Services Middleware

In Chapter 3, we introduced Odin - our prototype mobile service provisioning middleware. Odin was designed to allow developers to write scalable mobile services without concern for the reachability or mobility of mobile hosts. Furthermore, Odin was designed to gracefully handle changes in network conditions, including but not limited to mobility within a network, handover to alternative networks, and disruptions in network connectivity. As discussed in section 3.4 of that chapter, while Odin performed reasonably well in experimental conditions, it failed in terms of its ability to be used in a real-world environment. Performance and reliability were not sufficient, and it relied on the Jini platform, which is much less widely supported compared with more modern technologies. This unnecessarily increased the developer effort required to create simple mobile services, with a significant amount of Jini boilerplate code required of surrogate developers. Finally, cross-platform support was never considered for Odin - only Android devices are supported.

One of the primary goals of this thesis has been to achieve a mobile service provisioning solution which is sufficiently robust that it can be employed in real-world applications. Our original Odin prototype clearly did not meet this goal, hence the decision was made to redesign the middleware. Our redesigned Odin, dubbed Odin2, would be informed by the same goals as the original Odin, in addition to its successes and failures, and an increased understanding of the requirements of a robust middleware design stemming from our early work with NIHI. To help ensure these robustness goals were met, we also begun research into a test harness to simulate network changes on multiple mobile devices in a controlled, repeatable manner. This chapter details our redesigned Odin2, middleware, while this testing framework, the Odin Test Harness, is detailed in Chapter 6.
The remainder of this chapter is organized as follows. Initially in section 4.1 we lay out Odin2’s requirements, as informed by our original goals as well as lessons learned from our original prototype, and we provide a list of Odin2’s features. We then briefly introduce supporting technologies used by Odin2 in section 4.2. In section 4.3, we provide an overview of Odin2’s architecture and its important components Sections 4.4 through 4.7 then present further details on several of Odin2’s key features. Finally, we conclude in section 4.8 by showing how Odin2’s features enable it to meet its requirements. Elsewhere, we present an overview of service development using Odin2 in Appendix A.

4.1 Requirements and Features

From our understanding of the challenges inherent in mobile service provisioning (Chapter 2 section 2.1), our desire to support multiple types of mobile services (Chapter 1 section 1.1), and our evaluation and experience with our initial prototype (Chapter 3 section 3.4), we can elicit a set of functional and non-functional requirements for Odin2, as outlined in this section. Many of these requirements are similar to those outlined in Chapter 3 section 3.1

4.1.1 Functional Requirements

Odin2 should:

FR #1: Expose mobile services to internal and external clients - Odin2 must provide a method by which clients can locate and consume services. These clients may be external, or may be Odin2 applications themselves. In the latter case, Odin2 should be able to offer increased efficiency, and the same connectivity guarantees it offers to hosts (see below) to those clients.

FR #2: Support multiple modes of communication - Odin2 must be able to support request-response, publish-subscribe, and streaming communication between any client and service. Furthermore, services should have the ability to multicast information to any subset of all available devices. This will allow for reduced traffic due to data duplication in peer-to-peer scenarios.

FR #3: Maintain connectivity between clients and host devices - Odin2 must ensure that data is not lost or duplicated whenever a host’s network connection is interrupted, and must restore any previous communications once connectivity is restored.
FR #4: Support notification of successful data delivery - Odin2 services should be able to be notified when a piece of data sent to a client arrives at its destination.

FR #5: Provide infrastructure to support remote code execution and persistence - Odin2 must allow its services to split data persistence and code execution between the host device itself and a fixed-network component with superior computational resources. The exact details of this split must be configurable by service developers on a per-application basis.

4.1.2 Non-Functional Requirements

Odin2’s set of non-functional requirements is as follows:

NFR #1: Performance - Compared with leading mobile communication technologies in use today, such as the various push notification frameworks explored in Chapter 2, Odin2 should offer similar or increased performance over the same network medium, in terms of both bandwidth and latency.

NFR #2: Scalability - Multiple clients should be able to concurrently access an Odin2 service. Similarly, multiple Odin2 services should be able to be concurrently accessed by a single client (assuming sufficient client-to-middleware bandwidth, over which we have little control). The definition of what constitutes sufficiently scalable differs from service to service, but Odin2 should be able to offer much greater scalability when compared with direct communication with a single device.

NFR #3: Availability - Odin2 services should be available to clients at least whenever the host device has any network connection available. The service should furthermore appear to be available from the perspective of clients with currently-active service interactions, even when connectivity is temporarily lost. In other words, Odin2 should achieve functional requirement #3 above transparently to clients and host devices. Furthermore, the middleware should take steps to avoid unnecessary communication where possible. This will increase available bandwidth for necessary communication, in addition to reducing the load on a device’s battery. These will also contribute to increasing service availability.

NFR #4: Heterogeneity - Odin2 should provide some manner of cross-platform support for service developers.

NFR #5: Simplicity - Odin2’s APIs should be simpler to use than those of the initial prototype. Similarly, clients which consume Odin2 services should be easy to develop.
This will be achieved by using well-known software frameworks which currently see wide usage.

**NFR #6: Interoperability** - Clients need not have any knowledge of Odin2 in order to consume services, other than the protocol chosen as the client-to-middleware communication protocol. Furthermore, this protocol should be widely used and well known.

### 4.1.3 Features

Odin2 provides the following features:

**Feature #1: Service Reachability, Availability, and Scalability** through the surrogate architecture.

**Feature #2: Interoperability** through the use of REST services and other well-documented, mature technologies.

**Feature #3: Ease-of-use**, again through the adoption of mature, well-known technologies.

**Feature #4: Expressiveness** through a fully-featured messaging protocol supporting asynchronous, request-response, and streaming messages, multicast support, and a message delivery notification system.

**Feature #5: Performance** through a redesigned, TCP-based interconnect protocol.

**Feature #6: Reliability** through the redesigned protocol in conjunction with extensive testing (see Chapters 6 and 8).

**Feature #7: Cross-Platform support** via APIs for Android and Windows Phone.

Compared with our original prototype, features #2, #3, #5, and #7 are unique to Odin2, feature #4 is greatly expanded, and feature #6 has been subject to far more thorough verification. The remainder of this chapter describes in detail how each of Odin2’s features is realized.

### 4.2 Supporting Technologies

For the design of Odin2, several leading web technologies were adopted to serve as the underlying framework and client communication protocol. Each technology listed below
was chosen due to its widespread use, industrial strength, and ease of interoperability, both with external technologies and with each other.

In addition to the originally-supported Android platform, Odin2 additionally supports development of services hosted on the Windows Phone platform. This platform was chosen primarily due to it being a mainstream alternative to Android.

To meet Odin2’s interoperability requirement (NFR #6), the ubiquitous REST Services over HTTP protocol [74] was chosen as the protocol over which clients should consume services. By default, REST services support the request-response communication paradigm. Through REST services, publish-subscribe semantics are also trivial, and are achieved by either long polling or a callback URL [74]. In conjunction with REST services, the Web Socket protocol was chosen to enable streaming semantics for external clients. The web socket protocol is a standardized protocol for allowing low-latency full-duplex communication between clients and web servers [75].

Retained from the initial prototype is the Surrogate Architecture [39], due to its ability to allow us to meet Odin2’s remote execution requirement (FR #5), while greatly assisting with scalability and availability (NFR #2 and #3). As explained in Chapter 3, it does so by providing a Surrogate Host in a fixed network, onto which mobile devices may deploy Surrogates - software components which act on behalf of, and in concert with, the host device to provide the mobile service. The use of surrogates allows Odin2 to meet FR #5 by allowing developers to write code to execute on the surrogate and to create databases persisted on the surrogate host. Furthermore, this remote persistence can be used as a cache, which lessens the network burden on mobile hosts, increasing scalability. Availability is increased by having the surrogate always available to clients, even when the host device is temporarily disconnected. The surrogate architecture also helps meet FR #1 and #3, as the fixed-network surrogate host machine necessarily has a well-known address for clients, relieving them of the need to locate a mobile host.

While the surrogate architecture concepts from the initial prototype still remain, their implementation is substantially different. To help meet Odin2’s simplicity and interoperability requirements (NFR #5 and #6), Jini was dropped as the underlying framework, and replaced with the Spring framework\(^1\). Compared with Jini, Odin2’s Spring-based surrogate host is far simpler to deploy - it may simply be deployed as a web application in any standard Java EE container, such as Tomcat\(^2\) [76]. In addition to increased ease-of-use, this also allows Odin2 to take advantage of cloud-based web containers such as Amazon Web Services\(^3\) and Microsoft Azure\(^4\) to provide essentially “free” increases in

\(^{1}\)http://spring.io/
\(^{2}\)http://tomcat.apache.org/
\(^{3}\)http://aws.amazon.com/
\(^{4}\)http://azure.microsoft.com/en-us/
scalability. In addition to these benefits, Spring also provides the sub-framework *Spring MVC*, which provides the required REST service and Web Socket support.

The use of the Spring framework allows us to make use of the *Hibernate* persistence framework\(^5\), support for which is built-in to Spring. This framework is used to persist necessary information on the surrogate host, while also allowing service developers access to Hibernate-based persistence from within their surrogates, helping with FR #5 and NFR #5. On Android-based service hosts, we instead employ the ORMLite\(^6\) library for persistence, which may be used by service developers if desired. ORMLite is similar in nature to, yet more lightweight than Hibernate, and includes SQLite support, which is the database used by Android devices. For similar reasons, OpenNETCF.ORM\(^7\) is used on Windows Phone hosts.

### 4.3 Architectural Overview

Odin2’s high-level architecture is based on the Surrogate Architecture, as shown in figure 4.1. A host device must register itself with a surrogate host, and provide its surrogate to that machine if required. Clients may then access the mobile service via a well-known, static address provided by the surrogate host. This communication takes place via a REST service API, or by Web Sockets if streaming is required.

At this level, Odin2 appears similar to our initial prototype introduced in Chapter 3. Some key differences to note, however, include:

- Odin2’s surrogates are shared amongst all host devices exposing a particular service. This allows host devices to communicate and collaborate with each other directly.

- Previously, the *identity* of a particular host device was tied to the device itself (with a device ID). With Odin2, this is abstracted into a *User* system. A particular service provider could log out of one device and log into another, migrating its state to the new device. Clients would require no knowledge of this change in device.

- Odin2’s messaging protocol supports an expanded set of communication semantics. Specifically, *i*) multicast messages between Odin2 devices, and *ii*) the ability for devices to track messages they have sent, are now supported.

An alternative view of Odin2’s architecture can be seen in Figure 4.2. Here, we can see the distribution of application layers across a host device and surrogate host.

\(^5\)[http://hibernate.org/]
\(^6\)[http://ormlite.com/]
\(^7\)[https://orm.codeplex.com/]
Odin2 services deployed on host devices contain a single package, consisting of the Odin2 APIs for that platform, as well as the developer’s custom logic specific to that service. These packages take the form of the native deployment structure for a given platform (namely, APK files for Android, XAP files for Windows Phone). Odin2 app developers interface with the APIs via the Application Layer, which provides an abstraction of a two-way communication channel with a surrogate, along with several utilities to assist with the development of common tasks such as background communication.

The Odin2 Surrogate Host is deployed as a Java EE web application package (WAR file) to any web container, such as Apache Tomcat. The package contains, in addition to the surrogate host logic itself, a simple web application accessible via a browser. This allows administrators to configure the host and add new users to the system.

Each Surrogate is deployed to a surrogate host as a JAR file containing the developer’s own Surrogate logic plus the Surrogate API, which allows developers to interact with the host as well as with devices associated with that surrogate. If the configuration of a particular Odin2 installation is known prior to runtime, these JARs can be pre-installed.
Central to both the device and host deployments is the *Messaging Layer*. This layer is responsible for providing a messaging abstraction to the upper layers. Using this abstraction, components may send and receive arbitrary messages. Messages may be set to require a response message, or otherwise, and messages may be chained together to form a streaming protocol. Finally, the messaging layer can provide notifications of several different stages of message delivery and receipt. These messages are sent and received via the *Interconnect Layer*. The interconnect layer is responsible for converting messages into a platform-independent format and sending them across a network connection, while ensuring message delivery in the face of network faults.

The remainder of this section provides further detail regarding the components within each layer.
4.3.1 Shared Components

Many components within the Messaging and Interconnect layers are common to both the device and surrogate host. Figure 4.3 shows these shared components, with the messaging and interconnect layer components shown in green and blue, respectively.

Within the interconnect layer, the InterconnectChannel class is responsible for sending messages across the interconnect medium (e.g. Wi-Fi, 3G, Bluetooth...) to the recipient, and receiving messages and delivery notifications from the sender (see section 4.5). The MessageFactory singleton is used to marshal and unmarshal messages. When messages and notifications are received, the InterconnectChannel may notify any interested parties. To get these notifications, interested parties need simply to implement the ChannelCallback interface. ChannelCallback implementations will additionally be notified if the channel closes due to an IO error or a planned closure from the remote end. As part of Odin2’s error detection logic, the InterconnectChannel implements the HeartbeatListener interface so that it may perform heartbeats at intervals decided by the HeartbeatTimer (see section 4.6).

Within the messaging layer, the InterconnectEndpoint class is responsible for abstracting away the underlying networking and error detection logic from upper layers and provides the main interface through which these upper layers send and receive messages. By invoking methods on the endpoint, components may send messages, receive messages, and query the delivery status of messages. The endpoint notifies any registered EndpointCallback s when messages are received. Upper layers implement this interface to act upon those messages. For more information on Odin2’s messaging API, see section 4.5. Each InterconnectEndpoint requires access to an InterconnectChannel to send messages, which is obtained via a ChannelProvider. The implementation of ChannelProvider is different for devices and surrogate hosts.

4.3.2 Device Components

Figure 4.4 shows Odin2’s Android API, which includes device-only extensions to the messaging and interconnect layers, as well as the device application layer which developers can use to implement Odin2 apps. The Windows Phone API is similar - key differences are discussed in Section 4.7. Application layer components are shown in orange, while messaging and interconnect layer components are shown in green and blue as above.

Central to the device application layer is the OdinService class, which is an Android Service implementation, and as such is managed by the Android operating system lifecycle [77]. Developers gain access to Odin2 functionality simply by making sure the OdinService
Figure 4.3: Key components shared between the Device and Surrogate Host
(User Activities / Services)

<<Interface>>
IODinService

+open()
+close()
+sendAsyncMessage(msg: Message): MessageHandle
+sendRequest(req: Message): MessageHandle
+openStream(name: String, stream: Stream)

BroadcastReceiver

OdinServiceReceiver

+asyncMessageReceived(msg: Message)
+requestReceived(req: Message): Message
+streamOpened(name: String, stream: Stream)

TCPClientChannelProvider

+ctor(host: String, port: int)

<<Interface>>
ChannelProvider

InterconnectDeviceEndpoint

+open()
+close()

<<Interface>>
InterconnectEndpoint

Figure 4.4: Key components located on the device
is started (using \texttt{startService()}). OdinService’s \texttt{Binder}, returned whenever an Android component binds to a service, implements the \texttt{IOdinService} interface, through which developers may gain access to Odin2’s message sending features.

To receive messages, developers may register an instance of \texttt{OdinServiceReceiver} with Android’s \texttt{BroadcastReceiver} framework. The OdinService will broadcast received messages, which will be picked up by all receivers of the appropriate type.

To connect to a Surrogate Host, the OdinService uses the device-only implementations of ChannelProvider and InterconnectEndpoint - namely, \texttt{TCPClientChannelProvider} and \texttt{InterconnectDeviceEndpoint}, respectively. TCPClientChannelProvider takes the desired surrogate host’s address and port as arguments, and creates InterconnectChannel instances backed by TCP Sockets - part of Java’s Sockets API. The provider also may receive broadcasts from the Android OS informing it of changes in network state (e.g. entering or leaving a Wi-Fi zone). This information can be used, for example, to create channels over Wi-Fi instead of 3G when available. As a connection between device and surrogate host is initiated by the device, the InterconnectDeviceEndpoint contains the ability to \texttt{open} and \texttt{close} connections to the host (see section 4.4).

4.3.3 Surrogate Host / Surrogate Components

Figure 4.5 shows components of the Surrogate Host involved with communication between devices and surrogates. As we recall, the surrogate host is built upon the Spring framework, hence the UML diagram in figure 4.5 is stereotyped with Spring entity classes. All classes marked with the \texttt{Bean} stereotype are Spring beans, while \texttt{Repository} stereotypes are specialized beans which use Hibernate to persist data, and some methods of \texttt{Controller} stereotypes are made available to clients as web services.

The \texttt{SurrogateHost} bean provides an endpoint through which devices connect. It maintains a \texttt{TCPHostChannelProvider}, which wraps a Java \texttt{ServerSocket}. Each call to \texttt{accept} blocks until a device establishes a new connection, then returns an InterconnectChannel representing a connection to that device. The \texttt{EndpointRegistry} bean maintains a list of \texttt{InterconnectHostEndpoints} - one for each device connected to the host (each device being represented by a different \texttt{User} entity). An InterconnectHostEndpoint is an InterconnectEndpoint which will not try to re-connect automatically upon discovery of an IO error. Instead, it will lie dormant until a new InterconnectChannel is provided via the \texttt{switchChannel} method. This is because it is always the device’s responsibility to re-establish connectivity after a failure, as its point of network attachment may have changed. Whenever a new channel is opened, the SurrogateHost will check whether the channel’s associated user already exists in the EndpointRegistry. If so, that endpoint’s
Figure 4.5: Key components located on the Surrogate Host involved with surrogate device communication
channel will be swapped for the new one. If not, a new InterconnectHostEndpoint for the new user will be created.

The **DAOService** wraps a MySQL database, via Hibernate. Currently that database stores a list of users, as well as any surrogate-specific data.

When a new device connects, it will provide the name of its surrogate. If that surrogate has not yet been loaded, the SurrogateHost will delegate the loading task to the **SurrogateLoader** bean. This bean may load surrogates by name (in which case a JAR file with the corresponding name will be loaded from a pre-configured location on the host machine), or from a byte stream provided by the device. Surrogates must extend the **Surrogate** abstract class. Each loaded surrogate is stored in the **SurrogateRegistry** bean, mapped by name. Each surrogate instance is shared amongst all devices using surrogates of that name.

Figure 4.6 shows components of the Surrogate Host involved with communication between the host itself (including any surrogates) and clients.

To configure the surrogate host, an administrator may use a web browser to navigate to the URL handled by the **HostWebAppController**. Here, administrators may add or remove users, and upload surrogates to be pre-loaded.

To allow clients to access surrogates, which may be added dynamically, a custom **HandlerMapping** implementation was developed, namely, **SurrogateHandlerMapping**. Within the Spring framework, it is the responsibility of HandlerMapping implementations to delegate the handling of different URLs to the appropriate controllers based on the URL pattern. One of Spring’s default HandlerMapping implementations is the **DefaultAnnotationHandlerMapping** class. This allows controller classes to configure themselves using Java annotations. The DefaultAnnotationHandlerMapping instance then reads these annotations at runtime to determine which controller should process a given URL. However, as the object builds its URL-to-controller mappings at initialization time, it is difficult to include extra controllers which may be added later - such as Surrogate instances. To solve this issue the SurrogateHandlerMapping object was developed, which functions in exactly the same manner as the DefaultAnnotationHandlerMapping class, except that it also uses the SurrogateRegistry bean to obtain Surrogate instances which should handle various URLs. Any requests to **../surrogates/<name>/** will be mapped to the surrogate with the given **<name>** by the mapping object. All annotations supported by Spring controllers are supported by Surrogate implementations and are available for use by surrogate developers.
Chapter 4. Odin2

Figure 4.6: Key components located on the Surrogate Host involved with surrogate-client communication
4.4 Connection and Device Registration

When a host device initially connects to the surrogate host, it must be authenticated, optionally upload its own surrogate, and then undergo a synchronization step to account for previous network and/or device errors.

4.4.1 Initial Handshake

Figure 4.7 shows how a device uses its InterconnectDeviceEndpoint to open a connection to the surrogate host. The important steps are as follows:

1. The device’s endpoint receives a call to `open()`, presumably from the OdinService instance.
2. The endpoint invokes `getChannel()` on a `ChannelProvider` instance, which abstracts away the type of channel, and method of connecting. The provider creates an `InterconnectChannel` instance backed by a Java Socket with its remote endpoint set to the host’s address and port (hosts are assumed to have well-known addresses).

3. The `InterconnectChannel` will open a Socket connection to the host. When the connection is established, the `HostChannelProvider` will create an `InterconnectChannel` instance backed by the Socket representing this connection.

4. Authentication takes place to identify the connecting device. User information is written to the host’s `InterconnectChannel`, which includes username, id, and password hash. The host channel will authenticate this information using the `SurrogateHost` bean. At this time, the device also informs the host of the desired surrogate, either via a provided surrogate name (shown here) or a byte stream containing the surrogate data.

5. Once authentication is complete, an `OK` message is written back to the device’s `InterconnectChannel`, which completes its construction. The device’s `ChannelProvider` then returns the created channel to the device endpoint. The device endpoint then sends its sequence numbers to the host for synchronization (see section 4.4.2).

6. Concurrently, the host has used the created host `InterconnectChannel` to create a new `InterconnectHostEndpoint`, as in this sequence the device has not previously been connected. As part of the host endpoint construction process, sequence number synchronization takes place.

Following the creation of the `InterconnectDeviceEndpoint`, the `OdinService` will register itself as an `EndpointCallback` and then be able to send and receive messages on behalf of Odin device apps. Following the creation of the `InterconnectHostEndpoint`, the `SurrogateHost` will store it in the `EndpointRegistry`, mapped to the user information that was provided by the device during the initial handshake.

### 4.4.2 Sequence Numbers and Synchronization

Both the device and surrogate host endpoints keep track of three important numbers:

1. `lastAttempted`: The sequence number of the last message sent to the recipient (regardless of whether receipt has been acknowledged).

2. `lastSuccessful`: The sequence number of the last message we know to have arrived at the recipient (due to a received acknowledgment).
3. *lastReceived*: The sequence number of the last message we received from the sender.

   This is used to avoid duplicate or out-of-order messages (see section 4.5).

Sequence number synchronization is necessary in the most extreme error case, where a device completely crashes. If this occurs, to the host it would appear no different than an IO error - it will happily buffer new messages until a connection is restored. However, if we simply reset the device to the default starting sequence number (zero), then all messages will be dropped by the host as they will be assumed to be duplicates. Furthermore, the any messages from the host to the device will be buffered at the device indefinitely, never being delivered to the application layer, due to the non-zero sequence numbers of these messages being considered “too high” (causing the device to wait for the expected sequence number of zero before delivery).

Allowing synchronization of sequence numbers whenever the device opens a new endpoint, in conjunction with the fact that sequence numbers and pending messages are persisted in an SQLite database on the device, is enough to ensure that messages are assigned their correct sequence numbers unless the device’s database becomes corrupted. We have determined that the chance of this happening is sufficiently rare that we do not account for it in Odin2’s design.

### 4.5 Messaging

Odin2 allows surrogates and host devices to communicate via a messaging abstraction. Either party may send messages synchronously (requiring a response) or asynchronously. Furthermore, devices and surrogates may, at any time after a message has been queued for delivery, query that message’s delivery status. Possible states of delivery include:

1. **Pending**: The message has been queued for delivery but an attempt has not yet been made to deliver the message.

2. **Sending**: The first attempt to send the message across the interconnect will shortly be made.

3. **Sent**: An attempt has been made to send the message across the interconnect. The attempt completed without fault, but message delivery has not yet been guaranteed.

4. **Received**: Receipt of the message has been confirmed by the recipient, but the message has not yet been delivered to the application layer.

5. **Delivered**: The message has been delivered to the application layer on the recipient.
6. **UserAcknowledged**: A special form of delivery guarantee that allows us to determine when a certain user action has been executed on the message. As an example, consider an instant messaging application which provides a user with the knowledge of whether the recipient has “seen” a particular message.

![Sequence diagram](image)

**Figure 4.8**: Sequence diagram showing a high-level view of request-response messaging between device and surrogate under ideal circumstances.

Figure 4.8 shows a sequence diagram detailing the high-level communication between different layers on the device and surrogate host, when the device sends a request message to the surrogate and expects a response. The communication flow shown in the diagram assumes no communication errors. The path of the Request message, Response message, and delivery notifications are color-coded in green, blue, and red text, respectively. Note that the surrogate may use this same process to send messages to the device—messaging is two-way and may be initiated by either party. The steps involved with this sequence are described here, with reference to the numbered items in the figure:

1. The device app queues a request message to be sent to the surrogate host. Due to Odin2’s asynchronous nature, this request does not block until a response is
obtained - instead, a handle is returned to the app. This handle may be used to query for the current delivery status of the associated message, to wait for a particular state of delivery, and to wait for and retrieve a response, in the case of a request message. Immediately upon the queuing of the message, this delivery status is set to Pending.

2. Some time in the future, the messaging layer will ask the interconnect layer to send the request to the host. Before it does so, the message’s status is changed to Sending. This change is immediately reflected in the handle that was originally generated for the app. Following this, the interconnect layer attempts to send the message across the network medium. Assuming this does not immediately fail, the message’s status is changed to Sent. Note that at this stage, we cannot guarantee that the message has reached the host. Even in the case of TCP, a reliable communication protocol, Java’s Socket API will allow write operations, without failure for a significant period of time after an error has occurred (upwards of 20 seconds in some cases).

3. Assuming the request makes it to the host, the host will immediately send acknowledgment of receipt back to the device. We will assume for this case that the acknowledgments arrive successfully (see section 4.6 for more information on what happens when they do not, or when other errors occur). On receipt of the acknowledgment, the message’s status will be set to Received. This lets the device know that the request has arrived successfully at the surrogate and is waiting in its internal buffer for eventual delivery to the surrogate.

4. After notifying the device of message receipt, the host’s interconnect layer will send the request to the messaging layer, which will queue it for eventual delivery to the surrogate.

5. Eventually, the request is sent to the surrogate for processing. Immediately following this, the host will send a delivery notification back to the device, which will set the message’s status to Delivered. This status guarantees that the user-code on the surrogate has been allowed to execute on the request.

6. Eventually, the surrogate will produce a response message. In this diagram, the sending of the response is greatly simplified. In reality, the process of sending the response back to the device is the same as for the request, but in reverse (i.e. the surrogate sends the response message and receives delivery status notifications regarding that message). If the request was marked as requiring user acknowledgment, a UserAcknowledged status would also be sent at this time (not shown in this sequence diagram).
7. On the device, as the initial method to enqueue the request returns immediately, the device’s main thread may continue to execute arbitrary code until such time as it requires the response.

8. Once the user code on the device requires access to the response, a blocking call may be made to wait for the response. This method returns immediately upon receipt of the response message. Note that any message sender may wait, not only for a response, but also for any of the possible message statuses (Pending, Sending, Sent, Received, Delivered, UserAcknowledged).

4.5.1 Components Involved in Messaging

![Diagram of messaging and interconnect layers](image_url)

**Figure 4.9:** Overview of Odin2 Messaging and Interconnect layers

Figure 4.9 shows the components within the messaging and interconnect layers involved with messaging. The blue boxes and arrows in the diagram represent components and
data flow related to the *sending* of messages, while the green boxes and arrows relate to *receiving* messages. Certain components such as the `MessageFactory` and `ReceiverThread` are involved with both, and as such are shown multicolored in the diagram.

The primary component facilitating access to Odin2’s messaging functionality is the `InterconnectEndpoint`, which contains methods to send messages, and callbacks to receive messages and delivery notifications. Within the endpoint, an `InterconnectDataStore` contains a buffer of all messages to be sent to the recipient. Within the data store, each message is associated with a *sequence number* as well as a *timestamp* indicating the last time an attempt was made to send that message. The sequence number is sent along with the message to allow the recipient to determine the correct message order, while the timestamp is used to detect whether further attempts should be made to send that message (if a significant period of time has passed without receiving any acknowledgment of message delivery). A `DeliveryNotifier` component receives delivery notifications from the interconnect layer and removes messages from the data store once their delivery is confirmed.

When the application layer sends a message, the endpoint generates a *Handle* for that message. These handles are kept in a separate `MessageHandles` data structure within the endpoint, indexed according to the sequence number of the corresponding message. The handle for the specific message is also returned to the application layer where user code can access it to get the current delivery status of the message, wait for a particular delivery status, and / or wait for a response in the case of a request message. The DeliveryNotifier also notifies the MessageHandles when a new delivery status is received, so that the corresponding handle can be updated. The MessageHandles data structure stores handles as *weak references*, which allows them to be garbage collected by the system when not referenced anywhere else. This can happen when the user’s code doesn’t retain a reference to the handle (perhaps because they do not require access to its information). In this way, we conserve memory while allowing the system to automate the removal of “dead” handles.

When the interconnect layer receives a message from the sender, it will pass this message to the endpoint where it is stored in the `ReceiveBuffer`. The buffer maintains an *expected sequence number* - the sequence number of the next message that should be received. Messages arriving with lower sequence numbers than this are discarded as they are considered to be duplicates, while messages with numbers higher than this are buffered until the expected message arrives, at which time that message and each message with a higher contiguous sequence number is sent to the application layer. For example, if the expected sequence number is 5, then we receive message 3, that message will be dropped. If we then receive message 6, 7, and 9, those messages will be buffered. If we
then receive message 5, then messages 5, 6, and 7 will be forwarded, and the expected sequence number will be updated to 8. Message 9 will remain buffered until such time as message 8 arrives. This mechanism ensures that the application layer always receives messages in the same order that they were sent. This mechanism is necessary even in the case of a TCP connection with reliable ordering guarantees. For example, consider the case where message 5 is sent after a network error occurs but before it is detected, then message 6 is sent after the error is resolved, then message 5 is resent because it was determined not to have reached the recipient. In this case, the recipient will receive message 6 before message 5.

Also involved with messaging is the InterconnectChannel, which is responsible for marshalling and unmarshalling messages and delivery notifications, and writing them to / reading them from the interconnect medium. Each InterconnectChannel contains a MessageSendThread responsible for sending messages, and a ReceiverThread responsible for receiving messages and notifications. These are necessarily background threads, to increase application responsiveness by moving expensive network logic away from the UI thread. The InterconnectChannel delegates the task of marshaling and unmarshalling messages back to the messaging layer’s MessageFactory component.

Referring to figure 4.9, the processes involved with the sending of messages can be seen when examining the blue numbered arrows. Numbers (1) through (9) show the process of sending a message, while numbers (10) through (13) show the process of receiving a message delivery notification. The process shown in the diagram assumes no network errors.

When sending a message, the following occurs:

1. Initially, the application layer uses its reference to the InterconnectEndpoint to send a message. This message is immediately buffered in the InterconnectDataStore, and it is assigned a sequence number. When a new InterconnectEndpoint is created, the first valid sequence number is obtained (see section 4.4). Numbers then increase by one for each new message to be delivered. The message is also assigned an initial timestamp of zero to indicate that no attempt has yet been made to send the message.

2. A Handle is created for the message and stored in the MessageHandles table, indexed by its sequence number. Its initial delivery status is set to Pending and its initial response to null. Once created, the handle is returned to the application layer.

3. On a separate thread, the MessageSendThread within the InterconnectChannel will block until there exists a message within the InterconnectDataStore that has
not yet been sent, or a message which was sent more than 3,000 milliseconds ago for which we have not yet received a delivery confirmation. Once the data store contains one or more such messages, the sender thread will obtain all these messages. At this stage they are not removed from the datastore, however, their timestamps are set to the current time to indicate a delivery attempt.

4. To send a message across the interconnect medium, the message must be converted into a stream of bytes. The MessageSendThread delegates this task to the MessageFactory.

5. Before sending the message, the endpoint’s DeliveryNotifier component is notified that the send is about to occur.

6. The DeliveryNotifier, upon receiving notification of a message about to be sent, will locate that message’s handle and set its delivery status to Sending, assuming its status is not already Sent. This could occur if this is not the first time delivery of this message has been attempted.

7. Once marshaled, the serialized message is sent across the interconnect medium. For the TCPInterconnectChannel, this directly results in a write call being made on a TCP Socket. A detectable IO error may occur here, in which case the channel will close itself and notify the endpoint (see section 4.6).

8. If a detectable IO error did not occur when sending the message, the DeliveryNotifier is notified.

9. The DeliveryNotifier, upon receiving notification of a sent message, will locate that message’s handle and set its delivery status to Sent.

The message will eventually reach the recipient and then be delivered to its application layer. As a result of this, delivery notifications will be sent from the recipient back to the sender. At this time:

10. The InterconnectChannel’s ReceiverThread is constantly reading bytes from the interconnect medium. If an IO error is ever detected here, the channel will close itself and notify the endpoint (see section 4.6). Assuming no IO errors, the ReceiverThread will first read a header indicating whether the next bytes are a delivery notification or an incoming message. Assuming the incoming bytes correspond to a delivery notification, then additional information is read from the stream, including the sequence number of the message to which the notification applies, and the actual notification itself (consisting of a single byte - sufficient to represent the six possible types).
11. The ReceiverThread then forwards the notification to the endpoint’s DeliveryNotifier.

12. The DeliveryNotifier updates the corresponding message’s delivery status within its handle. The possible delivery notifications here include Received, Delivered, and UserAcknowledged, each stronger than the last. The notifier won’t update a handle with a weaker status than it already has. This could occur, for example, when a message is received by the recipient, and a Received status is returned, but lost due to a network error. The sender will resend the message due to the acknowledgment not being received, which will result in additional notifications being returned. The order of received notifications could be a Delivered or UserAcknowledged notification from the first attempt, followed by a Received notification from the second attempt, which would result in an inconsistent state if we did not perform this check.

13. In the case of a Delivered or UserAcknowledged notification, the message is deemed to have successfully reached, and been processed by, the recipient. At this stage it is deemed safe to remove the message from the sender’s buffer, and thus the InterconnectDataStore is notified to do this.

Referring to figure 4.9, the processes involved with the sending of messages can be seen when examining the green numbered arrows. Numbers (14) through (21) show the process of receiving a message, and sending appropriate delivery notifications back to the sender.

14. The InterconnectChannel’s ReceiverThread is constantly reading bytes from the interconnect medium. Assuming no IO errors, the ReceiverThread will first read a header indicating whether the next bytes are a delivery notification or an incoming message. Assuming the incoming bytes correspond to an incoming message, additional header information will be read, including the message’s sequence number, message type, length in bytes, and whether it requires a response. The message body is then read into a byte buffer.

15. The ReceiverThread delegates the task of unmarshalling the message to the MessageFactory.

16. Once the received Message object has been created, a Received notification is written back to the interconnect medium.

17. The received message is then passed to the InterconnectEndpoint, where it is buffered in the endpoint’s ReceiveBuffer. The ReceiveBuffer may discard the message if it is a duplicate, or hold the message until such time as it is ready to be delivered to the application layer.
Chapter 4. Odin2

18. Once the ReceiveBuffer determines the message can be delivered, it will be sent to the application layer.

19. If a response is required, it will be generated by the application layer and sent in the same manner as any other message.

20. The ReceiveBuffer will notify the InterconnectEndpoint that the message has been delivered.

21. The InterconnectChannel will write the Delivered status notification back to the sender.

4.6 Error Handling

Odin2 uses several methods to ensure message delivery in the case of network errors. Firstly, Odin2’s messaging system includes the ability for any interested component to ascertain whether a particular message has been successfully delivered (see section 4.5). Odin2 itself uses these notifications to determine if a message may have been lost due to a network error - if an acknowledgment is not received within a certain timeframe, a message is scheduled for redelivery. To deal with the case where the message was initially received, but its acknowledgment was lost, each message is assigned a sequence number. Messages arriving with lower-than-expected sequence numbers are detected as duplicates and dropped. The sequence number mechanism additionally deals with out-of-order messages - messages arriving with higher-than-expected sequence numbers are buffered until the missing messages arrive, then delivered to the application layer in-order.

The concept of sequence numbers originated from the Transmission Control Protocol (TCP) [78]. However, we implement our own sequence number system to provide an additional layer of reliability on top of TCP. Firstly, our experiments have shown that we cannot guarantee that data has reached the recipient if the socket’s write operation succeeds without error - recently terminated connections have been shown to exhibit this behavior. Furthermore, we wish to maintain a message sequence across multiple TCP connections, as these may be broken at any time due to network errors. Moreover, Odin2 is extensible with respect to its interconnect channels. While only a TCP channel is currently implemented, developers may wish to add support for other protocol types.

To detect network failures, Odin2 largely relies on the ability of the underlying operating system to report IO errors in the form of IOExceptions, thrown when attempting to read or write from a broken TCP socket. Any IOException thrown will cause the currently
open InterconnectChannel to be forcibly closed, and a new one obtained. Whereas Odin’s original prototype used its own logic to determine to which wireless network it should connect, Odin2 now relies on the underlying operating system to do this as such support is included in current versions of both Android and Windows Phone. On both platforms, when a more preferred network becomes available (such as a known Wi-Fi network instead of a 3G network), the operating system forcefully closes all connections over the former network, while allowing all new connections to be opened over the new one. Odin2 detects this as a network failure, and handles this case in the same manner as a real network failure (i.e. by asking the OS to provide a new connection).

In some network failure cases, IOExceptions are not thrown by the operating system in a timely manner, and additional logic must be employed in order to avoid an unsatisfactory period of disconnectivity. As an example, consider the case where a 3G network is being used. Mobile network operators often make such connections “stale” after unspecified periods of inactivity. Our tests have shown that attempting to communicate over a stale connection results in read or write operations blocking indefinitely with no exceptions. To mitigate this, a heartbeat mechanism is used. Periodically, Odin2 devices will send a heartbeat to the host, which will respond with a heartbeat acknowledgment. The host is aware of how often it should be receiving heartbeats. If the device does not receive a heartbeat acknowledgment within a certain period of time, its currently active InterconnectChannel will be forcibly closed, and a new one will be obtained. Likewise, the host will forcibly close an InterconnectChannel if heartbeats for that channel have not been received within the agreed-upon time period. The heartbeat period is determined dynamically according to the type of the current network (Wi-Fi or 3G) and the amount of activity on the channel. While sending messages, heartbeats are unnecessary and are suspended until periods of inactivity. Wi-Fi channels are not subject to being made stale, and as such these heartbeats occur at a greatly reduced rate when compared with 3G. We have set the heartbeat period during inactive messaging sessions at 3000ms over 3G and 20000ms over Wi-Fi.

4.7 Cross-Platform Support

Odin2 supports both the Windows Phone and Android platforms. Surrogates are always written in Java, and hosts may be deployed on any web container. To allow communication between devices, which may not be written in Java (Windows Phone uses C#), it was important to use a cross-platform message serialization approach rather than rely on a platform-dependent mechanism. For this, JSON is used as it is widely supported on
multiple platforms. Each platform has a JSON marshaling / unmarshalling API which can be used to translate between JSON and platform-specific objects.

In addition, when developing a cross-platform API, it was necessary to take into account various platform differences. Of particular importance is the difference between Android and Windows Phone’s support for background tasks. In this case, a background task refers to any code that is allowed to execute while the application is not in the foreground on the user’s device. Android Services allow arbitrary background tasks to execute at any point - there is no restriction. Services must take care to avoid consuming too many resources while running in the background, or the operating system may kill services to conserve memory or CPU bandwidth. However, on modern smartphones this limit far exceeds the resources required by Odin2. With Windows Phone, there are much stricter limits on background tasks. The most common background task is one which is allowed to execute periodically by the OS (around once every 30 minutes) and must finish in a timely fashion and use a small amount of CPU and memory. This is insufficient to allow the ongoing network communication typical of Odin2 applications. In fact, when Odin2 applications on Windows Phone move to the background, there is no way to continue timely, two-way communication. However, limited network communication is allowed in the form of push notifications. An app may be configured to receive push notifications containing arbitrary data. When a notification is received, an app is allowed to process the data (though severe CPU and memory constraints apply) and also notify the user via a toast message. If the user responds to the toast, the application will be brought to the foreground and full processing will be allowed to resume.

Odin2’s surrogate host can send push notifications to Windows Phone devices whose Odin2 apps are running in the background, using Microsoft’s Push Notification Service. It does so whenever that app’s surrogate sends a message to the device. When the device app is in the foreground, communication occurs exactly as with the Android platform (the steps of which are detailed in Section 4.5). However, when the app is in the background, a push notification containing the message will be sent instead. When the device receives this notification, it will be allowed to perform processing that does not involve updating the UI. It will then send a toast message to the user to notify them that their attention may be required.

Another platform difference is the lack of an equivalent to Android Services within the Windows Phone APIs. To overcome this limitation, the OdinService class for Windows Phone is implemented as a Thread. Rather than invoking the service using the startService method, and receiving a callback from the operating system when the service is established as occurs with Android services, we manually start a new Windows Phone Thread running
the OdinService instance, and then invoke its methods directly rather than through a service binder.

### 4.8 Summary

In this chapter we have introduced Odin2, which draws upon lessons learned from our initial prototype design to create a mobile services middleware sufficient for real-world adoption. Tables 4.1 and 4.2 show how Odin2’s design meets each of its functional and non-functional requirements, respectively.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>How is requirement met?</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR #1</td>
<td>Odin2 allows external clients to consume Odin2 services via REST services and web sockets. Clients need only know the address of the surrogate host, which is assumed to be well-known. Internal clients (i.e. other clients using the Odin2 middleware) may consume services via Odin2’s interconnect.</td>
</tr>
<tr>
<td>FR #2</td>
<td>For external clients, Odin2 allows request-response and publish-subscribe semantics via REST services, and streaming via web sockets. For Odin2 applications, Odin2’s interconnect offers request-response and asynchronous messaging (can be used for publish-subscribe). Streaming can be achieved by chaining messages. Odin2’s interconnect additionally allows multicasting of messages.</td>
</tr>
<tr>
<td>FR #3</td>
<td>Odin2’s interconnect achieves this through the tracking of message sequence numbers. Further details are available in sections 4.5 and 4.6.</td>
</tr>
<tr>
<td>FR #4</td>
<td>Odin2’s interconnect manages the delivery of message status updates (see section 4.5). Furthermore, service developers may send “user acknowledged” status updates themselves in response to a certain event, to indicate that a particular message has had the user’s attention.</td>
</tr>
<tr>
<td>FR #5</td>
<td>Odin2 allows developers to persist data and execute code on the surrogate, which is hosted within a fixed-network web container along with the surrogate host. The code and data to be located here is arbitrary.</td>
</tr>
</tbody>
</table>

**Table 4.1:** Summary of how Odin2’s design meets its functional requirements

In Chapter 7 we present a case study showing how Odin2 can be used as the underlying middleware for a complex real-world software system. Chapter 8, we evaluate Odin2 with respect to its performance, expressiveness, and dependability. An overview showing how to create a simple Odin2 mobile service for both Android and Windows Phone is given in Appendix A.
<table>
<thead>
<tr>
<th>Requirement</th>
<th>How is requirement met?</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFR #1</td>
<td>Odin2’s performance has been evaluated and this non-functional requirement has been found to hold when compared with various push notification systems in use today. The full evaluation is available in Chapter 8.</td>
</tr>
<tr>
<td>NFR #2</td>
<td>We have not run any comprehensive scalability evaluation. However, in Chapter 8 we show that Odin2 is scalable enough that, when the surrogate host is deployed on a single mid-range virtual machine, Odin2 supports significantly more than the maximum number of devices required at any given time by the REMOTE-CR case study. Furthermore, the surrogate host can be deployed in the cloud, allowing it to leverage the scalability options inherent in these platforms.</td>
</tr>
<tr>
<td>NFR #3</td>
<td>When Odin2 devices lose internet connectivity, any messages on either side are buffered by the interconnect, ready to be sent when connectivity is restored. Furthermore, clients remain connected to surrogates. These may, at the developer’s discretion, offer cached data and / or inform the client that the device is currently unavailable (surrogates are aware of when their associated devices are connected).</td>
</tr>
<tr>
<td>NFR #4</td>
<td>Odin2’s APIs have been developed for both the Android and Windows Phone platforms. The Windows Phone API takes into account platform differences when executing background tasks.</td>
</tr>
<tr>
<td>NFR #5</td>
<td>Odin2’s underlying framework, Spring, is far more widely used compared with Jini. The same is true regarding REST services and web sockets - the means by which clients consume Odin2 services.</td>
</tr>
<tr>
<td>NFR #6</td>
<td>Clients may consume Odin2 services using REST and / or web sockets. Neither of these protocols are specific to Odin2 in any way, and are both widely used.</td>
</tr>
</tbody>
</table>

Table 4.2: Summary of how Odin2’s design meets its non-functional requirements
Chapter 5

OdinTools: A Model-Driven Tool for Cross-Platform Development of Odin2-based Applications

One of our main research goals is to investigate means by which we can allow developers to increase their productivity while developing mobile services. To this end, Odin, and even more so Odin2, allow developers to build their mobile service applications without concern for issues such as device reachability or mobility. Moreover, Odin2 has been implemented for both the Android and Windows Phone platforms, and its APIs remove the need for developers to worry about certain differences between the platforms, such as the way each handles background tasks (see Chapter 4 section 4.7 for details). However, using Odin2, developers wishing to support multiple platforms must still manually write code for each of those platforms plus a Surrogate class to be provided to Odin’s Surrogate Host. This can potentially be a time-consuming process for applications of even a moderate size.

As a possible solution, we have reviewed the literature associated with Model-Driven Engineering (MDE), which has been shown to assist with cross-platform development. We have also investigated the current state-of-the-art tools used for cross-platform development of mobile applications. The literature review for these subjects can be found in Chapter 2. From our review, the Model-Driven Architecture (MDA) technique, as proposed by the Object Management Group [63], was shown to be an effective paradigm at aiding cross-platform development. We have developed an exploratory prototype tool, OdinTools, which follows the MDA paradigm and allows developers to use a Visual Language to partially specify a mobile service. Several essential service components are automatically generated for multiple platforms using OdinTools. As an example, the
Android and Windows Phone platforms were chosen. Each is representative of different programming platforms and has different characteristics.

The remainder of this chapter is organized as follows. Section 5.1 shows the models and transformations that exist within OdinTools, and shows how these are implemented using the Visual Studio Modeling SDK, our chosen development environment. Then, section 5.2 shows how an example mobile service can be implemented with the help of OdinTools, prior to the chapter’s conclusion in section 5.3.

5.1 Design

OdinTools allows developers to model certain parts of an Odin application and will then generate a partial implementation for the surrogate as well as Windows Phone and Android devices. OdinTools itself follows a model-driven approach, with its models and transformations closely resembling those detailed in the Model Driven Architecture (MDA). The MDA is a well-known MDE framework released by the Object Management Group (OMG) [63]. OdinTools has undergone several revisions, with the most current prototype being implemented using the Modeling SDK for Visual Studio 2013\(^1\). Section 5.1.1 details the models that comprise OdinTools, followed by a description of its implementation in section 5.1.2.

5.1.1 Model Design

5.1.1.1 Overview

OdinTools allows developers to model certain parts of an Odin application and will then generate a partial implementation for the surrogate as well as Windows Phone and Android devices. Developers may use OdinTools to specify: i) the different kinds of data required by an Odin application, ii) the types of messages the application should be able to handle, and iii) certain persistence semantics for its data. The information required to specify this information is modeled using OdinTools’ top-level model, the Odin Application Model (OAM). This model is the only one which is exposed to developers.

As shown in figure 5.1, there are additional levels of models, which are automatically generated by OdinTools during the code generation process. Initially, a Platform-Independent Model (PIM) is generated from the OAM. Within this model, we introduce the concepts of devices and surrogates, without yet specifying the exact implementation requirements of

those devices and surrogates. From this model, several Platform-Specific Models (PSMs) are generated. Each PSM consists of the source code, API, and configuration files for either the Surrogate or a particular mobile platform. These files can then be imported using the appropriate tool to generate the final implementation for each platform.

OdinTools can be described as a simplified implementation of the Model Driven Architecture (MDA), with its models and transformations closely resembling those detailed in this methodology. MDA is a well-known MDE framework released by the Object Management Group (OMG) [63]. Specifically, OdinTools’ OAM matches closely with MDA’s Computationally Independent Model (CIM). Traditionally this is the model with the highest level of abstraction - any developer using this model would require little knowledge of the underlying platforms or APIs. OdinTools’ PIM again matches closely with MDA’s PIM. This level of model may introduce some platform-specific concepts (in our case, surrogates and devices), but without details on how those specific concepts are implemented. The source code generated by OdinTools, and the subsequent implementation artefacts generated by the appropriate platform-specific tools, can be compared
with MDA’s PSMs and Implementation, respectively.

5.1.1.2 Odin Application Model

Figure 5.2 shows the Odin Application Model that’s specified by developers. A particular OAM implementation will contain an application’s data types, along with (optionally) the kinds of messages to be handled and information regarding data persistence. Developers may specify just the data types, or may also specify the data persistence and / or message handling information. The extent to which a developer specifies each of these three sections determines which sections of code will be generated in the final implementation.

![Diagram of Odin Application Model](image)

**Figure 5.2: Overview of OdinTools’ Odin Application Model (OAM)**

Each OAM implantation contains a single *OdinApplication* instance which acts as a container for all other generated or developer-specified model entities. A single *Database* instance is also generated and added to the model upon creation. This may be used by developers to specify data persistence (see below). Several *SystemType* instances are also generated and added to the model (also see below).
Each OdinApplication instance contains a list of types. These types are specified by DataType model entities and represent the kinds of data in an application. When an OdinApplication is created, its types list is pre-populated with a list of SystemType instances. These represent common data types likely to be used by many Odin2 applications, such as GPSLocation, and include primitives such as integer and string. In addition, developers may specify new data types for a particular application by adding CustomType instances to the model. Each CustomType has a unique name along with a list of Property instances. A CustomType’s properties represent the kinds of information contained within that type. Each Property has a name of its own, along with its own type, which can be any DataType specified within the model. This way, a developer can build a hierarchical data structure for their application within OdinTools.

To specify data persistence, two options are available. Primarily, developers may build a database. This is done by adding any declared CustomTypes to the Database instance’s tables list. Adding a CustomType to this list will allow objects of that type to be persisted in a database. Properties in persisted CustomTypes will be represented either by table columns or other tables, depending on whether the Property is typed with a SystemType or CustomType itself. Another way of specifying persistence is to add Property instances directly to the OdinApplication’s properties list. Properties added in this way will be persisted in an application’s configuration settings, rather than a database. Developers are free to use one or both of these methods for data persistence.

Each entry in the Database’s tables list and the OdinApplication’s properties list is associated with a locality, which can be specified as local, remote, global, or auto (default). This property specifies where a particular type is persisted. Local types are persisted at the device, remote types are persisted at the surrogate, and global types are persisted at both locations (with extra logic introduced for synchronization). Currently, auto will default to local, but in the future we plan to automatically determine the optimal locality based on the work in [7]. For more information, see section 5.1.1.3 below.

Finally, the OAM may be used to specify the kinds of messages to be handled by the application. Developers do this by adding Message instances to the OdinApplication’s handledMessages list. A method added here may be an AsyncMessage or a RequestMessage, corresponding to the different kinds of messaging semantics available in Odin (see Chapter 4 section 4.5 for further details). Each message contains a number of Property instances, to be specified by developers, which determine the kinds of information to be sent in the message. If the message is a RequestMessage, then an additional AsyncMessage must be defined as that message’s response. Similarly to database tables and application properties, messages also have a locality which determines where they are handled.
5.1.1.3 Platform-Independent Model

Figure 5.3 shows OdinTools’ Platform-Independent Model (PIM). This model is automatically generated by OdinTools, given a completed OAM. In the figure we can see that the Locality enumeration, along with all its references, have been deleted. The primary purpose of the PIM is to introduce the concepts of devices and surrogates - that is, we transform the developer’s locality specification into the structure of the PIM. For this purpose the OAM’s OdinApplication domain class is replaced with the OdinAppComponent class, from which three child domain classes inherit. The Surrogate and Device classes represent the data persisted in, and messages handled by, the device and surrogate, respectively. The Globals class contains the data and messages to be handled globally, along with all DataTypes (which are always shared between the device and surrogate). Each of these three components are assigned a Database model entity to represent the data that will be persisted at each location.

To transform the OAM into the PIM, firstly we create a new PIM instance, with a single instance of each OdinAppComponent child entity (Globals, Device, and Surrogate). Three Database entities will also be created, and one will be assigned to each of these three OdinAppComponents. Next, for each DataType contained within the OAM’s OdinApplication’s types list, a corresponding PIM DataType instance will be created and added to the types list of the Globals entity.

Next, a new PIM Message instance (either an AsyncMessage or RequestMessage) is created for each OAM Message in the OdinApplication’s handledMessages list. Each Property of the OAM messages is copied and assigned to the corresponding PIM message. In the case of RequestMessages, the AsyncMessage representing the response is also copied. Each Message is then added to the handledMessages list of either the Globals, Device, or Surrogate PIM entity depending on the value of the OAM message’s locality domain property: local, remote, or global message localities will be assigned to the Device, Surrogate, or Globals entity, respectively. Currently, any message with an auto locality is also assigned to the Device entity.

Similarly to the above method for adding Messages to the PIM, any DataTypes in the OAM with entries in the tables list of the OAM’s Database entity will have a corresponding entry added in the tables list of the Globals, Device, or Surrogate entity in the PIM, depending on locality.

Any Property in the OAM OdinApplication’s properties list is added to the properties list of either the Device or the Surrogate entity in the PIM, again in a similar manner to above. Currently the OdinTools prototype doesn’t support the generation of global
properties. Any OAM property defined as global will be added to the Device at this stage.

Finally, new messages are added to the PIM. For each DataType listed in the Globals Database, a `DataChangeMessage` instance (a subclass of AsyncMessage) will be created. A DataChangeMessage will be created with a single Property with the name "newValue" and the type of the corresponding DataType. DataChangeMessages will be sent from a device to the surrogate when the corresponding table in the device’s database changes, and vice versa.
5.1.1.4 Platform-Specific Models

Following the generation of the PIM, a Platform-Specific Model (PSM) for the surrogate and each supported device platform is generated. Currently, supported device platforms include Windows Phone 7 and Android. Rather than generate another set of model instances based on custom metamodels, we use the existing development language and APIs for the target platforms (namely C# and .Net libraries for WP7, Java and Android APIs for Android, Java and Spring APIs for the surrogate) as the metamodels as we saw little to be gained in adding an additional layer of abstraction at this point. Consequently, the PIM-to-PSM transformation in this case results in a set of source code and project configuration scripts being generated for each platform. These can then be loaded into the appropriate tool (Visual Studio or Eclipse) to generate the final, runnable device and surrogate packages.

For the surrogate and each mobile device platform, a folder is created. Within each of these folders, the following is created:

Android Device PSM Within the Android folder, initially, project configuration files are generated for Eclipse. These will allow the contents of this folder to be loaded as an Android application within the Eclipse IDE. The project is configured to contain a reference to the Odin2 Device API for Android. A folder structure is also created within the Android folder, to represent the application’s Java package structure. The top-level package `odin.<appname>` is created, with the sub-packages `data` and `messaging`.

For each CustomType in the types list of the Globals entity in the PIM, a corresponding Java class is created with the same name and added to the data package. Each Property of that CustomType is added to the generated Java class, in the form of a private member, and a public getter and setter method. For each of these types which is also listed in the `tables` list of either the Device or Globals PIM entity, additional code will be added to the corresponding Java class in the form of Java annotations for ORMLite [81]. ORMLite is an Object-Relational Mapping (ORM) implementation, designed to allow Android applications to more easily translate between objects and database schemas. Adding ORMLite annotations allows objects of a class to be persisted in Android SQLite databases. If any CustomTypes require the addition of ORMLite annotations, a reference to the ORMLite API is also added in the project configuration script. If the CustomType is listed in the Globals list, an additional Java class will be created and made to extend Odin2’s Message class. The class will be named `XDataChangedMessage`, where `X` is the name of the CustomType. Within each generated setter of the CustomType, additional code will created which will use Odin2’s OdinService class to send the corresponding DataChangedMessage as an asynchronous message to the surrogate. Additionally, a
new method will be created within that CustomType, called \textit{dataChanged}, to be called elsewhere (see below) when that DataChangedMessage is received from the surrogate.

To support messaging, a \texttt{MessageSenders} class will be generated and added to the messaging package. The generated contents of this type will depend on the defined messages, as explained further below.

To support storage of app properties, a Java class called \texttt{AppProperties} is generated and added to the top-level package. For each Property in the PIM Device’s properties list, two methods will be added to this class: \texttt{setProperty\_X} and \texttt{getProperty\_X}, where \texttt{X} is the name of that property. These methods will utilize Android’s \texttt{SharedPreferences} class to get or set the values of these properties.

For each Message listed in the \texttt{handledMessages} list of any of the three OdinAppComponent entities, a Java class representing that message will be created and added to the messaging package. The generated class will be made to extend Odin2’s \texttt{Message} class (regardless of whether the message is request-response or asynchronous). Properties will be added to the generated message similarly to CustomTypes above. For each RequestMessage generated in this way, its associated response AsyncMessage will be created in the same manner.

For each defined Message (not including any response messages), a method will be generated in the MessageSenders class. For a message named \texttt{X}, the method will be named \texttt{sendMessage\_X}. The method will take two parameters: an object of the corresponding message type, and a list of Odin2 \texttt{User} objects representing the users to which the message should be sent (see Chapter 4 section 4.4. The generated method will utilize Odin2’s \texttt{OdinService} Android service to send the message as an asynchronous message or request message as appropriate. The message will be sent to all given users, or to all users if the users parameter is null. If the message is a RequestMessage, the generated method will additionally contain code to wait for a response to be received, then return that response. Otherwise, the method will be generated as a void method and will return immediately upon sending the message.

If the GPSLocation SystemType is used anywhere within the PIM, an additional Android Service class, called \texttt{LocationService}, will be added to the app’s top-level package and configuration file. This class will utilize the Google Play Services APIs to detect when a user’s location changes, and forward this event to a registered \texttt{LocationChangeHandler} - an interface which will also be added to the generated application.

Finally, a class will be generated to initialize the app. The class will be named \texttt{XMainActivity}, where \texttt{X} is the name of the user’s app. It will be placed in the top-level package, made to extend Odin2’s \texttt{OdinActivity} class, and will be configured to be the main activity
which runs when a user starts the app. Within this class’ onCreate method, code will be added to configure the app’s SQLite database based on the data types to be persisted (see above). For each message defined in the Globals or Device PIM entities, a method will be generated in the main class. For a message named X, the method will be named handleMessage_X. The method will take two parameters: an object of the corresponding message type, and a User object indicating the sender of the message. If the message is a RequestMessage, the method signature will be generated to return an object of the appropriate response message type. Otherwise, the signature will be generated as a void method. These methods will be left as skeletons for the developer to implement, and will be called by the main class’ handleAsyncMessage and handleRequestMessage methods (required by OdinActivity), which will also be generated. These methods will examine incoming messages and forward them to the appropriate message handler skeleton method depending on the message type. If the LocationService service was generated, code to start that service will be added to the main activity’s onCreate method, and the activity class will be made to implement the LocationChangeHandler interface to receive location change events. Currently, the associated locationChanged method is left as a skeleton for the developer to implement.

**Windows Phone Device PSM** Code for the Windows Phone platform is generated similarly to the Android platform. Obviously, the generated code is C# rather than Java, and is written to use the Windows Phone .Net libraries and Windows Phone Odin2 APIs rather than their Android equivalents. Other than those differences, the following changes can be observed compared with the Android code generation above:

1. As C# does not have packages, instead .Net namespaces are used to structure the generated code. These function very similarly.

2. Rather than getters and setters for properties of CustomTypes and Messages, instead C# Properties are generated. By default, the default, blank property implementation can be used. In the cases where we are required to write additional code in the property setter (in the case of a Global CustomType), we instead generate a private member, then a public default getter, and a public setter containing the additional code.

3. OpenNETCF.ORM [82] annotations will be used for persistence, rather than ORM-Lite.

4. Rather than have the app’s main class extend OdinActivity, the Windows Phone equivalent is used - namely, OdinPhoneApplicationPage.
5. To detect changes in GPS location, we instead generate code that makes use of Windows Phone’s GeoLocation API\(^2\). Within the app’s main class, code is generated to create and start a new GeoCoordinateWatcher instance whenever the user navigates to the app, and to stop it when they exit. A callback method is also added to the class, which is notified by the GeoCoordinateWatcher when the user’s location changes. As with the Android version above, this method is left as a skeleton for the developer to implement.

**Surrogate PSM** Similarly, Eclipse Java project configuration files and a package structure similar to the Android project are generated for the Surrogate. Any defined CustomTypes are generated in the same way as with the Android project, with the exception that any required persistence annotations will be generated as Hibernate [83] annotations, rather than ORMLite.

Similarly to the device projects, Java classes will also be generated for the Surrogate to represent the messages to be handled. A MessageSenders class will also be generated, with methods allowing the Surrogate to send messages to any Odin2 User or Users.

The Surrogate’s main class will be generated, named appropriately, and made to extend Odin2’s Surrogate abstract class. For each message assigned to be handled by the Surrogate (by being listed in the handledMessages list of the PIM’s Surrogate entity), a blank skeleton method will be generated in the main class to handle that message. For each message assigned to be handled globally, skeleton method will also be generated. However, in this case, additional code will also be generated in that method to throw a NotHandledException, which will result in the message being forwarded to the device for processing. In other words, the generated behavior is such that, the Surrogate gets the first chance to handle any global message sent. If it decides not to (by throwing the aforementioned exception), the device will get the chance. A comment is also generated within these global handler methods, informing the developer of this behavior.

Additionally, within the Surrogate main class, methods will be generated to allow clients outside of the Odin2 infrastructure to send messages (and receive responses from request messages) through REST services. For each message type handled by the application, a method will be generated within the surrogate class to accept that message type from a client, and delegate it to either the surrogate’s handler for that message, or the device’s handler, depending on message locality. The generated method will take a list of User objects as message targets, and an object of the appropriate type to act as the message body. The method will be annotated with Spring MVC’s RequestMapping annotation,

and configured to be an HTTP POST request, with a JSON body, which must contain JSON representing the Users and Message.

### 5.1.1.5 Manual Completion

OdinTools provides developers with the ability to create a partial implementation of an Odin2 application for multiple platforms, however developers will still be required to write code to complete their application. Table 5.1 shows a summary of the components which are generated, compared with those which must be manually implemented. In particular, message handling and user interface logic must be specified manually. Section 5.2.2 shows this for an example application.

<table>
<thead>
<tr>
<th>Generated</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>API references</td>
<td>User interface</td>
</tr>
<tr>
<td>Configuration files</td>
<td>Message handling</td>
</tr>
<tr>
<td>Main Activity</td>
<td>Custom logic</td>
</tr>
<tr>
<td>Data types</td>
<td></td>
</tr>
<tr>
<td>Database</td>
<td></td>
</tr>
<tr>
<td>Message types</td>
<td></td>
</tr>
<tr>
<td>Skeletons for message handling</td>
<td></td>
</tr>
</tbody>
</table>

**Table 5.1**: Comparison of generated components with those that must be manually implemented on each platform

### 5.1.2 Tool Design

An initial prototype of OdinTools was implemented using Marama [84], a plugin for the Eclipse IDE allowing developers to create Domain-Specific Languages (DSL) using a graphical editor. Marama was chosen due to its integration with a popular IDE, along with its rapid prototyping potential. Subsequent versions of OdinTools have been implemented using Microsoft’s Visual Studio Visualization and Modeling SDK [85, 86], then later, the updated Modeling SDK for Visual Studio 2013 which includes greater support for DSL development. This platform was chosen due to its maturity and widespread support. Furthermore, the SDK supports import and export of the XML Metadata Interchange (XMI) format, promoting interoperability between different MDE frameworks should developers wish to extend OdinTools or create their own visual representations using alternative frameworks such as the Eclipse Modeling Framework (EMF).

OdinTools’ OAM and PIM were created using the Modeling SDK’s visual designer, which allows developers to create UML-like diagrams representing the entities and relationships
in their metamodels, as shown in figure 5.4. A visual language was also created using the SDK’s ability to define *Shapes* and *Connectors* that application developers can add to a graphical editor, as well as define mappings between these shapes and the underlying model instance. Developers may add shapes representing *CustomTypes*, *AsyncMessages*, *RequestMessages*, and the *Database*, and embed within them various *Properties*. They may do this by dragging various shapes onto the main work area from within the tool, as shown in figure 5.5. Section 5.2 illustrates this process in more detail.

The OAM-to-PIM transformation was implemented using a class library written in C# and added to the OdinTools Visual Studio project. This method was chosen over alternative methods, such as XSLT transformations, due to the ease with which Visual Studio allows the manipulation of its models via code. The transformation from the PIM to the source code for each target platform was implemented using the *Text Template Transformation Toolkit* (T4), developed by Microsoft and natively supported from within the Modeling SDK [86]. This toolkit is a templating engine which allows developers to write scripts which interleave C# (or other .Net) code with plain text. A T4 script has been created for each of the generated components in each of the target platforms.
Figure 5.5: A view of OdinTools running as a plugin to Visual Studio 2013

Figure 5.6: Adding a CustomType shape to an OdinTools diagram

5.2 Service Development with OdinTools

In this section, we demonstrate how OdinTools can be used to assist with the development of Odin2-based applications. As an example, we will consider a basic social networking service which allows users to send text messages and location updates to each other. We then show the code that’s generated, and detail the steps that would need to be taken to create a functional app from the generated code.
5.2.1 Use of OdinTools

In this application, a person will need to be represented in the system. As such, we will start by creating a Person CustomType. We can do this by dragging a CustomType shape from the toolbox onto the page, and giving it the name, “Person”. A person’s identification and location are important to know, so next a developer will drag two Property shapes from the toolbox onto the Person shape. We will give each of these properties a name ("Name" and "Location", respectively), and specify the types of those properties using a drop-down list shown on the table on the right-hand side of the screen in figure 5.6. We will specify that Name is a String and that Location is a GPSLocation. Both of these are built-in types and do not need to be defined. The full definition of the Person data type, as it appears to a developer, can be seen in figure 5.6.

A person’s details need to be stored, so next we will create a Database shape by dragging it onto the screen, and then drag a Database Table connector between the Database and Person shape to specify that Person objects should be persistable. By default, this will eventually result in a database table called Person being generated. We will change the table name to friends by editing this information as shown in figure 5.7. We will leave its locality set to Auto, which will result in the friends info being stored on the device.

We want to give users the ability to notify each other with text messages and location updates. These are the messages which need to be handled by devices. As such, we will next drag to AsyncMessage shapes onto the page. One will be named TextMessage, while the other will be named LocationChanged. Both of these messages will have a property of type Person that identifies the user who moved or sent the text message. This is defined by dragging a Property shape into each AsyncMessage shape - similarly to how we earlier specified that a person could have a name and a location. As shown in figure
5.8, we may now select *Person* as the type of these properties, as we have defined this custom type in our application. To complete the specification of our message types, we will add one more property to each - a *messageBody* property (of type *String*) and a *newLocation* property (of type *GPSLocation*) to the *TextMessage* and *LocationChanged* messages, respectively. We will leave the locality of the messages set to *Auto* to signify that they are handled by the device.

### 5.2.2 Generated Code

One we have completed the specification as detailed in section 5.2.1 above, we will use OdinTools to generate the Surrogate, Windows Phone, and Android projects. Partial implementations will be generated as specified in section 5.1.1. Table 5.2 shows a summary of the generated files and their contents. The *Platform* column shows whether the file exists on the Android (A), Windows Phone (W), and / or Surrogate (S) platform. The extension of each of the class files will be *.*java or *.*cs, depending on whether the file is a Java or C# class, respectively.

### 5.2.3 Completing the Application

To complete the application from this point, developers will be required to do the following manually for each device platform:

1. Build the UI
2. Call the `sendMessage_TextMessage` method from an appropriate place - presumably in response to a button click
3. Call the `sendMessage_LocationChanged` method from an appropriate place - presumably from within the `locationChanged` skeleton method generated by OdinTools.
<table>
<thead>
<tr>
<th>File</th>
<th>Platform</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>A, W, S</td>
<td>Class representing a person. Has properties for Location and Name. The Android and Windows Phone versions will additionally have ORMLite and OpenNETCF.ORM annotations defined for persistence, respectively.</td>
</tr>
<tr>
<td>TextMessage</td>
<td>A, W, S</td>
<td>Class representing a text message to be sent to a user. Contains properties detailing the user who sent the message, as well as the text message itself.</td>
</tr>
<tr>
<td>LocationChanged</td>
<td>A, W, S</td>
<td>Class representing a message to be sent when a user’s location changes. Contains properties detailing the user whose location changed, as well as the new GPS co-ordinates.</td>
</tr>
<tr>
<td>MessageSenders</td>
<td>A, W, S</td>
<td>Class allowing developers to send LocationChanged and TextMessage messages to other parties. Contains the methods <code>sendMessage_LocationChanged</code> and <code>sendMessage_TextMessage</code> to facilitate this.</td>
</tr>
<tr>
<td>LocationService</td>
<td>A, W</td>
<td>Class allowing developers to respond to GPS events from the operating system. Implemented using platform-specific APIs.</td>
</tr>
<tr>
<td>MainActivity</td>
<td>A</td>
<td>Android class which will be executed on startup. Configures the database, registers itself with the LocationService, and contains skeleton methods for developers to manually fill in to handle location change events, as well as TextMessage and LocationChanged messages from other users.</td>
</tr>
<tr>
<td>MainPage</td>
<td>W</td>
<td>Similar to MainActivity, but for Windows Phone devices. Extends OdinPhoneApplicationPage.</td>
</tr>
<tr>
<td>SurrogateMain</td>
<td>S</td>
<td>Main Surrogate class. Contains methods with Spring MVC RestTemplate annotations to accept TextMessage and LocationChanged messages from outside clients, and forward them to devices. No handler methods are generated in this surrogate, as all messages were specified to be handled by the device.</td>
</tr>
<tr>
<td>Project.sln,</td>
<td>W</td>
<td>Visual Studio configuration files for Windows Phone.</td>
</tr>
<tr>
<td>Project.csproj</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.classpath,</td>
<td>A</td>
<td>Eclipse configuration files for Android.</td>
</tr>
<tr>
<td>.project, AndroidManifest.xml</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.classpath,</td>
<td>S</td>
<td>Eclipse / Maven configuration files for the Surrogate.</td>
</tr>
<tr>
<td>.project, pom.xml</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Summary of generated code for the simple social network service
4. Complete the `handleMessage_LocationChanged` method by writing the appropriate code - presumely, display the provided location in a map component within the UI.

5. Complete the `handleMessage_TextMessage` method by writing the appropriate code - presumely, display the received message as a toast message and / or within a text field in the UI.

For this application, developers will not need to modify the generated Surrogate class, though they may wish to add custom logic, such as handling of offline users or changing the toast messages sent to background Windows Phone devices.

### 5.3 Summary

In this chapter we have introduced OdinTools, and showed how it can be used to aid in the development of an Odin2-based mobile service application. By specifying an application’s data types, messages, certain persistence semantics, and message and data locality, these service components will then be generated for the Surrogate, as well as the Android and Windows Phone mobile platforms. In this way, developer productivity is increased as it is quicker to use OdinTools to specify these application components as opposed to specifying them manually - particularly in the case where multi-platform support is desired. For a more thorough evaluation of OdinTools’ ability to increase developer productivity, we refer the reader to Chapter 8 section 8.3.

In the future, we wish to enable a more comprehensive cross-platform mobile service development tool. We believe that this can be achieved using a combination of our current model-driven engineering approach and an existing hybrid application development solution such as those introduced in Chapter 2 section 2.7. Using a combined approach, developers would specify the structure of a mobile service in much the same manner as OdinTools’ current implementation. From the specification, the tool will generate code targeted at a hybrid development solution, rather than a specific platform. Initially we will explore the use of Titanium for this purpose as it offers the ability to add platform-specific extensions, which OdinTools could leverage to generate native code for particularly performance-critical sections of code. The combination of Titanium and OdinTools will allow developers to quickly specify a service application’s structure visually, while being able to create a cross-platform user interface, further decreasing development time.
Chapter 6

Odin Test Harness

Our evaluation of our original middleware, Odin, revealed some shortcomings in the platform, as discussed in Chapter 3 section 3.4. Some of these shortcomings were related to Odin’s performance, reliability, and ease-of-use - shortcomings which we sought to address in our redesigned middleware, Odin2. Another issue we faced was that of testability. Several of Odin’s reliability issues came about as a result of an inadequate testing environment for the middleware. Ideally, we would be able to automatically deploy Odin services on many concurrent devices and have each device experience a range of network conditions. Furthermore we would be able to manipulate these conditions, and operate each device’s user interface, in a controlled and repeatable manner. Finally we would be able to test Odin’s surrogate host, to ensure that it can meet the demands of all mobile services and clients with which it communicates.

A small literature review (Chapter 2 section 2.6) identified what while there are tools which allow us to automate smartphone UI testing, and other tools which allow us to simulate various network conditions, there is no tool which allows us to perform both these tasks, concurrently, for multiple concurrent devices, in a precisely controlled manner. We set out to design and develop a system which could perform such testing. Primarily this system, the *Odin Test Harness*, was designed to assist us with the development and testing of Odin2 and any Odin2 services, such as the REMOTE-CR trial (Chapter 7). However the test harness could potentially be useful in the development and testing of any networked Android application. The test harness was implemented in collaboration with Habib Naderi, a member of the Department of Computer Science’s Research Programming team.

The remainder of this chapter is organized as follows. Initially we introduce the test harness’s requirements in section 6.1. We then discuss its design in section 6.2. In section 6.3 we then introduce the test harness’s API, prior to demonstrating some test harness
usage examples in section 6.4. We then describe how the test harness itself was tested in section 6.5, prior to concluding in section 6.6.

6.1 Requirements

This section details the functional requirements of the test harness. This set of requirements is intended to ensure that we can sufficiently test whether Odin2 meets its requirements as set out in Chapter 4 section 4.1.

The test harness must:

**FR #1: Simulate multiple concurrent devices** - Allowing us to simulate multiple concurrent devices will allow us to obtain test results faster by running multiple concurrent tests, and / or run the same test on multiple concurrent devices to increase confidence in the results. Additionally, simulating multiple devices will allow us to load test the surrogate host more easily.

**FR #2: Simulate multiple network interfaces per device** - One of Odin2’s requirements is that it must be able to deal with the loss of connectivity over a particular network interface by switching to an alternative. The test harness must support multiple network interfaces on its simulated devices to allow us to test this requirement.

**FR #3: Simulate a network environment for each interface for each device** - Another of Odin2’s primary requirements is to remain operational under a variety of network conditions, without data loss or duplication. To test this we must be able to simulate such conditions. Specifically, we must be able to control the bandwidth, latency, and loss rate of each simulated network interface, and change these parameters at any time during a test.

**FR #4: Configure a tested app** - The correct operation of many mobile applications - including Odin2 services - depends upon various data not available at the application’s install time. Such data could include datasets in an application’s database and / or configuration settings such as the address of an external host. To allow us to test these applications, the test harness must be able to modify a test application’s database and its application settings.

**FR #5: Automate a tested app’s user interface** - The vast majority of mobile applications require user input to navigate to various application features and otherwise interact with the app. The test harness must be able to simulate user interaction to allow an app’s full feature set to be tested.
FR #6: Allow us to repeat experiments - We must be able to describe a test to the test harness in such a way that a test is repeatable. This way, when a bug is identified and a fix implemented, we can rerun the same test to verify that the fix is effective.

6.2 Design

Figure 6.1 shows an overview of different physical and emulated hardware components within a test harness machine, and the network channels between them. The test harness machine itself runs Ubuntu 12.04 LTS [87] and is equipped with two network cards (labeled eth0 and eth1 in Figure 6.1) to allow the simulation of multiple physical networks - useful for testing the correctness of multiple network scenarios such as Vertical Handover between Wi-Fi and 3G. The Test Harness runs 15 Android-x86 [88] virtual machine
(VM) instances (labeled Android-01 .. Android-15) within VMWare Player 5.0.1 [89]. To facilitate this, the machine is built with eight CPU cores and 16 Gigabytes of RAM.

Within the test harness machine, two virtual network devices (labeled Vmnet1 and Vmnet2) have been created using vmware-netcfg, part of the VMWare Player Utilities [89]. Each acts as an intermediary between the Android VMs and the physical network cards, allowing us to control the properties of the networks as experienced by the VMs without affecting the network operation of other software components on the machine. Each Android VM is configured with two emulated network adapters to simulate the multiple network interfaces (i.e. Wi-Fi and 3G) available on physical devices. For each VM, eth0 is mapped to Vmnet1 and eth1 is mapped to Vmnet2. Each VM connection is mapped to a different channel on its corresponding virtual network device. Each of these channels has a Token Bucket Filter (tbf) Classless Queuing Discipline (qdisc)\(^1\) associated with it which can be altered to change the bandwidth of that channel, and a Network Emulation (netem)\(^2\) qdisc which can be used to alter the packet loss rate and latency.

---

\(^1\)http://linux.die.net/man/8/tc-tbf
\(^2\)http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

**Figure 6.2:** Software components within a test harness machine

Figures 6.2 and 6.3 show the location of various software components integral to the operation of the test harness. Such components are located in three key areas: (i) directly
within the test harness; (ii) within each test harness VM; and (iii) optionally within each app to be tested by the harness. A description of each set of software components is given below, while a detailed API for each provided library is given in Section 6.3.

6.2.1 Test Harness Software

To use the test harness, a tester will create a shell script which may use any standard shell functions, plus those defined in the Host Script Library. This library contains functions which:

- Synchronize the test harness clock
- Start and configure VMs
- Install and run Android apps within VMs
- Read from / write to a VM’s file system
• Dynamically alter the bandwidth, latency, packet loss rate, and preferred network of individual VMs

• Communicate with, and simulate user interaction on, a test-harness-aware app running within a VM

To synchronize the test harness clock, the script invokes the standard `ntpd` program installed with Ubuntu to synchronize with an NTP time server. `ntp.ubuntu.com` was selected for this purpose.

To start and configure VMs, the library executes one instance of the `vmplayer` software for each VM. Once a VM is running, a debug connection is opened to that VM over the Android Debug Bridge (ADB) included in the Android Platform Tools.

To install an app, ADB’s `install` command is invoked, simply providing the path to an APK file to install on a VM. To run an app, ADB’s `start` command is used. This command allows ADB to run any native application within a device. To run Android apps packaged as APK files, the native Activity Manager (am) app can be invoked in this way to start any Activity within any installed APK. ADB’s `push` and `pull` commands are invoked by the library to copy files to and from a running VM, respectively.

Using ADB, one can also open a shell connection using the `adb shell` command. Using this, we can invoke any valid shell command on a VM through ADB as if we were using a local console window. Using this functionality we can invoke any of the VM Scripts installed within a VM. These scripts are used to alter the preferred network for a VM (for network switch tests) and communicate directly with running Android apps, and are detailed in 6.2.2 below.

Finally, using the Host Script library we may alter the network characteristics of any VM individually. The library achieves this by invoking the Traffic Control (tc) software provided by Ubuntu and other Linux distributions to alter the properties of the qdiscs through which each VM sends its traffic. By altering these qdiscs we can alter a VM’s bandwidth, latency, and loss rate for either or both of its virtual network connections.

To use the test harness, a tester must simply import the Host Script library, named `odin_host_lib`, into their own shell scripts. Examples of test scripts are given in section 6.4.

### 6.2.2 Test Harness VM Software

Within each VM are a set of VM Scripts which may be run by the Host Scripts through the ADB. These scripts contain the functionality to:
• Synchronize the VM’s clock
• Simulate network switches and outages on a VM
• Configure and control test-harness-aware apps running within the VM

To synchronize a VM’s clock, the script simply invokes the ClockSync app installed on the VM. The app is preconfigured to synchronize with the same NTP server as that used by the test harness itself (ntp.ubuntu.com). By synchronizing the clocks of the test harness and all its VMs at the commencement of a test, we can ensure that the timestamps of all log files are synchronized enough to be comparable.

To simulate network switches, the library invokes the route network tool\(^3\) to change the default gateway through which the VM routes packets to the Odin2 surrogate host. Traffic bound for the host will favor the given gateway (either eth0 or eth1). To simulate network outages, the iptables network tool\(^4\) is used to add a filter which drops all packets bound for the given network interface.

To configure and control running apps, those apps must include the TestHarnessService library (see below). If they do, the VM script can open a local TCP port to that app and send commands to simulate user interactions and edit application data.

The VM Scripts are not intended to be invoked directly by testers. Testers achieve this functionality by invoking corresponding Host Script functions which delegate to the VM scripts, when necessary, over the ADB.

### 6.2.3 App Software

Optionally, developers may include the TestHarnessService library within an app to be tested using the harness. If this is done, the test harness gains the ability to:

• Invoke methods on the tested app
• Simulate user interactions on the foreground Activity
• Configure an app by writing to its database and / or preferences file

Furthermore, the app becomes test-harness-aware, allowing it to alter its behavior based on whether the app is running on a test harness or a real device. This can be useful, for

\(^3\)http://linux.die.net/man/8/route
\(^4\)http://linux.die.net/man/8/iptables
example, to disable various application sections which don’t function correctly in a VM scenario, such as Bluetooth interaction or utilization of some Google Play services.

To allow communication, the TestHarnessService listens on a local TCP port. This port is accessed by the corresponding VM script, which acts as an intermediary between the ADB and this service.

To invoke arbitrary methods on an app, the service uses standard Java Reflection. Currently, only methods that take zero or more Strings as arguments, and return Void or String can be invoked in this way.

To simulate user interactions on an Activity’s layout, Android’s android.content.res/Resources#getIdentifer() method is used to get the ID of a UI component from its name, then interactions are simulated using methods such as android.view.View#performClick() to simulate clicks. Android’s unit testing framework includes the UiAutomation library which is designed to allow the creation of unit tests which simulate user interactions to a greater degree of freedom than is allowed using the TestHarnessService. Originally we found that the library would be difficult to integrate with the test harness, however we will investigate this further in the future due to a promising API update5.

To interact with an app’s preferences and database, Android’s standard SharedPreferences and sqlite database libraries are used.

To allow the operation of the TestHarnessService, its library project must be included during app development. Furthermore, the service must be started from with an app’s main Activity (using the standard startService() method).

6.3 Test Harness API

This section details the specific functions available to testers when using the harness. Of most importance is the Host Script Library, to be used directly by testers. However, the VM scripts and TestHarnessService APIs are also documented here.

6.3.1 Host Script Library

The Host Script Library is to be referenced by testers when developing test scripts, and contains all necessary functionality to interact with the test harness machine and its VMs. The library contains several functions intended for use by testers (public functions), along

with several helper functions which assist in the implementation of the public functions. Each public function is detailed here.

- **setup <number-of-VMs>|<list-of-VM-names>:** Starts <number-of-VMs> VMs or those specified by <list-of-VM-names> and configures them for starting an experiment. For example, `setup 3` starts and configures three VMs starting from Android-01, while `setup Android-01 Android-03 Android-05` starts and configures three VMs named Android-01, Android-03 and Android-05. If a VM specified by this function is already running it will not be restarted; however it will still be correctly configured.

- **set_loss <VM-name> <device> <loss>:** Sets the packet loss rate for <device> on the specified VM for traffic bound for the Odin2 surrogate host. For example, `set_loss Android-01 eth0 25%` sets the loss rate to 25% for eth0 on Android-01. Only traffic bound for the surrogate host is affected.

- **set_delay <VM-name> <device> <delay>:** Sets the delay to <delay> milliseconds for <device> on the specified VM for traffic bound for the Odin2 surrogate host. For example, `set_delay Android-01 eth0 200ms` sets the delay to 200 milliseconds for eth0 on Android-01. Only traffic bound for the surrogate host is affected.

- **set_bandwidth <VM-name> <device> <bandwidth>:** Sets the maximum bandwidth for <device> on the specified VM to <value>. For example, `set_bandwidth Android-01 eth0 16kbit` sets the bandwidth to 16 kbit/s for eth0 on Android-01. Only traffic bound for the surrogate host is affected.

- **drop <VM-name> <device> <duration>:** Drops TCP connections to the Odin server via <device> for <duration> seconds on the specified VM. For example, `drop Android-01 eth0 10` drops TCP connections which have been or are going to be established through eth0 on Android-01 for ten seconds. Only traffic bound for the surrogate host is affected.

- **block <VM-name> <device>:** Blocks communication to the surrogate host through <device> on the specified VM. For example, `block Android-01 eth0` causes Android-01 to drop all packets which are sent to / received from the surrogate host through eth0 until further notice (via the unblock command).

- **unblock <VM-name> <device>:** Unblocks communication to the surrogate host through <device> on the specified VM. For example, `unblock Android-01 eth0` removes filters set by a previously issued block command.
- **start_app** <VM-name> <app>: Starts the native application <app> on the specified VM. Any command line arguments for the target app may be given as a list after the app name. For example, `start_app Android-01 "ping ntp.ubuntu.com"` runs `ping` on Android-01 with the command line argument `ntp.ubuntu.com`. This function may be used to start any installed Activity on a VM by using the `am` native app. For example, `start_app Android-02 "am start-n your.package.name/your.package.name .ActivityName"` will start the ActivityName activity located in the APK with the given package name on Android-02.

- **stop_app** <VM-name> <app>: Stops all instances of application <app> on the specified VM. For example, `stop_app Android-01 "ping"` stops all `ping` processes on Android-01.

- **config_app** <VM-name> <args>: Sends an arbitrary string to the test-harness-aware app running on the specified VM. This may be used to set application preferences, run database scripts or control the UI. For example:
  - `config_app Android-01 "set-property port=2002,server=130.216.33.11"` will send a list of key-value pairs (namely, `port=2002,server=130.216.33.11`) to the app running on Android-01, which will be assigned to the app’s preferences file.
  - `config_app Android-02 "exec-sql INSERT INTO my_table VALUES (a, b, c)"` will execute the sql statement "INSERT INTO my_table VALUES (a, b, c)" on the database of the app running on Android-02.
  - `config_app Android-03 "ui.click my_button"` will simulate a click event on the button named `my_button` on the foreground Activity of the app running on Android-03.
  - for a full list of available commands with the `config_app` command, please see Section 6.3.3.

- **config_all_apps** <num-devices> <args>: A convenient shortcut for invoking the `config_app` command with the given <args> on VMs named Android-01 through Android-<num-devices>. For example, `config_all_apps 3 "set-property port=2002"` will invoke the `set-property` command on Android-01, Android-02, and Android-03.

- **install_app** <VM-name> <app.apk>: Installs an application on the specified VM. For example, `install_app Android-01 /home/odin/odin-test.apk` installs odin-test.apk on Android-01.

- **uninstall_app** <VM-name> <package-name>: Uninstalls an application from the specified VM. For example, `uninstall_app Android-01 nz.ac.auckland.cs`
.odin.test uninstalls the application whose package name is nz.ac.auckland.cs.odin .test from Android-01.

- **switch_to <VM-name> <device>:** Changes the active network device on the specified VM. For example, `switch_to Android-01 eth0` causes Android-01 to change its active network interface to `eth0`. Since `eth0` on each VM has been bound to `eth0` on the test harness through Vmnet1, this change causes the packets from Android-01 to be forwarded through `eth0`, on the test harness, to the surrogate host. This will necessarily cause any such traffic to experience the loss, latency, and bandwidth constraints set by `set_loss`, `set_delay`, and `set_bandwidth` on `eth0` for Android-01, respectively.

- **switch_test <VM-name> <time-between-switches> <duration>:** Runs a network switching test on the specified VM for `<duration>` seconds. The test starts with blocking communication to the surrogate host on the active interface. Then it waits for `<time-between-switches>` seconds, before changing the active interface to another interface and waiting for another `<time-between-switches>` seconds. This process is repeated for `<duration>` seconds. For example, `switch_test Android-01 5 30` runs a network switch test for ten seconds, with a five second delay between switches, on Android-01.

- **get_logs:** Retrieves the log files from all VMs and stores them in a folder. The folder name is a combination of the current date and time. The log file name is set to the VM name.

- **finish:** Should be called at the end of the experiment to reset the network settings on the test harness and VM network devices. If it is used with `-s` as a switch, it additionally shuts down the VMs involved in the test.

- **copy_to <VM-name> <source-on-odin-test-host> <destination-on-VM>:** Copies data from the test harness to the specified VM. For example, `copy_to Android-01 /home/odin/data.dat /system/bin/odin` copies `data.dat` from `/home/odin` on the test harness into `/system/bin/odin` on Android-01.

- **copy_from <VM-name> <source-on-VM> <destination-on-odin-test-host>:** Copies data from the specified VM to the test harness. For example, `copy_to Android-01 /system/bin/odin/data.dat /home/odin` copies `data.dat` from `/system/bin/odin` on Android-01 into `/home/odin` on the test harness.

- **set_debug <on|off>:** Enables / disables debugging messages. It is disabled by default, but can be useful for troubleshooting purposes.
6.3.2 VM Scripts

The VM Scripts are installed on each VM. They can be run from a shell like any bash script, and are executed by the Host Script Library via ADB’s shell command. They are not intended to be executed manually by testers, but their functionality is included here for completeness.

- **odin_vm_lib**: This script provides a set of functions which are used by other VM scripts, and is referenced by each of the scripts below.

- **switch-to <device>**: Changes the active network interface by altering the VM’s routing table to preferentially send packets over `<device>`. For example, `switch-to eth0` will cause all traffic from that point forward to be sent over `eth0` when possible.

- **reset <default-router-eth0-address> <default-router-eth1-address>**: Resets the VM’s network settings to their default state and assigns the given IP addresses to that VM.

- **run-switch-test <time-between-switches> <duration>**: Runs a network switch test on the VM. A switch test will periodically switch between network interfaces on the VM for the given duration. For details of a network switch test, please see the Host Script Library’s `switch_test` function in 6.3.1 above.

- **init**: Sets the VM’s name and default network behavior. This runs when the VM boots, rather than being called by any Host Script function.

- **get_ip_addr <device>**: Returns the IP address of the given network interface on the VM.

- **block_dev <device>**: Completely blocks all traffic on the given interface bound for or incoming from the surrogate host.

- **unblock_dev <device>**: Unblocks all traffic on the given interface bound for or incoming from the surrogate host.

- **get-time**: Returns the current time in the format `hh:mm:ss.sss`.

- **app-config <port> <args>**: Sends the given `<args>` string to the given `<port>` on localhost over TCP, and returns a string response. It is expected that a TestHarnessService running on a test-harness-aware app on the VM is listening on that port. For more details, please see 6.3.3 below.
6.3.3 Test Harness Application Library

By including a reference to the *Test Harness Application Library* (THAL) and making sure to start the *TestHarnessService* Android Service from their applications, developers may make their apps test-harness-aware. This allows the test harness to control and configure an app, while allowing apps to be informed about whether they are running on the test harness. Figure 6.4 shows an overview of the core TestHarnessService logic.

To determine whether an app is running on a test harness, a developer must simply call *TestHarnessService.isTestHarness()*, which returns *true* if the app is running on the test harness, and *false* otherwise. To control the UI, a tester will send one of several
commands over a TCP connection on a local port, via the *app-config* script (see 6.3.1 above). The valid commands are as follows:

- **set-property** `<key=value[,key2=value2...]>`: writes the given list of key-value pairs to the app’s *SharedPreferences* file.

- **exec-sql** `<sql>`: Runs the given sql script against the app’s database.

- **ui.click** `<name>`: Executes a click event on the UI widget with the given name on the currently active Activity.

- **ui.go-back**: Simulates the user pressing the back button.

- **ui.invoke** `<method-name> <args>`: Executes the method named `<method-name>` on the currently active Activity with the given `<args>`. The method to execute may be any visibility, but must return either *String* or *Void*, and must take a number of *Strings* as method arguments equal to the number of arguments given to this command.

- **ui.select-index** `<widget-name> <index>`: Locates the UI widget with the given `<widget-name>`, and, if that widget is the container of a list of items (e.g. a *ListView*), this will select the item in that list at the given `<index>`.

- **ui.select-item** `<widget-name> <item-name>`: Locates the UI widget with the given `<widget-name>`, and, if that widget is the container of a list of items (e.g. a *ListView*), this will select the item in that list whose `toString()` method returns the given `<item-name>`, ignoring case.

Each of these commands will return a *String* representing the result of the operation. This will be either an “OK” message, an error message, or the direct result of calling a method (in the case of *ui.invoke* or *exec-sql*).

In addition to these commands, the THAL is extensible, allowing developers to add their own commands if necessary. Developers achieve this functionality in a manner consistent with the Android development guidelines, by registering a *BroadcastReceiver* object with the Android operating system which extends *CommandBroadcastReceiver* (see Figure 6.4). Any receiver registered this way will receive all commands issued by the test harness via a call to its `processCommand()` method. Here, developers filter commands intended for them by name.
### 6.4 Example Test Scripts

Some example test scripts are provided here to show how either simple or complex tests may be created using a combination of Unix *bash* commands and the functions provided in the Host Scripting Library. Listing 1 shows a simple test to see whether an app handles delay and network switches, while Listing 2 shows a script to simulate several users conducting exercise sessions using the REMOTE-CR trial software (see Chapter 7). This test script simulates a workout session of a customizable length on a customizable number of simultaneous VMs. For more information on the use of the test harness during the development of Odin2 and REMOTE-CR, please see Chapter 8 section 8.4.

```bash
1 # ODIN_SERVER=130.216.33.11
2 . /home/habib/odin/odin_host_lib
3 setup Android-01
4 set_delay Android-01 eth0 100ms
5 set_delay Android-01 eth1 200ms
6 switch_to Android-01 eth0
7 start_app Android-01 "ping $ODIN_SERVER"
8 sleep 5
9 switch_to Android-01 eth1
10 sleep 5
11 stop_app Android-01 "ping"
12 get_logs
13 finish -s
```

**Listing 1**: Simple script to test that the latency and switch commands are functional

In Listing 1, line one defines the Odin2 surrogate host IP address. This is required in all test scripts (as all drop-rate and latency commands only affect Odin-bound traffic). Lines four and five set the latency of *eth0* and *eth1* on Android-01 to 100ms and 200ms, respectively. Line six ensures that Android-01 will use *eth0* for all Odin2 traffic. The *ping* app is started on line seven and then allowed to run for five seconds before the active network interface is changed to *eth1* on line nine. The app is then allowed to run for an additional five seconds before being stopped on line 11. Logs are saved for later analysis on line 12, prior to resetting all network interfaces and shutting down the VM on line 13.

In Listing 2, we allow the test runner to configure the script on lines five and six by entering the number of VMs to use and the workout length (in minutes) as command line arguments. On line nine, a custom function called *nihi_setup* is invoked. This function is included in an extension to the default Host Script Library designed specifically for use with the REMOTE-CR app, and will start and configure the given number of VMs before installing, starting, and configuring a fresh copy of the app (using the *setup, install_app,*
#!/bin/bash

ODIN_SERVER=130.216.33.11
. /home/odin/odin/odin_host_lib

numDevices=$1
workoutLength=$2

# Shortcut to setup VMs and install and run NIHI app on all VMs
nihi_setup $numDevices

# Navigate to workout screen on all devices
config_all_apps $numDevices ui.click btnWorkout

# Wait for a bit to make sure Odin is connected
sleep 1.5

# Click the "start workout" button on all devices
config_all_apps $numDevices ui.click btnStartWorkout

# Simulate the workout
declare timeToSleep
let "timeToSleep = $workoutLength"
while [ $timeToSleep -gt 0 ]
do
    log "Performing workout ($timeToSleep seconds remaining)...
    sleep 'min $timeToSleep 60'
    let "timeToSleep -= 60"
done

# Navigate backwards on all devices. This will cause the workout
# session to finish and will cause the app to disconnect from Odin.
config_all_apps $numDevices ui.go-back
sleep 2

# Make sure all logs are written, then close the app and copy the
# log to the test harness.
nihi_finish $numDevices

Listing 2: Simple workout test script for REMOTE-CR app

start, and config_all_apps commands, respectively). On line 12, we simulate the user
clicking the “Workout” button on each VM, which will invoke the Activity responsible
for a workout session. The app is designed to connect to Odin as soon as this activity
starts, so we wait for a small amount of time for this to occur on line 15. Following
that, we simulate the user pressing “Start Workout” on each VM. Lines 20 through 28
then sleep for the required amount of time, while periodically logging status messages so
the tester remains informed of test progress. After the required amount of time, on line
32, we simulate a back-button press on all VMs, which will cause the workout to finish (according to the app requirements). Finally we invoke \texttt{nihi\_finish} on line 37, which is a convenience function for simulating the user disconnecting from Odin, shutting down the app, saving all logs to the test harness, and resetting all network behavior.

### 6.5 Test Harness Evaluation

As the test harness was being developed, it was tested to make sure its functional requirements were satisfied. Each test harness requirement was evaluated using a tool suitable for evaluating that particular requirement.

To test the ability to introduce latency (the \texttt{set\_delay} command), a test script was run which executed the \texttt{ping}\textsuperscript{6} program on a VM. The ping tool was set to ping the Odin server and report the \textit{round-trip-time (RTT)} for packets to reach that server. The actual RTT could then be compared with the expected RTT based on the latency that was set. For example, for a latency of 100ms, a RTT of just over 100ms was expected (comprised of the 100ms introduced latency plus the base RTT, which should be close to 0ms as the test harness and Odin2 surrogate host are both connected directly to the University intranet). This test was repeated with multiple latencies and on every VM. A similar approach was used to test the packet loss (\texttt{set\_loss}) and \texttt{block} commands, as the ping tool is also capable of reporting packet loss percentages. The average packet loss was tested to make sure it was comparable with the target loss rate set by the \texttt{set\_loss} command, while packet loss was tested to ensure 100\% loss in the case of the \texttt{block} command.

To test the \texttt{switch\_test} and \texttt{switch\_to} commands, again the ping tool was used to send packets to the Odin server. However, this time the \texttt{tcpdump}\textsuperscript{7} network tool was used to record when a packet went through a particular network interface. While pinging the surrogate host, one of the switch commands was run, and the output of tcpdump was examined to make sure the packets went through the expected network interface, both before and after the command was executed.

To test the THAL, a minimal Android application was developed which simply had a preferences file, a database, and a user interface with several components to control. Black-box testing was then conducted using this app, with each valid THAL command being executed against the app, and the results compared with expected results.

In all cases, the tests show that all functional requirements of the test harness are met. Following the test harness evaluation, we were confident in its ability to detect potential

\footnotesize{\textsuperscript{6}http://linux.die.net/man/8/ping  \textsuperscript{7}http://linux.die.net/man/8/tcpdump}
bugs with Odin2 and REMOTE-CR. The use of the test harness in the development of Odin2 and REMOTE-CR is discussed in Chapter 8 section 8.4.

6.6 Summary

Table 6.1 shows how each of the test harness’s functional requirements are satisfied.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>How is requirement met?</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR #1</td>
<td>The test harness supports the simulation of fifteen simultaneous Android devices. Each may be operated independently or in groups. Multiple test harness machines may be used in parallel if additional devices are required.</td>
</tr>
<tr>
<td>FR #2</td>
<td>The test harness provides each virtual device with two virtual network interfaces - <em>eth0</em> and <em>eth1</em>. The virtual device’s operating system recognizes one as 3G and the other as Wi-Fi.</td>
</tr>
<tr>
<td>FR #3</td>
<td>Each virtual network interface on each device is configurable in terms of bandwidth (via <em>tbf qdiscs</em>), latency and drop rate (via <em>netem qdiscs</em>). Furthermore each interface can be completely disabled at will.</td>
</tr>
<tr>
<td>FR #4</td>
<td>Apps to be tested may import the <em>Test Harness Application Library (THAL)</em>. This allows apps to receive commands from the test harness via a TCP socket. One of these commands, the <em>set-property</em> command, will allow testers to write to an app’s preferences file. Another, the <em>exec-sql</em> command, allows testers to execute arbitrary SQL scripts on an app’s database.</td>
</tr>
<tr>
<td>FR #5</td>
<td>The <em>ui.click</em>, <em>ui.invoke</em>, and <em>ui.go-back</em> commands, part of the THAL, allow testers to invoke click events on any UI widget, invoke public methods, and navigate backwards in the UI’s navigation stack, respectively. Android’s UIAutomation library allows greater flexibility than this, and we will investigate its use further.</td>
</tr>
<tr>
<td>FR #6</td>
<td>To run tests, testers write bash scripts which include a reference to the test harness API. These allow a particular sequence of UI interactions and network changes to be repeated at will.</td>
</tr>
</tbody>
</table>

*Table 6.1: Summary of how the Odin Test Harness’s design satisfies its functional requirements*

Evaluation of the test harness using a combination of Linux networking tools and black-box testing has ensured that each of its features have been correctly implemented, as discussed in section 6.5. Furthermore, the test harness has been validated in a real-world scenario, having been used to test both Odin2 and the REMOTE-CR trial software. This has allowed us to detect bugs which would be otherwise difficult to detect, such as bugs depending on rarely-occurring network conditions. This has helped us increase Odin2’s
dependability. Further discussion on the test harness’s use in this manner is available in Chapter 8 section 8.4.
Chapter 7

REMOTE-CR: An Odin2-enabled Telemed System for Cardiac Rehabilitation

One of the primary goals of this thesis is to provide a mobile service provisioning solution which is sufficiently performant, expressive and dependable to be used in a real-world scenario. We argue that Odin2 is such a solution. In this chapter we show how Odin2 has been used to implement a complex real-world telehealth service, REMOTE-CR. REMOTE-CR is a service which allows patients undergoing a cardiac rehabilitation programme to conduct exercise sessions associated with that programme without being constrained to a particular location, while still providing clinicians the ability to monitor those patients in real-time while providing motivational feedback.

Using REMOTE-CR, a patient’s smartphone acts as an Odin2 service. Physiological, location, and other data is gathered from internal and external sensors and is made available to clinicians on request. In this way, REMOTE-CR at its core is similar to the patient monitoring example service introduced in Chapter 1. However, REMOTE-CR’s list of requirements is significantly more complex, as shown in this chapter.

The remainder of this chapter is organized as follows. Initially in section 7.1 we present a brief background of the prior work associated with REMOTE-CR. We then summarize REMOTE-CR’s requirements in section 7.2. Section 7.3 provides a detailed description of the software, as well as its usage. Finally in section 7.4 we summarize how Odin2’s capabilities have been leveraged within REMOTE-CR, prior to concluding in section 7.5.
7.1 Background

Cardiovascular Disease (CVD) is a leading cause of death worldwide, accounting for 30% of deaths globally [90] and 40% in New Zealand [91]. In addition to high mortality rates, CVD is also associated with significantly reduced quality of life in sufferers, including pain, disability and fatigue. The prevalence of CVD has increased by 18.2% in the past decade, and is projected to increase a further 16.6% by 2030 [90, 92]. Exercise-based Cardiac Rehabilitation (CR) has been shown to be an integral component in risk management due to its ability to target several CVD risk factors [93–95].

Despite its benefits, participation in CR programmes is low in many countries [90, 92, 96]. Even amongst patients who attend, adherence to prescribed exercise programmes is also poor. One reason for this is patient difficulty in accessing CR programmes. Telehealth, which uses modern communications equipment to enable remote communication between participants and healthcare providers, is one solution to this and its effectiveness in managing medical conditions in other areas has been well documented [97–99].

Within the University of Auckland, the National Institute for Health Innovation (NIHI) has conducted the Heart, Exercise And Remote Technologies (HEART) trial, in which participants received regular personalized text messages and motivational material over 24 weeks. Results from the HEART trial show that text messaging and internet intervention is effective and cost-effective at increasing physical activity and health-related quality of life of trial participants. For more information about the HEART trial, we refer the reader to NIHI’s body of work [100–103].

Other work has explored the ability of smartphones to provide patients with a much richer experience, in addition to providing medical professionals with a greater ability to monitor and provide real-time feedback to participants. However, in these studies the need to address mobile service provisioning concerns has become apparent. For example, in a recent study [104], Worringham et al. provided participants with a smartphone, coupled to a GPS device and ECG monitor via Bluetooth. Participants would receive a phone call before and after a scheduled exercise session, while the smartphones would stream GPS and ECG data to physicians via a website. The study showed that the mobile CR solution improved walking performance and health-related quality of life compared to an in-clinic approach.

However, amongst other issues, the software was unable to adequately respond to periods of intermittent mobile connectivity. This resulted in 10% of sessions experiencing data loss, sometimes necessitating that the session be canceled. Furthermore, the participants’ smartphones were under-utilized - there is the potential for a greater degree of interaction between participants and physicians than a simple phone call before and after each session.
To address the issues above, NIHI commissioned the Remote Exercise Monitoring Trial for Exercise-based Cardiac Rehabilitation (REMOTE-CR) study. The study seeks to "compare the effectiveness of technology-assisted, home-based, remote monitored exercise CR to standard supervised CR in New Zealand adults with a diagnosis of ischaemic heart disease". The development of the software used in the study has been a collaborative effort between NIHI and the Department of Computer Science.

Initially, an early pilot study was conducted to assess the feasibility of the proposed vital sign sensors, and of Odin as a middleware platform [73]. As discussed in Chapter 3, the study showed Odin’s potential but also demonstrated the need for further development of the platform. Odin2 (Chapter 4) has addressed Odin’s shortcomings and is well suited to act as the middleware platform upon which REMOTE-CR is built. The Odin Test Harness (Chapter 6) has additionally proved invaluable during the development of REMOTE-CR.

7.2 Requirements

REMOTE-CR’s requirements were motivated from the needs of a cardiac rehabilitation programme, results of previous CR study outcomes [90, 92, 96, 100–104], and the desire to leverage modern smartphones more fully than previous studies. We present a summary of REMOTE-CR’s functional and non-functional requirements in this section; for a detailed functional requirements list we refer the reader to appendix B.

In these requirements, and elsewhere in this chapter, we refer to patients using the software as part of a CR programme as participants, while we refer to clinicians monitoring those patients as clinicians.

7.2.1 Functional Requirements

REMOTE-CR must:

**FR #1: Allow participants to access the system from any supported device** - Participants must have a username and password which they can use to log into the mobile app on any device with the software installed. Whenever a participant logs into a device, all of their information must be downloaded to that device, and, other than shared data such as routes (see below), it must be deleted from all other devices.
Chapter 7. REMOTE-CR

FR #2: Maintain a profile for any number of participants - A participant’s profile consists of his or her name, date of birth, height, weight, gender, resting heart rate, maximal heart rate, and additional health metrics obtained from prior medical assessments. Participants must be able to view and edit their own profile from within the smartphone app, while clinicians must be able to view and edit any participant’s profile, and profiles for new participants, using the web app.

FR #3: Monitor participants’ physiological and location data during exercise sessions - REMOTE-CR must interface with an appropriate monitoring device to read a participant’s heart rate and breathing rate. Furthermore, a participant’s location, speed, and distance traveled must be obtained via the smartphone’s GPS. Additionally, the values for the calculated performance metrics $\%HRR$ and $L$ (see appendix B.2) must be derived. Finally, the most recent ten seconds of Electrocardiograph (ECG) data must be cached on the smartphone. This data must be made available to clinicians and participants on-demand in a timely manner.

FR #4: Allow participants to report symptoms - During a session, participants must be able to notify clinicians of three types of symptoms associated with cardiac events, along with those symptoms’ severity. These are angina, dyspnoea, and syncope. These must be represented to the participant using their common names: chest pain, shortness of breath, and light-headedness, respectively. When a symptom is reported, a clinician must be immediately notified and presented with the information provided by the participant along with the most recent ten seconds of ECG data.

FR #5: Allow goals to be set for participants - Both clinicians and participants must be able to set weekly and per-workout goals. Such goals include workout time, cumulative training load, average heart rate, distance, and average speed. Participants should be notified when their goals are achieved.

FR #6: Allow clinicians to send motivational messages - Clinicians must be able to send text messages to participants. Participants must be notified of incoming messages, and these messages must be read to participants using text-to-speech. Clinicians must be notified when participants receive messages.

FR #7: Allow participants to plot and share routes - Participants must be able to record their routes traveled during exercise sessions and share them with other participants. Participants must be able to select routes created by themselves or others to follow during a session. Such a route must be displayed on a map.

FR #8: Allow participants and clinicians to review exercise sessions - Participants must be able to view captured information for any session they have undertaken,
in addition to monthly summary data. Clinicians must be able to view this data for any participant, in addition to weekly and yearly summaries.

7.2.2 Non-Functional Requirements

REMOTE-CR’s non-functional requirements are as follows:

**NFR #1: Resource Consumption** - REMOTE-CR’s mobile application must consume sufficiently few resources on a Motorola Moto G 2013 edition Android device that, when running in the background, participants are able to use their smartphones for other tasks without REMOTE-CR adversely affecting the performance of those tasks.

**NFR #2: Performance** - When a device has access to a network connection, all data sent by the device should arrive at the web app within five seconds. When a device’s network connection is disrupted, all data sent by the device during the period of disruption should arrive at the web app within five seconds after connectivity is restored.

**NFR #3: Reliability** - All possible steps must be taken to ensure that data loss does not occur, including during periods of intermittent connectivity. Furthermore, the mobile application should be stable enough that unexpected user input, or other events, do not cause it to crash.

**NFR #4: Availability** - REMOTE-CR’s server-side infrastructure will be allowed to undergo regular scheduled maintenance. Other than during these times, 99% reliability is desired.

**NFR #5: Scalability** - REMOTE-CR’s server-side infrastructure must allow every participant currently enrolled in the programme to be active concurrently. The maximum number of concurrent participants is 30.

7.3 Design Overview

REMOTE-CR primarily consists of an Odin2-enabled smartphone application used by participants, and an associated web application used by clinicians. During an exercise session, participants send physiological, location, and symptom data to the clinician in real-time, while clinicians send motivational messages and exercise instructions back to the participant.
An overview of REMOTE-CR’s architecture is given in Figure 7.1. On a Tomcat web server we host an instance of the Odin2 Surrogate Host and the REMOTE-CR web application. The surrogate host in turn hosts the REMOTE-CR surrogate. This is paired with the REMOTE-CR mobile application to form a mobile service accessible by the web app.

Participants use the mobile application, along with a Zephyr BioHarness 3 device\(^1\), which communicates with the smartphone via bluetooth. During an exercise session, a participant’s physiological data is recorded by the BioHarness and sent to the smartphone. Here, this is sent to the surrogate, along with location information. Participants may enter further symptom report information manually, which will also be forwarded to the surrogate. The surrogate stores all received data in a MySQL database, and notifies the REMOTE-CR web app that new data is available.

\(^1\)http://zephyranywhere.com/products/bioharness-3/
Clinicians use the web app to monitor one or more participants in real-time. When the REMOTE-CR surrogate informs the web app of new information being available, the web app will query the database for this information and update its UI. The clinician may also request ECG data from a participant. In this case, the web app will invoke a REST call to the REMOTE-CR surrogate, which will in turn send an Odin2 request message to the device for the latest ECG information. Upon returning, the new information will be returned in the REST response to the web app. Clinicians may also send text messages to participants - in this case, these messages are sent to the surrogate via a REST request, and are forwarded asynchronously to the participant’s device. The web app is subsequently notified of successful transmission using Odin2’s message delivery notification functionality.

The remainder of this section further explores the design and operation of REMOTE-CR.

7.3.1 Mobile Application Design

REMOTE-CR’s mobile application has been designed for the Android operating system, and as such, makes use of Android’s Activity class to provide the user interface, and its Service class to provide background services. The app’s architecture is a layered architecture with four layers. The Operating System layer contains low-level services provided by the Android operating system. The Service layer abstracts these operating system components and provides application-specific functionality to the User Interface (UI) layer. The Database layer exposes the app’s data model, which is shared between the UI and Service layers. A detailed overview of the interactions between components in these layers is given in figure 7.2.

The Android operating system provides developers with many services and APIs which can be used to ease application development. REMOTE-CR makes use of the Bluetooth API for communication with the BioHarness, the Location service for accessing a participant’s GPS coordinates, and the Networking API for Odin2 communication.

Within REMOTE-CR’s Service layer, the following components are located:

- The Location service abstracts away the complexity and boilerplate code involved with Android’s native location API, removing the need to manually subscribe to, and unsubscribe from, Google Play Services. When using this location service, any components interested in receiving location updates need only implement a single interface and subscribe to location updates. The observing components will be notified via a callback method whenever the user’s location changes.
• The BioHarness service allows upper-level components to access the BioHarness in an object-oriented manner. Components may simply subscribe to BioHarness updates, and will be notified via a callback method when the BioHarness transmits new vital sign data. The BioHarness service takes care of translating the raw Bluetooth data packets sent by the BioHarness into Java objects usable by upper-level components.

• The Odin service encapsulates an Odin2 connection within an Android service, allowing upper-level components to send and receive Odin messages.

• The Workout service is the core of the REMOTE-CR mobile application and encapsulates all of the requirements to do with running an exercise monitoring session within an Android service. This decouples the associated logic from the user...
interface and allows workout sessions to run in the background while participants concurrently use their phones for other purposes.

Within REMOTE-CR’s User Interface layer are a number of Activity components, each representing one screen on a participant’s mobile device. Figure 7.2 shows how each of these activities uses the lower-level service, database, and OS-layer components. Section 7.3.2 covers the app’s user interface in more detail, including the app’s navigation structure and screenshots, however a summary is presented here:

- The Home activity contains links to all main application areas, allowing participants to successfully access REMOTE-CR’s functionality.

- The Login activity allows participants to login to the system. This is achieved through Odin2’s built-in authentication feature (Chapter 4 Section 4.4).

- The Profile activity allows participants to view and edit their profile information. A local copy of the current participant’s profile is stored in the database, and any changes are synchronized with the REMOTE-CR web application via Odin2.

- The Workout activity is the user interface allowing participants to conduct exercise sessions. This allows participants to view their vital sign data, location (on a map), speed, distance traveled, and messages from clinicians, as well as report any symptoms they may have during a session. The actual business logic associated with this is located within the workout service in the service layer, which exposes a model to the workout activity, which acts as a view and a controller (according to the Model-View-Controller (MVC) paradigm).

- The Goals activity allows participants to view their progress towards their weekly goals, and to add, change, or remove those goals. Local copies of a participant’s goals are stored in the local database, while any changes are synchronized with the web application using Odin2.

- The Routes activity allows participants to view routes on a map that have been traveled by themselves and other participants. The routes activity uses the location service to allow these routes to be sorted by distance from the participant’s current location.

- The Review activity allows participants to view summaries of all past monitoring sessions they have undertaken.


7.3.2 Mobile Application User Interface

This section details the design of the mobile application’s user interface (UI), and shows the various pathways available to the user for navigation through the app. Figure 7.3 shows the navigation pathways available to the user, while figures 7.4 through 7.6 show the main UI screens available within the app.

As shown in figure 7.3, when the participant first starts the app, if they have not previously logged in, they will be presented with a login screen (figure 7.4 (a)). Following a successful login, they will be directed to the profile screen to enter their details, or the home screen if this is already done. If the user has already run the app before and logged in on a particular phone, they will be sent directly to the home screen when starting the app (i.e. they need only to login once, unless they explicitly log out, which can be done on the profile screen). From the home screen, participants can access any of the app’s functionality via a list of buttons (figure 7.4 (b)).

The profile screen consists of a scrollable form which allows participants to enter any of the personal information required by them. This can be seen in figure 7.4 (c) and (d). Any changes to the participant’s profile are synchronized between the mobile device and the web app using Odin2. Participants may also log out from this screen using the button which can be seen in figure 7.4 (d). This will completely clear all of their data from the app.

---

Figure 7.3: Overview of REMOTE-CR mobile application flow. The arrows represent possible navigation options, which are explained in section 7.3.2.

---

2Smartphone app graphics provided by Illumix Studio: http://www.illumix.co.nz/
smartphone (it will still be available on the Surrogate and will be downloaded should they choose to log in again from this device or any other) and redirect the participant back to the login screen.

The goals screen allows users to enter any weekly goals they may have. There are five different types of weekly goals available - these can be seen in figure 7.4 (e). Participants may add up to one goal of each type, using the “Add Goal” button on the bottom of
the screen, and remove them using the “X” button associated with each goal. When adding a goal, a participant is presented with the dialog shown in figure 7.4 (f), with the uppermost list box only containing valid options - i.e. those goals which have not already been added. Using the goals screen, participants can also view the progress of their goals for the current week. This progress is reported through a standard progress bar widget. As with a user’s profile, any goals are synchronized with the web app using Odin2.

Figure 7.5: Screenshots for the REMOTE-CR mobile application (2 of 3).
The review screen allows participants to view summaries of any exercise sessions they have undertaken. This can be seen in figure 7.5 (a). Daily and monthly summaries are available. The daily summary occupies the majority of the screen and is scrollable to view all necessary information, while the monthly summary is shown above that, adopting a more compact form as less detailed information is required. Participants may swipe left or right on the day and month view to view summaries for different days or months, respectively. They may also use a date picker dialog to quickly browse to a specific day. The button to access this dialog is on the top-righthand corner of the screen, and the dialog itself is shown in figure 7.5 (b). Note that for a selected month, only days on which an exercise session was conducted are shown.

The route screen allows participants to view maps showing all routes they have traveled during exercise sessions. The screen is organized as a list of routes, showing a small thumbnail of each route along with length and distance information, as shown in figure 7.5 (c). Using the tabs located near the top of the screen, participants may choose to view all routes, “nearby” routes (currently set to be any route within 2 kilometers of the participant’s current location), or routes marked as “favorites”. Clicking on a route in this list will expand a more detailed view of that route, including a larger thumbnail, details of the route’s creator, the option to add or remove the route from the favorites list, and a button which will allow the participant to start a workout session pre-configured to follow the selected route. This can be seen in figure 7.5 (d). Periodically, the app will use Odin2 to synchronize with other participants. Any routes traveled by all such participants will be shown on this screen, in addition to a user’s own routes.

The workout screen is used by participants to perform an exercise session. Immediately upon navigating to this screen, REMOTE-CR will connect to Odin2 (assuming the availability of an internet connection) and the BioHarness. During the initial connection phase, the dialog in figure 7.5 (e) is shown to the user. If Bluetooth is not enabled, the participant will be prompted to enable it. If a Bluetooth connection is available, but a previously-paired BioHarness could not be found, the participant will be presented with the screen shown in figure 7.5 (f). Here, the participant may select the BioHarness from a list of previously paired Bluetooth devices (shown in the upper half of the screen) or any newly discovered devices (in the bottom half of the screen).

Once an Odin and BioHarness connection is obtained, the participant will be able to view the workout screen in its entirety. The layout of this screen is shown in figure 7.6 (a). On the top of the screen are buttons to show or hide a map showing the participant’s location, and a button to change the current workout goal (see below). On the bottom of the screen are buttons which allow the user to report any symptoms they may have, and to view any notifications sent by clinicians. Just above these buttons is the button which
Chapter 7. REMOTE-CR

Figure 7.6: Screenshots for the REMOTE-CR mobile application (3 of 3).

will start or stop a session. As seen in figure 7.6 (a), a workout session in progress will display a “Stop” button here, along with a timer showing the current workout duration. If a workout has not yet started, this will instead display a green “Start!” button, along with the text “Start!” The central part of the screen displays four tiles, each showing one of REMOTE-CR’s available exercise statistics. Participants may click on these tiles to customize the statistic shown in each tile, to their own personal preference.
When the participant clicks the “Show Map” button, a map showing the participant’s location will be displayed. This map will occupy the entire screen, with all necessary buttons superimposed over the map. As shown in figure 7.6 (b), two small tiles displaying exercise statistics are shown instead of four large tiles, to allow the participant to view the map. If the participant is following an existing route during the current exercise session, this is also shown on the map view. An example can be seen in figure 7.6 (c).

By clicking the button on the top-right corner of the screen, the participant will be allowed to select a goal for the current workout. This differs from the goals shown earlier in this section in that a goal selected here is to be achieved during a single exercise session (as opposed to a week-long goal). The text displayed on this button will change according to the type of goal. by default, no goal is selected, in which case the button will display the text “Free Workout” as can be seen in figure 7.6 (a). Participants may set a duration, distance, training load or route goal using the dialog box that appears upon clicking this button. The dialog is shown in figure 7.6 (d). If a goal is set for a workout, a notification will be played to the user when that goal is reached, at which time they will be able to stop their workout, or continue.

During a workout, a participant may experience symptoms associated with their particular medical condition. As per the requirements, REMOTE-CR allows participants to report those symptoms to their clinician. This is achieved using the button in the bottom-left corner of the screen. Clicking this button will show the dialog in figure 7.6 (e). The upper half of this dialog shows any symptoms previously reported during the current workout, while the bottom half allows the participant to report a new symptom, along with the severity of that symptom.

During a workout, clinicians may send text messages to participants. Whenever a notification is received, it is displayed in the Android notification bar at the top of the screen. The message is also read to the participant using Android’s text-to-voice feature. Participants may view any messages received during a workout using the button on the bottom-right corner of the screen. Clicking this button will display the dialog shown in figure 7.6 (f). Each notification received will be displayed here in a list, along with a button for each notification which will replay that message using text-to-voice (the “speech bubble” button).

When the user completes a workout, the review screen will be shown (figure 7.5 (a)) with the completed workout pre-selected for perusal by the participant.
7.3.3 Web App and Surrogate Design

In addition to the mobile application used by participants, REMOTE-CR consists of a web application used by clinicians to monitor and review participants. The web application is deployed across three processes, shown in blue in figure 7.7: the Browser used by clinicians, the main Web Application Backend running in a web container and containing the majority of the server-side business logic, and the Odin2 Surrogate, running within a Surrogate Host, which interfaces with the mobile app and provides real-time data to the web app. The web application was implemented conjointly with the University of Auckland Research Programming team. The web application and surrogate components were developed as separate projects to ease this concurrent development effort.

The web application backend contains logic allowing clinicians to login, view and interact with in-progress exercise sessions, and review participants’ past sessions. All data associated with past sessions and login details is stored in a MySQL database, which is read from by components within the backend. This functionality is achieved through the use of web controllers (as part of the model-view-controller (MVC) programming model) which act in response to requests from clinicians (over the public internet via a web browser) or incoming live monitoring data from the surrogate. These controller components can be seen in figure 7.7. The web application is implemented using Spring MVC.

The web app components allowing clinicians to interact consist of JavaScript and HTML pages provided to the clinician’s browser. The JavaScript libraries used by the web app include Knockout\(^3\), Apache Tiles\(^4\), and JQuery\(^5\). Communication between the browser and the backend is through a REST interface.

The Surrogate manages interaction with individual participants. As well as managing participant login through Odin2’s built-in authentication feature (see chapter 4 section 4.4), the surrogate also contains a Participant Synchronization component which manages the persistence of all participant data. When participants first login using a particular device, and periodically thereafter, the surrogate will synchronize with that device to ensure both the local and remote copies of this data are accurate. The surrogate also may receive asynchronous messages from participants’ devices containing physiological and location data during an exercise session. These are received by the surrogate’s Incoming Data Manager, which persists the data and forwards it to the web app backend’s monitoring controller. Finally, the surrogate contains a Session Controller which exposes its own REST service to the backend, allowing it to send notifications from clinicians.

\(^3\)[http://knockoutjs.com/]
\(^4\)[https://tiles.apache.org/]
\(^5\)[http://jquery.com/]
to participants, as well as request current ECG data. The session controller translates these REST messages into Odin2 messages which are then forwarded to the appropriate participant.

### 7.3.4 Web App User Interface

To use the web application, clinicians must first log in using a simple form shown in figure 7.8. When they have done so, they are initially presented with a list of all participants currently enrolled in the trial. This can be seen in figure 7.9. This page shows the name of each participant, along with details such as when they last exercised, when the clinician last updated the participant’s prescription, and the total number of exercise sessions that participant has performed. From here, clinicians may add new participants by clicking the “Add Participant” button at the bottom of the page, or view a detailed summary for a single participant by clicking on that participant’s name.

When adding a participant, the clinician is presented with a page as shown in figure 7.10. Here they enter the participant’s name, as well as a username which the participant will use to log in using the mobile application. They may also enter warning thresholds for heart rate and breathing rate. These will cause an audible alarm to sound if exceeded during a live monitoring session. Finally they may enter initial values for a participant’s weekly goals.

When viewing an exercise history summary for a participant, a clinician is initially presented with a page as shown in figure 7.11. Here, the clinician can view the participant’s compliance with their exercise programme, as well as edit the warning thresholds for that participant. The “Disconnect” button at the bottom-right will stop that participant’s exercise sessions from appearing in the live monitoring view, and the “View Details” button may be clicked to view detailed information about a participant’s entire exercise history.

When a clinician clicks on the “View Details” button in figure 7.11, they are presented with screens such as those shown in figure 7.12. Clinicians may view daily, weekly, monthly, or yearly summaries for a participant. The daily and monthly summaries are shown in figure 7.12 (a) and (b), respectively. Using the daily view, a clinician may select from distance, heart rate, %HRR, breathing rate, speed, and load using the bar on the top of the screen, and select a date using the month selector just below this, the year selector to the left, and the day selector just below the year selector. Clinicians will then be presented with a graph showing how the selected exercise metric changes over the course of the exercise session performed on that day. In addition, any symptoms reported by the participants during that session, and any requests for ECG data made by the clinician,
Chapter 7. REMOTE-CR

Figure 7.7: Overview of components within the REMOTE-CR web app and surrogate, showing the connections between them and the utilization of the Odin2 middleware.
are visible on the graph at the appropriate timestamp. To the right of the primary graph, clinicians can view any of these symptoms and ECG requests in list form. Clicking on any item in this list, or on the primary graph, will show the ECG data associated with that symptom or request on a separate graph, shown on the bottom-right of the screen. Also to the right of a primary graph, the clinician can view any messages that were sent to the participant during this session.
The weekly, monthly, and yearly summary views instead show a bar graph for the selected fitness metric - this can be seen in figure 7.12 (b), which shows the monthly summary for a participant. Each bar on the chart groups all recorded data for that particular metric over a particular timespan. The data are grouped by averaging (heart rate, %HR, breathing, speed) or summing (distance, load), depending on the metric. On each bar, a symptom summary icon is shown. Hovering over this icon will show all symptom reports and ECG requests during the time period associated with that bar. The time period for weekly and monthly summaries is one day, while the time period for yearly summaries is one week.

REMOTE-CR’s primary functionality is to allow participants to undertake exercise sessions, and have themselves be monitored by clinicians in real-time. Clinicians can monitor several participants at the same time. Figure 7.13 shows the clinician’s view of a live monitoring session with four currently active participants. Each active participant is associated with a monitoring panel. For each participant, the clinician may view a graph showing how a selected exercise metric is changing in real-time. To the right of the monitoring panes, clinicians may view a map showing the current location of all active participants, along with the ECG data associated with the most recent symptom report by, or ECG request to, a participant. Clicking the “ECG” button on a participant’s monitoring pane will show the latest available ECG data for that participant, and cause the ECG pane to become associated with that participant. Finally, clinicians can see a Notifications pane, showing all symptom reports and ECG requests for all participants,
and a notification for each time a participant’s heart rate or breathing rate threshold is exceeded.

When the ECG pane on the live monitoring page is associated with a participant (either by clicking a participant’s “ECG” button or when a new symptom report from a participant arrives), clinicians may click the “Get New” button to retrieve a current ECG reading from that participant. When they do so, they are presented with a window as shown in figure 7.14 (a). This window takes up a large proportion of the screen, giving the clinician a more fine-grained view of the ECG data. Clicking the “Refresh” button will cause the latest ECG data to be requested from the participant. When the data arrives
Figure 7.13: REMOTE-CR web app live monitoring session in-progress with four active participants

(a) Detailed ECG reading  
(b) Message sending window

Figure 7.14: REMOTE-CR web app additional windows shown during live monitoring sessions

at the browser, the graph is automatically updated. Clicking the “Close” button will dismiss the pane.

Clinicians may send messages to a participant by clicking the “Message” button on that participant’s monitoring pane. When this is done, the clinician will be presented with a window as shown in figure figure 7.14 (b). Here, clinicians may type any text message, then send it by clicking the “Send Message” button. A notification is sent back to the web application once the message has been received by the participant.
7.4 Leveraging the Odin2 Middleware

REMOTE-CR is a non-trivial software system which must support timely, reliable exchange of many kinds of data between smartphones, a backend infrastructure, and a web application. Odin2 is sufficiently reliable, performant, and expressive enough to be used for such kinds of applications.

Referring to REMOTE-CR’s list of functional requirements, FR #1 requires participants to be able to authenticate from smartphones. Odin2’s authentication feature directly meets this requirement.

FRs #2, #5, #7 and #8 each require synchronization of various data between smartphones and the backend. Odin2’s surrogate host database maintains a copy of all such participant data, devices may send asynchronous data updates whenever a participant modifies the local smartphone copy. If communication is temporarily disrupted while the synchronization operation is in progress, Odin2’s interconnect will make sure data is not lost or duplicated, ensuring data integrity. Furthermore, as Odin2 maintains a two-way communication pathway between the surrogate and each device, any modifications made by clinicians using the web app are pushed to devices.

FR #3 requires smartphones to stream physiological and location data to a clinician’s web application. Odin2’s streaming message abstraction allows this data to be sent to the surrogate as it is generated by sensors. The clinician’s web app, which will have previously subscribed via REST interface to updates from the participant’s device, will be updated via a REST callback. This shows the benefit of Odin2’s streaming abstraction, in addition to its interoperability through supporting a REST interface. FR #4 is met by sending participant symptom reports to the surrogate in the form of asynchronous messages. Similarly to FR #3, the new symptom reports arriving at the surrogate will be transmitted to client via a REST callback.

FR #3 also requires that clinicians are able to request up-to-date ECG data from participants at any time. To avoid unnecessary bandwidth usage, this data is not continuously streamed to the surrogate. Instead, we truly leverage Odin2’s capability as a mobile service provisioning middleware by exposing the participant’s device as a REST service. The clinician’s web app acts as a REST client, making a REST request for the latest ECG data when required by clinicians. The surrogate translates this into an Odin2 request message, which is sent to the device. Here, the device responds with the latest ECG data, which is then translated back into a REST response before returning to the web app. Translation is handled automatically by the Odin2 middleware.
FR #6 requires that clinicians are able to send motivational messages to participants. Here we again leverage Odin2’s mobile service functionality by allowing the web app to send REST requests containing these messages. To avoid unnecessarily delaying the web application, this request returns immediately. At the surrogate, Odin2 sends an asynchronous message to the device. Using Odin2’s message delivery notification feature, the surrogate will be notified with a UserAcknowledged status when the message has been seen by the participant. This notification is then forwarded back to the web app.

Odin2 also helps REMOTE-CR meet some of its non-functional requirements. NFR #1 requires a relatively low resource consumption rate on participant smartphones, to allow the smartphone to be used concurrently for other activities. Odin2’s surrogate architecture allows some processing to be performed on the surrogate, contributing to NFR #1.

NFR #2 requires that important data reaches clinicians in a timely fashion. Odin2’s performance has been evaluated in Chapter 8 section 8.1 and has been shown to be sufficient to meet this requirement. Furthermore, test harness use (Chapter 8 section 8.4.2) has shown REMOTE-CR’s performance requirement to be met.

NFR #2 additionally requires that in the event of a network error, data should be sent as soon as possible. NFR #3 further requires that data loss should not occur. Odin2’s interconnect channel, whose reliability has been shown in the experiments in Chapter 8 sections 8.1 and 8.4, allows REMOTE-CR to meet these requirements.

NFR #4 requires high availability other than scheduled downtime. While we cannot guarantee that the system has the 99% availability specified in the requirements, we can say that REMOTE-CR as thus far been available when required (see Chapter 8 section 8.5 for further details). This availability may be partly due to Odin2’s dependability, though other decisions such as the choice of hosting environment for the surrogate host would also play a major role in meeting this requirement.

NFR #5 requires REMOTE-CR to support 30 concurrent active participants. We have verified this in Chapter 8 section 8.4.2.

### 7.5 Summary

In this chapter we have detailed the REMOTE-CR cardiac monitoring and exercise system, which uses Odin2 as its underlying middleware. As detailed in section 7.4, Odin2’s capabilities have been leveraged in order to meet REMOTE-CR’s functional and non-functional requirements. Odin2’s comprehensive messaging abstraction, in combination
with its REST service support, has allowed REMOTE-CR to meet its functional requirements and provide interoperability with the REMOTE-CR web application using standard web technologies. Furthermore, REMOTE-CR’s requirements for performance, reliability, and availability have been met by leveraging Odin2’s own non-functional requirements in these areas.

REMOTE-CR itself is currently being used at the time of writing this thesis, as part of an ongoing clinical trial. As shown by its evaluation in Chapter 8 section 8.5, it has proven to be dependable and has received positive feedback from NIHI. Currently, the trial has sourced participants from Auckland and Tauranga, New Zealand. An additional trial using the same software is due to start at the Department for Prevention, Rehabilitation and Sports Medicine\(^6\) at Technische Universität München, Germany in April 2015. This positive response shows the usefulness of REMOTE-CR, and, by association, Odin2.

\(^6\)http://www.sport.med.tum.de/en
Chapter 8

Evaluation

In chapters 4 through 7 of this thesis, we have introduced our middleware, Odin2, which aims to meet the challenges inherent in mobile service provisioning introduced in Chapter 2 while being expressive enough to create a diverse set of mobile services such as those introduced in Chapter 1. We then introduced OdinTools, which aims to generate partial service implementations for multiple platforms, and demonstrated how it can be used to help create a simple mobile service. Next we introduced the Odin Test Harness, which has proved instrumental in identifying particular bugs throughout the development process. In particular, the ability to simulate various network states for large numbers of devices in a controlled and repeatable manner has allowed us to track down bugs which would otherwise rarely appear during real-world operation. Finally we have shown that Odin2 is expressive enough to act as the underlying middleware for a real-world patient monitoring system, REMOTE-CR.

In this chapter we further evaluate each of these four solutions. In section 8.1 we evaluate the performance of Odin2. This includes an evaluation of Odin2’s messaging and vertical handover mechanisms, in addition to a comparison with leading industry solutions which shows that Odin2 provides comparable or superior performance. In section 8.2 we then show that Odin2 is expressive enough to create each of our example services, before showing some evidence in section 8.3 suggesting that OdinTools increases productivity by allowing more rapid development of cross-platform mobile services. In section 8.4 we then evaluate the test harness. Here we show how the test harness was used during the development of Odin2 and REMOTE-CR, and the kinds of bugs it helped us to detect. Next, in section 8.5 we then detail how REMOTE-CR is being used, and provide feedback from participants as evidence that the system is functional. This shows that Odin2 is a reliable middleware, and that the test harness has helped prevent a significant
Chapter 8. Evaluation

proportion of bugs from reaching production. Finally we conclude the chapter in section 8.6.

8.1 Performance Evaluation

One of Odin2’s primary goals is to achieve a performance level which is comparable to widely adopted mobile communication solutions in use today. In this section we show that this is the case by evaluating Odin2’s performance with respect to several push notification technologies. As discussed in Chapter 2, push notification is being increasingly adopted in today’s smartphone applications to asynchronously relay messages to mobile users. While not originally developed for mobile service provisioning, push notifications address its reachability and addressability challenges, and they do so in a fashion that is seen by the development community as being sufficiently performant and robust.

In this section we compare Odin2’s interconnect to five push notification technologies. These are Google Cloud Messaging (GCM), Apple Push Notification Service (APNS), Blackberry Push Service (BBPS), Microsoft Push Notification Service (MPNS), and IBM Message Queuing Telemetry Transport (MQTT). We compare Odin2 to these technologies with respect to their reliability and efficiency.

A solution’s reliability in this case refers to the extent to which it offers message delivery guarantees. A reliable solution will have to consider a mobile device’s transient network connectivity due to loss of battery life, mobile and lack of network coverage, or simply being switched off.

A solution’s efficiency can refer to several metrics: responsiveness, bandwidth consumption, throughput, and energy consumption. Responsiveness refers to a solution’s ability to deliver messages to a mobile device with low latency. Bandwidth consumption refers to overheads inherent in the messaging protocol with respect to the size of the payload itself. Throughput refers to the maximum amount of data that can be sent under certain network conditions. Energy consumption refers to a solution’s drain on the device’s battery life during operation.

8.1.1 Experimental Setup

To assess the performance of each solution, we ran a series of experiments evaluating each solution’s reliability, responsiveness, bandwidth consumption, throughput, and energy consumption. When performing these experiments we considered the potential threats to the validity of the results, requiring procedural rigor and attention to the study’s
inherent heterogeneity. Tables 8.1 and 8.2 identify these threats and explain how we addressed them.

<table>
<thead>
<tr>
<th>Use of different mobile carriers for hosting devices may introduce bias.</th>
</tr>
</thead>
<tbody>
<tr>
<td>We used a single carrier (2degrees). In all cases except for BBPS (where a plan was required with access to Blackberry gateways) we used a basic plan typical for everyday mobile users.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use of different tools for measuring bandwidth may yield data that cannot be meaningfully compared.</th>
</tr>
</thead>
<tbody>
<tr>
<td>For measuring Wi-Fi bandwidth, we used an AirPcap wireless packet capture device in conjunction with the Wireshark protocol analyzer. For 3G, we used device operating system APIs where supported. We measured the similarity in measurements over Wi-Fi obtained from device operating systems and AirPcap and found their variance to range between 0.3% and 4%.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use of different devices may introduce variability in measurements and lead to device-dependent results.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use of different devices is necessary since each push service is generally dependent on a particular device platform. However, bandwidth consumption and loss rates are device-independent, being determined by the protocol being used. Latencies are similar in that for all but the load tests, devices’ processor utilizations are low. Power consumption is device-specific since we measure battery discharge. To compensate for this, we devised an energy efficiency index to assess push protocols out of context of a particular device. We used the following four devices, all contemporary and similarly high-end at the time of the experiments:</td>
</tr>
<tr>
<td>• For BBPS: a Blackberry Torch 9860 smartphone with Blackberry 7.1</td>
</tr>
<tr>
<td>• For APNS: an Apple iPhone 4S running iOS 6</td>
</tr>
<tr>
<td>• For MPNS and Odin2 for Windows Phone: a Nokia Lumia 800 device running Windows Phone 7.1</td>
</tr>
<tr>
<td>• For GCM, MQTT, and Odin2 for Android: a Samsung Galaxy S II running Android 4.0.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Locality of push gateways and surrogate hosts may introduce bias.</th>
</tr>
</thead>
<tbody>
<tr>
<td>While we cannot control the location of the push gateways for BBPS, GCM, MPNS, or APNS, we made sure to locate the MQTT gateway and the Odin2 surrogate host as similarly to the other solutions as possible. To do this we hosted each on a Windows Azure virtual machine running in a West US data center.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 8.1: Threats to validity arising from heterogeneity, and their countermeasures.</th>
</tr>
</thead>
</table>

For responsiveness, reliability and bandwidth consumption, we ran two-hour experiments involving sending messages to devices, each with a 256 byte payload, at a rate of 3 per minute. We repeated each experiment over Wi-Fi and 3G network connections. For responsiveness, we measured the latency between sending a message and the target device receiving it. We measured bandwidth consumption in all cases over Wi-Fi networks, and
Variability in device positioning may lead to different results for latency, bandwidth and power consumption.
We used a common university lab for the all experiments ensuring that the same physical access point (cell tower or Wi-Fi switch) was used for all experiments.

Varying the time when experiments are run may lead to different results.
We limited our experiments to business hours (9am-5pm) Monday to Friday. Using a consistent range of times helps to minimize any variability arising from, e.g., network congestion and access point usage patterns.

Without synchronizing device and content provider clocks, latency figures are meaningless.
For the experiments involving Android, iOS, and Windows Phone, we handled clock synchronization using the NTP protocol. For Blackberry, we used Blackberry Desktop software 7.0.0.59.

Sharing of wireless networks may introduce noise from entities not involved in the experiments.
For Wi-Fi experiments, we operated a closed network through a switch connected to the University’s wired Ethernet. AirPcap monitored the link connecting the single hosted device to the switch. For 3G experiments, we have no control over sharing of the medium, but used common day / time ranges to minimize variance.

Participant devices and computers that perform arbitrary processing may affect data collection.
We ensured that the only app running on the device at the time of the experiment was the experimental app. Superfluous device features (e.g. the radio transceiver for Wi-Fi experiments, GPS and account syncing) were disabled. Similarly the client machine was dedicated to running that application.

Single experiments cannot be considered to be representative of monitored behavior.
We replicated each experiment 5 times, taking statistical measures including mean, median and standard deviation as appropriate.

Table 8.2: Threats to validity arising from experimental procedure, and their countermeasures.

We measured reliability over a stable network connection by measuring the drop rate of messages over both continuous and interrupted connections. In addition to the two-hour test, we ran a load test to determine the rate at which each solution could sustain requests with consequent messages being delivered to the target device. This also allowed us to measure throughput. We capped the number of push requests at 5000, this being the lowest quota limit (BBPS) across the service providers.

We measured each solution’s response to unreliable network connections by forcefully closing a device’s network connection for various periods of time while performing a
two-hour experiment as above, and logging the proportion of messages sent during the periods of disconnectivity which arrived at the device once connectivity was restored. To further evaluate Odin2, we conducted a more comprehensive version of the experiment, which involved alternative experimental durations, additional periods of network closure, and maintaining 3G connectivity while cycling Wi-Fi connectivity to test Odin2’s vertical handover performance.

For power usage, we extended the two-hour experiments to 24 hours duration, retaining a request rate of 3 per minute with a 256 byte payload. We set out to measure the remaining battery level after running the experiment for each solution. In cases where the battery became exhausted in less than 24 hours, we recorded the duration for which the device remained operational. We conducted the power usage experiments over both Wi-Fi and 3G connections.

We repeated each of the above experiments for each solution, other than the vertical handover tests. Note that MQTT and BBPS are configurable with respect to their message delivery guarantees. Over and above the base fire-and-forget (F&F) level of service, BBPS provides at-least-once (ALO) semantics, and ALO-ACK semantics, which require that explicit acknowledgments of message receipt are sent from devices back to message senders. MQTT also provides ALO, in addition to at-most-once (AMO) semantics. Odin2 supports both Android and Windows Phone devices. For each of these solutions, we have separately evaluated each variant of the software.

8.1.2 Results: Bandwidth Consumption and Responsiveness

Figure 8.1 shows the average bandwidth usage for each solution when sending 256-byte messages. In this and all subsequent figures, (A) refers to the Android platform, while (WP) refers to the Windows Phone platform, for the Odin2 variants. The y-axis in the chart shows the actual number of bytes sent across the wireless network to send a 256 byte payload. Data sent over and above the payload is indicative of that protocol’s bandwidth overhead. Note that, bandwidth usage figures for APNS, MPNS, and Odin2 for Windows Phone over 3G connections could not be obtained. As explained in table 8.1, we used OS-specific functions for measuring 3G network usage. Such services did not exist for iOS and Windows Phone 7.1 at the time these experiments were performed.

Figure 8.2 presents the bandwidth usage data in terms of an efficiency value. We define a solution’s bandwidth efficiency for a given payload size, \( b_p \), as follows:

\[
b_p = \frac{p}{d_p}
\]
where \( p \) is the payload size, in bytes, and \( d_p \) is the amount of data required to transfer the payload, also in bytes. A value of \( b_p = 100\% \) would indicate that for payload size \( p \), a solution sends no additional data - i.e. there are no overheads. A value of \( b_p = 50\% \) would indicate that half the data being sent across the network to deliver a payload of size \( p \) is attributed to protocol overhead.

From figures 8.1 and 8.2, we can see that Odin2 performs well in terms of bandwidth efficiency for the given payload size of 256 bytes when compared with the industry standard solutions, achieving efficiencies of between 64% and 66% - comparable to the best-performing industry solutions, BBPS F&F (3G), BBPS ALO (3G) and MQTT F&F (Wi-Fi).
Other than Odin2’s performance, one other notable result was that of the BBPS solutions. In all other cases where both Wi-Fi and 3G bandwidth usage could be measured, solutions consumed more bandwidth over 3G. This is likely to be due to additional protocol overhead to compensate for intermittent mobile connectivity (for example, Odin2’s interconnect sends keep-alive messages at more regular intervals over 3G to ensure TCP channels are kept valid). In BBPS’s case, the reverse occurred. This implies the use of compression by BBPS to reduce consumption of valuable 3G network resources.

**Figure 8.3:** Average protocol responsiveness over Wi-Fi

Figure 8.3 shows the responsiveness of each protocol over Wi-Fi, while figure 8.4 shows responsiveness over 3G. The y-axis on these charts show the delay, in milliseconds, between the experimental software initially beginning to send the message, and the device receiving the last byte.
From the figures we can see that Odin2’s responsiveness is comparable to the industry standard solutions. We can directly compare the latencies of Odin2 and MQTT, whose server-side components were hosted on the same US-based virtual machine. Here we can say that Odin2 offers comparable latency over 3G, and superior latency over Wi-Fi. The superior performance of Odin2 over Wi-Fi is due to its relatively efficient protocol (from figure 8.2). The difference between Odin2’s latency and that of protocols with similar efficiency (some BBPS and MQTT variants) could be due to additional client loading on the other push gateways (we are unable to host a private BBPS gateway, for example) or other internal mechanisms regarding these solutions - which are primarily black-box systems. The large difference disappears over 3G connections. Primarily this is due to 3G latencies being much higher in general. Another reason may be due to differences in the way our mobile network operator, Vodafone NZ, manages international traffic.

8.1.3 Results: Throughput

Figure 8.5 shows the average rate, in messages per second, at which 5,000 messages can be sent using each solution with 100% reliability over Wi-Fi. Wi-Fi was chosen over 3G for this experiment to shift the bottleneck onto the messaging protocols themselves, rather than the network.

For rates higher than those in the chart, each solution’s reliability broke down in various ways. BBPS and MQTT dropped messages. GCM would block the message sender from achieving a faster rate and, furthermore, would cap the rate of message delivery to devices at around 10 per second, regardless of the rate of the message sender. MPNS would disallow the message sender from sending at a faster rate, and would notify the sender via an HTTP error code in the response. APNS would simply overwrite any pending messages at the gateway, causing data loss. Odin2 for both Android and Windows Phone would not allow higher rates due to a CPU utilization bottleneck. The difference between the two solutions in the graph is due to the difference in CPU capacity of the two smartphones.

Here we can see that Odin2 for Android supports significantly higher reliable throughput than all industry standard solutions other than MQTT. Experimental results appear to show that the limiting factor in Odin2’s interconnect protocol is its CPU utilization when unmarshaling messages on the device. This implies that devices with increased CPU power will show higher throughput.

For the comparative throughput experiments shown in figure 8.5, the number of messages was capped at 5,000 due to this being the maximum number of messages allowed by BBPS for a single device in a single day. To further evaluate whether Odin2 could sustain
this throughput level for longer periods, we repeated the experiment with 10,000, 20,000, and 50,000 messages, retaining the same throughput rate shown in figure 8.5. During these additional experiments, as with all other experiments in section 8.1, no messages were lost or duplicated.

8.1.4 Results: Reliability

Section 8.1.3 above shows each solution’s throughput. A solution’s throughput is related to its reliability over a stable network connection - we did not consider a solution to have a certain level of throughput if messages were lost or duplicated at that level. To further evaluate each solution’s reliability, we forcefully disabled the network connections of experimental devices during the operation of a two-hour experiment, with 256-byte messages being sent three times per minute. During the experiment we caused a single blackout period of two minutes, and another of 30 minutes. Figure 8.6 shows the proportion of messages sent during the blackouts which successfully arrived at devices after network activity was restored following these blackout periods.

All solutions with the exception of APNS and MPNS delivered all messages during the experiment. APNS’ aggressive notification overwriting behavior meant that only the last message received by the APNS service was queued for delivery to the offline device, while MPNS buffered only 32 messages before discarding further requests. While BBPS and MQTT did not incur data loss, many messages arrived out-of-order following reconnection after a half-hour blackout period. Only GCM and Odin2 provided in-order delivery of all messages during this experiment. In the case of Odin2 this confirms that its method for detecting out-of-order messages is functional.
Chapter 8. Evaluation

To further evaluate Odin2, this experiment was repeated with a four-hour duration, during which connectivity was disrupted in additional ways:

1. A single, one-hour blackout period
2. A single, two-hour blackout period
3. Five minute periods of 30 seconds of connectivity followed by 30 a second blackout
4. Maintaining connectivity over 3G, but cycling Wi-Fi connectivity as above

Furthermore, in addition to other experiments in which an Odin2 client sent messages through the surrogate host to a device, the experiment was additionally repeated with the Odin2 host device sending messages back to the surrogate. In all cases, all messages arrived with no data loss, duplication, or out-of-order delivery.

During experiment 4 above, where 3G connectivity was maintained but Wi-Fi connectivity was toggled, Odin2 maintained connectivity throughout the experiment. Odin2 used Wi-Fi when available, and 3G otherwise. The average time taken to detect the network change and switch networks is shown in figure 8.7. Handover times did not depend on when the network change occurred in relation to a message delivery. This can be seen in 8.7 - the “messaging” bars indicate vertical handover which occurred while a message was in the process of being sent, while the “no messaging” bars indicate vertical handover which occurred while no message was being sent. This is to be expected, as Odin2’s interconnect is always listening over a TCP connection and should be notified of a network change, via an IOException, as soon as the change occurs, as opposed to
the next time a message delivery is scheduled. From the graph we can see that handover to 3G is slower than handover to Wi-Fi. This makes sense, as the times in the graph include the time taken to reconnect to the surrogate host - which, intuitively, is longer over the slower protocol.

### 8.1.5 Results: Battery Consumption

To test the battery consumption of each solution, initially we left each smartphone on, but idle, logging its battery level 20 minutes for 24 hours. We then recharged each device and then sent three 256 byte messages to each device per minute, for 24 hours or until the battery was depleted - again logging battery levels every 20 minutes. We repeated this experiment over both Wi-Fi and 3G networks. Figure 8.8 shows the remaining battery life for each solution over Wi-Fi.

From figure 8.8, we can meaningfully compare the BBPS variations with each other. We can additionally compare the MQTT options, GCM, and Odin2 for Android, due to these solutions using the same physical device. From here we can see that the more reliable BBPS options do not affect battery consumption over Wi-Fi. We can also see that Wi-Fi battery usage for GCM and Odin2 for Android are similar - and both of these solutions provide significantly lower battery consumption than any MQTT variant.

We cannot draw further conclusions from figure 8.8. Primarily this is due to several different physical devices being used. Furthermore, the MPNS, APNS, and Odin2 for Windows Phone solutions cannot be evaluated in any way from this graph due to their devices running out of battery before the conclusion of the 24 hour experiment. We cannot say that this is due to poor protocol energy efficiency - the MPNS device must
activate its screen at all times to receive messages, while the APNS device’s screen is temporarily activated each time a message arrives. Similarly, when running on Windows Phone, the screen must remain active to allow Odin2 to remain running in the foreground, and thus, perform necessary background processing. Because of this, we additionally left the screen on for these devices during the idle tests.

In an attempt to overcome these consequences of heterogeneity, we have devised an energy efficiency index for each solution. The energy efficiency index relates a device’s operational time when receiving messages over a certain network channel to its operational time when idle. To calculate the energy efficiency index, we initially calculate the raw index for a solution over a particular network, $r_s$ as follows:

$$r_s = \frac{op_s}{idl_s}$$

where $op_s$ is the operational time for a solution over a particular network when sending 256-byte messages every 20 seconds, and $idl_s$ is the idle time for the device on which a particular solution operates. We then calculate the energy efficiency index for a solution, $i_s$, as follows:

$$i_s = \frac{r_s}{r_{max}}$$

where $r_{max}$ is the maximum $r_s$ for all solutions over a particular network. This gives a value from 0 to 1. Higher values for $i_s$ indicate that a solution tends to have less impact on its host device’s battery life than solutions with lower $i_s$ values when utilizing the
same wireless network. Figure 8.9 shows the energy efficiency indices for all solutions over both Wi-Fi and 3G.

![Energy Efficiency Index](image)

**Figure 8.9:** Energy efficiency index for each protocol over 3G

From figure 8.9 we can see that the Windows Phone-based solutions have significantly higher $i_s$ compared with other solutions - though care must be taken when considering these values as the power drawn by the device’s screen may contribute a significant constant value which our equation does not consider. Similarly it is difficult to draw conclusions regarding the Apple solution. Comparisons amongst other solutions show that Odin2 for Android is comparable to GCM over Wi-Fi, and slightly more energy efficient over 3G. GCM in turn is comparable to BBPS and MQTT over 3G, and significantly more efficient over Wi-Fi. Overall we can say with reasonable confidence that Odin2 is, on average, no less energy efficient than the industry standard solutions.

### 8.2 Odin2 Expressiveness

One of Odin2’s primary goals is to enable a diverse set of mobile services, such as those introduced in Chapter 1. To demonstrate Odin2’s expressiveness, we have implemented each of these sample services using Odin2 as the underlying middleware.

Figure 8.10 shows the Android and Windows Phone user interface of each of our example services: a mobile media service, a patient monitoring service, and a social networking service. Each of these services has been implemented on both the Android and Windows Phone platforms, with the exception of the patient monitoring service, as Windows Phone 7 does not support a Bluetooth API sufficient to connect to a BioHarness. A brief description of each sample service is provided in this section.
Chapter 8. Evaluation

8.2.1 Mobile Media Service

A simple mobile service which allows clients to request photos from a device’s camera. Requested photos are cached at the surrogate host to preserve device bandwidth in case of multiple requests for the same image. In figure 8.10 (a) and (d) we can see that the mobile app’s user interface is very simple - it simply contains buttons for connecting.
to, and disconnecting from, the surrogate host. The application’s logic is driven entirely by requests from remote clients. Clients may connect using any REST-capable software, such as a web browser. They may request a list of available images by sending a GET request to 
\texttt{surrogate/mobile-media/listImages?username=<user>} - this will return a JSON-formatted list of image names. They may request a single image by issuing a request to \texttt{surrogate/mobile-media/getImage?username=<user>&image=<imagename>}. Figure 8.11 shows this service being accessed from within the Internet Explorer browser.

\textbf{Figure 8.11:} Mobile media service client, accessed from Internet Explorer

\subsection*{8.2.2 Patient Monitoring Service}

This service communicates with a Zephyr BioHarness, similar to REMOTE-CR (Chapter 7), to obtain a user’s heart rate and breathing rate. It additionally gets a user’s location from the smartphone’s GPS. It exposes this information as an Odin2 service. Clients may get a user’s location by issuing REST request to \texttt{surrogate/patient-monitoring/getLocation?username=<user>}. Similarly they may request vital signs from \texttt{surrogate/patient-monitoring/vitalSigns?username=<user>}. These web methods will each return a JSON string containing the requested data. Clients may also subscribe to changes in a user’s vital signs or location by issuing a request to \texttt{surrogate/patient-monitoring/subscribe?username=<user>}, passing in a callback URL in the request body. The callback will be issued with a JSON string containing vital sign and / or location data from the appropriate user, when such data changes. As shown in figure 8.10 (c), a user’s location and vital sign data is also shown to them, on their own device.
8.2.3 Social Networking Service

This service allows an arbitrary number of users to share their location and communicate via simple text messages. Each user’s smartphone both provides and consumes an Odin2 service. Location data is obtained from a smartphone’s GPS and is multicast to each of a user’s contacts. Users may also send text messages, which are also multicast in a similar manner. The surrogate keeps a record of each user’s contacts list. The screenshots in figure 8.10 - (b) and (e) - show a two-user network consisting of Andrew (Android) and Bob (Windows Phone). Screenshot (e) was taken before any text message exchange occurred, while screenshot (b) was taken afterward.

8.2.4 Discussion

One of Odin2’s primary goals is to allow the development of a diverse set of mobile service types, such as our sample applications introduced in Chapter 1. Table 8.3 shows how each sample service uses Odin2’s features to meet its requirements. This demonstrates that Odin2 is indeed expressive enough for this purpose. Furthermore, any services implemented using Odin2 as a middleware will inherit its performance and reliability characteristics without additional developer effort.

8.3 Productivity with OdinTools

The primary goal of OdinTools is to increase developer productivity by generating partial, cross-platform implementations of mobile services. In this section we introduce the Odin Application Model (OAM) for each of our sample services, as defined using OdinTools. We then investigate the proportion of generated code versus that which must be manually implemented to complete each service.

8.3.1 Mobile Media Service Definition

Figure 8.12 shows the mobile media service OdinTools definition. We have defined a new type - Image - which represents an image to be transferred. We have also defined two request-response messages - one which represents a request for a single image, and another which represents a request for a list of all images on a device.

From this definition, OdinTools creates an app skeleton for both Windows Phone and Android devices which allows them to receive and process these messages. It also creates
App Description | Odin2 Features Used
--- | ---
Mobile media service | • *Request-response messaging* allows the surrogate to forward requests for images on to the device
• REST support allows the service to support existing clients, such as web browsers
• The *Surrogate* allows us to cache data on the fixed-network, to preserve mobile bandwidth and CPU power

Patient monitoring service | • *Streaming* messages allows the device to stream changes in physiological data to the surrogate - but only when a client subscribes to that service, so as to not waste bandwidth
• *Request-response messaging* allows the surrogate to forward location and vital sign data requests to the device
• *Publish-subscribe* support at the surrogate allows clients to remain notified of the latest data updates

Social networking service | • *Asynchronous* messaging allows each device to send location updates and text messages without requiring a response from each peer
• *Multicast* support allows each device to send data to multiple peers

Table 8.3: Summary of Odin2 sample apps and how Odin2 enables them.

a *Surrogate* which allows clients and other Odin2 services to send these messages via REST methods or Odin2’s interconnect, respectively.

To complete the service, we implemented a simple user interface (seen in figure 8.10) for each device, and added code to its event handlers to connect to and disconnect from the surrogate host. We also added code to the appropriate method skeletons to scan a device’s image folder and return either a single image or an image list. Finally, we implemented a simple caching scheme on the surrogate to store images which had already been requested from devices.
8.3.2 Patient Monitoring Service Definition

Figure 8.13 shows the patient monitoring service OdinTools definition. We have defined a VitalSigns type which represents a user’s vital signs. We have defined two asynchronous messages - one which is to be sent from the device to the surrogate whenever a user’s vital signs are updated; another which does the same for a user’s location. We also defined two request-response messages - one which allows clients to directly request a user’s vital signs; another for location.

From this definition, OdinTools creates an app skeleton for Android devices which allows them to receive and process these messages\(^1\). It also adds a location service to allow us to respond to location change events, as the GPSLocation type was used in the definition. Finally it creates a Surrogate which allows clients and other Odin2 services to send the request messages.

---

\(^1\) OdinTools also created the app skeleton for Windows Phone, but we deleted it as it is impossible to complete the service for Windows Phone 7.1
Completing this service was more complex compared with the mobile media service. Initially we added code to the Android device to allow it to communicate with a BioHarness over Bluetooth. We then implemented a simple user interface (identical to that in figure 8.10) and hooked up its event handlers. We then added code to each of the request message handler skeleton methods on the device to send a user’s current vital signs or location to a requester. We then added code on the device to send VitalSignsUpdated or LocationUpdated messages to the surrogate whenever one of these values changed. On the surrogate, we added two additional REST methods - one each to allow clients to subscribe to vital sign or location updates. Finally we added code to the generated handler methods for VitalSignsUpdated and LocationUpdated messages on the surrogate which would forward these messages to REST clients who had previously subscribed.

### 8.3.3 Social Networking Service Definition

Figure 8.14 shows the social networking service OdinTools definition. We have created a new type, *Person*, to represent a user and their friends. We have added a *friends* table to the database to allow each device to store a friends list. We have created two asynchronous messages which will be broadcast from a device to each friend - one for text messages, another for location updates.

![Diagram of OdinTools OAM definition for the social network service](image)

From this definition, OdinTools creates an app skeleton for Android and Windows Phone devices which allows them to send and receive each defined message type. Similarly it creates handlers for these messages in the surrogate. Additionally it creates a location service on each device to allow it to respond to location updates, as *GPSLocation* was used in the definition. Finally, on the surrogate, a REST method is created which allows clients to send each of the defined messages.
To complete the service we initially implemented a simple user interface (identical to those in figure 8.10) for each device platform and hooked up its event handlers. In the event handler for the “Send” button we added code to send the contents of the text box as a text message to the surrogate. We implement the same for location updates in the location service’s event handler. On the surrogate, we implemented code to respond to incoming location change and text messages by broadcasting them to all active devices. Finally, modified the REST interface for external clients by deleting the method allowing external clients to send location updates.

8.3.4 Generated Code Evaluation

As shown in sections 8.3.1 through 8.3.3, OdinTools has allowed us to partially generate each of our example services within Visual Studio. In each case other than the patient monitoring service, partial Surrogate, Android, and Windows Phone implementations were generated. In the case of the patient monitoring service, we did not generate the Windows Phone implementation as the service is impossible to implement using Windows Phone 7.1. This is because the service requires Bluetooth support, which this platform does not offer.

Following this partial generation, we were required to manually implement portions of the code to complete each service. We have investigated OdinTools’ potential to increase developer productivity by examining the proportion of generated code for each sample service. We have used Source Lines of Code (SLOC) [105] as the metric for this purpose.

Figure 8.15 shows a code breakdown for each sample application using OdinTools. In the chart, the total amount of code for a particular application was obtained by summing the total amount of surrogate code and the average of the device code for both Android and Windows Phone (other than for the patient monitoring service, in which only Android code was considered). We noticed that the proportion of generated code for the Android and Windows Phone platforms was very similar - hence, we do not feel we are losing data by averaging in this way.

Each bar in figure 8.15 is divided into four categories. User Interface (UI) refers to all code which defines an application’s user interface, in addition to event handlers for the UI (e.g. listener methods for button-click events) All such code must be manually implemented. Config Files refers to all configuration settings required to import the generated project into an appropriate IDE for further development, in addition to any API references. 100% of this code is automatically generated.
Chapter 8. Evaluation

**Figure 8.15:** Breakdown of generated and manual code using OdinTools

*Generated Code* refers to all Java or C# code (depending on the platform) generated by OdinTools. The code generated by OdinTools in this manner is determined by the *Odin Application Model* (OAM) defined by service developers (see Chapter 5 section 5.1.1 for details). Primarily it consists of the following (see Chapter 5 for further details):

- Main class - either *Activity*, *Surrogate*, or *PhoneApplicationPage*, depending on the platform
- Class definitions for Data types
- Class definitions for Message types
- Database definition and persistence annotations
- Method skeletons for handling each message type
- Location services, if used within the OAM

Finally, *App-Specific Logic* refers to all Java or C# code that must be manually implemented by users, other than UI event handling code.

From figure 8.15 we can see that in the mobile media and social network services, there is a high proportion of generated code. This is due to the fact that these services have simple user interfaces, and little application-specific logic - the manually implemented logic is described in sections 8.3.1 and 8.3.3.

For more complex services, the proportion of generated code will be much lower. As an example, consider the patient monitoring service. This service requires interaction with a Bluetooth device - the BioHarness - as mentioned in section 8.3.2. The BioHarness...
interface and Bluetooth logic are not generated by OdinTools, resulting in a significantly smaller proportion of generated code. In real-world cases, the proportion of user interface code - which must be manually implemented - will also tend to be higher.

8.3.5 Discussion

OdinTools allows developers to partially generate cross-platform mobile services. We have evaluated the extent to which OdinTools generates a service implementation in section 8.3.4. The main benefit developers gain from the use of OdinTools is the ability to generate the structure of, and configuration files for, a mobile service, for each target platform. Using OdinTools, developers can be assured that the mobile services created by building upon the generated structures are interoperable.

We have attempted to show that OdinTools increases developer productivity by measuring the proportion of generated code versus that which must be manually implemented. Ideally, we would need to carry out a comprehensive user evaluation, in which we ask developers to write services both with and without OdinTools, and measure the time taken in addition to soliciting feedback. Time constraints have meant that this work could not be included in this thesis, but is an important item of future work. In addition, we have noted that the amount of UI code required for a non-trivial app can be significant and must be written manually. We plan to investigate the integration of PhoneGap or a similar cross-platform mobile development tool with OdinTools to provide a comprehensive cross-platform mobile service development platform.

8.4 Test Harness Evaluation

As discussed in Chapter 6, the Odin Test Harness allows testers to:

- Simulate multiple, separately configurable Android devices (15 for a single test harness machine)
- Install and run arbitrary Android apps on each device
- Configure the database and preferences of each installed app
- Launch and simulate user input on each app
- Simulate Wi-Fi and 3G networks for each device
- Enable / disable each simulated network for each device and configure their bandwidth, latency, and loss rate
During the development of Odin2 and REMOTE-CR, we used the test harness to assist with development and testing. The test harness’s ability to create complex test scripts consisting of any combination of the above tasks allowed us to use it as our primary testing tool, in combination with other testing techniques such as unit testing and user acceptance testing. Sections 8.4.1 and 8.4.2 below detail the kinds of tests we ran on Odin2 and REMOTE-CR, before we discuss the test harness’s benefits and shortcomings in section 8.4.3.

### 8.4.1 Odin2 Tests

For all tests in this section, an Odin2 test app was created with a simple UI. The app allows a tester to enter their Odin2 username and password, log in or out of the surrogate host, and stream data to or receive data from the app’s associated surrogate at a specified transfer rate, by using the various on-screen buttons. The app’s surrogate was installed on a surrogate host accessible by the test harness machine. Before each test run, the Android app was installed on each of the test harness’s virtual devices. After each run, the surrogate host and app’s logs were retrieved and saved to a directory on the test harness for examination, prior to uninstalling the app from each virtual device. This process was completed automatically, through the test harness’s bash extensions. During each test, each app would log into Odin2 using a username corresponding to the virtual device on which the app was running - e.g. “Android-01” through “Android-15”. The passwords were equal to the usernames. These users were added to the surrogate hosts’s database using the surrogate host web app (see Chapter 4 section 4.3.3) before testing commenced. For all tests where streaming was used, a 32 kbps transfer rate was selected unless otherwise stated. The results of all of these tests were determined by:

1. Watching the test harness in operation, in the case of very short tests, and
2. Examining device and surrogate host logs, for all tests.

#### Test #1: Network Disconnection

Initially, a single virtual device was set up with a single active network connection, and the test app was set up to stream data to the surrogate. The virtual device’s single network connection was configured to disconnect for 30 seconds every five minutes. The app was left running for 24 hours. The experiment was then repeated with data transfer occurring from the surrogate to the device, then with two-way transfer.

**Expected Result**

During each variation of the test, the device should change its connectivity status to
Chapter 8. Evaluation

Connecting when disconnections occur, and should return to Connected status once connectivity is restored. No data should be lost, duplicated, or arrive out-of-order.

**Bugs and Fixes**

None - this test worked the first time we ran it.

**Test #2: Dropped Packets**

The simulated app was set up to behave in the same manner as in Test #1. Rather than completely disconnect all network connectivity with virtual devices, the devices’ networks were set to drop 10% of all packets transmitted to the surrogate.

**Expected Result**

The device should not be disconnected. No data should be lost, duplicated, or arrive out-of-order. Odin2 should handle the re-sending of any data which cannot be resent by the interconnect’s underlying TCP protocol.

**Bugs and Fixes**

None

**Test #3: 100% Dropped Packets**

A virtual device was set up to stream data to the surrogate. After streaming had been operational for 5 minutes, the device’s network connection was set to drop 100% of packets. 30 seconds later, it was set to resume sending all packets.

**Expected Result**

The app should detect the network disruption and change its state to Connecting when the disruption occurs. When connectivity is restored, the app’s state should change to Connected. No data should be lost, duplicated, or arrive out-of-order.

**Bugs and Fixes**

During the initial run of this test, the virtual device would remain “Connected” for the duration of the experiment, though as expected, no data was received by the surrogate once the device’s network was configured to drop packets. Furthermore, the device never sent any data even when connectivity was restored. A bug was identified where the TCP socket implementation for Android would not detect this case as a failed connection, and would block indefinitely, never throwing an exception to alert Odin2 of the outage. We fixed this issue by modifying Odin2’s heartbeat mechanism (see Chapter 4 section 4.6). Initially we specified that the heartbeat should only occur over 3G, to account for stale connections. We added a heartbeat to Wi-Fi connections - albeit at a much larger interval - to address the results of this test.
Test #4: Vertical Handover

A virtual device was set up to stream data to the surrogate for 24 hours. The device was given two network connections: a high-bandwidth, low-latency connection registered as the device’s “Wi-Fi” link, and a low-bandwidth, moderate-latency connection registered as the device’s “3G” link. Initially both connections were active. Every five minutes, the Wi-Fi link was disabled for five minutes (i.e. the Wi-Fi link was active half the time).

Expected Result

The virtual device should always be able to communicate with the surrogate. No data should be lost, duplicated, or arrive out-of-order. While the Wi-Fi link is active, communication should occur over that channel. Otherwise, communication should occur over the 3G link.

Bugs and Fixes

None.

8.4.2 REMOTE-CR Tests

When testing RMEOTE-CR we made use of the ability to make apps test-harness aware. This is because the test harness’s virtual devices do not have access to GPS or Bluetooth functionality. We used the test-harness-aware feature to substitute mock GPS and BioHarness services which are capable of sending either randomized or specific location and physiological data. The output of the mock services can be controlled by the test harness.

Similarly to the Odin2 tests above, each REMOTE-CR test script would install and configure the REMOTE-CR app on each virtual device prior to test commencement, then capture the logs and uninstall the app after completion. We used the same surrogate host machine as in the above tests. For tests which seek to evaluate the appearance of REMOTE-CR data within the web app, we manually evaluated this in collaboration with NIHI.

Test #5: Web App Layout

Six virtual devices were configured to run REMOTE-CR under optimal network conditions. This is the maximum number of participants expected to fit on a single monitor for live monitoring on the web app. These were each set to begin a workout. Each app’s mock BioHarness service was set to output a pattern, which would easily be visible to a web app user. A separate test script was created which would stop
the workout on each device. This would be executed manually, once the web app’s user interface was sufficiently examined by testers.

**Expected Result**

Six panels should be visible on the web app - one for each virtual device. Each panel should display a device’s BioHarness information. The heart rate and breathing rate should be easily identifiable on each panel as being the pattern set by the test harness. On a browser running with a maximized window, at 1680x1050 resolution, all six panels should be visible on the web app without scrolling. When each workout session is ended, the web app should display this information within five seconds of a session ending.

**Bugs and Fixes**

Issues with the web app’s layout were identified and fixed, to allow all six panels to fit on a single screen.

**Test #6: Web App Performance**

The test setup was the same as for test #5, but with 15 virtual devices rather than six.

**Expected Result**

All 15 panels should be visible on the web app (though panels 7 through 15 would require scrolling in order to view). When viewed in the Google Chrome browser, on a 2012 Intel Core i5 laptop with integrated graphics, no lag should be perceivable by web app users.

**Bugs and Fixes**

Initially, displaying 15 concurrent workout sessions was far too resource intensive for the above computing setup. Specifically, the browser could not render the required number of Knockout Charts data points to display all 15 graphs. To fix this we modified the web app to only display the most recent one minute’s worth of data for each participant.

**Test #7: Web App Symptom Reporting**

The test setup was the same as for test #6, except that each device would also send symptom reports at random intervals while performing a workout session. REMOTE-CR logs each symptom report on the device along with its timestamp, which can be used for comparison.

**Expected Result**

All symptom reports should arrive at the web app within five seconds of being sent.
Chapter 8. Evaluation

Bugs and Fixes

None.

Test #8: Web App Map View

The test setup was the same as for test #6, except that each device’s mock GPS service was configured to report changes in location at regular intervals. The GPS coordinates reported by the service would follow a pre-defined path around the Auckland CBD, to allow a web app user to easily determine if the correct information was being displayed.

Expected Result

Each device’s reported location should be visible on the map shown on the web app. The map’s information should be updated within five seconds of a device sending new data.

Bugs and Fixes

None.

Test #9: REMOTE-CR Reliability

Tests #1 through #4 from section 8.4.1 above were repeated with the REMOTE-CR app. Furthermore, each test was conducted first with a single virtual device, then with 15. Rather than streaming data from the test app, workout sessions were initiated for each virtual device. Each workout session would stream mock GPS and BioHarness data, as well as generate random symptom reports.

Expected Result

The same results as expected for tests #1 through #4 are expected here. Primarily, no data should be lost, duplicated, or arrive out-of-order.

Bugs and Fixes

During these experiments, a race condition was identified in Odin2’s interconnect layer. This would, rarely, cause the interconnect to be stuck in a Connecting state when connectivity was disrupted. The app would then never reconnect to the surrogate host, even once connectivity was restored or a new network became available. The logs generated by REMOTE-CR’s mobile application allowed us to track down and resolve this issue.

Test #10: REMOTE-CR Scalability

Two test harness machines were set up, each running a 15-device configuration similar to that in test #6.
Expected Result

The surrogate host, and its associated internet connection, should not be placed under heavy load by 30 simultaneous workout sessions. 30 simultaneous sessions was chosen as the maximum number of participants which would conceivably be online at any given time in a real-world scenario for the REMOTE-CR trial.

Bugs and Fixes

None.

Test #11: REMOTE-CR Mobile App User Input Response

A single virtual device was set up. Using the test harness’s ability to simulate user input, we navigated to each area of REMOTE-CR’s user interface and tried many combinations of user input - including those combinations which are deemed “incorrect”.

Expected Result

When “correct” user input is provided, the app should operate as per its functional requirements in Chapter 7. “Incorrect” input should either be impossible (e.g. due to disabled buttons on the interface) or, when it is entered, should provide feedback to the user, rather than causing the app to crash or enter an undefined state.

Bugs and Fixes

Several unexpected combinations of user input were identified which would cause the app to crash. Each of these combinations was changed to either be impossible to recreate, or, provide feedback to the user.

8.4.3 Discussion

The test harness has proved useful in a number of ways. As we can see from test #3, its ability to repeatably generate a desired network environment for its test devices allowed us to detect and resolve a bug which occurs only under a rarely-occurring network state (i.e. when a Wi-Fi connection appears “live” but is actually broken). Furthermore, the ability to repeatably change network conditions during test operation allowed us to detect the bug in test #9. As this bug only rarely occurred, it required a large number of network change operations to occur in order to reliably track down. In both these cases, it is possible that, without the use of the test harness, these bugs would have remained undetected before being moved to a production environment, in which they could have disrupted clinicians’ ability to reliably monitor participant vital signs.

The test harness’s ability to simulate multiple devices has also proven useful. In the case of tests #5 through #8, this allowed us to verify that the REMOTE-CR web app was
capable of appropriately displaying data from multiple participants, without requiring that multiple smartphones and BioHarnesses be used and co-ordinated. As seen from tests #5 and #6, substantial changes to the web app’s layout and backend were made as a result of these tests. More generally, and as shown in test #10, the simulation of multiple devices allowed us to confirm that the surrogate host was able to handle the requisite amount of mobile traffic without requiring that number of devices.

Finally, the test harness’s ability to simulate user input has proven immensely useful. Without this ability, none of these tests would have been possible, as each test has involved the simulation of at least one button press and / or entry into a text field.

Along with its benefits, the test harness has some shortcomings. Primarily, some application areas are impossible to test - particularly anything that relies on Bluetooth or GPS data. While we can use the test-harness-aware feature to allow apps to mock this kind of data, this is not a sufficient substitute in all cases, as demonstrated by the test harness’s failure to detect the GPS-related error detailed in section 8.5 below. Furthermore the test harness’s ability to control an application’s user interface is limited compared with existing tools such as the Android UIAutomation library\(^2\). In the future we will investigate how this or another library can be integrated with the test harness, or work to increase the expressiveness of the test harness’s UI automation features.

Overall, the test harness provides a useful set of features - particularly \(i\) the ability to simulate and control multiple devices, and \(ii\) the ability to control those devices’ networks. In combination with other testing techniques, the test harness has allowed us much more confidence in the reliability of Odin2 and REMOTE-CR.

### 8.5 Real-World Usage

The use of Odin2 in the REMOTE-CR trial can be used to evaluate Odin2’s suitability as a mobile service provisioning middleware in a real-world scenario. In this section we detail how REMOTE-CR is used in production, show any bugs reported by users, and provide feedback regarding the trial.

#### 8.5.1 REMOTE-CR Trial Setup

In total, 162 participants will be recruited for the REMOTE-CR trial. Participants must meet the eligibility criteria in [106]. Amongst participants, half will be selected to undergo the REMOTE-CR intervention, while half will be selected into a control group

and receive a more traditional method of care. At the time of writing, 32 participants have currently been enrolled and have been performing workout sessions.

Each participant randomized into the REMOTE-CR programme receives a Zephyr BioHarness 3 monitoring device and a 2013 Motorola Moto G smartphone with a 2degrees 3G data plan and the REMOTE-CR software pre-installed. Before beginning the trial, participants’ initial fitness levels are assessed (see [106] for details) and they receive instruction regarding the use of the software. Each participant is given an individualized exercise programme, with frequency and intensity tailored to that participant’s particular fitness level and symptom severity.

During the trial, clinicians are available to monitor a participant’s sessions in real-time from 6am till 11am on Mondays, Wednesdays, and Fridays. Participants additionally are allowed to exercise at any other time - and these sessions will be recorded on REMOTE-CR’s surrogate host - though the presence of a clinician will not be guaranteed.

During any session undertaken by a participant in which a clinician is available, the clinician will monitor that participant’s vital signs and symptom reports and take any appropriate medical actions, which may include instructing the participant to increase or decrease their level of activity, and / or summoning medical aid to that participant’s location. Additionally, clinicians will encourage participants using personalized motivational messages.

Each participant is enrolled in the programme for a period of 12 weeks. During this time, participants are invited to submit feedback about their experience with the app, in the form of unstructured comments. At its conclusion, participants complete a survey to elicit further feedback. Furthermore, participants at any time may submit bug reports through the Google Play Store, or via email. Such bug reports include any exceptions generated, along with the log files for that participant’s device.

8.5.2 Bug Reports

Participants and clinicians may, at any time during the trial, submit bug reports via email or through the Google Play Store. Here, we detail all bug reports received thus far (the trial has been underway for eight months at the time of writing this thesis) and their associated fixes:

Bug #1: Web App Data
The web app would report data for incorrect time periods when using weekly and monthly views.
Fix

An error in the web app’s SQL query code was identified and fixed.

**Bug #2: Smartphone App UI**

Smartphone app crashes were identified in response to unexpected user input in the Routes and Workout screens.

Fix

Sufficient sanity checking was applied in each case.

**Bug #3: Google Static Maps API**

The smartphone app would sometimes crash when ending a workout session. When this occurred, the surrogate host would not be notified of the session’s end.

Fix

The crash was identified as an issue with the Google Static Maps API for generating route thumbnails. As this routine was called before the *end session* message was sent, this explains the host never being notified. Once the API usage was corrected, this issue disappeared.

**Bug #4: Android Application Lifecycle**

When a workout session was ended while mobile connectivity was broken, then the smartphone app closed, the surrogate host would never receive any data sent after the connectivity outage occurred (it should receive all such data once connectivity is restored).

Fix

Caused by a misunderstanding of when the Android OS is allowed to kill REMOTE-CR’s background service. A properly configured background service resolved this issue.

**8.5.3 REMOTE-CR Feedback**

As the REMOTE-CR trial has not yet completed, its results are not yet available. However, from the limited number of submitted bug reports, as shown in section 8.5.2 above, we can assume that both clinicians and participants have encountered few issues with the software thus far. This shows that Odin2 is capable of allowing the development of robust mobile services.
We solicited feedback on the trial’s current progress and the reliability of the software from NIHI. Jonathan Rawstorn, one of the trial’s primary investigators, was happy to provide this feedback:

“The Odin platform enabled rapid prototyping, development and refinement of a sophisticated mobile application designed to support remote delivery of an exercise-based cardiac rehabilitation programme. In conjunction with a complementary web application the Odin-based mobile application delivers a range of features required as part of cardiac rehabilitation including real-time remote physiological monitoring and communication during exercise, performance assessment and feedback, and goal setting and achievement feedback.

The Odin-based application provides exercise physiologists with essential data required to cardiac patients’ exercise prescription to optimize beneficial outcomes and minimize the risk of cardiac complications during exercise. Remaining features enable the delivery of a suite of evidence-based strategies for promoting the adoption and long-term maintenance of positive exercise behaviors that are proven to reduce the risk of recurrent cardiac events such as myocardial infarction.

The remotely delivered cardiac rehabilitation programme is now being evaluated in a randomized clinical trial. Results from the trial are not yet available; however, the mobile and web applications have thus far performed very well, and early feedback from participants has been very favorable. The Odin platform has played an integral role in the development and deployment of this programme. In conjunction with the feature rich, intuitive mobile application the platform is a foundation of a study which has the potential to substantially enhance the ways in which cardiac patients are treated in New Zealand, and around the world.”

8.5.4 Discussion

By examining the reported bugs in section 8.5.2, we can see that none of these bugs is an issue with the Odin2 middleware itself, but rather, other REMOTE-CR features. From this, along with Rawstorn’s feedback in section 8.5.3, we can be confident that Odin2 is dependable and performant enough to act as the mobile service provisioning middleware for a complex, real-world application.

In addition, by examining the bugs in section 8.5.2, we can further evaluate the test harness. Bug #2 shows a weakness in the test harness’s ability to automate UI testing. When compared with test #11 in section 8.4.2, we can see that several user interface errors were not detected by the test harness. This is our primary motivation to further
investigate the integration of a user interface testing tool into the test harness. Bug #3 is an example of a bug which is indirectly dependent on a feature which is unsupported by the test harness - in this case, GPS. As each test concerning location data in section 8.4.2 used a pre-defined set of mock GPS co-ordinates, this bug - which occurs only with certain GPS data - was not detected. Such randomly-occurring bugs dependent on features unsupported by the test harness necessitate that a testing regime with real devices is still followed - the test harness cannot fully replace user acceptance testing with real devices. Finally, bug #4 could have been detected by the test harness, had such a test been performed. It was undetected due to our lack of consideration that such a bug could even occur. This was a shortcoming with the testers, rather than the test harness itself.

8.6 Summary

In this chapter we have evaluated Odin2’s messaging performance with respect to leading industry push notification systems. As shown in section 8.1, Odin2’s responsiveness, reliability, bandwidth consumption, and battery consumption are comparable or superior to each of the evaluated industry systems. This shows that Odin2 is performant enough to be used for real-world software development.

We have also shown, in section 8.2, that Odin2 is expressive enough to create a wide range of mobile services by implementing a patient monitoring service, a mobile media service, and a social networking service using Odin2. Each of these has a distinct set of functional and non-functional requirements and leverages Odin2 to meet these requirements. Furthermore, each of the sample services is implemented on both the Android and Windows Phone platforms, with the exception of the patient monitoring service. This shows that developers are capable of creating cross-platform services using Odin2 as the underlying middleware.

In section 8.3 we have shown that OdinTools allows developers to automatically generate a cross-platform mobile service’s configuration files and application structure for each target platform. This allows developers to be assured that the mobile services created by building upon the generated application structures are interoperable. Furthermore, OdinTools has the potential to increase developer productivity through code generation, though further study is required to explore this more fully.

In section 8.4 we have shown how we have used the Odin Test Harness during the development of Odin2 and REMOTE-CR. We have shown that the test harness is capable of detecting bugs which would otherwise be difficult to detect - such as bugs
triggered by rare network conditions - by allowing us to simulate such network conditions across multiple concurrent devices in a controlled, repeatable manner. In the future we will investigate the integration of Android’s UI Automation Framework or a similar tool to improve the test harness’s ability to test an application’s user interface.

Finally, we have adopted Odin2 as the underlying platform for the REMOTE-CR cardiac rehabilitation software. In section 8.5 we have detailed the trial’s experimental setup, and have shown that the few bugs reported by trial participants have been caused by application components other than Odin2. Furthermore we have provided feedback from one of REMOTE-CR’s principal investigators which details how the trial has thus-far been successful and has received positive participant feedback. This shows that Odin2 is capable as a real-world mobile service provisioning middleware.
Chapter 9

Conclusions and Future Work

In this thesis, we have created a mobile service provisioning middleware, Odin2. We have shown how Odin2 overcomes challenges with reachability, scalability and availability which are inherent in mobile service provisioning, and we have demonstrated that it is performant, expressive, and dependable enough to be used in real-world software engineering scenarios. Furthermore, we have created OdinTools, a domain-specific language for mobile service development. OdinTools leverages Model-Driven Engineering (MDE) techniques to allow service developers to partially generate cross-platform mobile services. We have shown that OdinTools reduces the amount of code required of developers, and we believe this will aid in the rapid application development of mobile services. Finally we have created the Odin Test Harness to assist with the testing of Odin2 and associated mobile applications. OdinTools allows developers to create test scripts enabling fine-grained, repeatable control over many concurrent devices and their associated network conditions. We have shown how the test harness has been used to place Odin2 under test, and we have shown it to be capable of detecting otherwise elusive bugs. This has greatly increased our confidence in Odin2’s dependability guarantees, and has allowed Odin2 to be adopted by the National Institute of Health Innovation (NIHI) as the underlying middleware for their REMOTE-CR clinical trial.

The remainder of this chapter is organized as follows. In section 9.1 we review the research contributions of this thesis, before addressing the research questions in section 9.2. Finally, we briefly outline possible areas for future work in section 9.3.

9.1 Research Contributions

During the course of this thesis, we have:
• Completed a literature review into the extent of current mobile service provisioning, cross-platform development, and relevant mobile application testing techniques.

• In collaboration with Thiranjith Weerasinghe, designed, developed and evaluated Odin, our mobile service provisioning middleware. Subsequently, Odin has been redesigned based on the initial evaluation and further dependability, simplicity, and performance requirements to produce Odin2.

• Designed, developed, and evaluated OdinTools, a domain-specific language which uses model-driven engineering techniques to allow service developers to partially generate cross-platform mobile services.

• Designed and developed the Odin Test Harness, a tool to allow the controlled, repeatable simulation of multiple Android devices and changes in network state.

• Demonstrated the relevance of Odin2 and the test harness in industry through REMOTE-CR, a cardiac patient monitoring and exercise service.

In this section we summarize each of these research contributions.

9.1.1 Literature review of mobile service provisioning, cross-platform development, and mobile application testing techniques

Initially we conducted a literature review into the challenges inherent in mobile service provisioning. We identified reachability, availability, and scalability as the key challenges which must be addressed by a mobile service provisioning middleware. We then investigated the proposed solutions to these challenges in the research. The initial literature review can be found in Chapter 2 sections 2.1 through 2.1.3.

From the literature we identified an intermediary-based architecture as being the most appropriate architecture for our middleware. Firstly, an intermediary with a fixed address is easier for clients to locate than a mobile device. The use of an intermediary for this purpose has recently become well-known in industry, in the form of push gateways for push notification services. Furthermore, as a mobile device roams, an intermediary can shield clients from changes in its point of network attachment, along with partial or total disruptions in the mobile network. Finally, an intermediary can be used to offer a framework for the offloading of code, which the literature shows to provide performance and scalability benefits in some cases.

Later in the thesis, we additionally investigated testability and heterogeneity in the literature. These reviews are available in Chapter 2 sections 2.1.4 and 2.1.5, respectively.
To address heterogeneity, we investigated Model-Driven Engineering (MDE) along with Hybrid Mobile Development solutions such as PhoneGap. We found that hybrid approaches tend to offer greater flexibility in the types of applications which can be created. In comparison, MDE approaches are less generally applicable but have been shown to increase productivity for application development in certain domains, and offer increased performance due to native code generation. For our cross-platform development tool, OdinTools, we chose an MDE approach as we considered the mobile services domain to have the potential to be sufficiently well-defined, and valued the performance and scalability benefits to be gained from native code generation.

In the field of mobile application testing, the majority of the literature focuses on UI automation testing. This refers to running tests which automatically navigate a tested application’s user interface, possibly under different simulated device conditions such as CPU speed and network bandwidth. While there are solutions that offer the ability to simulate large numbers of devices and network conditions, there are none that provide this ability in conjunction with the ability to offer fine-grained repeatable tests, especially for scenarios which do not depend on user input. Such scenarios are rare for mobile applications in general, but far more common for mobile services. This review informed the goal of the Odin Test Harness - to offer the testing ability of the state-of-the-art solutions, but tailored to the requirements of mobile services.

### 9.1.2 Odin2 Mobile Service Provisioning Middleware

Initially, in collaboration with Thiranjith Weerasinghe, we designed and developed our mobile service provisioning middleware, Odin, detailed in Chapter 3. Odin is based on the Jini Surrogate Architecture, which introduces the concept of Surrogate Hosts. A surrogate host is an intermediary machine onto which mobile service host-devices upload Surrogates. Surrogates are software components which act on behalf of the host-devices in a Jini service federation. This allows Jini clients to locate and consume mobile services using Jini’s existing discovery mechanisms. Communication between the host-device and surrogate occurs over a custom Interconnect.

The main contribution to Odin in this thesis is its interconnect protocol. Odin supports multiple network types - both IP based (such as Wi-Fi or 3G) or otherwise (e.g. Bluetooth). Furthermore its interconnect is extensible, allowing developers to add support for new network types. Most importantly, Odin supports vertical handover between any supported network, and ensures that data transmitted during vertical handover, or other network disruption, is not lost, duplicated, and does not arrive out of order.
Odin was evaluated under laboratory conditions, in addition to its use by final year project students and an initial pilot study with NIHI. While the laboratory experiments were promising, the real-world tests showed several shortcomings. Most notably, Odin exhibited poor performance over 3G, and was less reliable and easy to use than anticipated. This informed the need for a more robust testing methodology for the middleware and associated applications, along with stricter requirements regarding performance and ease-of-use. These new requirements, along with the original goals to shield developers from mobile service provisioning challenges while being expressive enough to create a range of different service types, were used to inform the design and development of our updated middleware, Odin2. Odin2 is detailed in Chapter 4.

Odin2 still follows the general surrogate architecture, but forgoes Jini in exchange for more established technologies to increase usability. Specifically, the surrogate host is implemented using the Spring framework, with Hibernate for persistence. Odin2 surrogates expose mobile services using a REST interface. Not only does this make mobile services easier to create by allowing developers to leverage these technologies’ extensive documentation, but it also allows for much greater interoperability with clients that lack access to the Jini libraries.

In addition to these changes, Odin2’s interconnect was redesigned with a focus on performance, reliability, and ease-of-use. Odin2’s interconnect now provides developers with the abstraction of a messaging protocol. Developers may send arbitrary asynchronous, request-response, or streaming messages between devices and surrogates, which are serialized as JSON. Developers may also request the delivery status for any message. Internally, the interconnect operates on a TCP-based protocol, with extensions to account for inconsistencies in the sockets implementation on some platforms (see Chapter 4 section 4.6 for further details). This offers greater performance - particularly with two-way communication - compared with the HTTP-based interconnect utilized by our original prototype. Furthermore, Odin2’s interconnect protocol can be used within Odin2, but it is a novel contribution in and of itself which can be applied to other areas of computing. For example, in a patient monitoring scenario, the protocol can be adopted by the vital sign monitoring sensors themselves to ensure important medical data transmitted over the Bluetooth link is successfully received by the smartphone.

An evaluation of Odin2’s performance and reliability has been performed, in comparison with leading push notification service providers. The evaluation, detailed in Chapter 8, shows that Odin2’s messaging protocol is comparable to, or better than, these solutions in terms of data throughput, protocol overhead, latency, and reliability. Furthermore, we have demonstrated Odin2’s expressiveness by showing it to be capable of enabling a variety of mobile service types, including our complete set of example apps introduced in
Chapter 1. Moreover, Odin2 has been adopted as the middleware for REMOTE-CR, a remote cardiac patient monitoring and exercise service. This shows Odin2 is sufficiently expressive and dependable for use in a real-world software project.

9.1.3 Model-Driven Service Development with OdinTools

We have created a domain-specific language, OdinTools, to assist with cross-platform development of Odin2 mobile services. OdinTools, detailed in Chapter 5, allows developers to specify a service’s data types, messages, certain persistence semantics, and message and data locality. From this specification, OdinTools generates partial implementations for the surrogate, Android devices, and Windows Phone devices. OdinTools generates the following for each platform:

- Configuration files to launch projects in Eclipse or Visual Studio
- Main class - either Activity, Surrogate, or PhoneApplicationPage, depending on the platform
- Class definitions for Data types
- Class definitions for Message types
- Database definition and persistence annotations
- Method skeletons for handling each message type
- Location services, if the use of GPS is specified

A preliminary evaluation of OdinTools has been carried out in Chapter 8 which shows that OdinTools reduces the amount of code required of developers. Furthermore, developers can be assured of interoperability between Android and Windows Phone services when using OdinTools. Such code reduction is likely to increase productivity of service developers, though a thorough user evaluation of OdinTools is required to explore this more fully. In addition to its merit as a mobile service development tool, OdinTools also serves as a validation of current MDE techniques, showing that they are more widely applicable to the mobile computing domain than had previously been demonstrated in the literature.
9.1.4 Odin Test Harness

Following the initial evaluation of Odin, we identified the need for a more robust testing framework for Odin2 and its associated apps. The Odin Test Harness, described in Chapter 6, meets this requirement. The test harness allows testers to:

- Simulate multiple concurrent devices
- Simulate multiple network interfaces per device
- Simulate and precisely control the bandwidth, latency and drop rate for each network interface for each device
- Automatically install tested apps to each device, and configure their preferences and database
- Simulate user input on each device
- Create test scripts to control all of the above features in a repeatable manner

Compared with alternative frameworks identified in the literature, the test harness offers better repeatability for specific test cases, and more support for test cases independent of a device’s user interface. This allows it to serve as a testing framework for not only Odin2-based mobile services, but for any networked application where reliability in a dynamic network and operating environment is of importance.

The test harness was used to test Odin2, as detailed in Chapter 8 section 8.4. The testing process allowed us to detect otherwise elusive bugs, such as race conditions which only occur under specific network conditions, or inconsistencies with the Android socket API. The test harness was also used during the development of REMOTE-CR, and again managed to catch bugs that may have otherwise gone undetected until after entering production. The use of the test harness during REMOTE-CR development did identify the need for greater expressiveness when simulating user input on devices. This will be achieved in the future through the use of the Android UI automation library or a similar framework.

9.1.5 REMOTE-CR Case Study

One of Odin2’s primary goals is to be performant, expressive, and dependable enough to be used for complex real-world application development scenarios. Our experience with the REMOTE-CR trial, detailed in Chapter 7, has shown that we have met this goal.
REMOTE-CR, or the Remote Exercise Monitoring Trial for Exercise-based Cardiac Rehabilitation, is a trial conceived by the National Institute for Health Innovation (NIHI) to test the effectiveness of a technology-assisted cardiac rehabilitation programme for patients with heart conditions. The REMOTE-CR software consists of a mobile service which allows clinicians, via a web application, to access participants’ vital signs and location information in real-time. Furthermore, participants may report symptoms they encounter to clinicians at any time. Finally, clinicians may send motivational or instructional messages to participants, and request up-to-date electrocardiograph (ECG) readings from sensors attached to the participant.

REMOTE-CR achieves its functional requirements using Odin2. Primarily, this functionality is achieved using Odin2’s interconnect, which provides the asynchronous, request-response, and streaming message support used for REMOTE-CR’s motivational messaging, ECG requests, and vital sign and location data, respectively. Furthermore, Odin2’s message delivery notification support allows clinicians to be sure that important messages are reaching participants. Moreover, Odin2’s REST support provides interoperability with the web application. In addition to supporting REMOTE-CR’s functional requirements, Odin2’s reliability guarantees allow REMOTE-CR to meet its own non-functional requirement for high reliability Furthermore, Odin2’s relatively bandwidth-efficient protocol allows NIHI to save on data costs.

Prior to being released to participants, REMOTE-CR was tested with the test harness. The test harness’s ability to easily simulate multiple devices allowed us to optimize the web app’s layout to better support the intended number of concurrent participants, in addition to identifying bugs in response to unexpected user input and a previously uncaught bug in Odin2’s interconnect. Following deployment, all participants and clinicians were given the ability to send bug reports via the Google Play Store or via email. A combination of excellent feedback from NIHI and the relative lack of bug reports - particularly in relation to app features enabled by Odin2 - has given us confidence in Odin2’s dependability in a real-world environment.

9.2 Answers to Research Questions

In this section we consider how our contributions have helped answer our research questions.
9.2.1 RQ #1: What are the challenges inherent in mobile service provisioning? To what degree does the existing body of work address these challenges?

Our initial literature review in Chapter 2 identified the following challenges inherent in mobile service provisioning:

- **Reachability** - How can clients discover and consume mobile services despite ever-changing points of network attachment and mobile network operator firewalls?

- **Availability** - How can clients remain connected to services as they roam, and in the face of network disruption? How can we increase the duration of service availability related to a host device’s battery life?

- **Scalability** - How can mobile devices, with limited CPU, memory, and network bandwidth, host services which meet the demands of multiple clients?

The literature review showed that while there are solutions which seek to address one or more of reachability, availability, and/or scalability, none address all of these challenges in a sufficiently reliable manner (or the evidence that they do is not convincing).

9.2.2 RQ #2: What technologies and techniques are used commercially for mobile service development, if any?

There is little in the literature regarding the development of mobile services in industry. However, in recent years push notifications have been adopted as a way to allow machines to send notifications to mobile devices. Push notifications, detailed in Chapter 2 section 2.3.3, address the reachability problem identified in the literature, and this technology has been developed as one part of our mobile service provisioning middleware.

9.2.3 RQ #3: What are the existing approaches to cross-platform mobile development, and cross-platform development in general?

In the literature, we identified two main cross-platform development approaches: Model-Driven Engineering (MDE), along with Hybrid Mobile Development solutions such as PhoneGap. These approaches are detailed in Chapter 2 section 2.7. We found that hybrid approaches tend to offer greater flexibility in the types of applications which can be created. In comparison, MDE approaches are less generally applicable but have been shown to increase productivity for application development in certain domains, and offer increased performance due to native code generation.
9.2.4 RQ #4: How can we support more rapid application development of mobile services?

Our model-driven toolkit, OdinTools, allows developers to specify mobile services using a drag-and-drop interface. From this specification, OdinTools partially generates cross-platform mobile service implementations. While developers must still write code to complete a service, a preliminary evaluation in Chapter 8 section 8.3 has shown that OdinTools both reduces the amount of code which must be manually written, in addition to providing developers with assurance that Windows Phone and Android versions of their services will be interoperable. We believe that for these reasons, OdinTools enables more rapid mobile service development, though a full-scale usability evaluation of the tool is required in order to further explore this.

9.2.5 RQ #5: How can we sufficiently test our mobile service provisioning solution and associated apps to ensure they meet their requirements?

From our early work with Odin we identified the need to repeatably expose mobile services to specific, precisely-controllable network connections to test their reliability. A literature review, shown in Chapter 2 section 2.1.4, identified systems which allow multiple simulated devices to be exposed to changes in network state, but these solutions focus on user interface automation testing. We created The Odin Test Harness, presented in Chapter 6, to support more fined-grained network control via a scripting language, in addition to providing support for test cases which do not rely on user input.

The test harness has been used during the development of Odin2 and REMOTE-CR, where it has identified otherwise elusive bugs such as race conditions which depend on a specific network environment. This has prevented such bugs from being prevalent in REMOTE-CR’s production environment, as is evident from NIHI’s feedback coupled with a relatively small number of bug reports from users.

9.2.6 RQ #6: Is our solution dependable enough to be used in a real-world scenario?

Odin2 has been used as the middleware for NIHI’s REMOTE-CR cardiac rehabilitation trial. Following the initial REMOTE-CR development, bugs were present in both Odin2 and REMOTE-CR. However, the test harness allowed us to detect and remedy many of these bugs before the software was deployed to production. During the REMOTE-CR trial, participants and clinicians are given the ability to submit bug reports, either
through the Google Play Store or directly via email. Excellent feedback from NIHI, coupled with the relative lack of such reports - especially in relation to features enabled by Odin2 - gives us confidence that the current version of Odin2 is sufficiently dependable for real-world software development.

9.3 Future Work

Several potential research directions have arisen from the work in this thesis, concerned with Odin2, OdinTools, and the Test Harness. Furthermore we are continuing to collaborate with NIHI for the REMOTE-CR trial and other possible industrial opportunities.

9.3.1 With Odin2

We believe that Odin2 largely addresses the reachability and availability challenges identified in the literature, and is sufficiently robust due to the test harness. However, we believe we can further improve Odin2’s scalability. Early work on Odin by Weerasinghe [3] investigated surrogate migration as a potential method by which a surrogate host’s load can be managed. Results in Weerasinghe’s thesis were promising, but this work was not included in Odin2 as it heavily relies on Jini, which is no longer supported by Odin2 due to interoperability and usability concerns. We plan to investigate how Odin2’s architecture can support surrogate migration in a similar manner. Many surrogate hosts could be deployed within the cloud, and surrogate migration could be used to appropriate load-balance host devices.

9.3.2 With OdinTools

We wish to enable a more comprehensive cross-platform mobile service development tool. We believe that this can be achieved using a combination of our current model-driven engineering approach and an existing hybrid application development solution. Using this “OdinTools2”, developers will specify the structure of a mobile service in much the same manner as its current implementation. From the specification, OdinTools2 will generate code - not for a specific platform, but for one of the previously identified hybrid development solutions. Initially we will explore the use of Titanium for this purpose as it offers the ability to add platform-specific extensions, which OdinTools could leverage to generate native code for particularly performance-critical sections of code. The combination of Titanium and OdinTools will allow developers to quickly specify
a service application's structure visually, while being able to create a cross-platform user interface, further decreasing development time.

We also plan to design and run a usability evaluation for OdinTools. This can be used to elicit valuable feedback on the tool’s design, as well as to explore whether OdinTools does in fact provide useful increases in productivity, as we argue in this thesis.

9.3.3 With the Test Harness

While the test harness has proven extremely useful in its current state, it is not as expressive with regards to user input simulation compared with existing approaches in the literature. We will investigate the use of Android’ UI Automation framework to increase its expressiveness in this area. Furthermore, the number of concurrent devices a test harness machine can simulate is limited by its CPU and memory capability - our prototype with 16 Gbytes of RAM can simulate 15 devices. Other approaches in the literature have increased this limit by moving into the cloud. We will investigate the best way to do this while still maintaining the fine-grained control which the test harness currently affords testers.

9.3.4 In Industry

At the time of writing this thesis, the REMOTE-CR trial is ongoing in New Zealand. We will remain available to address any issues encountered with the software - though few are expected. We are also making preparations to begin an additional trial of a similar nature at Technische Universität München, Germany in April 2015. In collaboration with NIHI and UniServices, we hope to find other areas both within and external to the healthcare industry which can benefit from Odin2.
Appendix A

Service Development with Odin2

To demonstrate how a developer might go about building a mobile service using Odin2, we will consider the construction of a simple messenger app. Using the app, mobile device users may log in and broadcast messages to all active participants. In addition, a user may broadcast messages via a REST service, accessible from a web browser. Each message from a device or web-based user can be represented by a simple string, while each device will receive broadcasts consisting of those string messages plus information about which user originally sent the message. A high-level overview of this service is shown in Figure A.1. Figure A.2 shows how the completed application appears on both Android and Windows Phone platforms.

Figure A.1: Overview of example messenger service

To develop such an application, a developer must create two separate sub-projects - a device project, and a surrogate project. The device project must be created using the
appropriate platform tools for the target mobile device platform, while the surrogate project must be created using a Java build environment. Eclipse, with its support for Maven and Java projects, is an excellent choice of IDE for surrogate development and its use will be showcased in this section. In addition, its support for Android development, via the Android Development Toolkit (ADT), makes it ideal for the Android device project. Visual Studio will be used for the Windows Phone project.

![Android app and Windows Phone app](image)

**Figure A.2**: User interface for simple messenger app

### A.1 Surrogate

To create the surrogate project in Eclipse, we start by creating a blank Maven project. This will create the desired folder structure, along with an initial POM file. The POM file is the Maven configuration script for the project and will be used to specify project dependencies and other important information.

The creation of a surrogate depends on two APIs - the `odin2-core-api` and `odin2-surrogate-api`. The core API contains all Odin2 functionality that’s shared between devices and surrogates / surrogate hosts, while the surrogate API contains surrogate-specific utilities and extension points. Listing 3 shows how these dependencies are specified in the POM file.

Another important aspect which should be specified in the POM is the name of the surrogate’s JAR file which will be generated by the build tools. To do this, we configure
Listing 3: Surrogate dependency specification within the POM file

Listing 4: Surrogate name and class specification within the POM file

Maven’s JAR plugin to generate the JAR with a specific name. Listing 4 shows how this is specified using the finalName element on line eight. Specifying a name is important as the name of the JAR file must correspond to the surrogate name which devices specify to surrogate hosts upon connection. This allows the hosts to locate the correct surrogate JAR.

Finally, in the POM, we must specify the main surrogate class. When loading a surrogate JAR file, the surrogate host will examine the JAR file’s manifest for an entry with the
key 'Surrogate-Class'. As this is a non-standard manifest entry we must specify it in the POM. This can be seen on lines 10 through 14 in listing 4.

```java
public class MessengerSurrogate extends Surrogate {
    @Override
    public Message requestReceived(UserID userId, Message request) throws Exception {
        return null;
    }

    @Override
    public void asyncMessageReceived(UserID userId, Message message) throws Exception {
    }
}
```

**Listing 5:** Blank Surrogate class, with required method skeletons

Once the POM is defined we can start creating our Java classes. The main class needed is the surrogate class. When creating this, we make sure that the full name of the class (including the package) matches the Surrogate-Class definition in the POM. We also must extend the `nz.ac.auckland.cs.odin.surrogate.Surrogate` base class (part of `odin2-surrogate-api`). Listing 5 shows the blank surrogate class definition, including two method skeletons required to be implemented - `requestReceived` and `asyncMessageReceived`. These methods are called by the surrogate host when requests and asynchronous messages are received, respectively.

```java
public class SampleMessage {
    private String userName;
    private String messageContent;
    /* Getters and Setters */
}
```

**Listing 6:** SampleMessage POJO declaration

Odin2 has a message type - `BasicStringMessage` - which devices can use to send messages to the surrogate. We then want to broadcast those messages, along with a user name. We’ll need to define a custom class to represent a message to be broadcast. Listing 6 shows this definition - a simple POJO named `SampleMessage` which we’ll serialize as JSON using Odin2’s `JsonMessage` class. Odin2’s Android implementation utilizes the
Jackson JSON processor\(^1\) to automatically manage this process (and its surrogate host uses the same API for deserialization).

```java
public void asyncMessageReceived(
    UserID userId, Message message) throws Exception {
    // Get the async message sent by the device
    BasicStringMessage contentFromDevice = (BasicStringMessage) message;
    String content = contentFromDevice.getString();
    String senderName = userId.getUserName();

    // Multicast the message content and username
    // to all currently active devices
    SampleMessage toMC = new SampleMessage();
    toMC.setUserName(senderName);
    toMC.setMessageContent(content);
    JsonMessage jsonToMC = new JsonMessage(toMC);
    multicastAsyncMessageToAllActiveUsers(jsonToMC);
}
```

**Listing 7:** Completed `asyncMessageReceived` method body

When we write the device logic below, we’ll specify that each message to be broadcast will be sent as an asynchronous `BasicStringMessage`. Each of these will be received by the surrogate at the `asyncMessageReceived` method. Listing 7 shows our implementation of this method. Lines four through seven get the necessary information to broadcast, including the message content and the sender’s name. Lines 11 through 13 encapsulate this information in a `SampleMessage` instance, line 14 wraps it in a `JsonMessage` (provided by Odin2’s core API and allows the serialization of any POJO as JSON), then line 15 sends that message to all active users (an active user is a device currently connected to the surrogate - temporary network errors do not render a device inactive).

Our final surrogate implementation task is to create a web method which may be accessed as a REST service over HTTP. This method will take the value `message` request parameter and broadcast it to all active devices, representing the sender name as “Big Brother”. Listing 8 shows our implementation of this method.

As each surrogate is exposed as a Spring MVC controller by the surrogate host, we may use any valid Spring annotations within the surrogate to expose our web methods. We can see this in listing 8. Line one shows a `@RequestMapping` annotation stating that the method will handle any HTTP GET requests to the relative URL `/broadcast`. Line three’s `@RequestParam` annotation specifies that the `content` argument will be populated with the value of the `message` request parameter (e.g. if a request is made to

\(^1\)https://github.com/FasterXML/jackson
Appendix A. Service Development with Odin2

清单8：Surrogate web方法定义

`@RequestParam`（value = "/broadcast", method = RequestMethod.GET）

`@ResponseBody`  

public SampleMessage bigBrother(@RequestParam("message") String content)  
  throws InterconnectException {

  SampleMessage toMC = new SampleMessage();
  toMC.setUserName("Big Brother");
  toMC.setMessageContent(content);
  JsonMessage jsonToMC = new JsonMessage(toMC);
  multicastAsyncMessageToAllActiveUsers(jsonToMC);

  return toMC;
}

//broadcast?message=hello，然后content参数的值将为“hello”）。行二的@ResponseBody注解指定应返回的值应作为HTTP响应主体。可以省略此注解，让Spring选择返回一个View（例如一个JSP页面）。实现方法，我们只需将适当的值包装在一个SampleMessage中，将其转换为JSON，并向所有活跃设备广播，与上述的asyncMessageReceived实现类似。

A.2 Android

为了创建Android应用，我们使用Android开发工具Eclipse插件。这将创建一个空白Android应用项目，其中包括必要的配置文件，并允许我们轻松将应用部署到设备或模拟器进行测试。创建项目时，我们确保导入了Odin2 Android API项目作为库。

一旦项目创建后，我们需要编辑应用的manifest文件（Android-Manifest.xml）。我们需要添加三个uses-permission条目以允许我们的应用访问所需的系统功能。Listing 9展示了这一过程，从行6到11。所需权限包括i) 全网访问（INTERNET），ii) 检测网络环境变化的能力（ACCESS_NETWORK_STATE），和 iii) 将其写入外部存储，用于日志的目的（WRITE_EXTERNAL_STORAGE）。

\[http://developer.android.com/tools/sdk/eclipse-adt.html\]
<?xml version="1.0" encoding="utf-8"?>
<manifest ...
...
<uses-permission
android:name="android.permission.INTERNET"/>
<uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<application ...
...
<service android:name="nz.ac.auckland.cs.odin.android.api.services.OdinService" />
</application>
</manifest>

Listing 9: Important parts of Android Manifest file

In addition to adding permissions, we also need to expose the OdinService class as a service which may be consumed by our app. To do this we add a service entry to the manifest, as shown in listing 9 line 17.

Once the manifest has been properly defined, we can create the user interface. This can be achieved using the layout editor available within the ADT Eclipse plugin. We will create a basic UI consisting of a TextView to enter login details, another one to enter messages to send, buttons to login and send messages, and a ListView to display received messages. The UI can be seen in figure A.2 (a).

Along with the UI, we will create an Activity which will be run on application startup, and will display the UI, start the Odin service, and contain our business logic. Listing 10 shows the parts of that activity which are related to Odin. The activity extends OdinFragmentActivity, which provides utility methods to interact with Odin, and will automatically start the Odin service on creation, if it is not already created. It also implements the MessageHandler interface, which will allow it to respond to messages received from the surrogate. An OdinFragmentActivity will automatically register itself with Odin as a handler for received messages, if it implements this interface. The MessageHandler interface requires that we implement the requestReceived and asyncMessageReceived methods shown in the listing.
public class MainActivity extends OdinFragmentActivity
    implements MessageHandler {

    /* Instance variables, etc. */

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Define the database.
        DatabaseManager.getInstance()
            .defineDatabase("sample-messenger.db", 1,
                           OdinDBHelper.class);

        // Static preferences
        OdinPreferences.HostName
            .setValue(this, "odin.cs.auckland.ac.nz");
        OdinPreferences.Port.setValue(this, 8081);
        OdinPreferences.SurrogateName
            .setValue(this, "sample-messenger");

        /* Setup UI */
    }

    @Override
    public Message requestReceived(Message request)
        throws Exception {

        return null;
    }

    @Override
    public void asyncMessageReceived(Message message)
        throws Exception {

    }
}

Listing 10: Main Android application Activity

The activity’s `onCreate` method is called by the operating system when it is started. Here we can see how we define a database, on line 11. All Odin2 apps must define a database that supports at least the tables defined in the `OdinDBHelper` class (provided by the API). These tables persist Odin2’s sequence numbers and pending messages in case of an application crash. We may specify our own database tables by supplying our own DB Helper classes. These must extend `SQLiteOpenHelper`, part of the ORMLite
Object-Relational-Mapping framework used by Odin2’s Android API\(^3\).

Additionally, when the activity is created, we can set several application properties required by Odin. We provide an enumeration, `OdinPreferences`, containing all properties required. These include the name of the surrogate, as well as the host name and port, and any user information needed to identify ourselves to the surrogate host. We can provide the host name, port, and surrogate name on creation - user info will be added later.

```java
private class ConnectTask extends OdinConnectTask {

    @Override
    protected void onPreExecute() {
        btnLogin.setEnabled(false);
        String username = txtUsername.getText().toString();
        OdinPreferences.UserName.setValue(MainActivity.this, username);
        OdinPreferences.UserID.setValue(MainActivity.this, username.hashCode());
    }

    @Override
    protected void onPostExecute() {
        try {
            getResult();
            btnLogout.setEnabled(true);
        } catch (Exception e) {
            /* Error handling... */
        }
    }
}
```

Listing 11: OdinConnectTask usage in Android

Next, we will define an implementation of `OdinConnectTask`. This task is an extension of Android’s `AsyncTask` which will connect to Odin on a background thread, and possibly perform user operations. We may also override the `onPreExecute` and `onPostExecute` methods to execute code on the UI thread, both before the Odin connection is established, and after the work is completed. Listing 11 shows a task which, before connecting, will set the user name and ID based on the information the user has entered. After establishing the connection, we will check whether the connection was successful by examining the

\(^3\)http://ormlite.com/
result (line 17). This will throw an exception if the connection failed. Otherwise, we won’t do anything else here.

```java
@Override
public void onClick(View v) {
    String toSend = txtMessage.getText().toString();
    BasicStringMessage message = new BasicStringMessage(toSend);
    try {
        getOdinService().sendAsyncMessage(message);
    } catch (InterconnectException e) {
        /* Error handling... */
    }
}
```

**Listing 12:** Sending an Odin2 asynchronous message in Android

Next we will define a method which will respond to the user clicking the “Send Message” button. Listing 12 shows this method definition, in which we create a `BasicStringMessage` containing the user’s message, and send it to the surrogate.

```java
@Override
public Message requestReceived(Message request) throws Exception {
    return null;
}

@Override
public void asyncMessageReceived(Message message) throws Exception {
    JsonMessage json = (JsonMessage) message;
    SampleMessage msg =
        (SampleMessage) json.deserialize(SampleMessage.class);
    /* Display message on UI... */
}
```

**Listing 13:** Receiving an Odin2 asynchronous message in Android

Finally, we must display received messages on the UI. Listing 13 shows our implementation of `asyncMessageReceived` which will read the JSON message sent by the surrogate, and deserialize it in to our `SampleMessage` data object. To gain access to the `SampleMessage` class, we may simply copy it from our surrogate project to the Android project. In more complex apps with many shared classes, we may instead define a `shared` project, to be referenced by both our Android and Surrogate projects.
A.3 Windows Phone

To create the Windows Phone app, we must install the Windows Phone SDK\(^4\). We then create a new Phone project, and set the API level to 7.1. Finally we add a reference to the Odin2 API for Windows Phone. Unlike Android, Visual Studio automatically detects when certain features are used by apps developed using the IDE, and adds the necessary permissions (e.g. internet access) to the configs. There is no need to manually define these in Visual Studio. Similarly we do not need to add a secondary reference to any Odin2 services in the app’s manifest - simply referencing the API once is enough.

```csharp
public class SampleMessage : INotifyPropertyChanged
{
    private string _UserName;
    public string UserName
    {
        get { return _UserName; }
        set
        { _UserName = value;
          NotifyPropertyChanged("UserName"); }
    }

    private string _Message;
    public string Message
    {
        get { return _Message; }
        set
        { _Message = value;
          NotifyPropertyChanged("Message"); }
    }

    public event PropertyChangedEventHandler PropertyChanged;
    private void NotifyPropertyChanged(String propertyName)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (null != handler)
        {
            handler(this,
            new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Listing 14: SampleMessage class in C# for Windows Phone

\(^4\)http://dev.windows.com/en-us/develop/download-phone-sdk
Once the project has been created, we can initially define the `SampleMessage` class in C#.

The version shown in listing 14 utilizes the .Net property change notification mechanism, which will inform the user interface whenever this object’s properties change.

```csharp
public partial class MainPage : PhoneApplicationPage, IMessageHandler {

    /* Instance variables, etc. */

    public MainPage()
    {
        InitializeComponent(); // Setup UI

        Odin2.InitDatabase("sample-messenger.db");

        Odin2.SetPref(Odin2Pref.HostName, "odin.cs.auckland.ac.nz");
        Odin2.SetPref(Odin2Pref.Port, 8081);
        Odin2.SetPref(Odin2Pref.SurrogateName, "sample-messenger");
    }

    public Message RequestReceived(Message request)
    {
        throw new NotImplementedException();
    }

    public void AsyncMessageReceived(Message message)
    {
        throw new NotImplementedException();
    }
}
```

Listing 15: Main application page

When we create a new Phone project, a default user interface, `MainPage.xaml`, is created. We will modify this file to create the app’s UI, which can be seen in figure A.2 (b).

Along with the xaml file, a code-behind file, `MainPage.xaml.cs`, is also created. We will add to the generated skeleton as shown in listing 15. We let the class implement `IMessageHandler`, which will allow it to respond to messages received from the surrogate. This interface requires that we implement the `RequestReceived` and `AsyncMessageReceived` methods. Additionally, in the constructor beginning on line 6, we initialize the app’s database and preferences. Here we can see the use of the `Odin2` static class in the Windows Phone API. Essentially this class performs much the same functionality as the
OdinService class for Android. For specifying Odin2 preferences in Windows Phone, we cannot support stateful enumerations as with Android, as these are not supported in C#. Instead, we have a stateless Odin2Pref enumeration - essentially a collection of constants - and we supply these to Odin2’s SetPref method.

```csharp
Action before = new Action(() =>
{
    btnConnect.IsEnabled = false;
    Odin2.SetPref(Odin2Pref.Username, txtUsername.Text);
});

Action<ConnectResult> after = new Action<ConnectResult>(r =>
{
    try
    {
        r.GetResult();
        btnDisconnect.IsEnabled = true;
    }
    catch (Exception ex)
    {
        /* Error handling... */
    }
});

Odin2.Connect(this, before, after);
```

Listing 16: Connecting to Odin2 in Windows Phone

Next, using the visual designer, we can add a click event handler for our “Connect” button to MainPage.xaml.cs. In the body of that handler method, we will add the code shown in listing 16. Here we can see how C#'s support for delegates and lambda expressions is used by the Odin2 Windows Phone APIs. In the listing, an Action is a class which represents a method that takes some number of arguments. The syntax shown allows us to declare a method in-line to encapsulate within that action. In listing 16 we define two actions. On line 1, we define an action which should run before connecting to the surrogate host. This will disable the “Connect” button and set the username and user ID to the correct values. On line 8 we then define an action which should run once we are connected, or in case of failure. Here, we check whether the connection was successful on line 12, and perform error handling if not. On line 21, we actually perform the connect operation. This method is asynchronous - it will return immediately. The code specified in the two actions will be executed on the UI thread at the appropriate time.

Next we will define a method which will respond to the user clicking the “Send Message” button. Listing 17 shows this method definition, in which we create a BasicStringMessage
private void btnSend_Click(object sender, RoutedEventArgs e)
{
    BasicStringMessage message = new BasicStringMessage
    {
        Text = txtMessage.Text
    };

    try
    {
        Odin2.SendAsyncMessage(message);
    }
    catch (Exception ex)
    {
        /* Error handling... */
    }
}

Listing 17: Sending an asynchronous Odin2 message from Windows Phone

containing the user’s message, and send it to the surrogate.

public Message RequestReceived(Message request)
{
    return null;
}

public void AsyncMessageReceived(Message message)
{
    JsonMessage json = message as JsonMessage;
    SampleMessage msg = json.Deserialize<SampleMessage>();
    /* Display message on UI... */
}

Listing 18: Receiving an asynchronous Odin2 message in Windows Phone

Finally, we must display received messages on the UI. Listing 18 shows our implementation of AsyncMessageReceived which will read the JSON message sent by the surrogate, and deserialize it into our SampleMessage data object.
Appendix B

Detailed Requirements and Metrics Explanation for REMOTE-CR

B.1 Requirements

Initially, REMOTE-CR’s set of requirements were generated by NIHI’s Ralph Maddison and Jonathan Rawstorn. These were motivated by the necessary elements in a cardiac rehabilitation programme, along with previous CR study outcomes [90, 92, 96, 100–104] and the desire to leverage modern smartphones more fully than previous studies. The list of requirements was collaboratively refined into the list of requirements in this appendix.

Within the REMOTE-CR system, there are five primary application areas. These are: (i) Profile, (ii) Monitoring, (iii) Routes, (iv) Goals, and (v) Review. The requirements for these application areas are covered in sections B.1.1 through B.1.5, respectively.

B.1.1 Requirements: Profile

1. Each participant must have a username and password, which they can use to log into the REMOTE-CR from any Android smartphone on which the REMOTE-CR software is installed.

2. Each participant must have a profile containing that participant’s name, date of birth, height, weight, gender, resting heart rate, maximal heart rate, and blood lactate concentration data (obtained from a separate assessment in a clinic prior to trial commencement).
3. Participants must be able to view and edit their profile from within the mobile application.

4. Clinicians must be able to view and edit the profiles of any participants from within the web application.

5. Whenever a participant or clinician changes a profile, this change must be synchronized between the mobile and web applications as soon as mobile connectivity is available.

6. Whenever a participant logs out of the REMOTE-CR application, his or her profile information must be deleted from that device.

7. Whenever a participant logs into the REMOTE-CR application, his or her profile must be made available on that device.

B.1.2 Requirements: Monitoring

Requirements for participants:

1. Participants must be able to initiate an exercise monitoring session using their smartphones at any time.

2. The mobile application must be able to connect to a Zephyr BioHarness 3 device using a Bluetooth connection.

3. Physiological data captured by the BioHarness must be made available to clinicians using the web application as soon as possible given the mobile network conditions at the time the data was captured. Physiological data to be transmitted currently includes a participant’s heart rate and breathing rate.

4. The mobile application must be able to utilize a smartphone’s GPS receiver to capture a participant’s location. Whenever a participant’s location changes, the new location must be sent to the web application as soon as possible given the current mobile network conditions.

5. If there is no mobile connectivity when new physiological or location data is captured, this data must be buffered and sent once connectivity is restored. Data must not be lost.

6. In addition to the data captured from sensors, two exercise metrics should be streamed to the web application in a similar manner. The first is Percent of Heart Rate Reserve (%HRR) and gives an estimate of how hard a participant’s heart is
Appendix B. *REMOTE-CR Detailed Requirements*

Currently working. The second is *Training Load* and gives an estimate of how much cumulative effort a participant has put in during the entire exercise session to date. For more information on these metrics we refer the reader to section B.2.

7. During an exercise session, the mobile application must store a buffer of the most recent ten seconds of *Electrocardiograph* (ECG) data.

8. During an exercise session, participants must be able to inform clinicians of the occurrence of three types of *symptoms* associated with cardiac events. These are *angina*, *dyspnoea*, and *syncope*. These must be represented to the participant using their common names: *chest pain*, *shortness of breath*, and *light-headedness*, respectively. A participant must also be able to specify a *severity level* in a symptom report. The possible severity levels are:

   (a) Light, barely noticeable
   (b) Moderate, bothersome
   (c) Severe, very uncomfortable
   (d) Most severe ever experienced

9. When a symptom is reported by a participant, the most recent ten seconds of ECG data must be sent along with the symptom report.

10. During a monitoring session, a participant must be able to view the following information within the mobile application:

    • Location, on a map
    • A *route* path to follow, on the same map (see B.1.3 below)
    • Current heart rate, in *beats per minute* (bpm)
    • Average heart rate (bpm)
    • Current percentage of heart rate reserve (%HRR)
    • Average percentage of heart rate reserve (%HRR)
    • Current speed, in *kilometers per hour* (km/h)
    • Average speed (km/h)
    • Distance traveled, in *kilometers* (km)
    • Training Load
    • Any notifications received from the clinician, along with the ability to listen to those notifications (*text-to-speech*).

11. Before starting an exercise session, a participant must be able to choose a *goal* for that session, if they so desire. The possible goals are:
• A specific average heart rate or \( \%HRR \) over a period of time
• Distance traveled
• Average speed over a period of time
• Total load for a session

12. During a session, if a participant set a goal before starting that session and they achieve the goal, a visual and audible notification must be given to the participant. They must then be given the opportunity to continue or stop exercising.

13. Anytime during a session, participants must be able to stop that session. When they do, clinicians must be notified that the participant has done so, and the participant must be presented with a review of the completed session. The review must contain information as detailed in section B.1.5.

Requirements for clinicians:

1. Clinicians must be able to use a web browser to view in-progress exercise sessions of any number of participants.

2. For each active session, clinicians must be able to view all physiological, location, and metric data detailed above, as well as all symptom reports from participants. As new data arrives from participants, the web application must be automatically updated without the clinician needing to refresh the page.

3. When a participant’s symptom report arrives at the clinician’s browser, an audible notification must be played to alert clinicians.

4. Clinicians must be able to view any ECG data associated with symptom reports from any participant.

5. Clinicians must be able to request ECG data from participants. This must result in the most recent ten seconds of ECG data captured from that participant’s BioHarness being displayed to the clinician, as soon as mobile connectivity to that participant is available.

6. Clinicians must be able to send arbitrary text messages to participants. When such messages arrive, the mobile application must play an audible text-to-speech notification to that participant. A visual confirmation must be displayed to clinicians when a participant receives a message.
B.1.3 Requirements: Routes

1. During an exercise session, a participant’s location must be periodically recorded. These recordings form the Route which the participant has undertaken during that session.

2. Once a session is completed, the participant’s route during that session must be persisted on the smartphone if it is different from any route previously used by that participant.

3. Periodically, when mobile connectivity is available, routes must be synchronized between the smartphone and the web app. The routes to be downloaded onto a participant’s smartphone must include not only that participant’s routes, but any routes undertaken by other participants which are nearby to that participant’s routes. “Nearby” is currently set at five kilometers, but the figure must be easy to change.

4. Participants must be able to view a list of all routes currently stored on their smartphone. Each route must be associated with a total length, a distance from the participant’s current location, and a creator (i.e. the name of the participant which originally traveled that route).

5. When browsing the list of available routes, participants must be able to select an existing route which should be displayed on the participant’s map during an exercise session. Doing this must launch the smartphone app’s workout functionality (B.1.2), with the map being displayed in the foreground and the selected route clearly visible.

B.1.4 Requirements: Goals

1. In addition to the per-session goals specified in section B.1.2, participants must be associated with a set of weekly goals. The progress towards these goals accumulates over the course of a week (each new week begins on a Monday and resets the goals from the previous week). Metrics to be associated with these goals are:
   - Total distance traveled
   - Total exercise duration
   - Number of sessions per week
   - Average %HRR over a given week
   - Total training load for a week
2. Participants must be able to view and edit their weekly goals using the smartphone app.

3. Clinicians must be able to view and edit weekly goals for all participants using the web app.

4. Changes made by participants or clinicians must be synchronized when mobile connectivity is available.

\section*{B.1.5 Requirements: Review}

Requirements for participants:

1. Participants must be able to view a summary of any exercise session they’ve undertaken on their smartphone. The data to be made available to the participant for a particular session includes:
   
   - The date of the session
   - Session duration ($hh:mm:ss$)
   - Distance traveled ($km$)
   - Training load
   - Minimum, average, and maximum speed ($km/h$)
   - Minimum, average, and maximum heart rate ($bpm$)
   - Minimum, average, and maximum $\%HRR$
   - Any reported symptoms
   - Any received messages
   - A map showing the route traveled during the session

2. Participants must be able to view a monthly summary for any month during which an exercise session was undertaken. The following information should be made available for each month:
   
   - Total duration ($hh:mm:ss$)
   - Total distance ($km$)
   - Average speed ($km/h$)
   - Average heart rate ($bpm$)
   - Total training load
3. Whenever a participant, who has previously used REMOTE-CR, logs in using a different smartphone, all their exercise summary information must be made available on that new smartphone.

Requirements for clinicians:

1. Clinicians must be able to view a detailed summary for any session completed by any participant. The information to be made available for each session includes:
   - The date and time of the session
   - Session duration (hh:mm:ss)
   - Total distance traveled (km)
   - A plot of distance traveled vs. time
   - Total training load
   - A plot of load vs. time
   - Minimum, average, and maximum speed (km/h)
   - A plot of speed vs. time
   - Minimum, average, and maximum heart rate (bpm)
   - A plot of heart rate vs. time
   - Minimum, average, and maximum %HRR
   - A plot of %HRR vs. time
   - Minimum, average, and maximum breathing rate (bpm)
   - A plot of breathing rate vs. time
   - Any received symptoms
   - Any received ECG data, along with their associated symptoms, or by themselves if requested manually
   - Any sent messages

2. Clinicians must also be able to view weekly, monthly, and yearly summaries for each participant. These summaries should include plots of heart rate, %HRR, distance, breathing rate, speed, and load information. For weekly and monthly summaries, these plots should show the daily total (for distance and load) or average (for heart rate, %HRR, breathing rate, and speed), whereas the yearly summary should show weekly totals or averages. Each point on the plot should also show the total number of symptoms received that day / week.

3. Clinicians must be able to view a list of all participants. Each participant in the list must be associated with a figure indicating the total number of completed sessions, and the date and time of their most recently completed session.
B.2 Calculated Performance Metrics in REMOTE-CR

Along with the raw data generated by the monitoring device, two calculated values should also be streamed. These are Percent of Heart Rate Reserve (%HRR) and Training Load (L).

B.2.1 Percent of Heart Rate Reserve

Percent of Heart Rate Reserve, %HRR is a good indicator of how hard a participant’s heart is working. It is calculated as follows:

\[
%HRR = 100 \times \left( \frac{HR - HR_{rest}}{HR_{max} - HR_{rest}} \right)
\]

Where HR is a participant’s current heart rate, HR_{rest} is a participant’s resting heart rate, and HR_{max} is a participant’s maximal heart rate. Assuming HR_{rest} and HR_{max} are set appropriately by a prior clinical assessment, then %HRR ∈ [0, 100]. %HRR = 0 indicates a person is at rest, while %HRR = 100 indicates a person’s heart is working as hard as it can.

B.2.2 Training Load

Training load, L, is a measure of the total amount of work done by a particular participant during an exercise session up until the point at which the measurement is taken. It will steadily increase with time, and its rate of increase depends on changes in the participant’s %HRR over time, along with other physiological factors.

To define L, we first define L_i as being the instantaneous load at epoch i. An epoch is a discreet number given to a block of elapsed time between one measurement and the next. L_i is calculated as follows:

\[
L_0 = 0, \\
L_i = mins(t_i - t_{i-1}) \times %HRR_i \times y_i
\]

Where t_i is the time at epoch i, mins(t_i - t_{i-1}) is the elapsed time in minutes between epoch i - 1 and i, and y_i is an exponential function. When blood lactate concentration
data is not available, an estimate of $y_i$ can be given based on a participant’s gender. This estimate is as follows:

$$y_i = 0.64e^{1.92 \left(\frac{\%HRR}{100}\right)} \text{ for men, } 0.86e^{1.67 \left(\frac{\%HRR}{100}\right)} \text{ for women.}$$

When blood lactate concentration data is available for a participant, we can give a personalized estimate for $y_i$. To get this estimate, the participant performs an exercise test in a clinic, prior to the use of REMOTE-CR. The blood lactate concentration data from this test must be entered into the REMOTE-CR system as a number of $\%HRR$ values versus the participant’s blood lactate concentration while exercising at those intensities. The resulting $y_i$ estimate will be the exponential function relating $\%HRR$ and blood lactate concentration, and can be estimated from the exercise test data using a linear regression algorithm.

Finally, we can define $L$ at a given epoch to be the sum of all $L_i$ up to that epoch. Thus, if the current epoch is $n$, then the figure, $L$, representing the cumulative training load for the entire exercise session up to the current point can be defined as:

$$L = \sum_{i=0}^{n} L_i$$
Bibliography


227


of theoretical basis, use of behaviour change techniques, and mode of delivery on efficacy. *Journal of Medical Internet Research*, 12(1), 2010. doi: 10.2196/jmir.1376.


