Opportunistic Networking Applications: From Theory to Practice

Teemu Kärkkäinen
Opportunistic Networking
Applications: From Theory to Practice

Teemu Kärkkäinen

A doctoral dissertation completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Electrical Engineering, Remote connection link (e.g. Zoom), on 22 July 2021 at 16:15.

Aalto University
School of Electrical Engineering
Department of Communications and Networking
Supervising professor
Professor Jörg Ott, Aalto University, Finland

Thesis advisor
Professor Esa Hyytiä, University of Iceland, Iceland

Preliminary examiners
Professor Lars Wolf, Technische Universität Braunschweig, Germany
Dr. Elko Yoneki, University of Cambridge Computer Laboratory, United Kingdom

Opponent
Professor Aruna Balasubramanian, Stony Brook University, United States of America
Modern smartphones are the primary gateways to information, entertainment, social networks, and private relationships for billions of people. They typically achieve this by connecting to centralized Internet services via fast wireless links.

However, these smart devices have another, less used, set of communication capabilities: They can also talk directly over device-to-device communication technologies, and exchange content and messages without the need for the Internet. Message passing over these contacts creates an Opportunistic or a Delay-Tolerant Network.

This form of networking is particularly useful in rural and remote areas that lack infrastructure, and in urban areas where the infrastructure has been overloaded by a crowd or disabled by a disaster. Even when infrastructure is available, it allows networking without relinquishing control and privacy to a centralized service.

In this dissertation, we contribute to advancing the availability and capability of opportunistic networking. We advance the availability by improving the opportunities for creating opportunistic contacts for smartphones, and the capability by designing advanced mechanisms for both content and computation interactions.

For connectivity, we measure to what extent existing public Wi-Fi access points allow direct device-to-device communications. We then introduce mechanisms to help scale device-to-device communications in access points with many clients. As an alternative to using existing infrastructure, we introduce the Liberouter neighborhood networking system based on cheap standalone opportunistic routers.

On top of this basic connectivity, we design more advanced content access mechanisms to ease the development of opportunistic networking applications. In particular, we design a request/response mechanism for accessing content, particularly websites, that reside on servers outside the opportunistic network. We also design a query/response mechanism to search for content in other nearby devices. Finally, we present mechanisms for shared content editing, e.g., wikis, in opportunistic networks.

We also introduce mechanisms to support opportunistic computations. First, we enable interactive web applications in the Liberouter network by attaching custom computations to the content messages. Second, we design a framework for the Liberouters for the opportunistic composition of general distributed services.

Together, these contributions help advance the technical basis for smartphone-based opportunistic networking applications.
Viime vuosikymmenen aikana älypuhelimista on muovautunut miljardien ihmisten portti tietoon, viihteeseen ja sosiaaliseen vuorovaikutukseen. Tämän kehityksen ovat mahdollistaneet nopeat langattomat verkkoteknologiat, jotka kytkevät puhelimet Internetin suuriin, keskitettyihin palveluihin. Internet-yhteyksien lisäksi modernit älypuhelimet kykenevät kuitenkin myös avaan-aamaan yhteyksiä suoraan toistensa välille. Nämä vähemmän käytetyt teknologiat mahdollistavat viestien ja tiedon välittämisen, toisiin ilman Internetiä ja sen palveluja. Tietoverkkokoja, jotka perustuvat suoralle viestienvaihdolle laitteiden välillä, kutsutaan opportunistiksi tai viivesietoisiksi verkkoiksi.

Opportunistiset verkot ovat erityisen hyödyllisiä syrjäisillä alueilla, joissa infrastruktuuriverkkojen peitto on heikko. Lisäksi niistä on apua, jos tietoverkot ovat ylikuormittuneet tai suuren käyttäjämäärän seurauksena. Opportunistisissa verkoissa on myös mahdollista paremmin estää keskitettyjen palvelujen usein harjoittamaa seurantaa ja hallintaa.

Tämä väitöskirjatutkimus pyrkii parantamaan opportunististen tietoverkkojen käytettävyyttä ja kyyvkyyttä. Käytettävyyttä työssä parannetaan tekniikoilla, jotka lisäävät suoria laitteiden välisiä tiedonsiirtomahdollisuuksia. Verkkojen kyyvkyyttää työssä parannetaan kehittämällä uusia mekanismeja sekä tiedonsiirtoon että tiedojenkäsittelyyn opportunistisissa verkoissa.


Avainsanat opportunistiset verot, viivesietoiset verot

ISSN (painettu) 1799-4934 ISSN (pdf) 1799-4942
Julkaisupaikka HelsinkI Painopaikka HelsinkI Vuosi 2021
Preface

This dissertation collects works from a significant span of the evolution of opportunistic networking, with a particular DTN focus. The DTN community at its peak was truly special, with brilliant networking researchers ranging from Internet veterans to passionate new PhD students, and meeting venues ranging from the Himalayas to the Galápagos Islands. The same time period also saw the rise of wide-spread availability of networked general purpose computing in everyone's pockets, which enabled the innovative new ideas to be actually built in practice. It is this serendipitous combination of an inspiring community and an exciting new technology that resulted in this dissertation.

This work would not have been possible without the collaborations with many great colleagues. Remarkably, all of them have been not only extremely talented researchers, but also exceptionally nice people to work with. I am most indebted to my supervisor Prof. Jörg Ott for the countless opportunities he has provided to me over the years. With his remarkable combination of a vast breadth and depth of technical knowledge, and his deep dedication and caring for his team, it is impossible to imagine a better boss.

The early papers in this dissertation are mostly collaborations with Mikko Pitkänen, which was very enjoyable. Later collaboration with Marcin Nagy was likewise fruitful and memorable. The work done by Ari Keränen on The ONE simulator has been crucial for enabling this thesis, and many others in this field. Many talented master's and bachelor's thesis students at Aalto and TUM have been invaluable for this work, in particular Chrysa Papadaki, Mika Välimaa and Shourov Kumar Roy—working with all the motivated and capable thesis students has been one of the greatest parts of this job.

I am also thankful for the pre-examiners Prof. Dr.-Ing. Lars Wolf from
the Technische Universität Braunschweig, and Dr. Eiko Yoneki from the University of Cambridge Computer Laboratory, and my advisor Prof. Esa Hyytiä from the University of Iceland, for their thorough and insightful comments. Their expert feedback has been valuable in improving this dissertation.

The work has been funded in part by the Academy of Finland (DISTANCE, RESMAN), and the European Commission under the FP7 (SCAMPI), the H2020 (PRECIOUS, RIFE), and the EIT ICT Labs programs.

Finally, my family both in Finland and in Serbia have been essential to my happiness and well-being over the years. The regular Christmas and summer trips to both countries have always provided the much-needed opportunities to relax and disconnect from work. The continuing support and encouragement of my loving and caring wife Ljubica mean that I am never alone when things get tough. In the end, the pending arrival of the tiny Eino Nikola provided the final incentive to wrap up this work, to be better able to provide him with a happy life.

Munich, Germany, June 23, 2021,

Teemu Kärkkäinen
Contents

Preface 1

Contents 3

List of Publications 7

Author’s Contribution 9

List of Figures 13

List of Tables 15

List of Abbreviations 17

1. Introduction 23
   1.1 Contributions ................................. 25
   1.2 Methodology ................................. 27
   1.3 Structure of the Thesis ................. 28

2. Concepts of opportunistic networking 31
   2.1 Basic Concepts ................................. 32
      2.1.1 Human Mobility ............................ 34
      2.1.2 Wireless Device-to-Device Communication .... 38
      2.1.3 Store-Carry-Forward over Spacetime Paths ... 40
      2.1.4 Routing and Dissemination .......... 41
      2.1.5 Platforms and Middlewares .......... 48
      2.1.6 Summary ................................. 52
   2.2 Simulating Opportunistic Networks: Publication I .... 54
      2.2.1 Simulator Architecture ..................... 54
      2.2.2 Creating Contacts ....................... 56
      2.2.3 Store-Carry-Forward Messaging ........ 58
2.2.4 Outputs ........................................... 60
2.2.5 Example Simulation ............................. 61
2.2.6 Summary ....................................... 64

2.3 Implementing Opportunistic Networks: The Scampi Router 65
2.3.1 Router Architecture ................................ 65
2.3.2 Store-Carry-Forward Subsystems .................. 68
2.3.3 Control Subsystems ................................ 71
2.3.4 Application Interface ............................... 73
2.3.5 Extensions and Utilities ............................ 74
2.3.6 Summary ........................................ 74

2.4 Conclusions ....................................... 75

3. Creating Opportunistic Connectivity ............................. 77
3.1 Cross Traffic in Wi-Fi Access Points: Publication II ........ 80
3.1.1 Public WLAN System Model ...................... 81
3.1.2 Opportunistic Contact Model ....................... 82
3.1.3 Measurement Methodology ......................... 85
3.1.4 Results and Evaluation ............................. 87
3.1.5 Interlude: Protecting Subscribers with Worse-
than-Best-Effort Transport for Cross-Traffic ............ 90
3.1.6 Summary ........................................ 92

3.2 Adapting to Dense Wi-Fi Networks: Publication III ........ 93
3.2.1 Scenario Model ................................... 93
3.2.2 Testbed Setup and Methodology .................. 95
3.2.3 Controlling the Logical Topology ................. 97
3.2.4 Exploiting Spectrum Diversity .................... 100
3.2.5 Summary ........................................ 106

3.3 Neighborhood Networking with Liberouters: Publication IV 107
3.3.1 Liberouter System Design ......................... 108
3.3.2 Liberouter Implementation ......................... 109
3.3.3 Liberouter Applications ............................ 111
3.3.4 Simulation Evaluation ............................... 113
3.3.5 Summary ........................................ 116

3.4 Conclusions ....................................... 117

4. Finding and Manipulating Content ................................ 119
4.1 Accessing Web Content: Publication V .................... 121
4.1.1 System Design ................................... 122
4.1.2 Simulation Evaluation ............................... 129
This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.


Author’s Contribution

Publication I: “The ONE simulator for DTN protocol evaluation”

The ideas and concept were developed mainly by Ari Keränen and Jörg Ott. I designed and implemented the application modeling and testbed integration capabilities of the simulator, and contributed to various other aspects of the implementation. I co-edited the paper with the other authors.

Publication II: “Enabling Ad-Hoc-Style Communication in Public WLAN Hot-Spots”

The ideas and concept behind this work were jointly formulated with Mikko Pitkänen and Jörg Ott. I developed the measurement system jointly with Mikko Pitkänen. I designed and ran the data collection and analyzed the results. I co-edited the paper with the other authors.

Publication III: “Practical Opportunistic Content Dissemination Performance in Dense Network Segments”

I developed the ideas and concept behind this work, and refined the topology control ideas with Mika Välimaa. The testbed was implemented by Mika Välimaa, and topology control experiments run by Shourov Kumar Roy, and I analyzed the results. I designed and implemented the spectrum sharing system, and designed, ran and evaluated the testbed experiments. I created the simulator, and designed and evaluated the experiments. I co-edited the paper with Esa Hyytiä, Jörg Ott.
Publication IV: “Liberouter: Towards Autonomous Neighborhood Networking”

I formulated the ideas in the work, and refined them with a team of students from Aalto Design Factory Product Development Project 2012 course. I designed and implemented the networking platform and all the applications. I created the simulation model and did the evaluation. I co-edited the paper with Jörg Ott.

Publication V: “Opportunistic Web Access via WLAN Hotspots”

The ideas behind the paper were jointly developed with Mikko Pitkänen and Jörg Ott. I implemented the client and bridge node prototypes and did the practical evaluation. The simulation experiments were jointly designed and evaluated with Mikko Pitkänen and Jörg Ott. I co-edited the paper with the other authors.

Publication VI: “Searching for Content in Mobile DTNs”

The ideas behind the paper were jointly developed with Mikko Pitkänen, Janico Greifenberg, and Jörg Ott. I developed the simulation model. The simulation evaluation was designed and carried out jointly with Mikko Pitkänen. I co-edited the paper with the other authors.

Publication VII: “Shared Content Editing in Opportunistic Networks”

I formulated the ideas behind the work, developed the simulation models, and analyzed the results. I co-edited the paper with Jörg Ott.

Publication VIII: “Web-based Framework for Accessing Native Opportunistic Networking Applications”

The ideas behind this paper were jointly formulated with Marcin Nagy and Jörg Ott. I integrated the base system developed by Marcin Nagy into the Liberouter platform. I developed the original native version of the Android applications, except Here and Now, which was jointly developed with Paul Houghton. I designed and implemented the testbed evaluation
of the system and co-edited the paper with the other authors.

Publication IX: “Composable Distributed Mobile Applications and Services in Opportunistic Networks”

I formulated the ideas behind the work and refined them with Chrysa Papadaki. I guided the platform design, development and evaluation, with most of the practical work being carried out by Chrysa Papadaki. I co-edited the paper with Jörg Ott.
## List of Figures

1.1 Dissertation Structure ........................................... 28

2.1 Opportunistic Networking Overview ........................... 32

2.2 Evolving Graph .................................................. 40

2.3 PodNet Design ................................................... 48

2.4 Haggle Design ................................................... 49

2.5 DTN Architecture ................................................ 51

2.6 The ONE System Design ......................................... 55

2.7 The ONE Simulation Example Results ......................... 63

2.8 Scampi System Architecture .................................... 66

2.9 Scampi Communication and Storage .......................... 68

2.10 Scampi Routing Subsystem .................................... 70

2.11 Scampi API ..................................................... 73

3.1 Cross-Traffic in Hotspots ....................................... 80

3.2 Measurement Study Area ........................................ 84

3.3 Measurement System ............................................ 85

3.4 LEDBAT for Cross-Traffic ....................................... 90

3.5 Dense Network Scenario ....................................... 94

3.6 Dense Network Testbed ........................................ 95

3.7 Topologies of Dense Networks ................................. 97

3.8 Dissemination Results in Dense Topologies ................ 98

3.9 Divide-and-Conquer Concept .................................. 101

3.10 Divide-and-Conquer Experiment Results .................... 103

3.11 Divide-and-Conquer Simulation Results ..................... 105

3.12 Liberouter System .............................................. 108

3.13 Liberouter Hardware ........................................... 110

3.14 Basic Liberouter Apps ......................................... 112

3.15 Here and Now App .............................................. 113
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.16</td>
<td>Liberouter Simulation Results</td>
<td>114</td>
</tr>
<tr>
<td>4.1</td>
<td>Web Access System Model</td>
<td>123</td>
</tr>
<tr>
<td>4.2</td>
<td>Web Access Response Probability</td>
<td>130</td>
</tr>
<tr>
<td>4.3</td>
<td>Web Access Latency and Hop Counts</td>
<td>131</td>
</tr>
<tr>
<td>4.4</td>
<td>Web Access Implementation Overview</td>
<td>133</td>
</tr>
<tr>
<td>4.5</td>
<td>Client GUIs for Web Access</td>
<td>135</td>
</tr>
<tr>
<td>4.6</td>
<td>Search System Concept</td>
<td>139</td>
</tr>
<tr>
<td>4.7</td>
<td>Query Processing Model</td>
<td>140</td>
</tr>
<tr>
<td>4.8</td>
<td>Query Response Probabilities</td>
<td>148</td>
</tr>
<tr>
<td>4.9</td>
<td>Query Response Counts</td>
<td>150</td>
</tr>
<tr>
<td>4.10</td>
<td>Query Response Latencies</td>
<td>152</td>
</tr>
<tr>
<td>4.11</td>
<td>Mutable Content Concept</td>
<td>154</td>
</tr>
<tr>
<td>4.12</td>
<td>Branching and Combining of Mutable Content</td>
<td>156</td>
</tr>
<tr>
<td>4.13</td>
<td>Adoption Strategy for Mutable Content</td>
<td>161</td>
</tr>
<tr>
<td>4.14</td>
<td>Merging Strategy for Mutable Content</td>
<td>163</td>
</tr>
<tr>
<td>4.15</td>
<td>Hybrid Strategy for Mutable Content</td>
<td>165</td>
</tr>
<tr>
<td>5.1</td>
<td>Web App System</td>
<td>174</td>
</tr>
<tr>
<td>5.2</td>
<td>Web App Application Model</td>
<td>175</td>
</tr>
<tr>
<td>5.3</td>
<td>Web App Aggregate-Detail Model</td>
<td>177</td>
</tr>
<tr>
<td>5.4</td>
<td>Web App Example Application</td>
<td>178</td>
</tr>
<tr>
<td>5.5</td>
<td>Framework Design for Web Apps</td>
<td>179</td>
</tr>
<tr>
<td>5.6</td>
<td>Example Web Apps</td>
<td>182</td>
</tr>
<tr>
<td>5.7</td>
<td>Testbed Design for Web App Evaluation</td>
<td>183</td>
</tr>
<tr>
<td>5.8</td>
<td>Web App Overhead Simulation Results</td>
<td>184</td>
</tr>
<tr>
<td>5.9</td>
<td>Web App Coverage Simulation Results</td>
<td>186</td>
</tr>
<tr>
<td>5.10</td>
<td>Web App Testbed Results</td>
<td>187</td>
</tr>
<tr>
<td>5.11</td>
<td>Composable Service Concept</td>
<td>192</td>
</tr>
<tr>
<td>5.12</td>
<td>Composable Service Framework</td>
<td>196</td>
</tr>
<tr>
<td>5.13</td>
<td>Composable Service Framework Detail</td>
<td>197</td>
</tr>
<tr>
<td>5.14</td>
<td>Composable Service Component Detail</td>
<td>197</td>
</tr>
<tr>
<td>5.15</td>
<td>Polling App Concept</td>
<td>199</td>
</tr>
<tr>
<td>5.16</td>
<td>Polling App Detail</td>
<td>200</td>
</tr>
<tr>
<td>5.17</td>
<td>Polling App GUI</td>
<td>201</td>
</tr>
<tr>
<td>5.18</td>
<td>Composable Service Evaluation Sequence</td>
<td>203</td>
</tr>
<tr>
<td>5.19</td>
<td>Composable Service Evaluation Results</td>
<td>206</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Results of the access point cross traffic measurements. . . 87
List of Abbreviations

**AD-SIMPLE**  Adaptive Detection SIMPLE

**AP**  Access Point

**API**  Application Programming Interface

**APK**  Android Package

**ARP**  Address Resolution Protocol

**AWS**  Amazon Web Services

**BS**  Base Station

**BSS**  Basic Service Set

**CC**  Creative Commons

**CDF**  Cumulative Distribution Function

**CI**  Confidence Interval

**CPU**  Central Processing Unit

**CRDT**  Conflict-free/Commutative Replicated Datatype

**CSS**  Cascading Style Sheets

**CT**  Contact Time

**DAG**  Directed Acyclic Graph

**DD**  Direct Delivery

**DHCP**  Dynamic Host Configuration Protocol

**DNS**  Domain Name Service

**DTN**  Delay-Tolerant Networking
List of Abbreviations

**DTNRG** Delay-Tolerant Networking Research Group

**DTNWG** Delay-Tolerant Networking Working Group

**EB** Exabyte

**EID** Endpoint Identifier

**EJS** Embedded JavaScript

**ESS** Extended Service Set

**FC** First Contact

**GB** Gigabyte

**GSM** Global System for Mobile Communications

**GUI** Graphical User Interface

**HCS** Helsinki City Scenario

**HTML** Hypertext Markup Language

**HTTP** Hypertext Transport Protocol

**HTTPS** Hypertext Transport Protocol Secure

**IBR** Institut für Betriebssysteme und Rechnerverbund

**ICE** Interactive Connectivity Establishment

**ICN** Information-Centric Networking

**ICT** Inter-Contact Time

**IETF** Internet Engineering Task Force

**ION** Interplanetary Overlay Network

**IP** Internet Protocol

**IPN** Interplanetary Networking

**IPND** IP Neighbor Discovery

**IRTF** Internet Engineering Task Force

**ISM** Industrial, Scientific and Medical

**ISP** Internet Service Provider
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAR</td>
<td>Java Archive</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>KAIST</td>
<td>Korea Advanced Institute of Science and Technology</td>
</tr>
<tr>
<td>KB</td>
<td>Kilobyte</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LCA</td>
<td>Lowest Common Ancestor</td>
</tr>
<tr>
<td>LED</td>
<td>Light Emitting Diode</td>
</tr>
<tr>
<td>LEDBAT</td>
<td>Low Extra Delay Background Transport</td>
</tr>
<tr>
<td>LTE</td>
<td>Long-Term Evolution</td>
</tr>
<tr>
<td>MAC</td>
<td>Medium Access Control</td>
</tr>
<tr>
<td>MANET</td>
<td>Mobile Ad-hoc Network</td>
</tr>
<tr>
<td>MB</td>
<td>Megabyte</td>
</tr>
<tr>
<td>MHTML</td>
<td>MIME encapsulation of aggregate HTML documents</td>
</tr>
<tr>
<td>MIME</td>
<td>Multipurpose Internet Mail Extensions</td>
</tr>
<tr>
<td>MRD</td>
<td>Modified Random Direction</td>
</tr>
<tr>
<td>NAN</td>
<td>Neighborhood Awareness Networking</td>
</tr>
<tr>
<td>NAPTR</td>
<td>Name Authority Pointer</td>
</tr>
<tr>
<td>NASA</td>
<td>National Aeronautics and Space Administration</td>
</tr>
<tr>
<td>NTP</td>
<td>Network Time Protocol</td>
</tr>
<tr>
<td>ONE</td>
<td>Opportunistic Networking Environment</td>
</tr>
<tr>
<td>PARC</td>
<td>Palo Alto Research Center</td>
</tr>
<tr>
<td>PDF</td>
<td>Portable Document Format</td>
</tr>
<tr>
<td>PFIF</td>
<td>People Finder Interchange Format</td>
</tr>
<tr>
<td>PMS</td>
<td>Poll Management Service</td>
</tr>
<tr>
<td>PPS</td>
<td>Poll Participant Service</td>
</tr>
</tbody>
</table>
List of Abbreviations

PPW  Poll Participant Widget
RAM  Random Access Memory
REST  Representational State Transfer
RFC  Request For Comments
RGB  Red, Green, Blue
RPC  Remote Procedure Call
RSA  Rivest–Shamir–Adleman
RTT  Round-Trip Time
RWP  Random Waypoint
SA  ShareAlike
SCAMPI  Service platform for social Aware Mobile and Pervasive computing
SD  Secure Digital
SGBR  Social Groups Based Routing
SLAW  Self-similar Least Action Walk
SMS  Short Message Service
SQL  Structured Query Language
SSID  Service Set Identifier
SUMO  Structured Query Language
TCP  Transmission Control Protocol
TCPCL  TCP Convergence Layer
TLS  Transport Layer Security
TTL  Time to Live
UDP  User Datagram Protocol
UI  User Interface
UML  Unified Modeling Language
URI  Uniform Resource Identifier
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>UX</td>
<td>User Experience</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>WDM</td>
<td>Working Day Movement</td>
</tr>
<tr>
<td>WLAN</td>
<td>Wireless LAN</td>
</tr>
<tr>
<td>WPA</td>
<td>Wi-Fi Protected Access</td>
</tr>
</tbody>
</table>
1. Introduction

As you are reading this thesis, there is likely a smartphone nearby competing for your attention with its many online apps and services. There are in fact over five billion of them all over the world [6]. Coupled with fast Wi-Fi or cellular connectivity to the Internet, they are our primary gateways to information, entertainment, social networks, and private relationships. These smart devices have become integral parts of our lives.

However, even in many parts of the densely populated Central Europe, a fifteen-minute journey out of a major city onboard a train is often enough to break or severely degrade connectivity. The same occurs in many rural or remote areas that are not economical to cover with expensive infrastructure networks. And even where infrastructure does exist, it may become overloaded by a crowd or disabled by a disaster. Around the edges of our well-connect world are many fringes and pockets out of coverage—and without fast Internet connectivity, our devices become decidedly less smart.

It does not, however, have to be like this. Inside every modern smartphone there are multiple radios capable of transmitting tens or hundreds of megabits per second to other nearby smartphones. Even when we cannot reach the Internet, we can often still reach each other. Our smartphones, often filled with content, are fully capable of exchanging information and entertainment with those around us, without reaching out to the Internet.

We can take this concept even further if we collaborate with each other: I might not be interested in your photo or movie, but I might store a copy anyway in case I later meet someone else who is interested. Content and messages could travel between people like a baton in a relay race. As Milgram’s famous experiment indicates [184], this approach might have an unexpectedly wide reach—the world is small and strangers are often linked by a surprisingly short chain of acquaintances.

If we were to construct such a network, it might make sense to use it
even in the presence of high speed Internet connectivity. By exchanging messages directly with each other, we do not have to relinquish control and privacy to a centralized platform. Further, due to locality, the content that we discover around us may be more relevant to us than the content on global platforms. Local exchanges also have no monetary cost, and can help to save the capacity of capped cellular plans.

This style of networking is studied in the field of *Opportunistic Networking* [206, 79] and *Delay-Tolerant Networking* [102, 62]. The fundamental idea is to use periodic peer-wise communication opportunities, rather than fixed end-to-end infrastructure, for networking. It is based on a set of concepts that is distinct from classical packet-based networking: *mobility* of the nodes, wireless *device-to-device* communications, and *store-carry-forward* message passing. The resulting networks can transmit messages in environments where classical networking is not feasible, but with a radically different type of service. For example, while classical networks typically have end-to-end delays in the order of 1–100 milliseconds, in opportunistic networks it may be hours or days. This means that applications must be designed specifically to take advantage of opportunistic networking, both in terms of protocol designs and user interfaces.

Nevertheless, opportunistic networking has found many fields of application. The initial driver was space networking [34], which requires connecting nodes that are only periodically within reach of each other and where transmission latencies are in the order tens of minutes to hours. From there, the concepts have spread to disaster recovery [181], providing networking for remote communities [90], underground mines [112], rural smart farms [111], wildlife tracking in remote areas [147], and military networks [179], among others. In this thesis we focus on scenarios where the opportunistic network is composed of smartphones in every-day environments—sometimes called *Pocket Switched Networking* [133].

The goal of this thesis is to make opportunistic networking more available and more capable, particularly for networks created from interconnected smartphones. In particular, we aim to make opportunistic networking more available by advancing the use of open Wi-Fi infrastructure to generate opportunistic contacts, and by introducing an alternative lightweight opportunistic *Liberouter* infrastructure. We aim to make it more capable by designing advanced content and computational mechanisms on top of the basic message transport. This includes fetching content from outside
the opportunistic network, searching for content, and shared editing of content within the opportunistic network. Further, we design mechanisms for opportunistic computing by attaching executable code to messages, and by creating a runtime for opportunistic composition of distributed services.

Taken together, the set of technologies developed in this thesis all contribute to the building of a foundation for future smartphone-based opportunistic networking applications. The contributions are, of course, still at the level of ideas and initial feasibility studies, but they do indicate directions for more work on the path towards production systems in the future. The hope is that one day smart devices can benefit from both infrastructure-based and opportunistic networking technologies to provide a more diverse set of capabilities to their users. This, of course, will take more than just technical solutions to bring about, but this work provides some steps towards solving the technological problems.

1.1 Contributions

The contributions of this thesis are divided into three areas—connectivity, content, and computation—each with its own set of research questions. We start with advancing opportunistic connectivity, by asking the following research questions:

• **To what extent can we leverage existing open Wi-Fi infrastructure to create opportunistic contacts between co-located mobile devices?**
  We answer this by conducting a measurement study to survey the support for opportunistic communication in existing public Wi-Fi access points.

• **How should we choose which contacts to create in a dense environment with a large number of potential communication partners?**
  To answer this we study the behavior of different topology control algorithms in a dense Wi-Fi testbed.

• **Can we move opportunistic traffic away from the infrastructure access points so that it does not interfere with subscriber traffic?**
  We answer this affirmatively by designing a mechanism for moving opportunistic interactions to software access points created automatically by the smartphones.
• **Can the use of dedicated lightweight infrastructure components offer a practical basis for opportunistic networking?**

We demonstrate that this is possible by designing and implementing the *Liberouter* neighborhood networking system, which combines cheap single-board computers with an opportunistic networking middleware.

Next, we focus on content-based mechanisms and ask the following research questions:

• **Can opportunistic networking be used to satisfy request/response interactions from mobile devices towards infrastructure origin servers?**

We answer this by designing and evaluating a system for accessing Internet-based content, websites in particular, from the opportunistic network via dedicated bridge nodes.

• **Can the content available in the opportunistic network be made searchable via a query/response mechanism?**

To answer this, we design and evaluate a system for search query dissemination and response forwarding for opportunistic networks.

• **How can shared content editing be supported in opportunistic networks?**

We design a system based on applying distributed version control concepts to opportunistic networks, and evaluating it.

Finally, we contribute to opportunistic computing by asking and answering the following research questions:

• **Can we support interactive web applications in opportunistic networks?**

For this, we design a web app framework for the Liberouters, which can dynamically synthesize backends to serve interactive web apps to connected web browsers.

• **Can we provide interoperability between native and web browser-based opportunistic networking applications?**

The web app framework design works by attaching computations to the same content messages used both by the native apps and the opportunistic web app framework, which achieves interoperability.
**Introduction**

- Can an opportunistic network be used as a dynamic execution environment for distributed systems?

We answer this by designing, implementing and evaluating a distributed service composition runtime for Liberouters and Android phones.

In addition to the above, we contribute tools for simulating (The ONE) and building (Scampi) opportunistic networks, which are needed by our research methodology, and have also been used by other researchers.

### 1.2 Methodology

Opportunistic networking in general, and the type of research questions we pose in particular, involve studying systems whose behavior arises from complex interactions between many mechanisms and phenomena. This often makes it infeasible to meaningfully isolate the specific mechanisms that we study from rest of the system. For example, the way search queries behave—one of our research questions—is intrinsically linked to the way the underlying routing distributes the messages, which is linked to the underlying contact patterns, which depends on the underlying mobility and connectivity. And, in the case of prototypes, everything depends on the behaviors of the real hardware and software stacks. We therefore propose to study these systems as a whole, rather than parts in isolation.

Our work, therefore, follows the **systems methodology**—also called the **holistic** approach—which considers systems as a whole. This approach can capture the complexities of the relationships between the different parts of the system, and their resulting emergent properties, but the results tend to be descriptive and qualitative. The main tools of this methodology are measurement studies, simulation models, and prototyping. The systems methodology has been developed relatively recently, in the 1950s, by Bertalanffy, Buckley, Churchman, and Emery [38].

We apply a **constructive research method**—commonly referred to as the **design** paradigm [242, 88, 180] in the field of computer science. The constructive method, as applied in our work, consists of three parts: 1) **building** a construct, 2) **evaluating and analyzing** the construct, and 3) **theorizing** based on the evaluation. Here our constructs are prototypes of the systems under study—a natural consequence of the systems approach, which requires an instance of a whole system to study.

In this thesis, we build both virtual (simulated) prototypes (see sec-
Introduction

The building phase consists of three main activities: 1) studying the literature to understand the current state of the art, 2) designing the system or mechanism under study, and 3) implementing the prototype. These prototypes are then evaluated analytically against predefined criteria or benchmarked in various testbed environments or simulation scenarios in order to understand their behavior. The final step of the method is to synthesize new knowledge by theorizing. In other words, making and justifying more general claims based on a set of specific observations generated by the evaluation—i.e, scientific induction. In practice, these activities feed back to each other and are iterated multiple times until the work converges at a result.

1.3 Structure of the Thesis

This thesis contains four main parts, which build on and complement each other as shown in fig. 1.1. We progress bottom-up, from the more general to the more specific topics. The starting point is concepts and tools, which defines the basic concepts of opportunistic networking and develops the tools required by our research methodology. We follow this by creating connectivity between co-located devices, which is the fundamental enabler for opportunistic networking. Next, we build more advanced mechanisms on top the basic connectivity for both content and data, and computations. Taken together, these mechanisms create a foundation for building advanced opportunistic networking applications and services.

Chapter 2, Concepts and Tools, starts by defining the basic concepts underlying opportunistic networking and provides a review of the related
literature for each concept. It then builds the tools needed to do research on these concepts. The first tool is a simulator, *The ONE*, which allows us to build simulation models that integrate all the main opportunistic networking concepts in a single simulation—this is also the topic of Publication I. The second tool is a networking middleware, *Scampi*, which allows us to build prototypes that run on real devices.

Chapter 3, *Creating Connectivity*, focuses on opportunistic contacts, which are the foundation of opportunistic networking. First, in Publication II, we conduct a measurement study to understand whether existing open Wi-Fi access points can be leveraged for creating opportunistic contacts. As these open access points can be popular, we then develop mechanisms for optimizing contact creation in the face of large numbers of users in Publication III. Finally, in Publication IV, we propose lightweight opportunistic infrastructure, *Liberouters*, as an alternative to existing infrastructure.

Chapter 4, *Content and Data*, builds more advanced mechanisms for accessing and working with content in opportunistic networks. First, in Publication V, we consider a request/response mechanism for fetching specific content, in particular websites, that reside in origin servers outside the opportunistic network. Then, in Publication VI, we consider a query/response mechanism to search for content in the caches of nearby devices. Finally, in Publication VII, we go beyond immutable content, and consider shared content editing in opportunistic networks.

Chapter 5, *Computation*, focuses on the second part of distributed systems beyond data—the computations that operate on that data. Publication VIII extends the work on opportunistic web access to also support interactive web apps by transmitting executable code within the opportunistic network. Then, in Publication IX, we generalize these ideas to an opportunistic execution environment for dynamic distributed services.

Finally, we provide some concluding thoughts, lessons learned, and future perspectives in Chapter 6.
Introduction
2. Concepts of opportunistic networking

Opportunistic networks are based on the concept of opportunistic contacts, which are communication opportunities that occur when two nodes physically move into wireless communication range with each other. These pair-wise contacts are often short and infrequent and, therefore, the resulting network topology may never be fully connected. This in turn means that the classical packet-switched networking underlying the success of the Internet is not applicable as it relies on end-to-end paths that do not exist in opportunistic networks.

Enabling this type of networking is a set of concepts that is distinct from classical networking—mobility of the nodes, wireless device-to-device communication, spacetime paths over time-variant topologies, and store-carry-forward routing and dissemination algorithms. To understand the work in this thesis, we must first understand these concepts and their related research. Further, as our methodology relies on simulations and prototypes, we must also implement these concepts in a simulator and in a networking middleware. We cover these three aspects in this chapter.

We start by examining the basic concepts underlying opportunistic networking and provide an overview of the related research done on each concept. This includes the movement of the nodes, with a particular focus on human mobility, which is the main scenario for the systems in this thesis. We also cover the most relevant wireless device-to-device technologies that can be used to interconnect the devices carried by people, in particular smartphones. Together, mobility and wireless communication generate a time-variant network topology, which we briefly define formally. Various routing and dissemination algorithms are then needed to enable the end-to-end messaging in these topologies. Finally, we discuss the implementation of these concepts in real-world platforms and middlewares.

Then, in Publication I, we examine how to build a simulator and
simulation models for these concepts. This simulator, The Opportunistic Networking Environment (The ONE), integrates mobility, device-to-device connectivity, and store-carry-forward messaging in a single simulation environment, which makes it an ideal tool for the work in the following chapters of this thesis. We also build another tool, an opportunistic networking implementation called Scampi, which implements these concepts in a real middleware. Scampi is then used to build prototypes of the systems that we design in the remaining chapters.

2.1 Basic Concepts

One way to illustrate the ideas behind opportunistic networking is to compare it to the more familiar Internet-style packet-based networking (which we call “classical networking”), as shown in fig. 2.1. Classical networking is based on dedicated routers connected by stable links. In packet switching [44, 221], these links feed small blocks of data prefixed by a destination address into the router. The router consults a routing table to find the outgoing link for the incoming packet based on the destination address. Since the outgoing link may be occupied at the time instance when the router is processing an incoming packet, the packets are briefly queued until they can be transmitted. As the routers store the packets in queues before forwarding them on links, this type of networking is sometimes referred to as store-and-forward networking [44]. By organizing the routers in hierarchies (as shown in the figure), we can create compact routing tables that allow a packet from any source to be efficiently forwarded.
to any destination in very large networks [155]. From a higher layer
perspective, the network provides stable low-latency end-to-end paths
between communicating processes. These paths enable tight control loops
between the end nodes of the network, which in turn has enabled the
communication state to be held only in the endpoints and not in the
network—known as the end-to-end principle [227] and fate sharing [77].

Classical networking has been exceptionally successful in connecting
devices all over the world. It does, however, have limits to its applica-
ability. It is relatively easy to violate the implicit assumptions about the
path latency and to break protocols relying on classical networking: The
Transmission Control Protocol (TCP) [214] has a delay of 1.5 round-trips
(RTT) before any data can flow, relies on continuous feedback for flow and
congestion control, and implementations often have a two-minute fixed
timeout [106]. This means that, for example, trying to use standard TCP
for space networking scenarios quickly leads to failure [106]—already just
trying to run TCP over satellite links requires special mechanisms [202].
Beyond large latencies, TCP can also be broken if the packet paths are not
stable, e.g., in ad hoc networks [127].

While these problems with classical networking can to an extent be
patched over by tuning the protocols with different assumptions, it becomes
completely infeasible in scenarios that physically do not have end-to-end
paths. Such scenarios occur in deep space networking where nodes may,
e.g., be blocked by planetary bodies [34, 106], in disaster scenarios where
network infrastructure is inoperable [181], in remote communities that are
not covered by infrastructure [90], in underground mines [112], in rural
smart farms [111], and in wildlife tracking in remote areas [147], to name
a few. It seems apparent, that another model of networking is needed to
support these scenarios.

While it may never be possible to find the stable end-to-end paths re-
quired by classical networking in these scenarios, there might still be
wireless single-hop communication opportunities created by nodes moving
near each other. We could use these opportunities to forward data closer to
the destination step-by-step, similar to a relay race, as shown on the right
of fig. 2.1. Networking based on this idea is called opportunistic network-
ing [206], which generalizes the concepts from delay-tolerant networking
(DTN) [63]—these terms are often used interchangeably, but in this thesis
we use “opportunistic networking” to refer to the general concept of net-
working using opportunistic contacts, and “DTN” to refer to the specific set
of architectures, protocols, and specifications from the IRTF DTNRG [18] and IETF DTNWG [10]. Since this approach involves nodes physically carrying messages stored in persistent caches until an opportunity to forward them to another node arises, this mode of networking is often called store-carry-forward networking [277].

To understand and model opportunistic networks, we need to understand a set of key concepts: First, the underlying enabler for these types of networks is the mobility that brings different nodes near each other. Second, some form of wireless device-to-device communication is needed to create the opportunistic contacts between nearby devices. The system created by such opportunistic contacts can be modeled as a dynamic graph, over which we must find spacetime paths that can move data between senders and receivers. Many concrete routing and dissemination algorithms have been proposed to find these spacetime paths in various different scenarios. Finally, we need platforms and applications, to make such networks practically useful. We first explore these five key concepts, and then describe the tools that we have created to simulate and build them.

2.1.1 Human Mobility

Opportunistic networking relies on node movement to physically carry messages and to come in communication range with other nodes. The characteristics of this movement, therefore, ultimately determine how effective opportunistic networking can be—without mobility, messages cannot travel at all, while a high degree of mobility can distribute them widely. In fact, movement patterns can be the determining factor for the performance of networked systems that connect mobile devices [41, 162].

In this thesis our focus is on opportunistic networks based on human mobility, for which there are two important lines of existing work that concern us: 1) The collection and analysis of traces, with the aim of understanding the statistical properties of human mobility. 2) The design of mobility models, particularly those that can create synthetic mobility of varying levels of realism useful in simulations.

Trace analysis: Perhaps the earliest documented formal attempt to use data to understand human mobility is based on the 1881 census of the United Kingdom, which derives laws for population migrations within the country [217]. More recently, in 2006 Brockmann et al. [59] use banknote tracking data from the United States as a proxy for human movement, and find that the probability density function (PDF) of travel
Concepts of opportunistic networking
distance follows a power law and note that it is reminiscent of, but not
equivalent to, Lévy flights [239]. The proliferation of cellular phones has
created a source of detailed individual mobility traces, which Gonzalez et
al. [113] use for more detailed analysis and find that the travel distance
in fact follows an exponentially truncated power law. They further find
heterogeneity in the individual trajectories—most people tend to make
short trips, while some make very long trips. This can be measured by a
radius of gyration, which is a metric for how far from the center of mass
of the trajectory the person tends to move. When scaled by this measure,
the heterogeneous travel distance PDF collapses into a single probability
distribution. The same paper finds a strong tendency for people to return
to frequently visited locations—people are found in their two most frequent
locations 40% of the time, and their return time probability distribution has
prominent peaks corresponding to the 24-hour day cycle. Song et al. [246]
extend this analysis to show that the waiting time distribution that people
spend in a single location follows a truncated power law with a cutoff
consistent with the human awake times, and that the frequency of visiting
locations follows Zipf’s law. They also find that human movements have
93% predictability when using temporal data (historically visited locations).
Numerous other authors also use traces to characterize various aspect of
human mobility [139, 86, 177, 45] and vehicular mobility [280, 170, 274].

While understanding movement of individuals is important, the real
enabler for opportunistic networking is the occurrence of meetings between
the nodes—i.e., how frequently the movement brings a pair of nodes within
communication range (inter-contact time, ICT) and how long they stay near
each other (contact time, CT), and how often contacts occur in general
(inter-any-contact time). In an early direct attempt to characterize the
meetings, in 2005 Hui et al. [134] equip conference visitors with small
Bluetooth devices and track the contacts between them. They find that
the ICT has a heavy-tailed distribution that can be approximated with
coefficient 0.4 within a time range of 2 minutes to one day, and that the
CT distribution can also be approximated with a power law. Chaintreau et
al. [71] expand on this by analyzing eight different datasets from Bluetooth,
Wi-Fi, and GSM traces. Their study confirms that in all cases the ICT
can be approximated by a power law with a coefficient less than one
in the range of ten minutes to one day. In a later study, partially on
the same traces, Karagiannis et al. [148] find that ICT in fact follows a
power law distribution, but only for up to half a day, after which there
is an exponential decay. These findings of heavy tailed ICT and CT are important, because it has been shown that under such conditions naive opportunistic forwarding has infinite latencies and context-aware routing approaches are needed [206] (we return to routing in section 2.1.4).

While the pairwise ICT and CT metrics characterize contacts on a micro level, work has also been done to understand macro level structures of the contacts—e.g., the tendency of humans to form communities with frequent contacts within them. Hossmann et al. [128, 129] have done detailed analysis of these structures particularly in the context of opportunistic networking. They construct a contact graph where the vertices are mobile devices and the edge weights express the frequency or duration of pairwise contacts, and apply various metrics on the resulting graph to quantify its structural properties. Based on the analysis on a number of traces, they confirm the existence of the small-world property in the contact graph—i.e., the existence of short paths between the node pairs. They further find that the contact graph is highly modular, indicating the existence of communities (or clusters), but also that these communities are not homogeneous entities as there is a high variance in tie strengths within and between the communities. Their analysis shows important structural characteristics for these communities: 1) contacts within a community tend to happen within a small set of physical locations (“home locations”), 2) bridging links between communities occur when two nodes meet outside their home locations, 3) bridging nodes occur when a node visits the home location of another community. Many advanced routing and dissemination algorithms are based on finding and leveraging these structures (see section 2.1.4).

Mobility models: Possibly the earliest documented human mobility model is due to Karl Pearson in 1905, who in a letter to the journal Nature asks how far would a man be expected to travel when starting from a point \(O\) and repeatedly moving a distance \(l\) in random directions [203]. Lord Rayleigh answers with an expression for the probability distribution function known as the Rayleigh distribution \(P(r) \approx \frac{2r}{n}\ e^{-r^2/n}\), for very large \(n\), where \(r\) is the distance and \(n\) the number of steps) [218]. This is known as the Random Walk model (also called the “drunken man” model by Pearson [204]), one example of which is Brownian motion described by Einstein also in 1905 [94]. An important feature of Random Walk model (in one or two dimensions) is that it returns to origin with probability one [213], i.e., an agent following a Random Walk will not “escape”. The model can be modified by changing the probability distributions from which
the directions, distances or speeds are drawn—e.g., a constant speed with step distance picked from the heavy tailed Lévy distribution results in Lévy flights, which are believed to closely model animal foraging [262]. Random Walk mobility models are still in use in many fields, including in our simulator in Publication I—although mainly for completeness, since other models are preferred in opportunistic networking research.

One of the most frequently used mobility models in networking simulations is the Random Waypoint model [146, 58], in which individual nodes pick a random destination within the simulation area, move there with a random speed, and then pause for a random time period until repeating the process. This model, however, has two significant problems: 1) The stationary node density distribution is not constant [48], and instead is significantly denser in the middle of the simulation area. 2) The average speed of nodes continuously decreases and does not reach a steady state [271]. The density problem can be fixed by the Random Direction model [222] where the nodes pick a direction instead of a waypoint and move in that direction (either until the edge, or a random distance), but it does not fix the decaying speed problem. Fortunately, the speed decay can be solved by setting a non-zero minimum speed for the nodes [271].

An important theme in mobility modeling research has been to increase “realism” of the entity models by introducing more constraints and complexity to capture the human mobility characteristics more accurately. For example, the move distances can be explicitly made to follow the truncated power laws, and entities given heterogeneous radii of gyration, as found in the human mobility traces [165]. A popular way to constrain mobility is to allow the entities to only move on predefined paths, which may be regular patterns such as the Manhattan grid [182], or randomly generated to model human movement around obstacles [145], or real maps [224]—the simulator in Publication I contains the downtown Helsinki map, which we use as a constraint in many of the simulation studies later in this thesis. Another popular mechanism is to introduce location preferences. For example, assigning nodes to home areas that they are more likely to appear in can recreate the human tendency to spend most of their time in a few locations [249]. This can be combined with time-dependence to capture the temporal characteristics of human mobility such as day cycles [132], or with social relationships to make locations with more “friends” more attractive [51]. Finally, a number of models increase reality by giving the entities realistic agendas to follow. For example, the entities may follow
a typical working day agenda, moving between home, office, and leisure activities [95], or they may be visitors to an amusement park [263].

These are the key concepts directly relevant to our work on this thesis, but mobility modeling is a mature field of study and many excellent surveys exist both for traces [39] and mobility models [65, 188, 149, 257, 211, 124].

2.1.2 Wireless Device-to-Device Communication

The next key concept in opportunistic networking is the creation of wireless communication links between devices—known as opportunistic contacts. Three sets of key technologies have been proposed for directly connecting the devices that we target (phones, consumer devices): The 802.11 Wi-Fi suite of specifications, Bluetooth, and cellular device-to-device technologies.

**Wi-Fi:** From early on, Wi-Fi included the 802.11 ad hoc mode, designed for enabling direct device-to-device communications. While the ad hoc mode has been extensively researched with real devices [36, 35], it has not been widely deployed. Attempts to use it for opportunistic networking have also not been successful for various reason—e.g., Helgason et al. [123] report high power use and the need to root the device with Android.

While the 802.11 ad hoc mode is not well-supported in mobile devices, the *infrastructure* mode support often includes the ability for mobile nodes to create software Wi-Fi access points—this is to support Wi-Fi tethering, i.e., allowing the mobile device to share its cellular Internet connection over the Wi-Fi. Wirtz et al. [266] propose to replace the ad hoc mode with this tethering capability for opportunistic networking. The major problem with this approach is the high energy use in access point mode, which has prompted multiple researchers to propose energy-use optimizations [120, 259, 216]. Another problem with creating, and connecting to, software access points in the end user devices is that it prevents the interface from being used for normal Internet access. This, however, can be solved by multiplexing multiple virtual interfaces over one physical interface as suggested by Chandra et al. [72] and expanded upon by Nicholson et al. [191]—today most new Wi-Fi hardware has this capability.

Trifunovic et al. [258] extend the concept of software Wi-Fi access points in mobile devices, and design a system called *WLAN-Opp* to automate the creation of opportunistic contacts. They provide a randomized algorithm for nodes to choose when to create, join, or switch software access points. The approach has a number of attractive properties, including the ability to adapt to different node densities by changing the transition probabilities,
and fairness by rotating the power-hungry access point duties between the devices. Further, the algorithm can be implemented on a standard Android system without rooting the device.

Wi-Fi Direct [31] is a specification designed to connect Wi-Fi devices together without joining an infrastructure access point. In Wi-Fi Direct, nodes that want to establish connectivity create a group with a group owner. The owner acts as a normal Wi-Fi software access point and rest of the group members join it as stations—hence there is no new hardware support required. The new mechanisms in Wi-Fi Direct automate the group discovery, formation and maintenance, and are implemented fully in software. Camps-Mur et al. [66] perform simple experiments with Wi-Fi Direct and find that the group formation process takes only a few seconds. Conti et al. [78] evaluate Wi-Fi Direct specifically in the context of opportunistic networking, and similarly find that the group formation with small number of devices takes only a few seconds in most cases, but also find a significant number of cases where the latency is tens of seconds—when adding four or more nodes, the group formation delays grow to minutes in their experiments. They also note that the requirement for manual user intervention in group creation (push button security) is a practical obstacle for using Wi-Fi Direct for opportunistic networking.

The latest device-to-device specification from the Wi-Fi Alliance is Wi-Fi Aware [30], also known as Neighborhood Awareness Networking (NAN). It provides energy efficient continuous service and peer discovery via a new beaconing mechanism, but relies on Wi-Fi Direct (or some other external channel) for the actual data transmission [67]. Apple devices use a proprietary 802.11 based networking technology, which is believed to function similarly to Wi-Fi Aware but with a very different implementation [253].

**Bluetooth:** Perhaps the oldest and most widely deployed technology for device-to-device communications is Bluetooth [3]. It is a frequency hopping spread spectrum technology with a star topology, primarily intended to act as a cable replacement—e.g., the mobile would act as the hub and earphones and other peripherals connect to it. Pietiläinen et al. [209, 210] experiment with using Bluetooth for creating opportunistic networking contacts in three conference venues. They conclude that although Bluetooth has the attractive properties of low power consumption, short radio range, and wide availability, it is ultimately not well suited for opportunistic contact creation due to low success rates and poor scalability to dense environments—this is due to its design as a wire replacement, not as a
technology for creating opportunistic contacts. Other researchers have proposed various mechanisms to improve Bluetooth to make it more suitable for opportunistic communications [40, 92, 81]. Despite these efforts, Bluetooth has not become widely used for opportunistic networking.

Cellular device-to-device: Cellular technologies have historically been designed exclusively to connect mobile devices to a core network and, therefore, have not been relevant to opportunistic networking. Recently, however, both research and standardization work has been done on device-to-device communications in cellular networks [176, 178, 144]. The main benefit of these technologies is that they can use licensed spectrum rather than the shared unlicensed spectrum. The drawback is that the device-to-device communications are likely to be controlled by a base station and therefore require operator support instead of operating autonomously like Wi-Fi and Bluetooth. As these technologies are not yet widely available or deployed, we do not discuss them further—they may, however, provide new opportunities in the future if they become widely available.

2.1.3 Store-Carry-Forward over Spacetime Paths

The result of connecting moving nodes with wireless device-to-device communications is a network composed of opportunistic contacts. This network graph changes over time and may not include paths between all nodes at any point in time. Bui-Xuan et al. [268] and Ferreira [107] show that this can be formalized with a simple combinatorial model of evolving graphs,
as shown in fig. 2.2.

Classical networks can be modeled with a graph $G = (V, E)$ where $V$ is the set of routers and $E$ the set of links between the routers—additionally the edges may be directed and weighted to reflect the type of communication link. In opportunistic networks (and in mobile ad hoc networks, MANETs), the edges and vertices change over time, which makes the graph a time-variant dynamic graph $G(t) = (V(t), E(t))$. Time can be divided into contiguous epochs, such that during epoch $[t_{i-1}, t_i)$ the network is described by a static graph $G_i$. We can then define an evolving graph $G = (S_G, S_T)$ where $S_T = t_0, t_1, \ldots, t_T$ is the sequence of all the epochs and $S_G = G_1, G_2, \ldots, G_T$ are the corresponding graphs. An opportunistic network can be represented by an evolving graph with a new epoch every time an opportunistic contact starts or ends.

Strictly speaking, classical networks also have time-varying graphs since occasionally links or routers break or new ones get added. However, the assumption in classical networking is that there is an end-to-end path between all vertex pairs $(u, v)$ in every $G_i$, called space path—i.e., $\exists P = e_0, e_1, \ldots, e_k$ s.t. $e_0 = u, e_k = v, (e_j, e_{j+1}) \in V_i$. In opportunistic networks space paths are not assumed to exist, but it might still be possible to deliver a message to the destination by using store-carry-forward techniques. This can be modelled as a journey $J = (R_e, R_\tau)$, where $R_e = e_0, e_1, \ldots, e_k$ is a sequence of edges such that $e_0 = u, e_k = v$, and $R_\tau = \tau_0, \tau_1, \ldots, \tau_k$ ($\tau_i < \tau_{i+1}$) is a sequence of the time instances when the edge is traversed (and $R_\tau$ is in accordance with $G$ and $R$).

The problem of routing in opportunistic networks is then the problem of finding a journey, also called a spacetime path, on the corresponding evolving graph. Borrel et al. [56] use this model to create a framework for classifying opportunistic routing protocols, while Faragó and Syrotiuk [105] use it for an assessment framework for MANET routing protocols.

### 2.1.4 Routing and Dissemination

The job of moving messages end-to-end using the opportunistic contacts belongs to routing and dissemination algorithms. In simple terms, the difference is that routing attempts to deliver a message to a predefined destination, while dissemination tries to make the message available to any interested nodes.

These algorithms have a set of tools they can use to achieve their goals: storing, forwarding, replicating, and coding. Storing the message allows
the algorithm to leverage future contacts, and comes with the related problem of buffer management [175, 158] to decide what messages to drop when the storage is full. Forwarding (single-copy) or replicating (multi-copy) in turn allows the algorithm to use an opportunistic contact to move or copy the message to another node—this comes with the related problem of scheduling, i.e., deciding in which order to send the messages [159]. The algorithms can also use coding (e.g., erasure codes) to encode the message into multiple parts that can be forwarded independently, with only a subset needed to reconstruct the message [171, 73].

Routing: Opportunistic routing has been extensively studied particularly in the context of DTNs. These algorithms use the mechanisms of forwarding, storing, replicating, and coding to transmit a message to a predefined destination—typically a single destination (i.e., unicast), but also to multiple destinations (i.e., multicast, broadcast, anycast).

The first class of routing algorithms neither assumes nor tries to derive any context information to make routing decisions—we call these zero-context algorithms. Perhaps the simplest way to route in an opportunistic network is to simply wait until coming into contact with the destination. This is called Direct Transmission routing [118], and has been proposed for data mules ferrying data between a sensor network and infrastructure [234]. It has the benefit of only a single transmission, but the drawback of long latency due to not being able to use multi-hop paths.

A somewhat counterintuitive result by Spyropoulos et al. [252] states that forwarding a message reduces its expected delivery time—i.e., instead of holding onto a message until meeting the destination, nodes should attempt to pass it forward. Motivated by this, they define the single-copy Randomized Routing Algorithm, which randomly forwards the message over a contact with some probability. Another choice is to always forward the message over the first available contact, called First Contact routing [142]. These algorithms, however, are mostly useful as comparison cases in research studies, not as practical routing approaches.

The above single-copy algorithms essentially perform a random walk of the contact graph. An alternative approach we can take with no context information is to replicate the message over every contact. This is called Epidemic Routing [260], which floods the network with copies of the message. When there are no resource constraints, epidemic routing uses every available spacetime path for every message, and therefore will always find...
an optimal route. Unfortunately, in reality the resources are constrained and the high replication rate of epidemic routing quickly congests the network. This has prompted many researchers to propose limited-flooding routing algorithms, which behave epidemically but use fewer resources.

The original authors, Vahdat and Becker [260], propose a hop count limit, which is effective, but the limit has to be chosen for the underlying mobility. Groenevelt et al. [117] develop simple stochastic models for 2-hop limited and unlimited epidemic routing, and show that the message delay is in both cases a linear function of the inter-contact time (assuming exponential inter-contact time distribution). Zhang et al. [275] expand on this by developing a unified framework for studying epidemic routing using ordinary differential equations. In a more adaptive scheme, Gou and Shen [130] reduce the replication probability as a function of hops that the message has travelled (from which they approximate the number of message copies created), which reduces the number of message copies while slightly increasing the message delay.

Another approach to limit the epidemic spreading, proposed by Spyropoulos et al. [248, 251], is to explicitly limit the number of message copies that the algorithm is allowed to make. Their algorithm, Spray and Wait, attaches a copy count to every message and divides the process into two phases: First, in the spray phase the nodes flood the message until their copy count reaches one. In the binary variant each node forwards half of their remaining message copies over every contact. Second, in the wait phase the nodes with message copies follow the Direct Transmission algorithm, i.e., wait until one of them meets the destination. The authors show via theory and simulations that, in resource constrained scenarios, their algorithm has significantly higher delivery rate, lower delay, and lower resource use than unconstrained flooding. Spray and Wait is good choice for a generic, practical routing algorithm due to its simplicity and good performance, but it may require the messages to be modified during forwarding to track the message copies, which complicates implementations.

Other optimizations to reduce the resource use of epidemic have been proposed. For example, Small and Haas [244, 243] propose to flood delivery notifications to clear delivered messages from their epidemic-based opportunistic network of whales.

A readily available source of potentially useful context information for routing is the history of contacts. Spyropoulos et al. [252] define this for a
single-copy case as *Utility-Based Routing with 1-Hop Diffusion*, where the utility is a monotonically decreasing function of the time since the node has last met the message destination, and a node will forward the message to another node when it has a sufficiently higher utility. They also define a variant with *transitivity*, i.e., if a node $A$ frequently meets node $B$, and node $B$ frequently meets node $C$, then $A$’s utility for $C$ should be non-zero even if $A$ never meets $C$ directly. A further variation by the same authors, called *Seek and Focus*, uses the randomized routing when utility is low, and the utility routing when utility is high. Simulation results indicate that these more sophisticated single-copy routing algorithms have potentially competitive performance while consuming little resources in the network ([248] provides more evidence).

History of contacts has also been used in multi-copy algorithms. In an early work, Juang et al. [147], in their opportunistic network of zebras, track the frequency of each node coming to contact with a base station and only flood messages to nodes with higher contact frequency. Lindgren et al. [173, 174] provide a more substantial design, called PRoPHET, by defining a “predictability” metric that is increased directly and transitively during contacts and decreased over time. This way the predictability should correlate with the probability that the node can deliver a message to the destination, and thus messages are forwarded to nodes with higher predictability values. In MaxProp [60], each node tracks the frequency with which it has contacts with each other node, and calculates a cost metric that increases as the contact frequency decreases. The nodes then exchange their frequencies, allowing each node to construct a graph with nodes as the vertices and costs as the edge weights—routes can then be calculated as the shortest path on this graph. As further optimizations, MaxProp prioritizes the scheduling of new messages, and floods acknowledgements that allow other nodes to remove delivered messages from their stores. The authors of Spray and Wait have also proposed *Spray and Focus* [250, 251], where the wait phase is replaced by an encounter history based single-copy forwarding algorithm. The underlying assumption of all these algorithms is that the past encounters can predict future encounters. While this assumption does not hold for random mobility, in section 2.1.1 we saw that it likely does hold for human mobility.

The second important source of context information for opportunistic routing comes from the social relationships of the people carrying the
networked devices. This is relevant because people’s relationships influence their movement [241], the content they are interested in [183] and their willingness to share their resources [195], as noted by Boldrini and Passarella [52]. These algorithms interpret the contacts between nodes as social encounters, and use the collected and shared contact information to construct a social network graph—often the ego network [99], i.e., the node, its direct neighbors, and the links between those neighbors. Various metrics can then be calculated from this graph, such as centrality or similarity, and routing decisions based on those—e.g., a central node likely meets more nodes, and a dissimilar node likely meets different nodes. Hsu and Helmy [131] provide evidence for the feasibility of this approach by showing that the resulting graph has small-world characteristics and the metrics converge quickly.

An early routing algorithm, SimBet, by Daly and Haar [84] defines a routing metric based on the node’s betweenness centrality within the ego network and its similarity to the destination node. When nodes meet, they construct SimBet utility metric as a weighted sum of the similarity and centrality for both nodes, and forward messages towards the higher utility. In another early work, Hui and Crowcroft [135] propose the LABEL algorithm to route based on communities. The nodes are assumed to be labeled with their community name, and messages are only forwarded to nodes in the same community as the message destination—they show that this can significantly improve routing performance on a real-world mobility trace. The authors then extend this into a fully fledged routing algorithm, BUBBLE [136, 137], by applying two ideas: 1) the communities are automatically inferred using k-clique clustering, 2) centrality is used both globally and within communities as a routing metric. In the first phase of BUBBLE, the global centrality is used to route the message towards better connected nodes, until it reaches the destination community. In the second phase, the centrality metric within the community is used as the routing metric until the message reaches the destination. Many later routing algorithms offer variations on these ideas of centrality and community, e.g., PeopleRank [186] and Sociability-routing [100] are based on types of centrality, SGBR [32] and SOCKER [273] are based on communities, and Gao et al. [110] use both. Effectively most “social” routing protocols are just sophisticated ways to use the contact histories between the nodes, so they are all relatively simple to implement in real systems.
The previous routing algorithms are designed to work with uncertainty about the future contacts. In some cases, however, we might have exact knowledge about the future contacts, e.g., in space networks the movement and contacts can be known and scheduled ahead of time [34]. In that case, we can apply a deterministic routing approach as shown by Jain et al. [142]. Their approach is to define a graph where the vertices are the nodes and edge weights are the costs to transmit the message between the nodes—route can then be found by calculating the shortest path on this graph. While this is a standard formulation of a general routing problem, the authors adapt it to opportunistic networks by defining the edge weights to be time-dependent. In particular, the cost $w_{i,j}(t,m)$ is the delay to send the message $m$ from node $i$ to node $j$ if it arrives at node $i$ at time $t$. Major part of this delay is waiting for the next contact, which we know explicitly from the future contact schedule. Other parts of the delay are due to the previous messages that need to be transmitted first and the propagation delay of the link—knowledge about these improve performance, but knowing the contacts is the dominant factor. After defining the edge weights, the authors adapt Dijkstra’s shortest path algorithm for the time variant edge weights in a straightforward way (similarly to Ferreira [107]).

**Dissemination:** While the previous algorithms focused on delivering messages to specific recipients (i.e., unicast), another useful mechanism in opportunistic networking is to make the message available to everyone in the network (i.e., broadcast). More generally, this can be seen in terms of the publish/subscribe paradigm, where publishers inject content into the network, and subscribers express interest in that content, without either one necessarily being aware of the other [52, 98]. Dissemination can be done with the same store-carry-forward primitives provided by the opportunistic network, but unlike routing, the destination of a message is typically not known in advance.

In an early work, Lenders et al. [168] study an opportunistic dissemination system organized around channels composed of entries [151]—entries are published into the channels, and nodes can subscribe to channels to receive the entries. During opportunistic contacts, nodes exchange entries for channels in which they have interest, but also try to help the network by caching other entries. The authors define dissemination strategies based on content popularity, and show that intentional cooperation improves performance, which is one of the original catalysts for the research into
opportunistic data dissemination.

As with routing, social context can be applied to dissemination as well. Boldrini et al. [49, 50] propose a dissemination algorithm, Content Place, based on community detection. They define a utility metric for a (node, message) pair as a weighted sum of utilities that the message has to the communities that the node belongs to, and use this utility to make forwarding decisions. The utility that a message has for a specific community is taken from web caching work, and depends on the popularity, availability and size of the message, which are estimated based on information collected during contacts. This need to maintain statistics per message per community is the main limitation of the approach. In another social context based approach, Yoneki et al. [270] define an opportunistic publish/subscribe data dissemination system. They first detect communities and then assign a broker to each community based on the node’s centrality metric. The brokers form an overlay to disseminate published messages to the interested subscribers. Other solutions include SocialCast by Costa et al. [82], and MOPS by Li and Wu [169].

A geographically constrained approach to content dissemination, Floating Content, has been proposed by Ott et al. [199, 198]. They define an anchor zone \( (P, r, a) \) with a center point \( P \) and two radii \( a \) and \( r \) such that \( r < a \). During a contact a message gets replicated to a node with a probability that depends on its location w.r.t. the anchor zone: Within the radius \( r \) the message is always replicated and outside radius \( a \) it is never replicated, and in between it is replicated with a probability that decreases as a function of the distance. There is a complementary deletion mechanism that always deletes messages outside \( a \), never inside \( r \), and with random probability in between. Their simulations show that an anchor zone of a few city blocks is feasible in a dense downtown environment.

In a recent work, Borrego et al. [55] limit the number of nodes that can perform the store-carry-forward operations instead of limiting the number of message copies. They propose a metric based on the centrality, similarity, and reliability of the node, and apply the Optimal Stopping Theory [238] to choose the carriers. Their simulation show a significant improvement in message dissemination per unit of energy and message latency.

While this section gives an overview of the routing and dissemination work, it is one of the most intensely studied areas of opportunistic networking and many more algorithms and approaches have been devel-
Concepts of opportunistic networking

Figure 2.3. PodNet system design. Node architecture on the left, data structure in the middle, synchronization process on the right.

2.1.5 Platforms and Middlewares

The opportunistic networking mechanisms are often implemented in a separate middleware or router, which provides networking services to applications via simple interfaces. Three well-known platforms are PodNet, Haggle, and various implementations of the DTN architecture and protocols. We will briefly introduce these systems here and later in section 2.3 explain in more depth of our own middleware implementation (also based on the DTN protocols) used in this thesis.

**PodNet**: PodNet [168, 122] is possibly the first major opportunistic data dissemination middleware. It is a topic-based publish/subscribe design with a data model inspired by the Atom Syndication Format [194]. The content is grouped into feeds containing entries, and each entry can have an attached enclosure containing a binary file. Applications interact with the system via a publish/subscribe API using the feed identifiers. During opportunistic contacts, the nodes query each other for the available feeds and entries, and request copies of any entries for which they have local subscriptions. This results in pull-based dissemination model.

Figure 2.3 shows the PodNet system design. The node design, on the left of the figure, shows the middleware running on every device. On the bottom, various wireless technologies, typically either Bluetooth or Wi-Fi,
create opportunistic contacts with nearby peers. A *Convergence Layer* abstracts the heterogeneous wireless technologies from the rest of the system. The *Discovery Module* uses the underlying wireless technologies via the convergence layer to discover nearby peers. Once peers have been discovered, the *Transport Module* is used to implement a request/response protocol that can fetch the feeds and entries from a connected peer. The logic that decides which feeds and entries to fetch is implemented by the *Sync Manager*, which has access to the local content in a database. By default, only feeds that have local subscriptions are fetched during contacts, but additional co-operative caching can be implemented by fetching other feeds. The interactions with the applications, at the top of the figure, are done via a *Pub/Sub API*.

The PodNet authors have implemented the middleware and various applications on Android. The applications include an opportunistic media blog, a personal profile sharing, a collaborative music sharing, and a participative light show.

**Haggle:** Haggle [254, 192, 79] is another seminal opportunistic networking middleware, also based on data dissemination. It is a content-based publish/subscribe design in which data objects are tagged with metadata attributes describing their content. Applications express their interest in receiving content by subscribing with a metadata attribute name, value and

---

**Figure 2.4.** Haggle node design. Event-based kernel with a data store, surrounded by managers, surrounded by modules.
weight. The underlying platform then matches the applications’ interests against the metadata of locally stored data objects. During opportunistic contacts, the nodes exchange their local applications’ aggregated interests and send each other matching data objects from their local storage. This results in a push-based dissemination model.

Figure 2.4 shows the Haggle middleware architecture. It is a microkernel design with a small kernel that holds the shared state, including the data store, and provides an event bus for service communication. The major functionality of the middleware is divided into services, called managers, that use the kernel’s event bus to interact. Managers can further have modules for specializing their functionality.

Connectivity and Protocol managers create opportunistic contacts jointly: The connectivity manager manages multiple underlying communication technologies, such as Wi-Fi and Bluetooth, by tracking the available interfaces and running peer discovery on the active ones. The protocol manager implements the data object transmission between two nodes using some suitable protocols, acting as a convergence layer over different transport protocols. The dissemination functionality is implemented by the Forwarding manager supported by the Node manager that tracks information about the encountered peers. The forwarding manager has dissemination protocols as modules, which decide which data objects to forward.

Additional managers provide supporting functionality such as managing the resource use, providing security, managing the data object storage, and providing the API towards the applications. The highly decoupled event-based microkernel approach allows new managers to be added easily. Pietiläinen at al. [210] use this extensibility in their mobile social networking MobiClique system, which adds social profile management and social graph maintenance to the Haggle middleware.

The Haggle authors have created a cross-platform implementation that supports Linux, Windows, Mac, and Windows Mobile. In addition, multiple applications have been developed for the platform, including photo sharing, medical triaging, social networking and messaging, and voting.

**DTN:** The DTN architecture [102, 62, 104, 70] differs from the previous two platforms in that it is the result of an open, collaborative development and specification process, largely carried out in the IRTF and the IETF. The specifications include a common architecture [70], an opportunistic networking layer called the Bundle Protocol [233], protocols for opportunistic
The DTN architecture is an opportunistic networking overlay on top of heterogeneous underlying transports, as shown in Figure 2.5. The main store-carry-forward mechanisms are implemented by the Bundle Protocol, shown in blue in the figure, including message (called bundle) storage and routing algorithms. These bundles have a block structure (e.g., primary block for typical header information, and payload block for the data), which allows the bundles to be easily modified while they traverse the network by adding, removing and modifying the blocks. Bundles also have an expiration time, after which they are removed from the router storage. Bundle routers, which implement the protocol, open opportunistic contacts between each other using various communication technologies via convergence layers. For example, two routers in the same Wi-Fi access point might discover each other using the multicast UDP based IP Neighbor Discovery [96] and open a transport link using the TCP convergence layer [87]. It is then up to the routing protocol to decide which bundles to exchange over active contacts.

The basic architecture is similar to topic-based publish/subscribe designs: Applications send (publish) messages to destinations called endpoints, named by an endpoint identifier (EID). Each endpoint may contain multiple nodes, and applications wishing to receive messages sent to an endpoint will register (subscribe) their interest in the corresponding EID.
ing is used to map EIDs to the corresponding nodes, which means that the senders and receivers do not need to be aware of each other. There are, however, a number of advanced mechanisms that go beyond basic topic-based pub/sub: Various delivery semantics can be defined for when a message is considered to be “delivered” to an endpoint—e.g., when it is delivered to any node in the endpoint, or all the nodes in the endpoint. Further, the EIDs are Universal Resource Locators (URIs) [47], and extensions can define EID schemes with arbitrary semantics—e.g., the “find” extension allows EIDs to address non-DTN services such as email [85].

Other advanced functionality include reliability through a custody transfer mechanism, which allows bundle routers to explicitly forward the responsibility for bundle delivery between them, priority classes for quality of service differentiation, fragmentation to transmit large bundles over multiple short contacts, and delivery receipts to track the traversal of a bundle through the network.

Multiple implementations of the DTN architecture and related protocols exist: DTN2 [9] is the official reference implementation, IBR-DTN [89] provides a highly efficient implementation suitable also for constrained devices, ION [61] is a NASA implementation for space networks, and the recent DTN7 [207] implements the Bundle Protocol version 7, to name a few. Our Scampi router, which is used in this thesis and described in section 2.3, is also based on the DTN architecture and protocols.

2.1.6 Summary

In this section, we have covered an overview of the basic concepts that underlie opportunistic networking—human mobility, wireless device-to-device connectivity, spacetime paths, routing and dissemination, and middleware platforms—and the previous research done on them.

We observed that human movement has significant structure, including small number of highly preferred locations, temporal cyclicity, and heterogeneity of characteristic travel distances. The contacts created when this movement brings people close to each other also has significant microscopic structure, with inter-contact times and contact durations appearing to be power law distributed with exponential cut-offs. There is also macroscopic structure in these contacts, with detectable communities connected by bridge nodes and links. Many mobility models exist to provide approximations for this movement, ranging from very simple random models, to very complex approaches that try to model the human agendas in detail, to trace
based models that attempt to closely recreate the statistical distributions.

Wireless device-to-device connectivity is then needed to connect the nearby devices, creating opportunistic contacts. We discussed three main classes of these technologies—802.11 Wi-Fi, Bluetooth, and device-to-device cellular—each of which comes with its own challenges and benefits. In particular, no wireless technology specifically designed for opportunistic networking is available, but the existing technologies can still be leveraged to build real systems.

The opportunistic contacts alone do not make a network, so in addition we need routing and dissemination algorithms—routing algorithms attempt to deliver a message to a particular destination, while dissemination algorithms attempt to make the message generally available to interested nodes. A large variety of algorithms for both exist in the literature, built on the primitives of storing, forwarding, replicating, and coding. The two main classes of algorithms are: 1) those that use zero context information (e.g., epidemic or first contact), and 2) those that use contact histories. For the latter, there are many sophisticated algorithms that try to infer the social context from the contacts, and then apply social network analysis methods to derive routing metrics. Plenty of choice exists for practical implementations, although it is not clear that any single algorithm dominates—different algorithms are likely needed for different scenarios.

Finally, significant effort has been put into implementing these concepts in middlewares. There are innovative systems arising from specific research projects (e.g., PodNet and Haggle), and the DTN architecture and protocols that have been collaboratively specified in the IRTF and IETF. In practical terms, the DTN architecture appears to have the most development effort and deployment.

In the next two sections we will build the tools needed to study systems based on these concepts: First, we will develop a way to simulate all these concepts in a single simulator that combines mobility, opportunistic contacts, routing, and applications. After that, we will study the implementation of these concepts in a real middleware in more detail than what we already saw in section 2.1.5.
2.2 Simulating Opportunistic Networks: Publication I

The first tool that we need for the research work in this thesis is a simulator capable of modelling the opportunistic networking concepts described in the previous section. Various simulators exist that could be (and have been) used to model opportunistic networking: General network simulators such as ns-3 [23] and OMNET++ [24] can be extended with opportunistic networking models, but their focus tends to be on detailed packet-level simulation. Mobility can be simulated with special purpose mobility simulators such as BonnMotion [4] or SUMO [12], but then the network simulations must be carried out using another tool and the context of the movement is lost. At the time of this work, no simulator focused on integrating all the opportunistic networking concepts in a single simulation.

This is the motivation for The Opportunistic Networking Environment (The ONE), which specifically targets opportunistic networking by implementing all the key concept—mobility, wireless contacts, store-carry-forward messaging, and applications—in a single simulator. In this section we will present the high level design of The ONE simulator, explain how it models the different opportunistic networking concepts, and show via an example how it is used to build simulation scenarios. All the simulation work in the following chapters of this thesis use The ONE simulator, and it is also used to generate the mobility and connectivity patterns for the testbed evaluations.

2.2.1 Simulator Architecture

The ONE is an agent-based, discrete event, opportunistic network simulator. This means that it maintains a set of agents whose behavior it simulates at discrete points in time. In concrete terms, The ONE maintains a simulation state structured out of components that model opportunistic networking concepts, and continuously updates the state after advancing the simulation time by a fixed time step. The high level structure is reminiscent of game engines, where the frame rendering loop advances the game time by the frame period and then updates the game state (e.g., recalculates the enemy positions) by the amount of time advanced.

The ONE is designed specifically for opportunistic networking simulations, which is reflected in its internal component model, shown on the left of fig. 2.6. Each simulation run has a single instance of the highest level component, the World, which contains all the other components. In
particular, it contains a set of agents, called $\text{DTNHosts}$, which represent the network nodes, e.g., humans carrying mobile phones. The agent set is fixed for the duration of the simulation, but the agents may be active and inactive during different periods of the simulation. In addition to the agents, the simulation has a set of $\text{EventQueues}$, which can generate events, such as sending new messages, into the simulation.

Each host has three important internal components: a $\text{MovementModel}$, some number of $\text{NetworkInterfaces}$, and a $\text{Router}$. The movement model defines how the host moves in the simulation by implementing some synthetic mobility model. The peer-wise opportunistic contacts between hosts are modelled by the network interface components, of which the host can have multiple. The ONE is a network level simulator and does not model any lower layer mechanisms, so the network interfaces are abstracted to a (possibly time-varying) communication range and capacity—nodes within the communication range can exchange messages at the given capacity of the interface, while nodes outside the range cannot communicate. Message forwarding over these contacts is controlled by the $\text{Router}$, which contains the host’s message cache and implements some opportunistic routing or dissemination algorithm. In addition, the router can have any number of attached $\text{Applications}$, which can implement any application layer state and logic that goes beyond just sending and receiving messages (the latter of which can be done by the event queues).

Simulation scenarios are created by defining concrete types and parameters for all the components. In The ONE simulator, this is done by configuration files. When the simulator is launched, it will parse the configuration.
file and set up the initial state according to the component definitions and parameters in the file. In principle, every host in the simulator could have a unique configuration, but mostly hosts are configured in groups whose members are identical—e.g., one group for pedestrians and another for cars. The configuration also specifies concrete types (Java class names) for the various components, which the simulator will load dynamically.

Once the initial state has been configured, the actual simulation is executed by the simulation loop, as shown on the right of fig. 2.6. The loop starts by incrementing the simulation time by a fixed time-step (configurable, but typically 0.1–1 seconds). Then all the event queues are processed, which may directly update the simulation state, e.g., by creating messages or setting up contacts. Next, the locations of all the hosts are updated according to their movement models, and contacts between them are calculated by the network interfaces. After the generation of the current contacts, the routing algorithms in all the hosts are called to transfer messages over the active contacts. The routers also update their attached applications when new messages have been received. During the entire process, the components generate events that can be recorded by Report modules attached to the components. After the simulation loop finishes, these report modules will generate their output into report files.

The ONE is implemented in Java, which provides a good trade-off between performance and easy extensibility, and provides easy portability. The simulator is single threaded and cannot scale a single simulation to multiple cores. This is usually not a problem, since multiple simulator instances can be run in parallel on multiple physical cores, e.g., with different seeds needed to generate statistical averages. Not being able to scale a single simulation to multiple cores does, however, limit the complexity of the scenarios. Typically, a practical limit to the scenario size is around 10000 hosts, depending on how complex the routing and application models are. The limiting factor on scalability tends to be the connectivity calculation, which is quadratic in the number of hosts within the communication range (the distance between every node pair is calculated).

2.2.2 Creating Contacts

Contact creation in The ONE has two elements: Node mobility that determines how the nodes move in the simulation, and communication interfaces that determine whether, and at what rate, two nodes can exchange messages. Both of these have well-define extension points in the simulator,
which allows custom models to be built easily.

**Mobility:** Mobility models in The ONE need to implement two functions: Providing the *initial location* for the host, and providing movement *paths* when required by the simulation. The paths are arbitrarily long lists of waypoints and speeds, and may cover a single simulation step, or the entire simulation duration. There are two main types of mobility model implementations: Those that generate movements based on *synthetic models*, and those that generate movement by reading *external traces*.

This structure makes it very easy to create random free-space synthetic models. *Random Walk*, for example, can be implemented by providing a random initial location, and then creating paths by picking a random direction, a random distance and adding them to the current location—by always moving to the edge, this becomes the *Random Direction* model. *Random Waypoint* is the same, except it picks random locations when building the paths—constraints can be placed on how these locations are picked to create, e.g., clusters or areas with different popularity.

An important class of mobility models in The ONE are map-constrained models. These models load a map file as a graph representing the allowed movements (e.g., roads), and generate paths from this graph. In the simplest case, the model can generate the path by picking random sequences of edges, which results in a random walk on the graph. However, it is more typical to define sets of points-of-interest on the map, pick the destinations from those, and then calculate the shortest path to there along the graph—this is called *Shortest-Path Map-Based Movement*. The ONE has built-in support for multiple map layers, so that different hosts can be restricted to different paths of the map—this is useful to, e.g., differentiate between pedestrian paths and motorways.

More complex mobility models are typically constructed by combining a state machine model for the host state with a number of primitive mobility models that are switched based on the state. The Working Day Movement model [95] included in The ONE, for example, has six states for *home*, *office*, *evening activity*, *traveling to work*, *traveling home*, and *traveling to activity*. The host transitions between these states based on the time of day, and each state uses a different primitive mobility model.

An alternative to synthetic mobility models is to generate movement based on external traces. The ONE implements this via the same mobility model interface as the synthetic models. These models read external traces from files and turn them into paths, which are then used in the simulation
as above. The advantage of traces is that they recreate authentic mobility with the real statistical properties of human mobility, but the downside is that they cannot be customized and randomized to generate new mobility similarly to synthetic models.

**Network Interfaces:** The second part of contact creation is to connect nearby hosts, which is done by network interfaces. Since The ONE does not model any packet or link layer phenomena, these interfaces are quite simple: Hosts with the same interface can communicate with each other if they are within a predefined range. The communication rate is usually static, but time-variant capacity is supported—the simulator includes an interference-limited interface that scales down the rate as a function of the number of nearby nodes, and a distance-limited interface that scales down the rate as a function of the distance between the hosts. In addition, the interfaces support a *scanning frequency*, which limits the frequency at which new hosts can be discovered. This models the behavior of real-world protocols, which only scan for peers periodically.

The calculation of contacts between network interfaces requires comparing every host against every other host to see if they are in range. This leads to a runtime that is quadratic in the number of hosts in the simulation, which often becomes the performance bottleneck. To alleviate this, The ONE includes a grid-based optimizer that only compares the hosts that are conceivably within communication range. The behavior is still quadratic, but if the hosts are spread out there is a significant constant factor improvement in performance—if the nodes are clustered around each other this optimization does not help.

### 2.2.3 Store-Carry-Forward Messaging

The previous mechanisms generate the underlying opportunistic contacts, but define no mechanisms for using those contacts to forward messages. In The ONE, the store-carry-forward messaging is the responsibility of *Message Router* modules, and their *Application* modules. The routers implement the message forwarding, while the applications implement more complex application models that use the basic messaging service. Similarly to the previous components, both of these have well-defined extension points in the simulator, which makes creating custom models easy.

**Routing:** The message router modules are responsible for the entire store-carry-forward model: They maintain the message cache, manage the state of message transfers of the peer connections, make routing decisions,
and inform applications when messages arrive. However, much of this work is generic and implemented by a common superclass, allowing routing algorithm implementations to focus on their custom behaviors.

Most routing modules are structured around an *update* method, which is called in every simulation step, and additional callback methods that are called by the simulator when new contacts are created or message transfers have been finished. Within the update method, the routers first check if it is possible to start a new transfer (e.g., there are idle contacts), and if so, select which message transfer to start on which contact.

*Epidemic* routing, for example, is implemented by trying to send all messages over all contacts. This can be turned into *First Contact* routing simply by deleting the message from the router after it has been successfully transferred to the first peer. Alternatively, it can be turned into *Direct Delivery* routing by only forwarding to peers that are the destination of the message. *Spray and Wait* router attaches the copy count to the message as a property, and only forwards the message if there are copies left.

Algorithms that use the encounter histories, like *PRoPHET* and *Max-Prop*, implement their metric calculations in a callback invoked every time a contact state changes. PRoPHET, for example, uses the callback to calculate its predictability metric values, and then uses them to select messages in the update method.

The router modules have access to the full simulation state via the *World* object, which allows routing algorithms with full knowledge to be implemented. For example, an ideal *Geographic Router* can access the locations of every host in the simulation and make forwarding decisions based on that. While such routers are unrealistic, they can reveal upper bounds to possible performance.

**Applications:** While the mechanisms we have seen so far are enough to evaluate routing algorithms, which has been a major focus of research into opportunistic networks, many of the mechanisms that we study in this thesis require state and logic above the routing layer. For example, the mutable content mechanism of section 4.3 needs to model mutations and merging of message content, independently of the underlying routing algorithm. The ONE defines application modules for this purpose.

Application modules are implemented by defining two methods: A *handle* method for incoming messages, and an *update* method similar to the router modules. The handle method is invoked every time the router receives a message for that application, allowing the application to run its custom
logic. The application can then choose to forward, modify, or drop the message (e.g., if it has reached its final destination).

Applications can also create new messages either in response to received messages or from the update method. The application content is attached as properties to the message. For example, a ping application would use the update method to periodically generate a message with a “ping” property. Other application instances would then receive the message via the handle method, and respond by generating a new message with a “pong” property destined to the sender of the ping.

### 2.2.4 Outputs

The purpose of building simulation models is, of course, to produce some useful outputs that help to understand the system under study. This is implemented in The ONE by Report modules, which record events during the simulation run and output report files after the simulation has finished. In addition, The ONE can be used to control real opportunistic networking middleware to emulate intermittent connectivity between them—this is useful for the testbed experiments that we describe in the later chapters.

**Reports:** All the components we have seen so far generate events about their actions during the simulation run. These events can be received via Listener interfaces, defined for the components. Listener interfaces have been defined for applications, connections, routers, mobility models, and the simulation loop. For example, the connection listener interface is called every time a contact is created or destroyed, and the movement listener interface is called every time a mobility model generates a new path.

Report modules implement these interfaces to be informed about the relevant events during the simulation run. The simulator automatically connects the report module to the correct components based on the listener interfaces that it has declared. After the simulation run is finished, the simulator calls into the report modules to allow them to write their reports into output files.

The ONE comes with a variety of report modules, which can calculate metrics for the node mobility (e.g., flight length distribution and radius of gyration), contacts (e.g., inter-contact time and contact duration), and messaging (e.g., delivery probability and latency). In addition, report modules can be used to create traces for external tools, for example, a mobility trace for use in ns-2 or even a contact trace for use with The ONE itself so that the expensive contact generation does not need to be repeated.
in every simulation run.

**Controlling Testbeds:** Beyond just collecting statistic of the system behavior, simulators are useful for creating emulated contacts in testbeds of real opportunistic middle implementations. The ONE includes native support for controlling the contacts between DTN2 reference implementation instances. This is done by creating a mapping between middleware instances and hosts within the simulation. In the beginning of the simulation The ONE connects to the control interfaces in the middleware instances and creates links between them as contact events occur in the simulation. The Scampi router, described in the next section, includes a topology controller interface designed for this specific purpose.

### 2.2.5 Example Simulation

To see how The ONE can be used for opportunistic networking research, we can look at an example simulation. This involves building a simulation model, which defines how the initial simulation state is set up—how many nodes there are, which mobility models they use, what routing algorithm is used, and how are messages generated, among other parameters. This model is then simulated, typically numerous times, while varying some parameters such as the random number generator seed value or the algorithms under study. Afterwards, metrics are calculated from the simulation results to characterize the behaviors of interest.

In this example simulation, we will compare free-space random mobility, map-constrained mobility, and agenda based mobility, while using six different routing algorithms. We will characterize both the opportunistic contacts, and the messaging performance. The goal is not to provide new insights into opportunistic networking, but rather to demonstrate the methodology used in the rest of the chapters of this thesis.

**Scenario Definition:** We start by constructing the simulation models, often called *scenarios*. To do this, we need to define the *simulation area*, the number of nodes, the *mobility models*, the *communication interfaces*, the *routing algorithms*, and the *message generation*. We divide the simulations into three scenarios based on the complexity of the mobility model used: *Random Waypoint* (RWP), *Helsinki City Scenario* (HCS), and *Working Day Movement* (WDM). Within these scenarios, we have six sub-scenarios for six different routing algorithms. The specific values we pick are arbitrary, but chosen to be illustrative based on our experience with these scenarios.

The size of the simulation area, the number of nodes, the communication
Concepts of opportunistic networking

interfaces, and message generation are the same in each scenario. We place 544 nodes on a simulation area of 8300 × 7300 meters. They have communication interfaces with 10-meter range and 2 Mbps transmission rate, which models Bluetooth Classic connectivity. Each node has 100 MB storage with message sizes uniformly randomly distributed between 500 KB and 1 MB. Messages are sent once every 20–30 minutes from a random source to a random destination.

The mobility differs in the three scenarios:

1. In RWP nodes have a uniform random speed of 0.5–1.5 m/s, with a pause time of 0–120 seconds.

2. In HCS the nodes are constrained to the Helsinki map. Six of the nodes are trams with fixed routes at 25–36 km/h and pauses of 10–30 seconds. One third of the nodes are cars driving at 10–50 km/h with pauses of 10–120 seconds, and two thirds pedestrians with the same speed as in RWP. Both cars and pedestrians pick random destinations and calculate the shortest path on the map.

3. WDM is a complex scenario that models the human working day on the Helsinki map. Nodes have home locations, office locations, and evening activity spots, between which they commute by walking, driving, or public transport according to the time of the day. We use the scenario from section 5 of [95], but proportionally scale it down to 544 nodes.

We simulate six different routing algorithms (see section 2.1.4) for messaging: Direct Delivery (DD), First Contact (FC), Epidemic, Spray and Wait, PRoPHET, MaxProp. PRoPHET has initialization constant $P_{init} = 0.75$, aging constant $\gamma = 0.98$, and transitivity scaling constant $\beta = 0.25$ (see [173]), and Spray and Wait is limited to six message copies with binary spraying.

**Simulation Results:** To generate results, we run each of the simulation scenarios with different random seeds to account for variance (10 and 5 seeds for contact and messaging metrics respectively). We are interested in the resulting contact patterns, which enable opportunistic networking, and the resulting message routing performance with different algorithms in the different mobility models. Figure 2.7 shows the simulation results.

We characterize the opportunistic contacts by measuring both inter-contact times (ICT) and contact durations, whose complementary cumu-
As discussed in section 2.1.1, real human mobility traces tend to exhibit power-law distributed inter-contact times. We can see that neither RWP nor HCS result in a power-law behavior, which would be a straight line on the log-log graph. In fact, the RWP scenario is so sparse that there are hardly any contacts at all—to better show the behavior in a more dense scenario, we also plot the ICT for RWP with 1029 nodes (dotted line in the figure). This is as expected, since these random models are unable to produce realistic human mobility [114]. The WDM scenario produces arguably more realistic inter-contact times, with linear behavior in the range of $10^3$–$10^4$ seconds, followed by a cut-off.

The contact durations are shown in the middle of the figure. We can again see how the different models result in different characteristics: HCS tends to lead to longer contact durations than RWP because nodes are constrained to move along the same paths rather than being able to move to arbitrary directions. WDM contact durations seem to resemble a power-law with a cut-off, which is expected for human mobility. There is, however, a pronounced peak around 10 seconds, which corresponds to the contact duration that results when two nodes meet while walking in opposite directions at the mean speed of 1 m/s with radio range of 10 meters. WDM also has long contacts because the nodes spend long times near each other in the office during the day or at home during the night.

The resulting message delivery probabilities within 18 hours of simulated time are shown on the right of fig. 2.7. As expected from the very infrequent contacts in RWP, no routing algorithm is able to deliver messages with a high probability—the spacetime paths simply do not exist within the message lifetime of 16 hours. The results appear to show that MaxProp
would have the highest delivery likelihood in the RWP scenario. Since the underlying mobility is random, MaxProp’s contact prediction cannot be responsible for the good performance. Instead, it is likely that MaxProp’s optimizations to prioritize new messages and the cleanup of caches via delivery receipts improve its performance.

In contrast to RWP, multiple routing algorithms reach very high delivery probabilities in HCS—this is because nodes in HCS travel widely but are constrained to the map paths and therefore meet frequently. Epidemic, PRoPHET, and MaxProp all reach above 90% delivery rates. The underlying mobility is still random, so the contact prediction of PRoPHET and MaxProp should be making essentially random replication choices. It seems likely that the good performance is simply due to the aggressive message replication done by these algorithms—the Spray and Wait variant that we chose restricts the replication more, leading to worse performance. The algorithms that do no replication, FC and DD, perform poorly. The fact that epidemic does well, indicates that the network is not congested (as expected since the message generation rate is low). Typically, the performance of epidemic routing collapses as the network becomes congested.

WDM again shows significantly lower delivery rates than HCS for all the routing algorithms. This is because the underlying mobility in WDM creates communities (e.g., workers in the same office) within which nodes spend most of their time. Transmitting messages within these communities is quick, but finding routes between different communities is difficult. As we pick the sources and destinations randomly, they are likely in different communities and therefore routing between them is more likely to fail.

Overall, this example shows how important mobility and routing algorithms are for the performance of the system. It shows the importance of selecting a range of different models when evaluating a system, which is what we do in the simulations in the following chapters.

### 2.2.6 Summary

In this section we saw how to build a simulator focused on implementing the opportunistic networking concepts, and how to use the simulator to construct simulation models and generate results for evaluation. This will be one of our primary tools as we evaluate the various mechanisms presented in the remaining chapters of this thesis. The ONE has also proven popular for opportunistic networking research more widely, and has been used by many other authors within the field.
2.3 Implementing Opportunistic Networks: The Scampi Router

As discussed in section 2.1.5, real-world implementations of opportunistic networking are typically done by middlewares or routers, which then provide a simple interface to applications. Further, prototyping is a major element of the systems research methodology applied in this thesis—simulations, as described in the previous section, can show theoretical feasibility and scalability properties, but showing practical feasibility requires building prototypes. This means that we need an opportunistic networking platform implementation.

At the time this work started, there were research focused middlewares, such as PodNet [122] and Haggle [192], which were developed by small teams for specific research projects. Basing work on such external platforms with fully custom architectures and protocols, and uncertain future(s), is risky. Alternatively, the DTN architecture and protocols were being specified in an open, collaborate process in the IRTF—this often results in longevity and better thought out solutions, due to a larger and more diverse set of people being involved. A number of DTN implementations already existed, but they were focused on either being faithful reference implementations (DTN2), or targeting specific use cases (e.g., NASA's ION). This made them unattractive platforms for research, where flexibility and extensibility are the most important characteristics.

These considerations lead us to develop Scampi, whose design we will explore in this section. It is based on the DTN architecture and protocols, which gives it interoperability and a solid design basis, but focuses on enabling flexible experimentation and prototyping rather than full and exact implementation of the DTN specifications. Here we will cover the basic structure and design of Scampi, while later chapters describe extensions, advanced features and applications in detail—the prototype implementations in this thesis use Scampi as the underlying platform.

2.3.1 Router Architecture

The high level structure of the Scampi router, shown in fig. 2.8, is similar to the Haggle design we saw in section 2.1.5: A simple event-processing core that holds the shared state of the router and facilitates interactions

\footnote{Scampi router in fact originated in the SCAMPI project, which was a follow-up project to Haggle with many of the same researchers. However, Scampi is developed independently and does not share any code or protocols with Haggle.}
Concepts of opportunistic networking

![Figure 2.8. Scampi system architecture overview. Event-processing core in the middle surrounded by the major subsystems.](image)

between subsystems that implement the various functionalities that comprise the system. The major subsystem groups are the store-carry-forward group that implements the opportunistic communications, the control group that implements control plane functionality, and the API that implements the application interface. In addition, there is a utility subsystem that implements logging, monitoring and management interfaces, and various extensions that implement more advanced features (such as those discussed in section 4.2 and section 4.3).

The store-carry-forward group comprises the communication, routing, and storage subsystems. The communication subsystem is responsible for creating opportunistic contacts, which it does by first detecting local communication interfaces, then running peer discovery on these interfaces, and finally opening transport links to the discovered peers. These contacts are then used by the routing subsystem, which can run multiple routing and dissemination protocols in parallel to push and pull messages over the links. The messages are persisted by the storage subsystem, which implements differentiated cache management—messages can be treated differently depending on whether they are generated by a local application or received from peers, and whether they should be permanently stored.

In addition to the basic store-carry-forward functionality, the router also needs various control-type functionalities. In Scampi this includes the
identity subsystem, which implements (cryptographic) identities—by default each Scampi instance has an identity based on a public-private key pair, which can be used to bootstrap security mechanisms like message authentication. As discussed in section 2.1.4, many advanced opportunistic routing and dissemination algorithms rely on analysis of past encounters (e.g., to calculate social metrics). Since this information is needed by so many other parts of the system, we aggregate it in a single encounter context subsystem, which maintains the history of encounters and the social context derived from it. Finally, the signalling subsystem implements the various peer-wise control signalling mechanisms, such as time and location synchronization, and content vector exchanges.

Scampi provides applications with a topic-based publish/subscribe API. Applications compose Scampi messages and publish those to topics (opaque strings), and other applications wishing to receive them subscribe to the corresponding topic. The content in Scampi messages is a typed key-value map where the values can be primitive data types or binary blobs—this is in contrast to most pub/sub systems that treat messages as opaque binaries. The applications can tag the message with metadata that then becomes available to the routing and other subsystems.

The Scampi implementation comprises roughly 50000 lines of Java code, including the DTN protocol implementations (e.g., Bundle Protocol [233] and TCPCL [87]). It runs on both the Java Virtual Machine (JVM) and Android. The core is single threaded, but the subsystems use internal threads—this provides a balance between needing complex locking mechanisms for the shared state and the ability to exploit parallelism. The main synchronization point is a priority queue into which the subsystems add prioritized events which the core then processes in priority order. The core does not act as an event bus, passing events between the subsystems, but rather executes code attached to the events. This trades off some decoupling in order to be able to represent multi-subsystem control flows as linear code, rather than having the code for a single logical operation being spread in various event handlers in various subsystems.

The implementation uses Java’s dynamic class loading extensively, allowing many parts of the system to be dynamically loaded and configured—from individual routing, topology control, and cache management algorithms, to entire communication stacks. This is particularly useful when running on Android, where the many Android-specific libraries can be developed and loaded as separate modules while keeping the majority of the
implementation in pure Java. We have run Scampi on many devices from low powered Android phones, to Raspberry Pis, to Linux servers—however, the choice of Java does mean we cannot target very low-end devices the way the C-based IBR-DTN [89] can, for example.

2.3.2 Store-Carry-Forward Subsystems

The store-carry-forward group of subsystems implements the major opportunistic networking concepts: The communication subsystem creates transport links for the opportunistic contacts, the routing subsystem passes messages over the contacts, and the storage subsystem persists messages so that they can be carried. Scampi provides a framework for each of these, which can be used to add implementations for specific technologies (e.g., routing algorithms or wireless stacks). Scampi also provides implementations of, e.g., Wi-Fi and DTN based protocols on the framework.

**Communication Subsystem:** The first major subsystem, an example configuration of which is shown on the left of fig. 2.9, is the communication subsystem, which is responsible for generating the opportunistic contacts. It has two major parts: interface discoverers and communication interfaces. The communication interfaces further comprise peer discovery, topology control, and transport services.

The communication capabilities of mobile devices vary at runtime—IP addresses are added and removed as the device connects to different Wi-Fi access points, and Bluetooth is toggled on and off as the user tries to save
energy, for example. Interface discoverers track these changes and detect when new capabilities become available. The figure shows an example of an IPv6 interface discoverer, which is implemented by periodically polling the underlying system for configured IPv6 addresses. When a new local address is found, the interface discoverer creates a new instance of an IPv6 communication interface with the new address (this is a Scampi internal “interface”, not to be confused with system interfaces).

Communication interfaces manage the various services needed to create opportunistic contacts, including peer discovery, topology control, and transport services. The example in the figure shows two peer discovery methods: 1) IP multicast based discovery, e.g., the DTN IP Neighbor Discovery protocol [96], which discovers peers by sending out periodic UDP beacons to an IPv6 multicast address. 2) Unicast TCP discovery, which discovers peers via direct connections to their known addresses. Other technologies, such as Bluetooth and Wi-Fi Aware, have their own discovery mechanisms that can be mapped into peer discovery modules.

After discovering peers, we need to decide which contacts to create—there may be a very large number of them, or the underlying technology might only support one concurrent connection, for example. These decisions are made by a topology controller attached to the interface. Scampi implements a full-mesh controller that connects all discovered peers, and a random-mesh controller that maintains a configurable number of random contacts—these are analyzed in more detail in section 3.2. Finally, the contacts are created by transport services, which are also bound to the interface (e.g., TCPCL server shown in the figure). Various other utility services bind to the interface (remote monitoring, logging, console, API, etc.), but are not shown in the figure for the sake of clarity.

This approach has proven to be quite flexible, allowing various communication technologies to be used in Scampi. These include the standard TCP/IP stack, Bluetooth on both Linux and Android, Android Wi-Fi tethering based communication (similar to WLAN-Opp described in section 2.1.2), and even an experimental implementation of the LTE D2D interface running on software-defined radios.

Storage Subsystem: Scampi persists messages in a storage subsystem with differentiated storage policies, as shown on the right of fig. 2.9. On a high level, the storage is split into three different caches: local persistent cache, local rotating cache, and peer cache. The local persistent cache behaves like database, allowing local applications to publish messages that
are guaranteed to be stored until explicitly replaced or deleted by the application. This form of cache is useful for data such as a social networking app profile, which the app will publish once but should remain permanently in the cache. Other messages published by local applications are stored in the local cache, and will be ejected based on cache management policies when the cache is full. Messages received from peers are kept in a separate cache so that a high churn from received messages does not push out the messages created by local applications.

Each cache has a configurable set of cache management functionalities. These include the queuing policy that ejects messages when the cache is full (e.g., first-in/first-out), enforcement of time-to-live limits on the messages, deduplication, and versioning. The underlying persistent storage is abstracted from these mechanisms, and can be anything from the file system to a relational database. By default, Scampi simply persists Bundle Protocol bundles into files on the file system (along with additional metadata), which enables an efficient zero-copy implementation of the message transport (e.g., Java `FileChannel.transferTo()` API).

**Routing Subsystem:** After opportunistic contacts have been created, it is the responsibility of the routing subsystem to use the contacts to exchange messages with the peers. The Scampi routing subsystem supports both push and pull based operation, with multiple concurrent routing and dissemination algorithms. The high level design is shown in fig. 2.10.

The routing subsystem is shown on the left of the figure, with its integration with the control subsystem on the right. The routing mechanisms are triggered by incoming content vectors (lists of messages that the peer
has in its storage) from the control subsystem. This vector first passes through a filter, which can drop entries to prevent their routing—this can be used, e.g., to filter out specific applications, or to only route messages from whitelisted peers.

Each routing algorithm then gets to give each message a priority value based on the importance of the message. These priorities get aggregated and passed to the control system, which then requests (pulls) the messages from the peer in priority order. The routing system can also push messages, to implement push-based dissemination algorithms. There is also a facility for the routing algorithms to access the control channel in order to do any out-of-band signaling (e.g., exchange Spray and Wait copy counts).

While this structure is flexible, it can also result in high complexity when multiple routing algorithms are used. Some routing algorithms may have conflicting goals, which requires careful configuration—e.g., if both an epidemic router and floating content are configured, then the epidemic router will request even messages that are outside the floating range. Often a configuration will have a form of epidemic routing with a low priority providing some baseline distribution, combined with a more intelligent algorithm with a higher priority.

2.3.3 Control Subsystems

In addition to the actual communications described above, routers require various control functions to organize and support their operations. In Scampi these include the identity subsystem that gives each router instance a unique identity, the control signalling subsystem that implements a signalling channel with connected peers, and the encounter context subsystem that tracks encounter histories and maintains social metrics.

**Identity Subsystem:** Scampi instances have unique identities that are not bound to any underlying communication technology. These identities are cryptographic, which allows the identity subsystem to implement certain security mechanisms tied to the node identity. By default, the identity is a locally generated RSA key pair, the public part of which is the identifier included in communications (e.g., as the sender of a message).

By using the cryptographic identity, nodes can sign messages generated by local applications, which allows recipients to verify that the messages are coming from the same origin. Nodes can also encrypt messages using the identity so that they can only be decrypted by the intended recipient. These identities could also be strongly bound to other identities. For exam-
ple, certificates could be used to tie the node identity to an organization or a person, and a common root of trust would allow every node to verify the identity. This could even be done by an external service, such as a social network, to allow the social network identity to be strongly bound to the identity of the user's Scampi instance—this, however, would require support by the external service providers.

**Control Signalling Subsystem:** Many mechanisms require control message exchanges between the connected peers. The way the Scampi router organizes this is shown on the right of fig. 2.10. Each connected peer (“neighbor”) has an associated *neighbor controller* instance, which serves as the endpoint for the signalling between the peers. The signalling is implemented as normal messages that are scheduled on the transport links along with normal traffic, but will not be persisted or forwarded by the communication subsystem. The scheduling mechanisms in the link multiplexer are used to give the control messages higher priority than content messages, and further logic could be implemented to pin the control signaling to a particular link for additional robustness.

As described above, one of the main functions of the control signalling is to exchange content vectors and request messages from the peers. In addition, the control signalling is used to establish the Scampi identity of the remote peer, if it cannot be established from the peer discovery signalling. Scampi nodes also use control signalling to roughly synchronize their time (which is a requirement for the DTN protocols), so that nodes without a real-time clock can take part in the system. Various other functionality also uses signalling, including localization that allows nodes to learn their geographic location even if they do not have localization hardware, and encounter history exchanges for transitive peer discovery.

**Encounter Context Subsystem:** The final major control functionality in Scampi is the maintenance of the encounter context, including deriving the social context used by various advanced routing and dissemination algorithms (see section 2.1.4). This is done by the encounter context subsystem, which is triggered by the peer discovery mechanisms when they detect a peer has been detected or has moved away. The subsystem records these events and calculates various metrics based on them.

The exact behavior of the subsystem is configurable. In the default configuration it only tracks the previous direct contact to each discovered node. This can be extended to tracking the historical frequency of contacts and the cumulative contact duration. Nodes that are not directly encountered
can be discovered by enabling transitive discovery control signalling, where nodes exchange their encounter information. Arbitrary metrics can be calculated by extension modules based on the basic data. One example of this is the AD-SIMPLE [54] community detection algorithm, which Scampi uses to classify the encountered peers that belong to the same community.

### 2.3.4 Application Interface

Scampi provides networking services to application via a topic-based publish/subscribe API, as shown in fig. 2.11. The applications send and receive Scampi message objects, shown on the left of the figure, which structure the content as typed key-value maps, instead of the more usual flat data—although the values can be arbitrarily large binaries, so applications may still do their own serialization if necessary. Applications may also tag the content with metadata descriptors, which are exposed to the subsystems for, e.g., making routing decisions. In basic operation, the topic is an opaque service name string, which is used to match publications to subscriptions. In more advanced operation, the topic is replaced with a search query and evaluated against the message metadata.

The Scampi messages used by the applications are mapped to Bundle Protocol bundles, as shown on the right of the figure. This is done by encoders that translate between the header data in the bundle primary block and the control fields of the Scampi message, the metadata tags in the message and bundle metadata extension blocks [255], and the Scampi content map and the bundle payload block. This means that the messages can be understood by any standard DTN implementation.
Although Scampi is implemented in Java, the application developers should be able to use other languages in various different platforms. We achieve this by defining the API as a local TCP protocol. This has the advantage that the client side of the API can be implemented in any language supporting TCP on any platform with a TCP stack—the downside is the increased implementation effort compared to, e.g., binding to a standard C API library. Scampi provides a Java implementation of the client API library, which can be used as-is on both JVM and Android.

2.3.5 Extensions and Utilities

As Scampi is focused on providing a platform for research prototypes and evaluations, it needs good support extensibility and data collection. The event-based design, subsystem modularity, many pluggable internal interfaces, and the dynamic loading facilities of Java are the basis for this extensibility. Some extensions, such as routing protocols, can be done simply by implementing a standard internal interface, compiling it into a Java class file, and configuring Scampi to dynamically load and instantiate it. This is used, for example, in the topology control work in section 3.2.3. Extensions that are wider in scope can be implemented as new subsystems. Examples of these include the mutable content in section 4.3 and the dynamic network creation of section 3.2.4.

The research focus also requires collection of runtime data, which can be used to analyze the system performance. Beyond the standard human-readable logs, Scampi produces runtime events in an easily machine-processable JSON format. These events include contact events, signaling events, message transmissions, application events, and performance counter update events. They can be saved to the file system, or streamed over a TCP link for central collection or real-time display. The testbed evaluation results in this thesis rely on processing of these event streams.

2.3.6 Summary

This section covers the high level structure and major design decisions of the Scampi middleware. It provides a flexible and extensible basis for research prototyping, while building on the DTN architecture and protocols. In the following chapters, we will explore numerous more advanced concepts of opportunistic networking, most of which have been prototyped using Scampi.
2.4 Conclusions

This chapter forms the foundation for the rest of the work in this thesis. It does so by establishing both the underlying concepts and the tools required by our research methodology.

We saw how opportunistic networking builds on a set of concepts that is distinct from classical packet-based networking. These concepts—human mobility, wireless device-to-device communications, and store-carry-forward messaging—can be used to build networks in challenging environments where classical networking is not feasible. A significant amount of research has been done in each of these concepts: Human mobility has been characterized both in terms of movement and the inter-person contacts that it generates, and various synthetic mobility models have been developed to approximate it in simulators. Various device-to-device communication technologies exist, and researchers have evaluated their use for creating opportunistic contacts—while a perfect solution does not yet exist, there are technologies that are feasible for practical implementations. Many opportunistic routing and content dissemination algorithms of various levels of sophistication and complexity have been proposed in the literature, and some implemented in practice. Again no single perfect solution exists, but many are good enough to serve as a basis for real systems, multiple of which have been proposed and built.

The contributions of this thesis to the body of work on opportunistic networking lie between these basic concepts and the practical applications. In particular, we aim to provide a richer set primitives on top of the basic store-carry-forward messaging that makes building opportunistic networking applications and services easier and more practical.

This chapter also develops the tools that we will need in the remaining chapters. The ONE simulator integrates mobility, wireless networking, store-carry-forward messaging, and applications into a single simulation environment. It allows us to build simulation models of the systems that we propose, and to evaluate their behavior and feasibility realistic scenarios with hundreds of simulated nodes. While simulations are valuable, proving practical feasibility requires real prototype implementations. We now also have the Scampi middleware as a tool that can be used to build and evaluate these prototypes on real devices.
Concepts of opportunistic networking
3. Creating Opportunistic Connectivity

The starting point for practical opportunistic networking is the creation of opportunistic contacts. As described in the previous chapter, opportunistic networking relies on transient peer-wise contacts between co-located devices created by wireless communication technologies. In principle, any two devices that can transmit and receive on the same radio frequency and are within close enough proximity can communicate with each other. In practice, however, most mobile devices are designed to connect to the Internet via dedicated links to infrastructure networks—Wi-Fi access points, cellular base stations—and have limited support for direct connections between the devices. Many technologies have been proposed and deployed for direct peer-to-peer wireless communications, such as Bluetooth, Wi-Fi Direct, Wi-Fi Aware, 802.11 ad hoc mode, but all have drawbacks that make them poorly suited for the kind of automatic, continuous contact creation required by opportunistic networking: Bluetooth can in practice only connect two devices at a time and the discovery process is energy hungry; Wi-Fi Direct requires a heavy setup overhead to create software access points in the mobile devices, and in many implementations (e.g., Android) requires manual user intervention; Wi-Fi Aware is faster at setting up connections, but does not support multiple concurrent connections well; 802.11 ad hoc mode is not exposed in the majority of available consumer devices. These problems motivate us to ask whether we can design practical mechanisms for automatically and autonomously creating contacts between mobile devices to support opportunistic networking.

In this chapter we examine three approaches to creating opportunistic contacts: 1) Using existing classical network infrastructure—Wi-Fi access points designed to provide Internet connectivity—to carry cross traffic directly between connected stations. 2) Creating ad hoc connectivity among co-located nodes by using direct communication capabilities between the
Creating Opportunistic Connectivity

1) Creating opportunistic connectivity devices. 3) Putting in place dedicated, light-weight infrastructure specifically designed to aid in opportunistic networking.

First, Publication II describes a measurement study conducted to establish the feasibility of using existing open Wi-Fi access points for creating opportunistic contacts. In this study, we systematically cover an area of downtown Helsinki and measure the ability of discovered open Wi-Fi access points to carry cross traffic between connected stations. The measurements are taken by two smartphones that run a custom communication test suite, which attempts to send various types of traffic between themselves via an open access point. We focus on studying the feasibility of fully automatable opportunistic contact creation without the need for human interaction, e.g., we do not enter any passwords or interact with any captive portals. The results show that the probability of success depends significantly on the particular protocols used to send the traffic, but that the probability of getting data through using at least one of the tested methods was high enough to make the approach practical.

Second, in Publication III we continue the study of using existing Wi-Fi access points by focusing on the scalability of the system when many nodes attempt to use access point cross traffic for opportunistic networking. For this, we create an experimental testbed with 50 small embedded Linux devices connected to the same Wi-Fi access point. The devices run our own Scampi opportunistic networking platform, which uses peer-wise TCP connections over the Wi-Fi access point to disseminate content between the devices. We find that a major limiting factor in the scalability of the system is the logical topology of the TCP links created between the devices. The simple full mesh topology—where every device is directly connected to every other device—scales very poorly and is not suitable for practical deployments. While the poor scaling of a full mesh was expected, it also has unexpected interactions with the forwarding process, leading to even worse performance than expected. However, we find that the scalability can be significantly improved by limiting the logical topology either randomly (every device connects to a random subset of other devices, creating a random topology) or in controlled way (every device connects to the same hub device, creating a star topology).

Also in Publication III, we propose an additional way to improve the scalability of the system by exploiting ad hoc connectivity directly between the devices. Instead of using the infrastructure access point to carry the cross traffic, we use the devices themselves to create transient
access points (often referred to as “softAPs”) to carry the traffic. This has two major benefits: 1) the opportunistic data traffic will be offloaded from the infrastructure access point and it will only be used for control signaling, and 2) we can exploit spectrum diversity by creating multiple transient access points on multiple different Wi-Fi channels. We present a divide-and-conquer algorithm for creating the transient access points and use simulations and testbed experiments to prove that it improves opportunistic content dissemination performance.

Finally, Publication IV explores using dedicated, light-weight, opportunistic infrastructure instead of exploiting existing networking infrastructure. We present our Liberouter neighborhood networking system, which combines cheap hardware (e.g., Raspberry Pi) configured to run as standalone Wi-Fi access points (without Internet connectivity) and our Scampi platform. The system is open and decentralized, allowing anyone to independently build and deploy Liberouter devices into the network. While this approach does require additional devices to be deployed, it has a number of benefits over the alternatives of using existing infrastructure or trying to create direct contacts between mobile devices: 1) Liberouters appear as standard Wi-Fi access points, which ensures seamless and automatic interoperability with all existing Wi-Fi enabled devices. In contrast, dedicated device-to-device standards, such as Wi-Fi Aware or Wi-Fi Direct, typically require special support from the hardware, drivers, operating system, and applications—and even when supported, often require active user interaction to use. 2) Liberouters run the opportunistic networking platform, storing and disseminating copies of the content to any connecting client device—while existing infrastructure only creates contacts without providing any opportunistic networking layer services. 3) Liberouters are long-lived and stationary, helping to disseminate content even in very low density deployments (i.e., when there are so few mobile nodes that they rarely come into direct contact with each other). In contrast, neither existing infrastructure nor direct device-to-device connectivity approaches work if the mobile nodes are never co-located. 4) Liberouters can bootstrap mobile devices into the network by presenting a captive Web portal for downloading the necessary opportunistic platform software—and they can be extended with further functionality to support “legacy nodes” (nodes that do not run the opportunistic networking software) as we will show in chapter 5.
Figure 3.1. Wi-Fi system model showing cross-traffic in a WLAN. The measurement study in Publication II evaluates the feasibility of using cross-traffic for creating opportunistic contacts between co-located mobile devices.

3.1 Cross Traffic in Wi-Fi Access Points: Publication II

Wi-Fi has been one of the most successful wireless communication standards for mobile devices and still continues to gain momentum. There were an estimated 169 million public Wi-Fi hotspots in 2018, and this is expected to grow to 628 million in 2023 [6], carrying over half of the smartphone data traffic (54–59% in 2017–2022) [7]. This makes existing Wi-Fi infrastructure an attractive choice for opportunistic networking.

The ubiquitous Wi-Fi infrastructure is built to connect mobile devices to the Internet, often implicitly assuming underlying client/server-based protocol designs (e.g., assuming that the mobile devices do not need public IP addresses). Typically, the Wi-Fi network operators do not design explicit support for direct peer-to-peer protocols, such as those used in opportunistic networking, and might actively seek to block it due to security concerns. In principle, however, the design of the Wi-Fi infrastructure enables an efficient packet path between two co-located clients via a single hop through the access point.

This leads to the research question that we address via a measurement study in Publication II:

- To what extent can we leverage the existing open Wi-Fi infrastructure to create opportunistic contacts between co-located mobile devices?
3.1.1 Public WLAN System Model

For the purposes of this study, we assume a simplified model of public WLAN infrastructure as shown in fig. 3.1—where the object of the study, cross-traffic, is shown in red. These deployments are designed to carry traffic between user equipment on the right of the figure to the Internet on the left. The minimal set of functionality provided by the infrastructure is the 802.11 medium access control and frame forwarding by the access point, autoconfiguration of the clients (via DHCP), and IP packet routing towards an upstream Internet service provider (ISP) by the gateway.

In the measurement study, we only consider open, public networks that do not employ any 802.11 security mechanisms for access control (e.g., Wi-Fi Protected Access, WPA) as these allow any nearby devices to connect and are therefore ideal candidates for generating opportunistic contacts. We also only focus on clients sharing the same physical layer, i.e., a single Basic Service Set (BSS) in the 802.11 terminology, and do not consider clients connected to different access points within the same WLAN (i.e., Extended Service Set, ESS). This limits the focus of the measurement study to physically co-located users, who are likely to connect to the same access point even in a multi access point network. We relax these assumptions in section 3.2 where the discussion on scalability is applicable to both access controlled networks and to larger WLANs composed of multiple access points interconnected by a distribution system.

There may be additional services provided from the local area network (LAN), most importantly captive portals used to authenticate and authorize clients before their packets are forwarded to the Internet. The typical technical implementation of a captive portal is to redirect HTTP requests of unauthorized users to a (logically if not physically) local web server serving the captive portal website. This means that the network must provide even unauthorized users with at least ARP, DHCP and DNS services in order for them to be able to access the captive portal. Further, importantly, the captive portal interactions typically happen on the LAN segment, above the link layer and outside the wireless segment, in principle making it possible for the access point to forward Layer 2 traffic between two clients regardless of their authorization status with the captive portal. Many public WLANs employ these portals due to, e.g., legal requirements or user tracking. They may require different levels of user interactions from a simple click to agree to terms, to multi-step account creation verified via
Creating Opportunistic Connectivity

SMS. While mobile operating systems and third party apps can automate the captive portal interactions, in the measurement study we do not use any such functionality or manually interact with the portals in any way. This is due to our use case of fully automatic creation of opportunistic contacts without the need or practical possibility for any user interaction.

3.1.2 Opportunistic Contact Model

The opportunistic contact creation process between devices has two main parts\(^1\): 1) *discovery* mechanisms that allow devices to find each other and to learn their transport parameters, and 2) *transport* mechanisms that move the payload data between the devices. Both mechanisms can be implemented by various different protocols, each of which may be impacted differently by the design decisions taken in a WLAN deployment—e.g., some might block all TCP traffic except a few well known protocols, and may do so either by simple port-based filtering or by complex deep packet inspection. Therefore, we test cross-traffic using a range of different mechanism for both discovery and transport.

The first step in the opportunistic contact creation process is for two nodes connected to the same Wi-Fi access point to become aware of each other and their transport layer services. The existence or peers and their services can be discovered in a single step or via separate steps using a variety of mechanisms.

The most popular approach to combined peer and service discovery protocols is to use a local IP multicast service to broadcast a discovery request to which peers will respond with the services that they provide (“solicited discovery”), and the peers may also periodically advertise their services (“unsolicited discovery”). Examples of widely used protocols that take this approach are Multicast Domain Name Service [75] with Domain Name Based Service Discovery [74] (mDNS/DNS-SD), and Universal Plug-and-Play [140] (uPnP), as well as the opportunistic networking specific DTN IP Neighbor Discovery [96] (IPND). *In the study, we measure the success rate of the popular mDNS/DNS-SD and the DTN specific IPND peer and service discovery protocols.*

Unfortunately, IP multicast over 802.11 Wi-Fi has many well known problems, leading to Wi-Fi network operators often blocking it in the access points [208]. For example, since the multicast frames are meant to

\(^1\)Between these two parts is a *topology control* step that decides to which peers to open transport links, which we will discuss in section 3.2.
be received by all stations simultaneously, including those furthest away with the poorest connectivity, the lowest common denominator modulation and coding parameters must be used. This not only means low throughput for IP multicast traffic, but leads to unfairness to unicast streams as the slow multicast streams occupy the physical medium for longer time periods \cite{76, 125, 93, 156}. Further, the simultaneous reception of multicast frames also prevents feedback from the stations as it would cause a reply storm, leading to high error rates due to lack of retransmissions—this can also compound the unfairness as the sender does not detect collisions and therefore cannot back off. Finally, multicast traffic requires special handling by the access point if any of the stations use the power saving mode. In such cases, the access point will buffer the multicast traffic and broadcast it intermittently (every few hundred milliseconds to a few seconds) leading to bursty delivery \cite{237}, which may interfere with the timing-sensitive discovery protocols. In the study, we measure the success rate of both IPv4 and IPv6 multicast using various different address spaces.

Because of the potentially widespread blocking of the IP multicast-based peer and service discovery protocols, other discovery mechanisms may be needed. To discover just the existence of potential peers within the same access point, without their capabilities, we suggest leveraging the Address Resolution Protocol \cite{212} (ARP). The protocol is critical to correct operation of basic connectivity and therefore likely to be well supported by access points. The functionality of ARP is very similar to the solicited discovery above: A client wants to learn a piece of information about a peer (the MAC address corresponding to an IP address), so it broadcasts a query into the local network segment, to which the peer with the given IP address responds. The packets sent by ARP are broadcast on the link layer similarly to multicast packets, but use the broadcast MAC address instead of the IPv4/IPv6 multicast MAC addresses, and are likely to be treated differently from IP multicast traffic. Nodes can use this for peer discovery by: 1) listening to ARP requests and responses (which carry peer IP and MAC addresses), and 2) periodically broadcasting unsolicited ARP response packets to make peers aware of them. The service discovery step can then be done using unicast mechanisms, i.e., via direct connections to the potential peers. In the study, we measure the success rate of ARP and unicast TCP echo between two stations.

The second part of opportunistic contacts is the establishment of a transport connection to move messages between the peers. These connections
are typically unicasting since each pair of devices will exchange different sets of messages, as governed by the opportunistic routing algorithm. The obvious choice is TCP as it is widely used for reliable and congestion-controlled delivery of byte streams between two devices over IP networks. However, since Web access—HTTP over TCP—is a primary use case for open access points, and captive portals are their primary means of authorization, it is possible that TCP traffic may be diverted to the captive portal rather than being delivered to the peer. Whether these diversions impact cross traffic depends on the exact implementation of the captive portal, of which there are many. For example, a popular approach is to use a local DNS server to resolve all domains to the captive portal web server IP, which will not impact any cross traffic since domain names are not used. In contrast, the access point could be filtering all TCP packets from unauthorized users, except those headed to the captive portal, which would prevent cross traffic. Moreover, a firewall that inspects traffic and redirects all unauthorized HTTP requests to the captive portal may or may not impact cross traffic depending on where in the network topology it is placed. In the study, we test TCP and HTTP/TCP to various ports without DNS resolution².
3.1.3 Measurement Methodology

We study the feasibility of cross traffic in a random sample of 50 open access points in downtown Helsinki (Figure 3.2). Two smartphones running a custom test framework connect to the access point under test and attempt to send various types of cross traffic between themselves (Figure 3.3). Based on the measurement results we then analyze the success rates for various parts of the opportunistic contact creation process.

We sample open access points by systematically walking in public areas in downtown Helsinki. Figure 3.2 shows the area covered by the sampling, which is focused around the busy commercial district. We choose this area because it has both high pedestrian traffic and many open access points offered by businesses, making it a good candidate for creating opportunistic contacts via cross traffic. The measurements are taken mainly from the pedestrian streets, with some inside shopping centers, as it allows us to evaluate the contact opportunities for mobile nodes. The sampling is done in a single day during working hours.

We develop a custom framework to automate running of the test procedures, shown in Figure 3.3, which minimizes the manual effort and standardizes the test procedure so that it is repeatable and the results are not impacted by differences in taking the measurements. The framework

---

Figure 3.3. Measurement process of the testing framework using two Android smartphones. Control signaling is done over a Bluetooth connection and measurements over a Wi-Fi access point.

---

2 We discuss an alternative worse-than-best-effort transport in section 3.1.5.
is an Android application that runs on two smartphones: a leader that makes all decisions, initiates all measurements and collects the results, and a follower that follows instructions from the leader. We use a Bluetooth link between the two devices for a persistent control channel that is not impacted by the Wi-Fi communications.

There are three phases to the measurements, which repeat indefinitely: Setup, measurement, and teardown. In the setup phase the leader selects an open access point to test and instructs the follower to connect to it. Once the follower has connected to the same access point, the leader will instruct the follower to start its side of the measurement modules (e.g., to start the TCP listening sockets) and passes the required parameters (e.g., which ports to listen on). When everything is set up on the follower side, the leader sets up its side of the measurements and the procedure moves to the measurement phase. In the measurement phase, the devices generate test traffic, e.g., by sending IP multicast packets or by attempting TCP connections. Both sides log the resulting events into log files, which are post processed offline to derive the experimental results. Once the leader has conducted all the desired measurements, the process moves to the teardown phase, where the leader instructs the follower to stop its measurement modules and stops its own modules. After the teardown is complete, the entire process can repeat from the beginning. During the test procedures, feedback on the status is given in the application GUI to the experimenter, which allows the experimenter to, e.g., avoid moving out of access point range during a measurement.

In total, we run 19 different test cases for each access point, including IP broadcast, IP multicast and TCP tests. Two test cases are for IPv4 broadcast, done by sending UDP packets to the local and subnet broadcast addresses. For IPv6 we test multicast to the link-local scope all nodes address [126], which replaces IPv4 broadcast. We further test IPv4 multicast to four address blocks: local subnet, internetwork control, AD-HOC, and administratively scoped [83]. And IPv6 multicast to five further address scopes: Realm-Local, Admin-Local, Site-Local, Organization-Local, Global [91]. In addition to IP multicast (UDP), we test TCP connections between the devices using four different ports: 80, 443, 8080, 4556. The three first ones are well known HTTP ports, while the last one is the TCPCL standard port [87]. In all cases we send HTTP requests as the payload traffic over the TCP connection, to see if a captive portal interferes. Finally, we have three tests that combine multiple protocols: 1) ARP re-
quest followed by TCP echo, which tests our proposed ARP based discovery with TCP connection. 2) Service discovery using DNS-SD over mDNS, which tests a popular and widely deployed service discovery protocol. 3) A combination of IPND and TCPCL, which are the standard DTN protocols for discovery and transport between DTN routers.

### 3.1.4 Results and Evaluation

The results of the measurement study are given in table 3.1 for 50 access points. The table includes both the measured success rates for the various tests, and the HTTP traffic capture rates for captive portals.

The majority of the results concern one-to-many communications—mostly used for the discovery portion of contact creation—with the success rates of individual mechanisms in the range of 28–40%. For IPv4 multicast, there is no difference in success rate when using arbitrary multicast addresses and protocols compared to using the standardized and widely-deployed

<table>
<thead>
<tr>
<th>Test case</th>
<th>Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPv4 broadcast: Local</td>
<td>30%</td>
</tr>
<tr>
<td>IPv4 broadcast: Subnet</td>
<td>32%</td>
</tr>
<tr>
<td>IPv4 multicast: Local subnet</td>
<td>30%</td>
</tr>
<tr>
<td>IPv4 multicast: Internetwork Control Block</td>
<td>28%</td>
</tr>
<tr>
<td>IPv4 multicast: AD-HOC block</td>
<td>30%</td>
</tr>
<tr>
<td>IPv4 multicast: Administrative scope</td>
<td>28%</td>
</tr>
<tr>
<td>IPv6 multicast: Link-Local</td>
<td>40%</td>
</tr>
<tr>
<td>IPv6 multicast: Realm-Local</td>
<td>40%</td>
</tr>
<tr>
<td>IPv6 multicast: Admin-Local</td>
<td>40%</td>
</tr>
<tr>
<td>IPv6 multicast: Site-Local</td>
<td>40%</td>
</tr>
<tr>
<td>IPv6 multicast: Organization-Local</td>
<td>40%</td>
</tr>
<tr>
<td>IPv6 multicast: Global</td>
<td>38%</td>
</tr>
<tr>
<td>ARP + TCP echo</td>
<td>30%</td>
</tr>
<tr>
<td>mDNS + DNS-SD</td>
<td>30%</td>
</tr>
<tr>
<td>TCP - port 8080</td>
<td>30%</td>
</tr>
<tr>
<td>TCP - port 4556</td>
<td>26%</td>
</tr>
<tr>
<td>IPND + TCPCL</td>
<td>20%</td>
</tr>
<tr>
<td>Completely open</td>
<td>16%</td>
</tr>
<tr>
<td>Some communication</td>
<td>48%</td>
</tr>
</tbody>
</table>

Table 3.1. Results of the access point cross traffic measurements.
mDNS/DNS-SD discovery protocol—all have roughly 30% success rate. Further, IPv4 broadcast and multicast both have the same roughly 30% success rate, which could indicate that the same technical mechanisms and policies are applied to block all of them. However, a detailed analysis of the logs shows that individual access points do not simply uniformly block all the broadcast and multicast traffic. Instead, we can find many different combinations of blocking particular broadcast and multicast mechanisms, including blocking everything, blocking only mDNS while letting all other IPv4 multicast traffic though, blocking some but not all address blocks, and blocking local broadcast but not subnet broadcast. This seems to indicate that the operators and the equipment manufacturers do not have a consistent policy towards IPv4 multicast.

Interestingly, many access points that block IPv4 multicast will still allow IPv6 multicast, with IPv6 multicast having overall 10% higher success rate. Since IPv6 multicast is not conceptually different from IPv4 multicast, blocking one but not the other is somewhat surprising. This could be due to technical deficiencies in the filtering technology used, e.g., lack of support for IPv6. It is also possible that IPv6 multicast is treated differently because the IPv6 standards make more widespread use of multicast than IPv4 does—e.g., DHCPv6 [185], IPv6 Duplicate Address Detection [256], and IPv6 Neighbor Discovery Protocol [190].

In addition to the multicast protocols that are mainly used for discovery, we test TCP for the transport part of opportunistic contacts. We find that 30% of access points allow cross traffic on the port 8080, which is an alternative HTTP port in the non-privileged port range. A further 12% of this traffic is captured by a captive portal, leaving 58% fully blocked. Since we do not interact with the captive portal in any way, it is possible that some of the captured connections would succeed after a simple user interaction (e.g., accepting usage terms). The success rate is somewhat lower (26%) when using the TCPCL assigned port 4556. However, there are very few cases of capture by a captive portal (2%) when using this port, leaving 72% of the traffic fully blocked. We do not test cross traffic in the usual HTTP ports 80 and 443 since listening on them is not possible on most consumer devices, as they require privileged access that most mobile operating systems do not give end users or their applications. We do, however, attempt to connect to those ports on the peer IP address to see if the traffic is captured by a captive portal. We find that 28% of attempted connections to the (non-listening) port 80 on the peer device get captured
by the captive portal, while only 4% traffic to the HTTPS port 443 gets captured—this makes sense, since the TLS security mechanisms should prevent a captive portal from capturing the sessions anyway. The return codes from the captive portal are 302, 307, and 404, but not the expected 511 code [193].

From the results we can see that no individual mechanism for discovery or transport has a success rate above 40%. And the success rate for an opportunistic contact creation using the IPND and TCPCL combination is only 20%—only slightly more than the 16% fraction of completely open access points. This seems to indicate that opportunistic communications will fail in 80% of the access points. However, the results also show that some form of communication is able to transmit bytes between the two devices almost half the time (48%). For example, we can improve the IPND+TCPCL success rate from 20% to 26% by directly scanning for the port 4556. Using the port 8080 instead of 4556 would further increase the success rate to 30%. And a fallback to multicast-based content dissemination instead of peer-wise contacts would get the success rate up to the 48%.

The major implication of the results is that randomly selecting an open access point and attempting to create opportunistic contacts with a single standard approach will have a low success rate. Instead, more intelligent contact establishment mechanisms are needed—for selecting the protocols and the access point. As shown above, by using a variety of mechanisms in different combinations, cross traffic is possible in half the open access points. A framework similar to Interactive Connectivity Establishment [152] (ICE) could be developed to systematically probe and set up the connection. This then leaves the issue of which access point to connect to, since the wide deployment of Wi-Fi means often there will be multiple options. If the contact establishment process can achieve a 50% success rate, it is feasible to simply randomly select access points until one with opportunistic peers is found—this should succeed in two tries on average. More complex decisions mechanisms could be designed to ensure that enough peers connect to the same access points to generate enough contact opportunities, while preventing too many nodes from connecting to the same access point and overloading it with cross traffic. In section 3.2, we present a scaling mechanism where the nodes use an infrastructure access point only to bootstrap a creation of separate ad-hoc access points to use for the actual contacts.
3.1.5 Interlude: Protecting Subscribers with Worse-than-Best-Effort Transport for Cross-Traffic

Cross traffic between two devices in Wi-Fi infrastructure mode is not sent directly between the devices, but rather has to be forwarded by the access point. This means that the cross traffic frames will transit the same wireless interface queue as normal subscriber frames coming from the Internet. When TCP is used for the cross traffic, it will attempt to fill the queue and to compete with the subscriber Internet traffic for its share of the bottleneck capacity. This interference could cause WLAN operators to attempt to block the cross traffic.

We propose to alleviate this problem by using a cross traffic transport protocol that will attempt to get out of the way of the TCP subscriber traffic rather than competing with it. Such transport protocols are called worse-than-best-effort [231], a popular example of which is Low Extra Delay Background Transport (LEDBAT) [235].

As an initial feasibility check, we conduct simple testbed experiments and simulations\(^3\), as show in fig. 3.4. The experiment setup, shown in the left of the figure, consists of five devices: A Wi-Fi access point serving three stations (AP), a ThinkPad Linux laptop as a cross traffic sender (A), an N900 Linux smartphone as a cross traffic receiver (B), a MacBook acting as an Internet subscriber (C), a local Linux server \((S_1)\), and a remote server \((S_2)\). In the experiment, subscriber C performs TCP uploads and downloads to \(S_1\) and \(S_2\), while A and B exchange cross traffic with TCP and

---

\(^3\)The experiment is in Publication II, our simulation study is reported in [163].
LEDBAT. The simulation scenario, shown in the middle of the figure, is similar to the testbed setup, but adds more realism by simulating mobility and modeling HTTP usage for the subscribers and bundle transfers for the cross traffic. Ten nodes generate cross traffic by exchanging bundles while moving according to the random waypoint model. For the cross traffic, we evaluate TCP, LEDBAT, and a fair variant of LEDBAT (fLEDBAT) that attempts to solve a late-comer advantage problem in the standard LEDBAT. Five stationary nodes generate subscriber traffic by making HTTP requests to a server.

The testbed experiments show that the mean bit rate of the subscriber TCP connection to the local server (C to S₁) drops by 5–20% when TCP cross traffic is generated between A and B. When using LEDBAT for the cross traffic, the subscriber connection to the local server is largely unaffected. The behavior is somewhat different for a remote server (C to S₂): The reduction in bitrate for the subscriber is only 5–10%, and this is roughly similar regardless of whether the cross traffic uses LEDBAT or TCP. In both cases we confirm that LEDBAT yields to the subscriber traffic, but the degree to which it yields is dependent on the rate of the subscriber TCP stream—the higher bitrate C–S₁ connection causes LEDBAT to yield more than the lower bitrate C–S₂ connection. While these are simple qualitative results, they do point to the desirability of a cross traffic transport protocol that yields to subscriber traffic.

The simulation study shows the impact of cross traffic transport choice more clearly. The results on the right of fig. 3.4 show how the page retrieval time by the subscribers changes as a function of increasing TCP, LEDBAT and fLEDBAT cross traffic. When the cross traffic uses TCP, the impact on the subscriber traffic is substantial, with the web page retrieval time rapidly increasing (solid red line). Using LEDBAT instead of TCP for the cross traffic significantly cuts the impact on the subscriber traffic (purple dotted line), while fLEDBAT shows almost no impact on the subscriber website retrieval latency (blue dashed line)—despite this the average cross traffic throughput for all three protocols is almost identical.

Overall, these results show that using a worse-than-best-effort transport, e.g., LEDBAT, has the potential to protect the subscriber traffic from the cross traffic. This is important to prevent access point operators from blocking cross traffic if opportunistic networking becomes widely deployed.
3.1.6 Summary

The research question posed at the beginning of this section was to determine to what extent can existing open Wi-Fi infrastructure be leveraged to create opportunistic contacts. The results show that in principle around 50% of the open access points can be used to transport cross traffic in some way. However, various types of blocking are widespread and intelligent mechanisms for creating opportunistic contacts are required—using a single standard approach is likely to lead to an unreasonably high failure rates. We further propose that cross traffic should use worse-than-best-effort transport in order to not interfere with regular subscriber traffic and thus create an incentive to increase the blocking. In the next section we explore mechanisms needed to scale opportunistic communications when a large number of devices are attempting to use the same infrastructure access point for cross traffic.
3.2 Adapting to Dense Wi-Fi Networks: Publication III

The approach of using Wi-Fi infrastructure for cross traffic—studied in section 3.1—presents two scalability challenges when a large number of devices are connected to the same wireless network segment: 1) Each node has a large number of peers that it could potentially create contacts with—creating all contacts results in $O(n^2)$ links in the topology. 2) The increasing amount of cross traffic will consume transmission capacity from normal consumer Internet traffic. We formulate these as two research questions for Publication III:

- **How should we choose which contacts to create in a dense environment with a large number of potential communication partners?**

- **Can we move opportunistic cross traffic away from the infrastructure access points so that it does not interfere with subscriber traffic?**

In this section, we design and evaluate mechanisms to answer both of these questions. First, we evaluate and propose topology control algorithms that reduce the number of transport connections between the peers by selecting only a subset of potential opportunistic contacts to create. Second, we present a divide-and-conquer algorithm for moving opportunistic contacts from an infrastructure access point into ephemeral software access points created by the devices themselves.

3.2.1 Scenario Model

Our focus in this section is on dense network segments, by which we mean link layer domains that contain at least tens, but potentially even hundreds, of nodes. In such dense network segments, a local discovery mechanism (such as mDNS) will find a set of peers that is so large that it is infeasible to create parallel contacts between all of them at the same time. The obvious question then is which contacts to create and when. These conditions occur naturally in open wireless networks as users cluster in areas covered by wireless access points—as human mobility is characterized by long periods of time spent in dense clusters with infrequent flights between the clusters [114, 220, 166].

Figure 3.5 illustrates the model in three layers and in three discrete points in time. For the link layer, we assume an 802.11 infrastructure
Creating Opportunistic Connectivity

Figure 3.5. Scenario model for dense network segments. This section studies the logical topology shown in the blue middle layer.

mode network where multiple mobile stations are attached to an access point. The wireless segment could be a single access point (similarly to section 3.1 and as shown in the figure) or there could be a distribution system connecting multiple access points into a single domain—the important factor is mutual discoverability via a local discovery mechanism. This results in a star-like link layer topology, where every station is connected to a central access point. Nodes in the network create opportunistic contacts by opening TCP streams between each other. These TCP connections form an arbitrary logical topology, independent of the link layer topology, which we can control via an algorithm that decides when a node opens TCP connections to other nodes. Finally, the content (e.g., Bundle Protocol bundles) are forwarded between peers using the TCP contacts. This content routing layer could use arbitrary opportunistic routing or dissemination protocols.

In contrast to section 3.1, where we use a simplified model of opportunistic contacts that only considers discovery and transport, here we use a realistic contact model. The model includes: 1) peer discovery, 2) topology control decisions, 3) TCP link creation between peers, 4) control messaging including content vector exchanges, and 5) content exchange.

Contact creation starts with a node using an IPv4 multicast-based discovery mechanism to find directly reachable peers. The discovering node will then run a topology control algorithm that decides which contacts to create by opening TCP connections. We consider a full mesh topology where every node opens every possible link, a limited mesh topology where nodes open a limited number of links (“n-link limited mesh”), and a star topology where all nodes open links only to a designated hub node. Nodes use the links for control signaling to exchange content vectors that describe which content they have, and to request copies of content from their
peers. The peer transmits content items requested from it in the order it received the requests. This content vector exchange, content request, and content transmission process continues for as long as the link is up. The selection of the content to request is done by a routing algorithm. In our case nodes request all contents from the peers, resulting in epidemic dissemination—this serves as a baseline that more advanced routing or dissemination algorithms may improve upon.

We focus on a scenario where a node connects to an access point with a large number of stations. The entering node carries content items with it into the network segment, while rest of the nodes have empty caches. The content items will then spread to the already connected nodes epidemically via a process that is mainly determined by the logical topology (as fig. 3.5 visualizes). We study the impact of different logical topologies on the spreading process in order to understand how this model of opportunistic networking scales in dense environments.

### 3.2.2 Testbed Setup and Methodology

We realize the scenario described above with a testbed setup that is made up from 50 embedded Linux devices connected to a Wi-Fi access point. The nodes run the Scampi router (see section 2.3), which implements the contact creation and content routing processes as described in the scenario model. This creates a realistic real-world environment including physical, transport, network and application layers on commercial off-the-shelf hardware and software.

Figure 3.6 shows the system model and the physical realization of the testbed. The experimental system has four main components: 1) a Wi-Fi
access point, 2) a device grid, 3) a local services node, and 4) an experiment controller. The devices in the grid connect to the access point to create a dense wireless segment. Local services, such as DHCP and NTP, are provided by the local services node, which is connected to the access point via a wired connection. The experiments are run by a custom testing framework in the experiment controller, connected to the access point. Since the grid devices only have a single network interface, the controller must do control signaling over the same network as the experiments. To minimize the impact of this on the results, our testing framework sets up the experiment and then avoids any signaling during the run time.

The right side of fig. 3.6 shows the physical hardware and setup of the testbed. The testbed is portable, and we conduct experiments both in an office environment and in a radio frequency-shielded room to see the impact of interference from other networks and devices. We use Intel Edison computers-on-modules with Intel breakout boards as the grid devices. They have dual core Intel Atom x86 processors running at 500 MHz, 1 GB of RAM, 4 GB of flash, and an 802.11n wireless modem running at both 2.4 GHz and 5 GHz bands—roughly equivalent to a smartphone at the time. They run a custom operating system based on the Intel Yocto distribution (ww05-15, Linux 3.10.7), from which we remove various components not needed for our tests, but which is otherwise close to standard. We use Scampi as the opportunistic networking layer, which runs on a Java 8 virtual machine in each Edison, uses a custom IPv4-based discovery mechanism and TCPCL [87] for transport. The access point is D-Link DWL-6600AP, which can support 50 associated stations simultaneously.

The procedure for the experiments is carried out by a custom test framework running in the controller: First, a random subset with the desired number of nodes is picked from the grid—this prevents systematic errors due to differences in performance of the different devices (which can be significant). Then, the initial state of the nodes is set up by clearing the Scampi caches, synchronizing the clocks, and copying over the configuration files for the experiment. Next, another random device from the grid is picked as the arriving node that carries new content into the dense segment. It is walled off from the testbed with local firewall rules and its Scampi cache is populated with content. The experiment is started by disabling the firewall rules, which allows Scampi in the node to discover and connect to the other devices in the testbed. Since we do not want to send any experiment control signaling traffic during the experiment,
we instead monitor the power use of the nodes in the grid, and stop the 
experiment when the power use has fallen back to the idle level. Finally, 
we collect the logs from the devices for off-line analysis.

### 3.2.3 Controlling the Logical Topology

The spreading of the content within the dense network segment is largely 
determined by the logical topology (as illustrated in fig. 3.5), which we can 
control with algorithms that decide which contacts to create. Figure 3.7 
shows three types of logical topologies: *full mesh*, *limited random mesh*, 
and *star*. The first of these, full mesh, results from every node opening 
every possible contact. We use this as the baseline, as it is likely to be the 
default behavior of opportunistic routers in the absence of more advanced 
topology control functions. The second case, limited random mesh, is 
a straightforward extension of a full mesh, where every node opens a 
limited number of links to random peers. The main tuning parameter for 
the limited mesh is the number of links each node opens—we call this 
parameter $N$, and the resulting topology *$N$-link limited random mesh*. 
Both of the previous topologies can be created by a purely local topology 
control algorithm without any coordination between the nodes. Distributed 
algorithms that do coordinate between the nodes can be used to build more 
organized topologies. The figure shows a simple example of this, the star 
topology, in which the nodes all agree to connect to the same hub node, 
which will then route all the traffic between the nodes.

To establish the baseline, we pick $N - 1$ nodes ($5 \leq N \leq 25$) connected 
in a full mesh topology with empty caches. We then have the $N^{th}$ node, 
carrying a single 10 MB message, open contacts to all the other nodes 
to transmit the message. The time it takes for the message to spread
to all the other nodes is shown in the top-left of fig. 3.8 as the circled points. If we assume each of the $N - 1$ nodes receives a single copy of the message, then the effective peer-wise transmission rate achieved is 5–6 Mbps in the office environment, and a few Mbps higher in the shielded room. This is an order of magnitude less than we would expect for a typical 802.11n network [240]—the available transmission capacity is either not fully utilized or it is being wasted on overheads.

We can see the reason for the low goodput in the top-right of fig. 3.8 (circled points), which shows the number of message transmissions during the experiment. The assumption that each node only receives the message once does not hold—e.g., in the case of 25 nodes, we would expect 24 transmissions but instead see over 100 in the graph. These unnecessary transfers of the message are taking the majority of the available transmission capacity.

More careful analysis of the logs reveals that the cause of the extra transmissions is a phenomenon we call request stacking: When the node
with a new message enters the network, it will will start concurrently transmitting copy of its message over all the contacts—in the case of full mesh topology to all the other nodes in the network. When the first of these transfers finishes, the receiving node makes it available to all of its peers (i.e., all the other nodes). In an aggressive strategy, all the peers will then request the message from the new node even though they have an ongoing transfer of it from the original node. The process repeats for every node that receives the message, resulting in a large number of redundant transfers. This aggressive strategy of trying to get a message over every possible contact makes sense in highly dynamic networks where the contacts may break at any time, but causes many unnecessary transfers.

The number of unnecessary transfers due to request stacking is limited by the logical topology since each node will advertise its messages only to its direct peers—the fewer peers the node has, the fewer transfers it can start. The baseline full mesh topology is the worst-case scenario, where every node will start transfers to every other node. If we use a limited random mesh instead of a full mesh, we expect to see fewer duplicate transmissions. This is in fact the case, as the top-right of fig. 3.8 shows (crosses and triangles). Using a 1-link limited random mesh (crosses in the figures) results in almost exactly the expected number of transmissions, with goodput between 20–30 Mbps (top-left of the figure)—still in the low end of what is expected, but explainable by inefficiencies in the opportunistic router implementation. Adding more links to the mesh increases the number of transmissions and correspondingly decreases the goodput. For example, going from 1-link to 2-link mesh almost doubles the number of transmissions. When viewing the spreading process over time (bottom-left of fig. 3.8), we can see that it follows the same pattern—at every point in time the 1-link limited mesh has made the most progress and the full mesh the least, with the 2-link and 3-link cases in between.

The star topology (right of fig. 3.7) is an interesting case where the topology prevents duplicates due to request stacking—every node can only request content from the hub node and nobody else. Our experiment shows that the star topology synchronization performance is consistently between the 1-link and 2-link limited mesh topologies (center-left of fig. 3.8). Since the star topology has no duplicate transmissions, its slower synchronization time compared to 1-link limited mesh cannot be explained by request stacking. Instead the difference is likely caused by the different pattern of transfers causing different congestion characteristics in the medium...
access and transport layers. While the 1-link limited mesh has only a few parallel transfers as the content spreads hop-by-hop along the topology, the star topology will start all the transfers in parallel. This means that all the nodes are competing for access to the medium at the same time, and multiple flows are competing for the queue slots in the access point.

We can look at a more realistic case where the arriving node has multiple content items to disseminate into the dense segment. The basic behavior (center row of fig. 3.8) is the same as with a single content item—the smaller the mesh, the faster the dissemination. However, the spreading behavior is different (bottom-right of fig. 3.8). The almost vertical lines show that the dissemination process finishes almost at the same time in all the nodes, while in the single message case the finishing times are more spread out. This makes sense since, in the multi-message case, the finishing time is the sum of the finishing times of the individual small messages, which has a lower variance than the finishing time of a single large message. The final figure also illustrates an important downside of the star topology: There is a discontinuity in the graph for the star topology. This corresponds to a freeze in the entire dissemination process due to the hub node getting disrupted for some reason (e.g., a long garbage collection pause), which highlights that the hub node is a single point of failure.

### 3.2.4 Exploiting Spectrum Diversity

So far we have seen that controlling the logical topology can lead to good opportunistic content dissemination performance in dense wireless networks, and that LEDBAT can be used to reduce its interference on subscriber Internet traffic. However, even with these mechanisms, the opportunistic contacts can only use a small part of the available communication resources—they are limited to the single Wi-Fi channel of the access point, where they compete with the subscriber Internet traffic and with other access points on the same channel. In other words, the opportunistic contacts can only use 20 or 40 MHz of the up to 745 MHz of potential bandwidth on the ISM bands. We propose to leverage the ability of smartphones to act as Wi-Fi access points (“softAP”) to spread the opportunistic contacts over more Wi-Fi channels than the one used by the infrastructure access point.

The basic idea is as follows: When a node enters a dense segment with new content, instead of trying to transmit it to every peer, it will pick some subset of the peers as *leaders* and transmits the content only to them. Further, it divides the rest of the peers as *followers* and assigns them to
the leaders, along with a set of resources (e.g., Wi-Fi channels) that each leader can use. The leaders will then create an access point on the assigned channel to which their assigned followers will connect. If a leader has more than a single resource, the process can be applied recursively.

In a practical system with multiple nodes wanting to disseminate their content, an election mechanism could be used to assign the leader. The process could also be applied to multiple node groups in parallel if different nodes are interested in different content. Further, if the devices support multiple virtual Wi-Fi interfaces (see section 2.1.2), the whole process could be run on a secondary virtual interface.

Figure 3.9 shows an example of this process with 28 nodes \((c_1, \ldots, c_{28})\), one infrastructure access point on channel 1, and five additional Wi-Fi channels to be used \((6, 11, 36, 40, 44, 48)\). Initially, all nodes are connected to a single infrastructure access point on channel 1. The initial leader \(c_1\)—the node with content to disseminate—chooses \(c_5\) and \(c_{17}\) as leaders and assigns them to channels 6 and 11 respectively. Clients \(\{c_6, \ldots, c_{16}\}\) and additional Wi-Fi channels \(\{36, 40\}\) go to \(c_5\), clients \(\{c_{18}, \ldots, c_{28}\}\) and channels \(\{44, 48\}\) go to \(c_{17}\) (two top arrows in the figure). The leader \(c_5\) and \(c_{17}\) then apply the same process recursively, each splitting off two leader-follower groups into their two additional Wi-Fi channels (lower arrows in the figure). The figure shows the end result with seven channels, each with a leader and three followers. After each leader has disseminated the content to its followers, the nodes can connect back to the original network.

There are multiple possible failure modes for this process. For example,
Creating Opportunistic Connectivity

an elected leader may fail to create the network or leave during the dissemination process, or a client might not be able to use the frequency range assigned to it. However, as a general failure recovery strategy, the nodes can always attempt to connect to the previous leader in the chain, all the way back to the original infrastructure access point—any active leader will be able to serve them. This will likely lead to suboptimal resource use, but will still be an improvement since more resources will be used for the dissemination process overall.

We can formalize this idea as a recursive divide-and-conquer algorithm as shown in algorithm 1. The algorithm, executed by a leader, is split into two phases: 1) breaking off a subset of clients to be served by others (lines 7–22), and 2) serving the content to local clients (line 24). In the beginning of the first phase (lines 8–13), the leader first splits the followers and resources into $k$ subsets according to the branching factor of the recursion ($k = 2$ in fig. 3.9). The leader then activates each subset by picking a leader and calling the algorithm recursively (lines 15–17), and by instructing the followers to connect to their new leader (lines 19–21). Finally, the leader disseminates the content to its own followers (line 24) and finishes.

To test the idea in practice, we implement the algorithm in the Scampi router and conduct a testbed experiment. In the experiment we use the three non-overlapping 2.4 GHz band Wi-Fi channels (1, 6, 11) and 12 devices. This splits evenly into three test cases: 12 devices on channel 1; 6 devices on channels 1 and 6 respective; and 4 devices on channels 1, 6, and 11. The twelve nodes start on the channel 1 and an additional node carrying content enters the wireless network and starts the process as the initial leader. The experiments are conducted in an office environment with uneven congestion on the different channels.

The left side of fig. 3.10 shows the dissemination process over time for the three test cases. We can see that using two channels instead of only one halves the dissemination time from about 300 seconds to 150 seconds (triangles and circles in the figure). The figure shows two unexpected features: First, examining the 2-channel graph more closely, we can see an abrupt change in the curvature between time 100 and 120 seconds. Second, adding a third channel (crosses in the figure) has very little impact in the total dissemination time (6 second improvement). Both of these are caused by the work being split sub-optimally between the channels. In the 2-channel case, the abrupt change is caused by the first leader becoming idle while the second leader is still disseminating content. In the 3-channel
Algorithm 1 Divide-and-Conquer

\begin{algorithm}
\begin{algorithmic}[1]
\Procedure{NODE.DISTRIBUTE}{$R, S_m, S_c, S_R, F$}
\State $\triangleright R$: resource to be used (e.g., Wi-Fi channel)
\State $\triangleright S_m$: set of messages to disseminate
\State $\triangleright S_c$: set of clients to which to disseminate the messages
\State $\triangleright S_R$: set of additional resources (e.g., Wi-Fi channels)
\State $\triangleright F$: branching factor for the recursion, $F \in \mathbb{Z}^+$
\State $\text{T.HI.S.ACTIVATE}(R)$  \Comment{Activate the resource, e.g., start SoftAP}
\State $S_f \leftarrow \text{SELECTFOLLOWERS}(S_c), S_f \subseteq S_c$  \Comment{Served by other leaders.}
\State $S_c' \leftarrow S_c \setminus S_f$  \Comment{Served by this node.}
\State $\triangleright$ Pick the number of sub-problems for recursion.
\State $k \leftarrow \text{BRANCHINGFACTOR}(F, |S_R|, |S_f|), k \in \mathbb{Z}^+, k \leq |S_R|, k \leq \frac{|S_f|}{2}$
\State $S_{R1}, \ldots, S_{Rk} \leftarrow \text{DIVIDERESOURCES}(S_R, k), S_{Ri} \neq \emptyset \forall i \in \{1, \ldots, k\}$
\State $S_{f1}, \ldots, S_{fk} \leftarrow \text{DIVIDEFOLLOWERS}(S_f, k), \bigcap_{i=1}^{k} S_{fi} = S_f$
\For{$i \leftarrow 1, k$}
\State $s \leftarrow \text{PICKLEADER}(S_{fi}), s \in S_{fi}$
\State $r \leftarrow \text{PICKRESOURCE}(S_{Ri}), r \in S_{Ri}$
\State $\text{S.DISTRIBUTE}(r, S_m, S_{fi} \setminus \{s\}, S_{Ri} \setminus \{r\}, F)$  \Comment{Connect clients to the new leader.}
\ForAll{$c \in S_{fi} \setminus \{s\}$}
\State $\text{C.CONNECT}(s, r)$  \Comment{E.g., SSID and Wi-Fi channel}
\EndFor
\EndFor
\State $\triangleright$ Directly disseminate the content to the remaining clients.
\State $\text{DISSEMINATE}(S_m, S_c')$
\EndProcedure
\end{algorithmic}
\end{algorithm}

Figure 3.10. Divide-and-conquer experiment results.
case, we can see that the last three data points—corresponding to the third leader—have high variance and seemingly slow down the dissemination while the two other leaders are idle. We can see the cause more clearly by looking at an example.

A representative example of the dissemination process is shown on the right side of fig. 3.10 for the case with three channels. We can see that Leader 1 takes 15 seconds to send the content to Leader 2, after which it takes 9 seconds for the leader to start dissemination, and an additional 37 seconds until all its followers have received the content. This 9 seconds includes the time it takes for the leader to create the access point, for the three followers to connect to it, and for Scampi to discover and start the transfers to the followers. For Leader 3, both the activation and transfers take roughly twice as long (19 and 77 seconds respectively). This is caused by the channel 11 used by the Leader 3 being heavily congested compared to the other channels—in the example the average transmission speeds for channels 1, 6, and 11 are 3.1 MB/s, 2.7 MB/s, and 1.3 MB/s respectively.

The widely varying channel performance has a significant impact on the efficiency of the entire process. In the optimal case all the leaders finish their dissemination at the same time—otherwise a leader that finishes early could have been given more followers to serve, causing the entire process to finish earlier. In the example case all the leaders finished at different times, with Leader 2 finishing already halfway through the process. For this case the optimal split would have been 5-4-1 (vs. 4-3-3) followers to the leaders 1-2-3 respectively, which would have resulted in the process completing after 111 (vs. 128) seconds, with all three leaders finishing within 20 (vs. 67) seconds of each other. In principle, this can be done in algorithm 1 with a DIVIDEFOLLOWERS() subroutine (line 13) that takes into account the varying performance of the channels.

To understand the behavior of the algorithm with a larger number of clients and channels we conduct a simulation experiment. Figure 3.11 shows the results of a simulation with 500 nodes and up to 22 channels. The simulation implements algorithm 1 and models both the leader creation process and the content dissemination process in a simple discrete event based simulator. It does not model any physical, link, or transport layer behavior and assumes fixed channel capacities. The simulator includes an optimizing mode that finds the optimal assignment of followers for each leader.

The left side of fig. 3.11 shows the dissemination process over time for the
cases with 1, 2, 3, 7 and 22 channels and optimal split of followers to each channel. The single channel case takes 500 seconds (squares, not visible in the figure), while adding another channel halves the dissemination time as expected. Adding more channels further improves the performance, with the realistic case of seven channels—the three non-overlapping channels in 2.4 GHz band and the four widely supported channels in the 5 GHz band—taking only 15% of the time a single channel would take.

The left side of the figure also shows the performance of a hypothetical case where the leaders activation would take no time, which puts an upper bound on the achievable performance (marked “max” in the figure). We can see that this upper bound is very close to the results, with cases with larger numbers of channels being further away from a bound. This shows that the overhead of the recursive leader creation process (lines 8–22 in algorithm 1) is negligible with a realistic number of Wi-Fi channels.

An important finding from the experiment was that the split of followers to the different leaders according to the relative channel capacity is critical for the performance of the algorithm. While the channel capacities are static and equal in the simulation, and hence do not exhibit this behavior, we can study another source of imbalance between the leaders—the time it takes to activate a leader (marked “Leader N received/start” in the right side of fig. 3.10 for the testbed experiment).

Since the leader can only start serving its followers after it has been activated, and the leaders are activated sequentially, those leaders that get activated later should have fewer followers assigned to them. The right
side of fig. 3.11 shows the optimal split of followers to channels for different activation delays. In the case of static and equal channel capacities and no leader activation delay, the optimal split of followers to leaders is roughly even. However, a realistic 10 second leader activation delay causes the optimal follower split to deviate significantly, such that the last leader should serve only 40% of the followers of the first leader to achieve optimal performance. With 20 second leader activation delay (which was observed in the experiment), it is optimal to not activate the last 7 out of 22 leaders at all. This effect, along with the uneven channel capacities, needs to be taken into account when designing the `DIVIDEFOLLOWERS()` subroutine of the algorithm in order to achieve optimal performance.

### 3.2.5 Summary

In this section, we posed two research questions arising from the desire to scale opportunistic communications to large numbers on nodes in the same wireless access points. The first question is how should we choose which opportunistic contacts to create when there are a large number of potential peers. Based on the experiments, there are two viable options: 1) each node creates a contact to one random peer (1-link random limited mesh topology), or 2) each node opens a contact to the same hub (star topology). Both of these limit the number of links in the topology, which reduces the request stacking phenomenon and number of competing transport flows through the access point. The first can be created without any coordination, while the latter requires coordination to choose the hub. The second question is whether we can exploit spectrum diversity (the many available Wi-Fi channels) to move traffic completely away from the infrastructure access points. For this we present a divide-and-conquer algorithm that moves the traffic from the infrastructure access point to softAPs created by the smartphones themselves. We show that this approach is capable of not only protecting the subscriber Internet traffic in the wireless network, but also cutting the total dissemination time to a fraction of what it would be if only using the access point. These two mechanisms are likely to be crucial for enabling large scale opportunistic networking in practice.
3.3 Neighborhood Networking with Liberouters: Publication IV

So far we have seen that opportunistic contacts can be created using existing open Wi-Fi infrastructure and improved by using the softAP capabilities of the smartphones themselves. There are, however, two significant issues with this approach: First, the existing Wi-Fi infrastructure only provides packet forwarding services. It cannot, for example, provide message caching services at the opportunistic networking layer, or bootstrap new devices by allowing them to download the necessary router software. Second, at least two devices must be co-located in the same access point at the same time for contacts to be created—and without contacts the system does nothing. This creates a bootstrapping problem where a critical mass of users must be created before the system will function.

One potential way to solve these problems is to create dedicated infrastructure to support opportunistic networking. We formulate this as a research question for Publication IV:

- Can the use of dedicated lightweight infrastructure components offer a practical basis for opportunistic networking?

Our proposed solution is called the Liberouter—a combination of cheap hardware (Raspberry Pi) and the Scampi router that acts both as an open Wi-Fi access point and as an opportunistic networking router. The Liberouters do not provide Internet connectivity, but do allow any connected Android device with Scampi to connect and exchange messages with them. The devices provide further services, in particular an opportunistic app store accessible with a web browser, from which connected devices can download new opportunistic networking applications (including the Scampi router itself). Stationary support nodes in opportunistic networking are often referred to as “throwboxes” or “infostations” [42, 43, 115, 278].

In this section we present the system design for Liberouter based neighborhood networking, followed by description of its implementation. We also study the performance of an opportunistic network composed of stationary Liberouters and mobile smartphones via simulations. We further extend the capabilities of the Liberouter devices in chapter 5 by adding the ability for them to execute opportunistic applications.
3.3.1 Liberouter System Design

The Liberouter system implements a concept we call *neighborhood networking*. We loosely define a neighborhood as a set of users with shared interests—and hence a desire to communicate—that frequent the same physical locations and often come into range of the same wireless networks. Examples of such neighborhoods are students in a school campus, residents on the same street, shoppers downtown, and visitors of a camping site. The goal is to connect a neighborhood and enable the users to communicate with each other. The goal is not to connect the neighborhood to the Internet, but rather to provide a parallel, local network for the members.

Figure 3.12 shows the high level system model for a Liberouter neighborhood network. The system is composed of islands of local wireless connectivity (blue in the figure) created by the Liberouter devices acting as Wi-Fi access points. The opportunistic routing layer, in this case the Scampi router, uses the wireless connectivity to create contacts between the Liberouter and connected smartphones (shown on the left side of the figure). Scampi will then synchronize the buffers of the Liberouter and the connected smart phones, allowing the users to pick up and drop off messages. For smartphones that do not have the Scampi router, and therefore cannot create opportunistic contacts, the Liberouter provides a web portal accessible via a standard browser.

This proposed model of networking with stationary Liberouters and mobile smartphone users has a number of desirable properties. First, since
all contacts are between a Liberouter and a smartphone, we do not require
the mobile devices to be co-located and to create direct contacts. This solves
the critical mass problem since just two users from the same neighborhood
that never meet directly can still communicate via a Liberouter—the
minimum size of a useful system is one Liberouter and two users with
smartphones. Second, while each Liberouter can function independently,
the users can act as carriers that spread messages between multiple
independent liberouter instances. This means that users do not need to
connect to the same Liberouter in order to exchange messages as long
as some set of carriers are ferrying messages between the Liberouters.
Potentially this inter-Liberouter distribution could also be done by a well-
connected backhaul, e.g., the Internet or direct wireless point-to-point
links, but such functionality is outside of our scope here. Finally, the
resulting system is highly distributed in nature. Copies of the content
are potentially stored in multiple Liberouters and smartphones, making
the system robust against data loss. No centralized coordination, control
or management of the system is required, instead anyone can deploy a
Liberouter device, which will automatically become a loosely coupled part
of the neighborhood network with any other deployed Liberouters.

The last major component of the Liberouter system is the web portal
and app store functionality. When a user connects to the Liberouter Wi-Fi
network and opens a standard web browser, a captive portal intercepts
the traffic and displays a web portal page that explains the system. The
page also includes an app store where the user can download the Scampi
router, allowing their smartphone to become a part of the opportunistic
network. This then allows the Liberouter network to bootstrap new devices
completely independently of the Internet or app stores controlled by the
device or operating system vendors.

3.3.2 Liberouter Implementation

We implement the Liberouter system by combining the Scampi opportunis-
tic router with a low cost single board computer, such as the Raspberry
Pi. The rest of the functionality—Wi-Fi access point and captive portal—
is implemented with standard Linux tools. The entire software stack is
distributed as a fully configured operating system image built on a mod-
ified official Raspbian distribution. This results in a low cost (20–30€)
device that can be built simply by flashing the distribution onto a memory
card, after which the Liberouter device is fully functional and requires no
additional management or configuration.

The basic requirements for a Liberouter device hardware are a single board computer with a CPU and enough RAM to run Scampi on the Java Runtime Environment (e.g., 500 MHz x86 with 1 GB of RAM as in section 3.2.2), persistent storage for the content (e.g., 1–10 GB), and Wi-Fi module supporting access point mode. The majority of popular Linux based single board computers on the market today are sufficient for use as Liberouters, with more powerful devices able to serve more concurrent users—even as the network grows, a single Liberouter is likely to serve only a small number of nodes concurrently. Figure 3.13 shows one possible hardware configuration for a Liberouter device: Raspberry Pi Zero W (1 GHz ARM core, 512 MB RAM, 802.11n at 2.4 GHz), an SD card with the Liberouter distribution, an optional RGB LED to signal the state of the router to the user, and a custom 3D printed enclosure.

The Liberouter device is configured to act as an open Wi-Fi access point using hostapd [17] with DHCP and DNS services provided by dnsmasq [11]. The captive portal functionality is created using the combination of dnsmasq and lighttpd [20] web server: The DNS resolver resolves all domain names to the static local IP address where the web server, which then redirects the traffic to the local Liberouter domain (liberouter.local), which contains the portal website. In most smartphones that we tested, this results in the operating system automatically presenting a captive portal dialog with the local Liberouter website to the user. This website contains information about the Liberouter system and an app store page where the user can download the Android Scampi router and other Liberouter apps.

We use the Scampi router to implement the opportunistic routing func-
tionality of the Liberouter system. Each Liberouter runs a Scampi instance, which is accessible to peers to via the Wi-Fi access point. The Scampi instance is configured to act as a hub node, so that each Liberouter device with its connected smartphones form a star topology. This has multiple benefits: First, our analysis in section 3.2.3 shows that the star topology has relatively good performance characteristics. Second, the star topology ensures that the Liberouter sees all the messages, allowing it to persist and distribute them to future peers. Finally, having a natural central node allows us to apply leader-based techniques like the divide-and-conquer algorithm presented in section 3.2.4. By default, we use epidemic routing, which fits the one-to-many content dissemination model required by our target applications (described below). Other DTN routing algorithms—such as PRoPHET [172, 173] or Spray-and-Wait [248]—can be used if a network with more efficient one-to-one routing is desired (see section 2.1.4).

### 3.3.3 Liberouter Applications

As we have seen, the Liberouter system is an independent neighborhood network and does not provide Internet access. Further, the unreliable, delay-tolerant message delivery semantics do not fit the classic Internet applications—a web browser where the user enters an address and waits, potentially hours, for the website to be delivered by the Liberouters is clearly not realistic. Instead, new applications that fit the data delivery semantics of the Liberouter network are required. **Non-realtime, broadcast-based** applications with no global consistency requirements are well served by the Liberouters—particularly apps that collect content in the background while the user moves around an area with deployed Liberouters. While these are quite limiting constraints, many popular application types fit these semantics, e.g., most content sharing and social apps.

Figure 3.14 shows a basic set of Android applications that we have developed for the Liberouter system. On the left of the figure is the Scampi router app, with a specialized interface for Liberouter, through which the user can control the router. Scampi itself runs in the background continuously—searching for peers, creating contacts, and forwarding messages—and provides services to multiple concurrently running apps. The figure shows three such apps: **Guerilla Pics** photo sharing, **Guerilla Tags** discussion boards, and **People Finder** disaster recovery app. All the apps share the same basic semantics. The apps collect content while in the background (photos, text messages, or notes on missing persons),

111
which are displayed to the user when they open the app. This satisfies the *non-realtime* requirement, since the user is never busy-waiting for content to arrive. The users can publish their own content, which Scampi will then disseminate in the background to Liberouters for other nodes to pick up. I.e., the message delivery semantics are *broadcast-based*. Each user in the system will have picked up a different, random subset of the content in the network, out of which the apps compose meaningful, locally consistent views to their users. This means that there is no need for *global consistency*. The applications differ in the structure of the content. Photo sharing messages are relatively large and contain the image data and metadata, discussion board messages are very small and contain only short text messages, while People Finder messages contain People Finder Interchange Format [25] (PFIF) data structures.

We have also worked with Futurice Oy, a software consultancy, to create a demonstrator of a more complex and realistic mobile application. The app is called *Here and Now* [150] and is shown in fig. 3.15. While the underlying principles are the same as for the basic apps, the data models and user interactions are more complex. There are three main primitives in the app: discussion *topics*, content *cards*, and user *profiles*. Cards are the container for content, and include photos, text and metadata (such as the topic, a physical location, and likes). Topics organize cards into logical groups, and are created automatically from the topic names attached to the cards. The application dynamically creates three types of topic lists based on the content it has collected—*trending*, *nearby*, and *recent* (brown, yellow, and red tabs in the figure respectively). This demonstrates the approach of building locally consistent views of randomly collected content,
rather than trying to create a synchronized state across all instances of
the application. In addition, the app allows users to publish their profiles,
which can be collected, displayed, and searched similarly to content cards.

### 3.3.4 Simulation Evaluation

We conduct a simulation study to understand the properties of Liberouter-
based networking. As a use case, we consider a photo sharing application
(e.g., Guerilla Pics from section 3.3.3) in a hypothetical Liberouter network
in downtown Helsinki. The users will move around in the city and occasion-
ally take photos and publish them using their local application. When they
come in contact with a Liberouter, they will pick up and drop off photos
from their smartphones. The metric we are interested in is how many
unique photos are delivered to each user’s device from the Liberouters—we
define this as the *goodput* of the system. This goodput metric reflects the
value that a user gets from a photo sharing application better than the
typical DTN metrics such as delivery probabilities or latencies.

We build a simulation scenario using The ONE simulator. The simulation
area is a $4000 \times 3000$ m$^2$ section of the downtown Helsinki map that is
distributed with the simulator. Within this area, we consider up to 80
pedestrians moving according to the map constrained random path mobility
model with a speed drawn from the uniform distribution $[0.5 \, \text{m/s}, 1.5 \, \text{m/s}]$.
In addition, we randomly place up to 20 Liberouters in the simulation area.
We assume a conservative communication range of 20 meters at 2 MB/s,
which models a scenario where the Liberouter is placed inside a building and pedestrians connect to it from the street outside the building. The opportunistic contacts are only between Liberouters and pedestrians, with no contacts directly between the pedestrians. We model the photo sharing application by having each pedestrian generate 1 MB messages with a period randomly drawn from the uniform distribution (0 min, 30 min] and a lifetime of 6 hours. The pedestrians have 100 MB storage for messages, while the Liberouters have 2 GB storage. When the storage fills up, the nodes will drop the oldest messages to make room for new ones.

Top-left of fig. 3.16 shows the goodput of the system as the volume of unique content a single user receives over a week. We can see the goodput initially increase rapidly as the number of users in the system increases. This is because the users play two roles in the network: They generate and consume the content, and they provide storage and forwarding resources. Initially the new users bring more content and resources into the network, improving the goodput for everyone. However, the goodput seems to quickly reach a steady state where additional users do not increase goodput. The goodput achieved is significant, in the order of gigabytes per week—more than a typical cellular subscription data cap in most of Europe.
In the steady state, generating more content into the network no longer results in other users getting more goodput, i.e., the system is congested. We can see this congestion on bottom-left of fig. 3.16. The figure shows what fraction of the messages that were dropped from the users’ devices was due to congestion rather than reaching the six hour timeout (the Liberouters have enough storage capacity that they do not drop messages due to congestion). This, too, rapidly increases as the number of users increases, followed by a steady state, similarly to the goodput.

In this steady state regime, the Liberouters contain so much content that each time a user connects, the entire contact duration can be filled with transfers of unique content. Since every Liberouter contact results in the user getting the maximum goodput possible, the entire system’s goodput also reaches the maximum—hence the steady state in the goodput. Similarly, the drops due to congestion are caused during these contacts as the user devices make room for the new content coming from the Liberouters. The rest of the drops are caused by timeouts happening mostly during the periods that users are not connected to the Liberouters.

One implication of the steady state behavior is that as the number of content items in each Liberouter increases, the probability that a particular content item gets picked up during a given contact decreases. This also means that the content item is less likely to be carried to another Liberouter for further distribution. While this effect is partly offset by the increasing number of contacts, overall the effect is a falling reach per message, as shown in top-right of fig. 3.16. The figure shows that in a lightly loaded network, each message will be seen by a significant fraction of all the users. However, as the number of users increases, each message will reach a smaller fraction of all users. The figure also shows an interesting phenomenon, where the reach first increases before starting to decrease. This is an effect of the dual roles of users as content creators and content forwarders—with a very small number of users, there are too few forwarders in the network to effectively spread the messages.

Another implication of the steady state is that to scale up the network we should increase the number of Liberouters in the system. This is because in the steady state each Liberouter contact will contribute more goodput, and more Liberouters means more contacts. This is indeed the case, with the top-left of the figure showing that increasing the number of Liberouters increases the goodput. The bottom-right of fig. 3.16 shows this scaling as a fraction of the maximum possible goodput (every node
receives every message). We can see that adding Liberouters to the system improves the relative performance linearly. The effect is stronger with a small number of users, and weaker in the steady state—this is likely due to the increase in resources from the additional Liberouters being relatively smaller compared to the number of users (and hence load in the network).

In addition to scaling horizontally by adding more Liberouters into the system, we can also scale vertically by adding more resources to the existing devices. We explore this by increasing the size of the user device buffers from 100 MB to 500 MB, which prevents any drops due to congestion. This, however, does not have any impact on the goodput, which is natural since the Liberouter contacts are the limiting factor in the steady state—increasing the buffers does not impact the amount of unique content that a node gets from a Liberouter in the steady state. We can increase these contacts by increasing the radio range, which should result in higher goodput since more unique messages can be transmitted over the contacts. In the simulation, when we increase the radio range from 20 m to 100 m, the system reaches close to the maximum goodput even with 80 users.

Overall the simulations show that the system is stable and the goodput reaches a steady state as the number of users increases. From the application perspective, the user will get the same number of new photos, but fewer users will see the same photo—i.e., an increasing load leads to more diverse and localized content. This means that the Liberouter network is well suited for broadcast applications where the goal is to let users publish and receive a diverse set of localized content. Conversely, the network is not well suited for applications that require a wide distribution of the same piece of content to many users, or to a specific receiver, even under load.

3.3.5 Summary

The research question we posed in the beginning of this section is whether lightweight infrastructure components could be used as a basis for practical opportunistic networking. The Liberouter system shows that such a network is practical to implement using cheap off-the-shelf components and standard Android phones. The system has multiple attractive practical properties, including fully autonomous deployment and ability to bootstrap new users into the network without requiring an Internet connection. The simulation study further shows that a larger network built with Liberouters has attractive scaling properties and can deliver significant goodput to the users even with a relatively small number of Liberouters.
3.4 Conclusions

This chapter establishes the practical basis for opportunistic networking—the creation of opportunistic contacts between mobile devices. Based on the results, three methods are practically viable: 1) using existing open Wi-Fi infrastructure, 2) creating direct device-to-device connections using softAPs, and 3) deploying dedicated, opportunistic infrastructure components. Each of these has its benefits and drawbacks.

Using existing Wi-Fi infrastructure is convenient since it is widely deployed in public areas, and many users will be connecting to them anyway in order to get Internet access. These access points, however, are intended for Internet traffic, and will block and filter cross traffic in various ways—this is likely to get worse if significant amounts of cross traffic will start interfering with the Internet traffic. While our measurement study shows that roughly half the access points can in theory be used for cross traffic, it will require a complex and evolving system capable of getting around the filtering.

In case the public access points become widely used for cross traffic, we offer multiple ways to improve the performance. First, we have some practical and simulation evidence that indicates that using a worse-than-best-effort transport, such as LEBAT, may protect subscriber Internet traffic while providing good cross traffic throughput. Second, we show that management of the logical topology is crucial. Devices in the same access point should not attempt to open contacts to every possible peer, but instead should either open just one random connection, or negotiate a structure such as a star topology or a spanning tree that minimizes the number of parallel connections. Finally, the infrastructure access points could be used to bootstrap a direct device-to-device dissemination process, which moves the cross traffic away from the access point. This also has the benefit of being able to exploit more than one Wi-Fi channel concurrently for the transmissions, greatly increasing performance. The drawback in these methods is that all the devices must support the same, relatively complex mechanisms, with no obvious way to bootstrap new devices into the network.

As an alternative to using existing infrastructure, we propose to build dedicated opportunistic infrastructure, in the form of the Liberouter system. This has the advantage of being easy to use—the devices appear the same as any other open Wi-Fi network—while providing additional ser-
vices, such as the ability to directly download the software needed to join the network. Such infrastructure can be very low cost, due to the recent popularity of cheap single-board computers like the Raspberry Pi, and can be independently deployed by anyone without the need for coordination or management. The drawback is that it requires new hardware to be built and deployed, while using existing open access points requires no additional devices.

Next, we will move from creating opportunistic connectivity to actually using it, and look at more advanced mechanisms than the simple messaging based applications that we saw in this chapter.
4. Finding and Manipulating Content

The previous chapter showed multiple ways of creating opportunistic contacts, and the Liberouter system showed how public broadcasting-based content sharing applications can be built on this basic messaging service (section 3.3.3). Richer applications, however, may require richer services from the opportunistic networking layer. For example, the Web is based on request/response interactions with specific origin servers in the form of Representational state transfer [108] (REST)—not just picking up random websites from the Liberouters as done by simple content sharing applications. More generally, an application may wish to search for content that matches some specific criteria, such as a music sharing application searching for the user’s favorite artists rather than just receiving a random collection of music. Further, many popular applications, such as Wikipedia or Google Docs, are based on shared editing of content. To enable such applications, we need to consider mutable content rather than just immutable content such as photos or music files. In this chapter, we will see how to build these three types of richer primitives.

First, in Publication V we design support for accessing static web content via an opportunistic network\(^1\). The nature of Web access is that the resources, identified by Uniform Resource Identifiers [47] (URI), reside in origin servers accessible via the Internet. Users then have request/response interactions with these origin servers to, for example, get copies of the resources. Typically, a website retrieval is made up from many of these interactions, but we show that it can be bundled into a single request/response interaction per website. Further, we show how open infrastructure—Wi-Fi access points or Liberouters—sitting between the opportunistic network of the users’ smartphones and the Internet connected origin servers, can be extended to enable web access without

\(^1\)We extend our work to dynamic web apps in chapter 5.
the end users having Internet connectivity. We show via simulations that a significant number of requests can be satisfied without requiring an always-on infrastructure, provided that users are willing to tolerate some response delay. We further demonstrate the practical feasibility of this approach via a prototype implementation.

Second, in Publication VI we also consider interactions where the users are interested in receiving specific content, but with two differences to the Web request/response model: 1) Instead of being interested in a particular named content item, the users are interested in any content item matching a particular description (a set of constraints)—e.g., all content items that are of type “MP3” and the artist is “The Beatles”. 2) The content does not have a specific origin server, but is instead published by the users themselves and resides in the caches of the opportunistic routers in the system. To achieve this, we tag each message in the system with metadata that describe its content, and allow search queries to be evaluated against the metadata. Based on this, we present a system design that enables nodes to search nearby message caches for content in which they are interested. We study the behavior of the system under different mobility models and routing strategies for the content and queries via simulations.

Finally, in Publication VII, we consider applications where the content items are not immutable, but instead are modified by the users. The fundamental problem with shared mutable content is that two users may modify the same content item at the same time, which creates two different versions that must later be reconciled. In well-connected systems this can be done by a server that imposes ordering on the modifications and ensures that there are no conflicts (e.g., Wikipedia). The centralized approach, however, is not applicable in opportunistic systems where nodes must synchronize their content versions via peer-wise contacts. To enable shared content editing, we apply concepts from distributed version control systems (e.g., git [14] or Mercurial [21]). In particular, we model content as lists (e.g., lines of text) whose elements the nodes can add, remove, and replace—each modification creates a new version of the list. When two nodes meet, we attempt to merge their lists together into a single list that incorporates changes from both sides. It is not possible to do this automatically in the general case since some merges will require human understanding of the content semantics, but we show that in many instances the merging can be automated. We further propose heuristic methods for automatically dealing with the cases where human intervention would be needed.
4.1 Accessing Web Content: Publication V

The first class of applications that cannot be supported well by the simple opportunistic message dissemination that we saw in the previous chapter, are those based on request/response interactions with origin servers on the Internet. The canonical example of this is the Web, where browsers communicate with servers using REST [108] interactions. It is not feasible to make up to date copies of all the websites in existence available in an opportunistic network, such as the Liberouter network, since this would require significantly more resources than are likely to be available. Instead, we need a system design that bridges between the website requests from clients in an opportunistic network and responses from the origin servers on the Internet. This leads to the research question for this section:

• Can opportunistic networking be used to satisfy request/response interactions from mobile devices towards infrastructure origin servers?

We answer this question by designing and implementing support for Web access in opportunistic networks. To do this, we extend the work on open Wi-Fi access points and Liberouters in chapter 3 to provide bridging between clients in an opportunistic network and Web servers on the Internet. In particular, we focus on three key aspects of the problem: 1) Bundling website fetching into a single request/response interaction, rather than the multiple REST interactions usually required to fetch an entire website. 2) Routing of the requests and responses in the hybrid system created by bridging an opportunistic network and the Internet. 3) Exploiting the inherent caching capabilities of opportunistic networking in order to satisfy requests without the need for a full roundtrip to the origin server. For these aspects, we present a system design, a simulation study, and a prototype.

The main limitation of the work in this section is the assumption that the website content is static, with only local interactivity (no remote API calls). This assumption is needed so that we can bundle an entire website into a single response message, which in turn allows us to satisfy a full website request with a single response, possibly with a nearby cached copy. While this model fits, or could in principle be made to fit, many existing websites (e.g., YouTube video page, Instagram image page), in practice many of today’s websites are built as dynamic web applications. These applications require multiple interactions where the client browser calls
the servers’ APIs that dynamically generate the content of the website—such designs cannot be easily converted into a form supported by our design. We will, however, tackle this problem of dynamic web apps in opportunistic networks later in section 5.1.

While the particular scenario we use in this study—web access in downtown Helsinki—is today a largely solved problem due to the proliferation of cheap and ubiquitous cellular connectivity, the underlying model and results have wider applicability. Two particularly relevant areas of applicability are: 1) cellular offloading [121, 219, 267, 279], and 2) alternative network deployments [225, 226].

The need for cellular offloading has been driven by the rapid growth in mobile data demand—estimated to grow from 12 EB/month in 2017 to 77 EB/month in 2022 [7]. The idea of cellular offloading is to handle this growth by moving some mobile traffic away from the cellular infrastructures into alternative networks. In practice, the most popular offload destination is Wi-Fi access points (which already carry the majority of mobile data traffic [7]), but the research community has proposed various other alternatives, such as offloading to opportunistic networks [267]. The work we present in this section can be classified as an opportunistic, delay-tolerant mobile offloading system, suitable for content requests that can be time-shifted rather than having to be immediately satisfied.

Alternative network deployments—as opposed to large, commercial public infrastructure networks—have emerged over the past decade to bring alternative communication services to poorly served areas and communities. These deployments are highly heterogeneous, with technologies ranging from optical fiber, to licensed cellular spectrum, to Wi-Fi, while the entity building and owning the network may be public, private or the community itself. The common model is to deploy an alternative fronthaul network for the subscribers, and to connect it to the Internet via a shared, possibly highly constrained, backhaul. [226] This matches the model we use in this section, where the opportunistic network created by the mobile devices and the Liberouters serves as the fronthaul, and the Internet connections from the Liberouters serve as the backhaul.

4.1.1 System Design

The goal of our system design is to combine the HTTP interaction model with the Liberouter (or open Wi-Fi AP) based opportunistic networking model described in the previous chapter. Figure 4.1 shows the desired
functionality of the system, including the communicating entities, their interconnections, and the application layer request/response interactions.

On the left of the figure, mobile devices form an opportunistic network, while in the middle of the figure Wi-Fi access points or Liberouters bridge the opportunistic network and the Internet (we will refer to these as bridge nodes). We assume that the origin servers are web servers that are only reachable via the Internet, and may or may not support opportunistic networking natively. We further assume that the mobile devices can communicate with nearby peers, as well as the bridge nodes, via opportunistic wireless links. The bridge nodes have fixed Internet connectivity, and this fixed connectivity is also used to inter-connect the bridge nodes—we assume that Liberouters, when used as bridge nodes, must have Internet connectivity unlike in section 3.3 where they are autonomous throwboxes.

Users in the opportunistic network make website requests \((\text{req}(A))\) in the figure), which the opportunistic network forwards until they reach a bridge node. Since both the bridge nodes and the origin servers are connected to the Internet, the request can reach its destination in a single hop after reaching a bridge node. On receiving a request, the origin server generates a response \((\text{resp}(A))\) in the figure), which contains the entire page with all its resources in a single message. The system will then forward the response message to the bridge node(s) and eventually to the requester. As a side-effect of the replication often done by the opportunistic network routing algorithms, response copies will reside in multiple nodes within the

Figure 4.1. System model for web content access from an opportunistic network.
opportunistic network. These copies can be used to satisfy future requests for the same content without the need to contact the origin server again (shown on bottom-left of the figure).

From this, we can see that the system design has three main parts, which we will address in turn: **Bundling** the incremental website fetching process into a single request/response interaction. **Routing** the request from the opportunistic network to the origin server, and the response back to the requester. Exploiting the inherent **caching** to reduce the need for full end-to-end roundtrips to the origin server.

**Bundling:** The first problem we need to solve is how to bundle an entire website fetch operation into a single request/response interaction. In the basic operation of HTTP, each resource (HTML, CSS, JavaScript, image, etc.) is requested separately, which is viable only in highly reliable networks as every independent request must be satisfied before the website can be fully rendered. In an opportunistic network, the likelihood of at least one of these requests getting lost—and hence preventing page loading—is high. And even if successful, the latency would be unbearably large.

The performance degradation caused by multiple requests also affects the classic Web, although not to the same extreme extent as it would in opportunistic networks. As a result, multiple protocol enhancements have been proposed with the goal of reducing these roundtrips—unfortunately these are not directly applicable to our work here. One proposal to address the problem is to use **early hints** [196] response with **preload** [116] links to indicate to the browser what resources will be needed by the site. This does not solve our problem because it only bundles the identifiers for the resources, leaving the fetching of those resources up to the client. HTTP/2 **Server Push** [46] addresses this by enabling the server to send responses to requests that it knows a client will make before the client actually makes them, allowing it to pre-emptively push all the site resources to the browser. While this allows the server to potentially transmit an entire site with multiple resources in response to a single request, it still retains the multiple request/response semantics rather than bundling the resources into a single unit. Web developers have created various ad-hoc ways to bundle multiple logical resources into a single request, one of the most popular of which is webpack [29]. Webpack is primarily targeted at flattening the JavaScript dependency graph of a website into a single source file, but its flexibility and extensibility has lead it to be used for bundling various types
Finding and Manipulating Content

of resources beyond scripts. While webpack can bundle website resources together, it is not designed to create a single bundle for the entire site as required by our work—nevertheless it could potentially be extended to generate the type of output that we require.

For the purposes of this work, we use MHTML [201] as the bundling format (as originally proposed in [200]). MHTML encapsulates an entire website in a single MIME [109] multipart/related structure—originally motivated by the desire to include complete websites in e-mail messages. The MIME message contains the HTML index file as a body part with Content-Type: text/html header. Other resources referred to from the root HTML file are included as body parts with appropriate types (e.g., image/gif). The mapping between a resource referenced in the HTML (e.g., by the src property of an img tag) and the body part that contains it is done by a matching value in the Content-Location header of the body part. This gives us the two main properties that we need: 1) Encapsulation of all the resources in a single message, and 2) the mapping from the HTML source to the resources. Additionally, the server could use S/MIME [230] to sign the message to prove its authenticity. While MHTML is particularly well suited for our requirements, alternatives such as simply compressing the directory hierarchy of a website into a Zip [141] archive are also possible.

An important remaining question is which entity does the bundling. If the origin server has native support for opportunistic networking, it will receive the request message from the client and can perform the bundling of the response. This is the optimal case, as the server has explicit information about which resources comprise the requested website (e.g., from the web authoring tools), and the server could even generate dynamic content specific to the requester (assuming credentials are contained in the request message, e.g., by S/MIME signing it with the client’s keys). If the origin server does not support opportunistic networking, the bridge node could perform the bundling operation. In such case the bridge node would request the desired website over standard HTTP as if it was the client’s web browser, and then bundle the received resources using MHTML to create the response message. The major limitation is that these mechanisms are only applicable to websites that can be represented by a set of static resources—we address the case of dynamic websites in section 5.1.

**Routing:** After we are able to bundle the website fetching operation into a single request/response interaction, next we need to route the request to
the origin server and the response back to the client.

The first issue we need to solve is to define the identifier on which the routing operates. In standard web operation this is done in two steps: First, the domain name portion of the website's URL is resolved to an IP address of the origin server, and then the IP address is used to route the packets to the server. In the opportunistic operation, we want to avoid the separate name resolution step, which would add an indefinite delay before the actual request/response operation could begin.

We therefore propose to route directly on the website URLs—or their static mappings. In the DTN protocols, which underlie our practical implementation, endpoints are identified by URIs called EIDs [233]. DTN bundles are destined to these EIDs and the routing algorithms attempt to deliver copies of the bundles to the DTN nodes that are registered to the corresponding EIDs. We propose a simple mapping from a website URL to the origin server's EID by using the form dtn:http://domain/*.

This approach to naming allows the standard endpoint-centric DTN routing algorithms to route the request and response messages between the client and the server, while also exposing the content name so that is can be used by caching mechanisms—i.e., naming both the endpoints and the content.

We can split the actual routing problem in two dimensions: 1) Routing the request from the client to the server vs. routing the response back, and 2) routing in the mobile segment vs. routing in the fixed segment.

For routing requests in the fixed segment, we can leverage the Internet name resolution infrastructure. Since the domain name of the origin server is encoded as a part of the EID, the bridge node can do a DNS lookup to find the IP address of the destination and then open a direct opportunistic contact with it. The routing algorithm in the bridge node will then see a direct contact with the destination of the request bundle and delivers it to the server. The reverse, however, is not true.

After receiving a request message, the server will generate a response message destined to the source of the request. This response should be routed back to the bridge node over the contact created to deliver the original request. However, the the EID of the bridge node will not match the destination of response message and therefore we have no guarantees that the routing algorithm will use that contact. There are at least three ways we can solve this: 1) Loose source routing, 2) splitting the interaction at the bridge node, and 3) cross-layer optimization between the application
and routing layers. In the first approach, we could define a return routing extension that the bridge node would insert to the request message, which would then cause the response to be routed back the same way. Another way to achieve the same effect would be to use a more advanced EID scheme that allows encoding path information (such as that proposed in [101]) and have the bridge node modify the source EID to insert itself into the return path. An alternative to the loose routing approaches is to split the interaction into two parts: Client-to-bridge, and bridge-to-server. In such a scheme the bridge would not forward the original request, but rather generates a new request for the same website and forwards that instead. This way the response message will be destined to the bridge and thus will be routed back correctly, after which the bridge can use it to craft a response to the client’s original request.

The downside to all the above approaches is that they all involve modifying the message or the interaction in ways that are likely to break end-to-end security mechanisms. For example, modifying the message by inserting an extension block or rewriting the source EID will break any digital signature that the source has applied. Similarly, splitting the interaction is equivalent to a man-in-the-middle attack, which most security mechanisms are designed to defend against. This indicates that security protocol designs should take these requirements into account and provide explicit support for them.

Request and response routing are not symmetric problems: The request only needs to be routed to any nearby bridge, while the response must be routed to a specific bridge near the user. Finding the correct bridge node for the response is analogous to the paging function in cellular networks, for which many optimization strategies exist [272, 119, 160]. The cellular network mechanisms, however, typically rely on the mobile devices continuously registering their location and having a centralized mobility management entity in the network—both of which we want to avoid.

Our proposed mechanism to route the response back to the bridge node where the original request came from is a first order approximation for solving this problem without the need for location awareness. Since the fixed segment routing for the request and subsequent response is likely fast (in the order of seconds), if a client was near a bridge node when originating the request, it is likely to remain near it when the request is returned. Problems arise, however, if the client is not near a bridge
when generating the request. In such cases the client may have moved significantly during the time taken to route the request to a bridge node, and therefore another bridge should be used for routing the response.

To solve this issue, we propose \textit{k-nearest bridge routing} for the responses. This is a simple extension of the previously presented response routing: Upon receiving a response back for a request, the bridge node will replicate the response to its \(k\) nearest peers—which trades off resource use against response delivery probability. This leverages the fact that the bridge nodes are geographically fixed and inter-connected by the fixed network segment, which allows them to maintain their neighbor sets. While in this work we study fixed values of \(k\), it is also possible to make the value dynamic. For example, the value of \(k\) could be a function of the latency from request creation to reception by a bridge, which would increase the response distribution area for requests that came from further away. In addition, the bridges could run opportunistic community detection algorithms \cite{138, 54} and prioritize bridges that have recently been in the same community with the client.

In the mobile segment (left of fig. 4.1), we rely on opportunistic routing algorithms. Unlike in the standard DTN routing problem, we are not trying to reach a specific node, but rather any of the bridge nodes. Regardless of this difference, standard probabilistic routing algorithms are well-suited for our purposes: In \textit{Direct Delivery} routing \cite{142} the client holds onto the request until it connects directly to a bridge node—in such cases the response will typically be immediately generated and delivered to the client. \textit{Epidemic} routing \cite{260} can be used to deliver messages without needing the client to directly connect to a bridge. It will replicate copies of the request and responses between the nodes in the mobile segment until one of the copies reaches the intended destination (client or bridge). This comes with the problem of high resource use, but reaches optimal latencies and delivery probabilities in uncongested networks. Real networks are not expected to be uncongested, so a tradeoff mechanism is needed between resource use and delivery performance. \textit{Spray-and-Wait} routing \cite{248} is well suited for tuning this tradeoff in our use case—the initiator of the message sets an upper limit on the replicas, which limits the resource consumption, and the algorithm reverts to direct delivery after all copies have been made. More sophisticated algorithms, discussed in section 2.1.4, as similarly applicable. In each case all the bridge nodes are configured to
accept website request messages by matching the destination EID prefix \texttt{dtn:http} (as explained above). The response messages are destined to the EID of the initiating client, hence no special mechanisms are needed.

**Caching:** The probabilistic multi-copy routing algorithms that we propose for the mobile segment have the side-effect of spreading message copies into multiple devices. Normally these copies would be wasted resources, but in the case of messages containing bundled websites, we can leverage them to create an opportunistic caching mechanism. Both the mobile nodes and the bridge nodes can act as caches—the former allows subsequent requests to be satisfied from nearby peers without the need for a roundtrip to the origin server, while the latter turns the bridge nodes into edge caches that take load away from the origin server. The main limitation of the caching approach is that only static content that is not personalized to each client can be cached.

To achieve caching, each node must have logic to match incoming request messages against cached response messages. Conceptually the matching is simply a comparison of the request URL with response URL, but to enable it we need to expose the URLs for the matching logic. There are a number of ways this can be done: First, if the full URLs are encoded into the EIDs as explained above, the matching can be done directly in the routing layer without interpreting the message contents. Alternatively, the URLs could be encoded as special metadata attached to the messages (e.g., in Bundle Protocol using the Metadata Exception Blocks [255]), which also allows the mechanism to be implemented without the need to interpret the content of the messages. Finally, the implementations could parse the MHTML headers contained in the payload of the messages to learn the URLs, which is not ideal since it would require application layer logic to be inserted into the forwarding process.

### 4.1.2 Simulation Evaluation

We validate the feasibility of the system design via a simulation model in The ONE simulator [153]. The model simulates an urban city scenario with pedestrians as the clients and Wi-Fi APs or Liberouters as the bridge nodes—the model is similar to the one we use for the Liberouter simulations in section 3.3.4. With the model we study the system behavior for different bridge node densities and routing algorithms.

The simulation area is $4.5 \times 3.4$ km area of downtown Helsinki, where the
node movement is constrained to the city streets (Helsinki City Scenario, HCS). As a comparison case, we also consider random waypoint mobility within the same area (Random Waypoint Scenario, RWP). We model 140 pedestrians and 17, 109 or 325 bridge nodes placed according to real world open Wi-Fi access point locations. The wireless range of the nodes is 50 meters and transmission speed 10 Mbps—this results in roughly 200k contacts between nodes during the 12 hour simulation. Bridge nodes are connected together with a fixed network dimensioned to not be a bottleneck.

For the application model, we choose top 50 websites in Finland and measure their size and access latency to parametrize the simulation scenario. Each mobile node uniformly randomly picks one of the 50 websites to request every 5 minutes on average (uniformly distributed over 280–320 seconds). For message routing, we use the three different probabilistic routing algorithms proposed above: Direct Delivery, Spray-and-Wait (binary, 10 copies), and Epidemic.

To begin with, we want to understand the success probability for the request/response interactions. Figure 4.2 shows the probability of receiving a response to a request for different opportunistic routing algorithms and message lifetimes. By increasing the message (request and response) lifetimes we allow more time for the interactions to succeed, but cause more resource use in the network.

The first feature we can see in the figure is that the bridge node (AP) density is a important factor for the response probability. In the densest case (~21 bridges/km²) the success probability reaches over 90%, while dropping to 80% for the medium density (~7 bridges/km²) and down to 40% in the low density case (~1 bridge/km²). These probabilities are reached when the time-to-live for the messages is 40 minutes, which makes
Finding and Manipulating Content

Figure 4.3. Response latency CDF for HCS and 40 min lifetime (left) and response hop count for HCS lowest density case (right).

the system practically feasible—not for interactive use like normal web browsing, but for, e.g., automatic background fetching of frequently visited sites. Furthermore, if we cut the message lifetime in half to 20 minutes, the success probabilities fall but remain practically feasible: 80%, 60% and 20% for the three densities.

Interestingly, the delivery probability does not appear to depend on the routing algorithm used in the opportunistic network segment. The difference in success probability between Direct Delivery (i.e., no opportunistic hops) and the multi-hop capable algorithms is less than 10% in all cases. In other words, the majority of the interactions do not travel over multiple opportunistic hops, but are rather completed when the requesting client comes into a direct contact with a bridge node. This means that in our scenario it is likely that a mobile client will come into direct contact with a bridge faster than the opportunistic network is able to transmit the request and response to and from a bridge. This is supported by the fact that we see the largest differences in the lowest density case, where the clients are less likely to directly meet an access point and therefore the opportunistic multihop routing is likely to help. The right side of fig. 4.3 shows the epidemic lowest density case in more detail. In the figure, we can see that a large majority of the responses only take a single hop, but that a longer lifetime allows a significant number of additional responses to arrive via multihop paths. As a result, the k-nearest bridge routing does not improve performance for nodes that can directly connect to the bridges—although it is useful for purely opportunistic nodes as we will see below. Finally, the HCS scenario, which forces nodes to meet more often, has a consistently higher delivery probabilities than the RWP scenario where nodes create fewer contacts between each other and the bridge nodes.

While the previous results show the success probability of the interac-
Finding and Manipulating Content

tions, we can understand the system behavior more deeply by looking at the response latencies. The left side of fig. 4.3 shows the cumulative distribution function (CDF) for the response latency for the different opportunistic routing algorithms. Again, we see that the routing algorithms do not have a large impact due to most messages being delivered directly. For the Direct Delivery case, the latency is effectively measuring the time until the requesting client meets a bridge node. If the client does not meet a bridge node within the 40-minute lifetime of the request, the interaction cannot succeed as indicated by the flat probability in the figure after $x = 40$ minutes for Direct Delivery—in contrast the multihop capable algorithms can support interactions up to 80 minutes of duration. In all cases, we can see that a significant fraction of the successful interactions finish quickly: For the high density case, half the successful interactions finish in 5 minutes, for medium density in 10 minutes, and for low density in 20 minutes. This further supports the practical feasibility of the system design.

Since the majority of successful interactions happen via direct contact between the requesting client and a bridge node, we will consider clients that cannot connect to bridge nodes in order to evaluate the purely opportunistic behavior of the system. When we configure 20 nodes in the medium density Spray-and-Wait scenario without the ability to connect to the bridge nodes, we find that they achieve success probability of 30% when the lifetime is 40 minutes—significantly less than the 80% probability achieved previously in the same scenario. However, this is where the k-nearest response routing becomes effective, and we find that it increases the success probability to 50%. This is further increased by the caching mechanisms in the opportunistic network segment, which otherwise have little impact—we find that we can double the success probability compared to just obtaining copies from the bridge nodes.

Overall, the simulation results indicate that the system is practically feasible for website requests that can tolerate delays in the order of tens of minutes within urban areas. This is clearly not feasible for the usual web browsing workflow of entering a website address and busy-waiting for the response. We believe, however, that the web browsing UX can be changed to accommodate this new mode of web access, as we will demonstrate next in the prototype implementation section.
4.1.3 Prototype Implementation

To establish the practicality of our design, we build a prototype of the entire system using the DTN protocols [70, 87, 233]. The prototype includes clients for Android and iOS, a bridge node, and a webserver with native support for opportunistic web access. The implemented mechanisms include website bundling both at the origin servers and at the bridge nodes, routing from bridge nodes to the origin servers over the Internet, k-nearest routing between the bridge nodes, and custom UX for the mobile clients.

Figure 4.4 shows the overview of the prototype system implementation. The purple elements of the figure contain the opportunistic web specific logic, which are all implemented from scratch for the prototype. The blue elements correspond to the standard DTN protocols that create the opportunistic network—here we use our own implementations (Android, iOS, bridge node) and the DTN2 reference implementation (web server). The grey-green elements represent standard Internet protocols, i.e., HTTP over TCP, and contains no opportunistic web specific modifications.

The figure also shows the supported interactions (black text and arrows): 1) Opportunistic web clients create website requests and receive responses from the bridge nodes. 2) Bridge nodes create opportunistic contacts with the origin servers over the Internet in order to route the requests and responses between the opportunistic and Internet segments. 3) Bridge nodes create contacts between each other in order to implement the k-nearest
routing mechanism. 4) Bridge node fetches websites from unmodified web servers and bundles them into opportunistic web responses.

As as the origin server, we use a custom web server written in Ruby using DTN2 as the underlying opportunistic router [205]. The server registers to EIDs matching the domains that it serves using a one-to-one mapping scheme between the website domain and the EID. We also create DTN NAPTR records for the EIDs that resolve to the TCP Convergence Layer parameters of the server, which allows the bridge nodes to resolve the EID and create a direct contact with the server. When the server receives a request message to one of the served domains, it reads the requested resource (e.g., “index.html”) in the message payload formatted as a standard HTTP GET request. The server then gathers all the resources for the requested website—either by reading a configuration file or by parsing the HTML of the requested site—and bundles them into an MHTML response message. The response is then sent to the requester via DTN2.

The bridge node is a custom Java application designed to run in fixed hotspots and implement the bridging functionality between the opportunistic and the Internet segments. The bridge node needs to perform some non-standard processing of the request and response bundles, and thus we use our own DTN protocol implementations rather than DTN2 (or Scampi): First, the bridge needs to resolve the destination EID to convergence layer parameters in order to open a direct contact to the origin server—in principle we could set up a static DTN routing overlay that would be capable of routing the messages, but we believe dynamic contact creation is a more practical solution. Second, the bridge implements response caching by taking a previously seen response message to the same website and rewriting the destination and timestamp fields to match the new request. In addition, the bridge implements a mechanism for fetching websites from standard non-DTN web servers and bundling the responses into MHTML response messages, allowing clients to request any website from any web server. Finally, the bridge nodes can be interconnected into a mesh to provide k-nearest response routing by replicating the response message from the serving bridge node to its geographic neighbors.

On the client side, we need custom logic to create the web requests and to receive, unbundle, and present the response to the user. A simple approach would be to run a local HTTP proxy that the user’s web browser is configured to use. This would allow the users to continue to use their unmodified web browsers, decreasing the barrier to adoption of our system.
The problem, however, is that the standard web browsers are designed around a “busy-waiting” approach where the user inputs the website (or clicks a link) and the website appears almost instantaneously. In a system like ours, where the simulations predict latencies in the order of tens of minutes (section 4.1.2), busy-waiting is not feasible—the users would assume the system is broken long before the responses would arrive.

We have therefore designed custom web clients with a user experience (UX) that makes it apparent to the user to not expect the standard web browser behavior. The user interfaces for our Android and iOS clients are shown in fig. 4.5. In our UX paradigm the client maintains a list of websites on which the user has long-lasting interest—e.g., their favorite news sources, organizational pages of their employers, or social media sites of people that they follow. This could be manually entered, imported from bookmarks, or automatically derived by recommender systems. The user interface lists these sites and shows their status, e.g., fresh new response, old response, or no response. In the figure, these are shown as bubbles with different background colors indicating the websites and their status (Android client on top of the figure), or as a list view divided into different segments for different stats (iOS client on the bottom of the figure).
Behind the user interface, as the user moves around with their devices in their pocket, the client is continuously maintaining requests for the websites. When responses are received, the client unpacks the parts of the MHTML message and stores the resources on the local filesystem and updates the status database. When the user enters the application, they are presented with an up-to-date view of the currently available content. The user can then select one of the websites with responses, which will render the website from the local storage using the standard operating system web view component (WebView on Android, WKWebView on iOS). We also implement a fall-back mechanism where the user can choose to fetch the website over a cellular connection (if one exists).

We validate the functioning of the system in an emulated testbed where the opportunistic network segment comprises busses running a circular route between the clients and bridge nodes 10 km away from each other. We find that the system is able to deliver requests and responses as expected, allowing the clients to access websites without fixed network connectivity with a latency proportional to the travel time of the emulated busses.

While the simulations and implementation show that the design is feasible in practice, it does have a number of limitations. The main limitation is the latency inherent in opportunistic communications that is many orders of magnitude larger than what the users of standard web browsers would expect. However, we believe that new UX approaches, such as those described above, can still produce a valuable service to the users. Further limitations come from the need to deploy multiple custom elements (clients, bridges, servers) for the system to be fully functional. There are, however, some paths to incremental deployment: First, our bridge implementation is able to fetch and bundle websites from any web server, meaning that the origin server does not need to have native support. Second, some of the bridge functionality could be implemented into the client. For example, the logic to fetch and bundle websites could be executed by the client when it is connected to the Internet. Finally, the opportunistic web client functionality could be implemented in standard web browsers, meaning that users would not need to install custom clients.

### 4.1.4 Summary

In this section we have seen that an opportunistic network can be used to extend the reach of the Internet for static websites. The simulations show that the system can reach high enough success probabilities to be feasible
in practice, while the prototype implementation shows that the design is also realizable in practice using existing platforms. However, even in urban scenarios we would expect latencies in the order of tens of minutes, which means that the UX of the web browsing needs to be changed from the busy-waiting paradigm of existing browsers. While this work focused on fetching websites as the primary use case, the underlying techniques should be applicable to any request/response style interactions between opportunistic clients and servers on the Internet. The work in this section is also limited to static websites that do not have elements that require frequent interactions between the clients and the servers—we will see how to support dynamic websites in opportunistic networks in section 5.1.
4.2 Searching for Specific Content: Publication VI

In the previous section, we saw how to build a system to support the fetching of specific, named content items (websites) in opportunistic networks. However, even on the web, the users often do not know exactly which sites they want to access, but instead have a general idea of what content they are interested in and use a search engine to find the specific website URLs.

The search engines continuously crawl the web to build a search index against which the users' queries are executed. However, implementing a search engine in an opportunistic network has significant obstacles: First, building an index via web crawling depends on all the content being accessible from a centralized location—which holds for the Internet, but not in opportunistic networks. Second, the system also depends on the index and the search engine itself being accessible via the opportunistic network. This access would have to have a low latency since the search step must finish before the actual content retrieval can begin—in opportunistic networks this extra round trip has a potentially unbounded delay, possibly blocking the content access for an extended period.

An alternative to a centralized index-based search engine, is to enable the clients to directly query and fetch content from their peers in the opportunistic network. This seems attractive since the multi-copy routing algorithms often used in opportunistic networks create copies of content within the network, which could be made available to nodes other than the original destination via the search mechanism. Further, in many mobile applications the clients are not only consumers, but also prime producers of content, further increasing the amount of content available in nearby mobile devices. We therefore pose the following research question:

- *Can the content available in the opportunistic network be made searchable via a query/response mechanism?*

We address this research question by designing a system that integrates the search logic into the opportunistic network. The design attaches metadata describing the content to each message and allows search queries to be executed against this metadata, with matching content items being routed back to the querier. The focus in this work is on the query distribution and response routing in the opportunistic network—we propose multiple algorithms for these and study their behavior by building a simulation.
model. The findings show that the query mechanism is feasible in a wide range of scenarios—the network will respond with some results with a high likelihood—but that exhaustive searches or searches for specific rare content items are likely to fail. This indicates that a query mechanism can be a useful building block for applications, but it should be used as an asynchronous background mechanism to continuously search content on behalf of the user rather than the typical Internet search experiences where lists of responses are immediately returned to the user.

This idea of basing networking on expressions of interest in particular content, rather than on addressing data to particular hosts, has also been proposed in the context of well-connected networks. In particular, recently various architectures have been proposed under the Information-Centric Networking (ICN) [33, 269] umbrella. These approaches name the data objects and then have the network transport them from the producers to interested consumers based on the names. In contrast, our query mechanism allows arbitrary queries against the data contents—this is similar to Content-based Networking [69], where receivers express their interest via predicates against the message contents. In most cases these architectures have assumed a well-connected network, while our work assumes only an opportunistic network.

### 4.2.1 System Architecture

The goal of the system design is to allow users in an opportunistic network to find and retrieve content stored in the devices of other users. The approach we take is to use opportunistic forwarding to spread queries within the network, and have the peers evaluate those queries against their local
Finding and Manipulating Content

content to generate responses. This is in contrast to the popular approach used in well-connected networks where a search engine provider builds a centralized index of off the content and their locations—as explained above, such approach is unlikely to be viable in opportunistic networks.

Figure 4.6 shows the proposed concept on a high level. The node wishing to search for content, shown in black in the middle of the figure, crafts a query message $Q$ and forwards it over opportunistic contacts to encountered peers. This forwarding process is determined by two mechanisms: 1) the underlying opportunistic routing protocol (e.g., Epidemic or Spray-and-Wait), and 2) a query termination mechanism that can stop the propagation based on search-specific criteria. In addition, each node that the query passes through may modify it ($Q \rightarrow Q'$), e.g., to track the number hops taken or the number of responses generated. Nodes that have matching content items will generate responses destined to the originator of the query—these are delivered by standard opportunistic routing algorithms.

Figure 4.7 shows the model of query processing performed by all search enabled nodes. The figure divides into two parts vertically: On the bottom of the figure, we model the opportunistic networking layer as a message cache that the routing algorithms synchronize with peers over opportunistic contacts. On the top of the figure, the application layer logic reads and writes messages to and from the cache via an API (e.g., send/receive, publish/subscribe). The messages flowing between these two layers are new incoming queries from the opportunistic network into the search logic layer, and response messages from the search logic into the opportunistic network—some mechanisms discussed in the next section also modify the query message, which would ideally use a replace or delete operation to substitute the updated query message for the original one.
The main processing flow for the system is performed by the search layer logic. The content that the search layer operates on is stored in a content database (top-left of the figure). We model this database as a conceptually separate entity, but in a real implementation it can take various forms: It could, for example, be a lightweight abstraction over the content in the router cache rather than a separate and independent storage element. Alternatively, it could be a database shared by various local applications that wish to make their content (e.g., images, music, text) searchable.

The way in which queries are executed against the database is arbitrary and many options exist, such as SQL statements against a common schema or regular expressions against text-based content. To make the content in the opportunistic router caches searchable, we propose that applications attach metadata about the content to each message that they send into the network. The metadata are key-value pairs that describe the content, e.g., ID3 tags for messages containing music, or geographic coordinates for photos. Queries can then be written as predicates against the metadata, e.g., $ID3\text{.artist} = \text{\textquote{The Beatles}}$. This approach allows us to define the search as a generic layer between the opportunistic router and the applications without the need for any application specific logic (e.g., having to understand application specific query language or database schemas).

The rest of the search layer comprises three main functions: Content matching, response selection, and query processing (shown in the figure as three colored ellipses). When a query message is received from the opportunistic networking layer, the matching function runs the query against the content database entries, producing a set of matching local resources. As the set of matching resources is potentially very large, we define a selection function to filter the resources down to a set of responses which will then be sent to the opportunistic networking layer. The selection function also gets a copy of the original query message, which may carry additional information for the response selection—e.g., maximum number of desired responses, or a ranking function for selecting the best matches. Finally, the query processing function updates the incoming query by first deciding whether to terminate or continue to the query propagation, and then modifying the contents of the propagated queries. The query modification may include incrementing the generated response count, and recording additional information useful in the future processing—e.g., the node degree, which is used in one of our termination algorithms described in the next section. Altogether, each incoming query message results in
Finding and Manipulating Content

a (possibly empty) set of response message being created and the query message being dropped or modified.

### 4.2.2 Query and Response Propagation

In this work our focus is on the core mechanisms of query and response propagation in the opportunistic network: The originating node hands a query to the opportunistic search layer, which propagates the query to other nodes, to which nodes with matching content generate responses, which the opportunistic network then propagates back to the originator. An ideal solution would propagate the query via the shortest possible paths to the smallest set of the nearest nodes that have enough content to satisfy the originator, and then route back a non-overlapping set of responses. Unfortunately, such an ideal solution is not possible in practice in an opportunistic network, since we cannot know beforehand which nodes have the content and the nodes cannot coordinate in real-time to only return a non-overlapping set of responses. Hence, the typical resource use vs. success probability trade-off of opportunistic networking applies: The more we spread the query and the more responses we generate, the higher the likelihood that the originator receives an adequate result set, but the more network resources we end up using.

The problem, therefore, becomes to devise algorithms to limit the spread of the query (and the generation of results) in a way that allows us to control the resource use vs. success probability trade-off. We study four different heuristics for terminating the query: 1) First-response drop, 2) hop-count limit, 3) time-to-live limit (TTL), and 4) utility estimation. These heuristics are mainly based on path information, that is information that is attached to the query by the nodes during processing (e.g., hop counts, number of responses generated). The query termination mechanism is independent of the underlying routing protocol, but the two together determine the spread of the query—we will study three different routing algorithms in combination with the four different query termination approaches via simulations in the next section.

The simplest query termination heuristic we propose is to terminate the query when generating responses. The assumption behind this heuristic is that the first set of generated responses will satisfy the initiating node, which means that continuing to propagate the query would use network resources without producing additional value. When using a multi-copy routing algorithm, the query will be replicated between nodes that do
not have matching contents, but will not be replicated by nodes that produce results—a single-copy routing algorithm will have at most one node generating responses. This heuristic for the forwarding decision for query $q$ can be expressed as:

$$\text{Forward}(q) : |\text{Responses}(q)| = 0$$  \hspace{1cm} (4.1)

The next query termination heuristic we propose is a hop-count limit. Nodes record the number of hops taken in the query message and stop forwarding the message after some limit is reached. The limit could be set by the originator of the query, which allows for explicit control over the query behavior—the user can choose a small hop-count limit to only search nearby and get the potential responses quickly, or a larger hop-count to search in a wider area with results taking longer to arrive. Alternatively (or in addition) the nodes in the network can impose their own limits, allowing the network to, e.g., adapt to increasing load or misbehaving nodes. This heuristic, with a hop limit $L$, can be expressed as:

$$\text{Forward}(q) : \text{HopCount}(q) < L$$  \hspace{1cm} (4.2)

An alternative to the hop-count limit is a time-to-live limit, which limits the amount of time the query is forwarded rather than the number of hops. TTL limit can be easier than hop-count limit to map to user requirements such as tolerated wait time or desired freshness of responses. However, unlike hop-count limit, TTL limit does not explicitly limit transmission resource use in the network as the query can be forwarded an unlimited number of times during its lifetime. If the underlying opportunistic network has synchronized clocks (such as the DTN architecture [70, 233]) the limit can be directly enforced by the networking layer. Otherwise, each node must record the time it held the query before forwarding as path information and enforce the forwarding limit itself. TTL limit of duration $T$ can be expressed as:

$$\text{Forward}(q) : t_{\text{now}} - \text{CreationTime}(q) < T$$  \hspace{1cm} (4.3)

Or using path information if the nodes do not have synchronized time:

$$\text{Forward}(q) : \sum_{n \in \text{Path}(q)} \text{ProcessingTime}(q, n) < T$$  \hspace{1cm} (4.4)

Finally, we propose a more complex heuristic that estimates the total number of generated responses globally, and sets a threshold for it. The
motivation behind this is to automatically adapt to queries that produce more or less matches: Hop-count and TTL limits will spread each query the same amount regardless of how many responses are generated. As a result, queries that generate many responses are propagated further than necessary, and queries that return few matches are not propagated enough. Generally, the mechanism is based on the assumption that the marginal value of responses decreases as the total number of generated responses increases. This is due to the further responses having a higher probability of being duplicates and the likelihood that the originator of the query has already received an adequate set of resources. Therefore, we want to devise a metric \( u_{\text{est}}(q, n) \) that is directly proportional to the number of responses generated globally as estimated by node \( n \), and set a threshold \( R \) for it:

\[
\text{Forward}(q, n) : u_{\text{est}}(q, n) < R \tag{4.5}
\]

We cannot get global knowledge of the number of responses generated, but we can use the path information for the query to estimate it: We can record how many hops the query has taken on its way to us, and how many responses those nodes generated, and use that information to derive \( u_{\text{est}}(q, n) \). In fact, for single-copy routing algorithms the response count in the path information is the total count. For multi-copy routing algorithms, there will be multiple copies of the query message taking multiple paths through the network, with each one generating some number of responses. With fixed-copy algorithms, such as Spray-and-Wait, the number of copies is explicitly known from the routing information. With epidemic routing, we propose to estimate the number of copies created by estimating the node degrees—each node will generate roughly as many copies as the number of nodes that it meets. Each node will estimate its degree by tracking the number of opportunistic contacts it has had in the previous \( \text{TTL}(q) \) time window. This degree can be used directly as an estimate for the average degree in the network, or it can be recorded in the query as path information and an average of all the node degrees on the path can be used (assuming the degree distribution has a well-defined average value):

\[
\text{nodes}_{\text{est}} = \text{DegreeEstimate}(q, n)^{\text{HopCount}(q)} \tag{4.6}
\]

We can then multiply the estimated node number with the estimated number of responses generated by each node. This estimate can be calculated from the number of responses recorded in the path information, or by using a constant estimate \( C \) if no responses have been generated:
Finding and Manipulating Content

\[
u_{\text{est}}(q, n) = \begin{cases} 
\text{nodes}_{\text{est}} \cdot \frac{|\text{Responses}(q)|}{\text{HopCount}(q)} & |\text{Responses}(q)| > 0 \\
\text{nodes}_{\text{est}} \cdot C & \text{otherwise}
\end{cases}
\]  

(4.7)

As mentioned before, these termination heuristics combine with the routing algorithm to produce the query propagation behavior. Single-copy routing algorithms result in the query taking a (random) walk between nearby nodes, while multi-copy routing results in some form of limited flooding for the query. Either way, once a query reaches a node with matching content, responses are generated, and need to be routed back.

The response routing problem is the standard opportunistic unicast routing problem—the originator of the query is the source of the query message, which allows the responding nodes to simply address the response messages to the originator. It is then up to the standard routing algorithms to deliver the response message to its destination, and we do not need to define any additional search-specific functionality. However, one possible optimization can be made: Multiple different nodes may respond with the same content item, producing different response messages with the same content. Although these messages are semantically identical, they appear different to the routing layer and will be routed to the destination. We propose to expose a separate content identifier as metadata in the message, which allows routers to identify and drop these semantic duplicates.

4.2.3 Simulation Evaluation

We will study the behavior of the search system via simulation models. The evaluation focuses on the core mechanism of query and response propagation, and abstracts from details such as how the queries are represented or matched against content items. We study the combinations of three different routing algorithms with the four query termination heuristics in four different simulation scenarios. The main metrics we are interested in are the probability of finding a specific content item, the number of different content items returned by a query, and the latency of responses.

We define four simulation scenarios with increasing amount of realism and heterogeneity: 1) Basic Scenario (BS), 2) Random Waypoint (RWP), 3) Helsinki City Scenario (HCS), and 4) Working Day Movement (WDM). The basic scenario is a static topology with an easily understandable topology for the query spreading. RWP adds mobility with random, unconstrained free-space movement of the nodes, and HCS adds structure by constraining
the mobility to a map (of downtown Helsinki) and dividing the nodes to three different mobility types (pedestrians, cars, trams). Finally, WDM adds more realism by more carefully modeling the daily human behavior.

**Basic Scenario (BS):** A static topology with a single central node and five concentric rings around it. Each ring is 100 units of distance away from the next inner one and contains six more nodes evenly spaced along the ring—i.e., the first ring around the central node has a radius of 100 and contains six nodes, the next ring has a radius of 200 units and contains 12 nodes, etc. This results in 91 nodes in the topology, out of which we use the central node to generate queries and the remaining 90 to carry the content items. Each node is connected to every node within 150 units, which results in a degree between 3–6 for every node.

**Random Waypoint (RWP):** Unconstrained random movement in free-space. We place 125 nodes in an area of $100 \times 100$ meters. Each node continuously picks destinations from the simulation area and moves there with a uniformly randomly picked speed in $[0.5, 1)$ m/s.

**Helsinki City Scenario (HCS):** Map-constrained random movement with three node types [153]. The nodes are constrained to the Helsinki map, and divided into three types: Pedestrians (80), cars (40), and trams (6). Pedestrians and cars continuously pick random destinations from a set of points-of-interest and move there along the shortest path, while trams follow predetermined routes that match real-life tram routes.

**Working Day Movement (WDM):** Model of daily human behavior. We add home, work and evening activity locations for the nodes, which they commute between with a daily schedule using various means of transport. We use the default scenario defined in [95] but scale down the number of nodes from 1029 to 544 by scaling down all group sizes.

In the scenarios with mobility (RWP, HCS, WDM), we generate opportunistic contacts via 2 Mbps radios with 10-meter range. Over these contacts, the nodes will exchange messages using three different routing algorithms: 1) First Contact [143], a single-copy routing algorithm where the node forwards the message over the first possible contact, 2) Spray-and-Wait [248] limited flooding with 10 message copies, and 3) Epidemic routing [260] where every node tries to send every message over every contact. The nodes have unlimited storage for the messages so that we can focus on the query spreading behavior rather than being constrained by storage bottlenecks and message dropping strategies. We simulate all the query termination heuristics described in the previous section (first
response, hop count, time-to-live, and response estimation).

The first metric of interest is the probability that the search mechanism is able to find and return a content item. To study it, we set up a simulation with a single content item with 5–30 copies in the network, and record the probability that a node is able to retrieve the item. We test every combination of scenario, routing algorithm, and query termination heuristic. The results are shown in fig. 4.8.

In the static topology (top row of the figure), we can see the significant impact that the routing algorithm has on the system behavior. In the case of First Contact routing—which results in a single copy of the query and response messages (randomly) traversing the network—we have a high probability of success (50–70%) even in the smallest case where less than 6% of the nodes have the content item, and reach close to 100% probability when little over 20% of the nodes carry the content. There is a clear difference in the performance of the query termination heuristics: Hop and time limited terminations have roughly 20% higher success probabilities than the first drop and response estimation heuristics. This makes sense, as the hop and time limited terminations will allow the query message to circulate in the network for longer, and thus generate more responses, than first drop termination that only generates at most a single response. The response count estimation heuristic with both thresholds behaves almost identically to the first drop heuristic, which suggests that it is also resulting in query termination once a single response has been generated (which is the expected behavior with the threshold set to one).

The behavior of Spray-and-Wait in the static topology (top-middle of the figure) is to search a random subset of the nodes for the content item (10 out of the 90 nodes). For example, when there are 5 content items placed randomly, the probability that one of the query message copies finds a match is about 56% (10 × $\frac{5}{90}$). The response probability is then a combination of the probability of finding a match and being able to route back the response—in the figure we can see that for the case of 5 content items this is 25–30%. As expected, the figure shows that as the number of content copies in the network increases, so does the probability of finding a copy. There is no discernible difference between the query termination heuristics, which indicates that in the static topology, the routing is able to distribute the query copies before the termination heuristics are triggered.

Epidemic routing in a static topology without resource constraints is
Figure 4.8. Probability of receiving a response to a query (note the different x-axis for the last row).
Finding and Manipulating Content

guaranteed to succeed regardless of the number of content items in the network, which matches what can be seen in the figure (top-right). The probability is slightly below 100% due to the way the results are collected (queries near the end of the simulation do not have time to complete).

Next we add mobility to the simulation in the form of free-space Random Waypoint mobility and map-constrained Random Waypoint in the Helsinki City Scenario (two middle rows of the figure). When we do this, we can see the success probabilities for the First Contact routing fall significantly. This is because in a static, fully-connected topology the query message can quickly propagate between the nodes, while in a case with mobility the propagation is over less frequent opportunistic contacts. The division of the termination heuristics into two groups (hop and time limited vs. first drop and response estimation) is still visible. This difference is more pronounced in the map-constrained mobility case than in the free-space mobility case, indicating that allowing the query to circulate longer provides more benefit in mobility models with more heterogeneous structure.

In contrast to First Contact routing, Spray-and-Wait routing seems to result in a remarkably similar outcome as in the static topology. This indicates that similarly to the static scenario, the routing algorithm manages to distributed all ten available query message copies to the peers and the route back the responses—overall response probabilities are slightly lower due to the larger number of nodes in the simulation (125 vs. 90). Interestingly, in the map-constrained variant the hop limited query distribution performs better than the other heuristics. This could be explained by the contacts being less frequent, meaning that hop count limit allows the query message to spread further than the other termination heuristics.

When we use Epidemic routing in the RWP and HCS cases, the response probability is 40–70% compared to 100% in the static topology. As is typical for Epidemic routing, the performance appears to be limited by congestion: First, the response probability is flat or falls as the number of content items increases in the network. If the performance is already bottlenecked by congestion, then adding more resources to the network will not help. Second, Spray-and-Wait is able to achieve higher response probability than Epidemic routing. Since uncongested Epidemic routing is guaranteed to find the optimal paths for every message, but fails to do so in this case, the network is likely congested. Since we have unlimited storage, the congestion is likely in the transmission capacity—either the number or the volume of messages is larger than the capacity of the contacts.
In the most realistic scenario, Working Day Movement, the system behavior appears qualitatively similar to RWP and HCS. However, the difference between the two groups of termination heuristics is amplified. The heuristics based on simple network layer metrics (hops, TTL) outperform the first drop and response estimation heuristics by a margin of 10–20%, at least when the response estimation is tuned to return a single response. Overall, all routing algorithms are practically feasible with either hop or time limited query termination, reaching response probabilities between 60–95%. The main difference will be in resource use, with First Contact only doing single-copy routing, Epidemic replicating to every node, and Spray-and-Wait limiting the copies to ten per message—higher resource use will result in higher response probability.

Next we add multiple different content items that match the query into the network. We take the HCS model and add 5–20 different content items that match the query. The number of copies of the different content items are Zipf-distributed with the most popular item having half the total number of copies—we choose the Zipf-distribution since it has been shown to model the popularity of Web content items well [57]. We then record for each query the mean number of unique responses as a function of the number of most popular resources.
of the content item popularity in fig. 4.9 (two responses are “unique” if they were generated by different nodes, even if the matched content item is the same). The figure only shows results for the time-limited and first drop termination heuristics because, as we saw before, the other heuristics behave almost identically to one of these two in most cases.

From the figure, we can see that the behavior matches what we would expect based on the previous single-item case results. For example, First Contact routing with time limited termination results in roughly 0.35 unique responses with the most popular content item (which has 10 copies in the network), which corresponds to the response probability of roughly 0.3 in fig. 4.8 (row 3, column 1, dotted line, 10 resources). The other matching content items have fewer copies in the network according to the Zipf-distribution, which we can see in the figures as correspondingly falling number of unique responses.

Epidemic routing has a notably better performance in this scenario (note the different y-axis) and returns multiple copies of the most popular content item, and more than half the time will also find few of the next most popular items. In contrast, neither First Contact nor Spray-and-Wait will find even the most popular item half the time—although they have a good chance of finding some item that matches the query. As explained before, the higher likelihood of responses with Epidemic (and Spray-and-Wait) comes with the cost of using more network resources.

Overall, the results indicate that within this simulation scenario, using Spray-and-Wait with TTL query termination provides a good trade-off in resource use vs. probability of finding some content matching the queries. None of the mechanisms are capable of doing an exhaustive search, or reliably returning rare content items. There is, however, a good chance that a query made by the user will find and return some matching content—and this will grow if there are more message copies in the network.

Finally, we can analyze the time it takes to search and retrieve items in the system. We do this by recording the mean latency of the responses in the order they are received in the WDM scenario with 300 content items in the network. Figure 4.10 shows the latency from starting to query to receiving the \( n \)th response using the three different routing algorithms.

The figure shows that even in the best case it takes 30 minutes to receive the first response, and then roughly 15 minutes for each additional response. Epidemic routing has the longest mean latencies to receive the content, but is also significantly more likely to find some content—this is
Finding and Manipulating Content

Figure 4.10. The mean response latency in the WDM model with 300 content items.

due to it spreading the queries more widely and hence finding more distant content. This again shows that any user experience that depends on user busy-waiting for the results of their queries will not be usable. Instead, similarly to the Web system (section 4.1.3), an asynchronous approach to building applications is needed. For example, the user's music app could make queries in the background to gather new songs from the most played artists, discovering fresh content for the user when they open the app.

4.2.4 Summary

In the beginning of this section we set out to answer the question of whether searching for content in opportunistic networks is feasible. We laid out a design for a system based on queries and responses, and developed heuristics for controlling the query distribution under different routing algorithms. We then created multiple simulation models to study the system’s behavior under different conditions. Overall, the simulation results show that the system is capable of finding content within the opportunistic network, but has significant limitations: It is not feasible to do any kind of exhaustive search, reliably find rare content items, or find content that requires many hops to get to—and the duration of the query process is in the order of an hour. This rules out the popular Internet use cases of search engines that instantaneously return thousands of content items to the user. Nevertheless, the results do show that such a query mechanism can be a useful mechanism for applications that can run long-lasting background queries and have asynchronous user experiences.
4.3 Shared Editing of Content: Publication VII

So far, the mechanisms we have studied have been for distributing, fetching, and searching for immutable content items. These are suitable for any application that can package its information into immutable messages, such as photo or video sharing, static web pages, and text messaging. Some applications, however, are based on shared editing of content and cannot be handled by the previous mechanisms.

The classical examples are wiki-pages (e.g., Wikipedia), which allow anyone to modify the articles, and shared document editing such as Google Docs. In well-connected networks, shared content editing is often implemented with a central server as the single source of truth, which accepts and orders (or rejects) modifications proposed by the clients. This, however, is not feasible in opportunistic networks where the round-trips to a server have unbounded delays, often in the order of hours in realistic scenarios.

Without special support for mutable content, the best applications can do is to send out a new version of content item every time the user modifies it locally. This will quickly result in numerous messages with different versions of the same content item congesting the opportunistic network. Further, it is unclear what the applications are supposed to do with all the different versions of the same content item that they continuously receive from the network. This motivates the research question that we address in this section and Publication VII:

- How can shared content editing be supported in opportunistic networks?

To answer this question we present a system design for a content modification mechanism that opportunistic networking applications can use for shared content editing. The content items are modeled as shared lists that applications can modify by deleting and inserting elements—Google Docs document or a wiki article can be modeled as a list of words, or a source code file as a list of lines of code. When two nodes with different versions of the same content item create an opportunistic contact, they will attempt to create a new common version of the content. We present three different ways of creating this common version: 1) Adoption, where the nodes choose to use one of the versions and discard the other, 2) merging,
where changes from the two versions are combined into a new version, and 3) hybrid strategy that combines the two other strategies. We then conduct a simulation study to evaluate the feasibility and behavior of the design.

The simulation study shows that neither adoption nor merging alone is likely to be a feasible basis for a real system: Merging can effectively propagate changes, but will eventually fail to make progress due to accumulating conflicting versions. Adoption does not suffer from this failure, but instead discards large amounts of the changes. However, the hybrid strategy that combines these two approaches reaches very high sustained rates of change propagation and could serve as a basis for a real system.

The results are limited by the fact that we do not present a real system (and thus cannot study actual user experience), and by the limited scope of the simulation study. Nevertheless, there is strong initial evidence for the viability of shared content editing in opportunistic networks.

### 4.3.1 Shared Mutable Content

Our model for shared mutable content is based on the same basic opportunistic networking mechanisms discussed in the previous sections. However, we need to modify the synchronization process performed when nodes open an opportunistic contact. In the normal case, when nodes meet they exchange content vectors that specify which messages they are carrying in their local message caches. The routing algorithm will then decide which message copies to forward between the nodes—if the peer,
for example, already has a copy of a particular message, it will not be forwarded again. We modify this process so that each node holds a local copy of a content item, which local applications can modify. When nodes with the same content item meet, they attempt to form a new, joint version of the content item—i.e., before a contact two nodes can have differently modified versions, but after a contact they will have a common version.

In this work, we model the content as an ordered list of elements that can be compared for equality. Content mutations are element addition and deletion operations on the lists. We choose lists as the underlying datatype since they map well to popular shared editing applications: Google Docs are lists of characters, wikis are lists of words, source code is lists of lines of code. Other data types, such as sets, trees or DAGs, could be useful for some applications, but they are out of scope in this work—the adoption strategy we define below will work on any datatype, but in practice more intelligent datatype-specific strategies are likely needed.

The process is shown in fig. 4.11: First, on the left of the figure, node $N_0$ creates an initial version $V_0$ of a new content item. Later, at some points in time, $N_0$ meets nodes $N_1$ and $N_2$, which do not yet have a copy of the new content item and will therefore request $V_0$ from $N_0$. Both $N_1$ and $N_2$ will independently modify their local copies of the content item to create new versions $V_1$ and $V_2$ respectively. At this point there are three different versions of the same content item in the network, although each node is only aware of their local version.

The problem arises when two nodes with different versions come into an opportunistic contact. This is shown in the middle of the figure where $N_1$ and $N_2$ meet after each has made local modifications to the content item. The nodes must then somehow reconcile $V_1$ and $V_2$ to produce a new version $V_3$ (red question mark in the middle of the figure). There are different approaches that could be taken to produce $V_3$, two examples of which (adopt and merge) are shown in the figure.

The modifications and combinations form a directed acyclic graph (DAG), where vertices are the different versions and edges show which version(s) are used to produce another version—an example is shown in fig. 4.12. In the DAG, each version, except the initial version, has either one incoming edge (modification) or two (combination). At any given point in time, a subset of the vertices is mapped to a subset of the opportunistic nodes such that each node has at most one version and each version can be in an arbitrary number of nodes.
We can identify two competing processes generating the DAG: The local modifications create a branching process (shown in blue in the figure) that causes the versions in the nodes to diverge. Opposing this are the opportunistic contacts that create a combining process (shown in red in the figure) that causes the versions in the nodes to converge. Essentially, the local modifications of the branching process create new information and the combining process distributes that information to other nodes. Our goal is, therefore, to devise strategies for the combining process such that as much of the information as possible is transmitted between the nodes.

The main hurdle to achieving this goal is conflicts. With the list datatype, two remote nodes may insert a different element in the same position in the same version of the list. If we later try to combine the two modified versions, we will have two conflicting elements competing for the same location in the list. In the face of conflicts, different combining strategies can take various actions ranging from terminating the operation, to using heuristics to resolve the conflict, to asking the user for help. Generally, how to resolve this conflict depends on the semantics of the content and cannot be reliably done automatically.

This work focuses on two combination strategies—adoption and merging—and their composition into a hybrid strategy. Adoption is a simple baseline strategy, which can be applied to any datatype and is not impacted by conflicts, but loses information in every combination. Merging is a more complex, datatype-dependent strategy that only loses information in conflicts—and even then heuristics can be used to preserve the informa-
tion, although at the cost of potentially breaking the content semantics. The hybrid strategy combines these two approaches to try to achieve better performance than either of them alone.

**Adoption Strategy:** When two nodes with different content versions meet, they will agree to pick one of the content items as the new version and discard the other. In this approach, information in the adopted version flows to a new node, but the information in the other version is discarded. The advantages of this approach are its simplicity, the ability to always make progress, and the independence of the content type—the downside is that it will continuously, and often unnecessarily, lose information.

The adoption strategy can be tuned by changing the heuristics used to select which version is adopted and which discarded. One obvious heuristic, which we study in the simulations, is to always select the version with more content (e.g., larger or more edits). This should maximize the information flow in the system, since the version with more information is preserved and the one with less information discarded every time a combination is performed. Alternatively, we can choose the version with the latest changes. This should promote the speed of information flow at the expense of the amount of information flow—we also evaluate this approach.

**Merging Strategy:** Instead of discarding one set of changes as in the adoption strategy, the merging strategy attempts to combine the two versions into a new version that incorporates changes from both versions. This is often possible since the modifications can be to different parts of the content item, and hence can both be applied to the new version without conflicts (see fig. 4.11)—in such cases the adoption strategy would needlessly discard half of the changes. This operation is called *merging* and is the basis for version control systems such as *git* [14].

There are many known merging algorithms that can be used for our system, but we base this work on the popular *three-way merge* algorithm known from the Unix *diff3* tool and many version control systems including *git*. The three-way merge is based on the ancestor relationship between the content items in the version DAG. When nodes $N_a$ and $N_b$, carrying versions $V_a$ and $V_b$ respectively, meet the merging proceeds as follows: If $V_a$ is an ancestor of $V_b$, then the result is $V_b$, and vice versa—this is called a *fast-forward* merge. Else, the lowest common ancestor (LCA) of $V_a$ and $V_b$, called $V_{LCA}$, is found in the version DAG by a breadth-first search along the reverse parent pointers. The initial version is an ancestor of all the other versions, so $V_{LCA}$ will always exist. As an example, in fig. 4.12 the LCA of
$V_1$ and $V_2$ is $V_0$, the LCA of $V_7$ and $V_8$ is $V_2$, and the LCA of $V_9$ and $V_{10}$ is $V_1$. Next, we calculate the smallest sets of addition and deletion operations, $O_a$ and $O_b$, needed to turn $V_{LCA}$ into $V_a$ and into $V_b$ respectively—this can be done by solving the Longest Common Subsequence problem [37]. The addition operations specify which element to add to which location, and the delete operations specify which position to delete from the list. If the sets of all locations in $O_a$ and in $O_b$ are disjoint then there is no conflict, and we can generate the combined version by applying all operations from $O_a$ and $O_b$ against $V_{LCA}$. Else, if there are operations in the two sets that have the same location, then we have a conflict. For example, both versions might have an addition of a different element to the same location of $V_{LCA}$, in which case we must decide in which order to add the elements, or decide to add only one of the elements, or decide to add neither, or decide to stop the whole merge operation. As discussed before, this conflict cannot be resolved in the general case without understanding the content semantics. We can, however, develop different heuristics to automatically solve the conflicts, or just pick randomly and leave it to the user fix the content if the merge breaks the content semantics.

The merging algorithm requires access to the version history DAG in order to operate. This means that each node must store the version history of their content version along with the content itself. During a contact, the nodes will exchange their version histories and combine them into a joint version history DAG. Since the version histories of each node contain all the ancestors of their versions, the combined DAG must contain the LCA. This also means that the storage required by the content item will grow continuously. However, this is also true for version control systems, where it has not prevented practical systems from being developed.

This approach can be seen as a type of operational transformation [97], which is a common way to build collaborative editing tools. Each content version is a set of addition and deletion operations on a sequence of ancestor versions. The merging algorithm transforms the operations of two versions into operations against a common ancestor, which then allows us to apply them to generate the merged version.

**Hybrid Strategy:** Both of the previous strategies have their strengths and weaknesses: Adoption can always make progress but discards information, while merging discards less information but can become unable to make progress due to unresolved conflicts. As a result, the natural hybrid strategy is to try to use the merging strategy and fall back to the
adoption strategy if the merging fails due to conflicts. This should allow us to not lose information unnecessarily, but continue to make progress in cases where conflicts arise. This can be further combined with different approaches on when to automatically resolve the conflict using heuristics and when to use adoption instead.

**CRDT Strategy:** An alternative strategy, which we do not study further in this work, would be to use Conflict-free/Commutative Replicated Data Types [236] (CRDTs). Like the merging strategy, the CRDT strategy has a series of operations (add, remove) that are applied on the list that represents the content item. However, while the merging strategy produces a carefully transformed set of operations to apply during contacts, CRDT operations are designed so that they can be applied in any order at any time. In other words, the operations in CRDTs are commutative, and the nodes in our system can simply apply any received operations against their local content versions in any order. Many CRDTs that are designed for collaborative text editing—such as TreeDoc [215] and WOOT [197]—are applicable to our problem since text is just a list of characters.

Even with a list CRDT, the fundamental possibility of a semantic conflict still exists: Two remote nodes can still insert a different item into the same location, which cannot be resolved without understanding of the content semantics. Heuristics can still be applied to attempt to automatically solve these conflicts. Furthermore, while the CRDT strategy does not require a full version history like the merging strategy, list CRDTs designed for collaborative text editing accumulate index metadata instead [215, 197].

### 4.3.2 Simulation Study

We conduct a simulation study to evaluate the feasibility and behavior of our proposed approach to shared content editing. The study focuses on the first three strategies outlined above—**adoption, merging, and hybrid**—and their ability to disseminate content modifications using opportunistic contacts. The main metric of interest is the number of modifications that nodes receive over time, which measures the information flow in the system. We construct three simulation scenarios in The ONE simulator [153]: 1) A static topology, 2) unconstrained free-space random movement, and 3) realistic trace-based movement.

The first of the three scenarios, a **static topology**, represents an optimal case where the interactions are not limited by intermittent connectivity. The topology has 61 nodes organized into a hexagonal grid so that each
node has six neighbors (except those along the edges of the topology, which have less). We choose this topology because of its regularity and ease of understanding. While the topology is well-connected with many paths between each node pair, the message propagation is still limited by the standard behavior of The ONE simulator: Only one link of each node can transmit at once and only one message per simulation step (100 ms) can be transferred. This means that end-to-end interactions are not instantaneous even though we dimension the network resources (link and storage capacity) to not become bottlenecks.

As the opposite extreme from a static topology, we use the Modified Random Direction [223] (MRD) free-space random movement model. In this scenario we place 60 nodes with 30-meter communication range randomly in an area of $1 \times 1$ km. Each node first picks a random direction and then a random point in that direction along the line from its current location to the edge of the simulation area (both uniformly random). The node then moves to the chosen point at 1 m/s, pauses for 0–120 seconds, and then repeats the process for a total of 20000 simulated seconds. We choose this model instead of the typical Random Waypoint Mobility because it results in a more even node density distribution [223, 271]. This scenario serves as the fully opportunistic case, where the connectivity is mostly via opportunistic peer-wise contacts without large connected components.

Finally, we define a more realistic scenario that attempts to model real human mobility characteristics. While in other studies in this thesis we have used the Helsinki City Scenario as a more realistic model, here we want a model that better replicates the user communities within which the shared content editing occurs. Therefore, we choose the SMOOTH [187] trace-based mobility model with 60 nodes and seven days. SMOOTH is a synthetic mobility model with a small set of parameters whose values can be derived from real-world mobility traces. When parametrized this way, the resulting mobility characteristics should closely match those in the trace. We use the parameters derived from a human mobility trace at the KAIST university campus [164] exactly as reported by the authors of the model (Table 4 in [187]). We call this the SMOOTH-KAIST scenario, and it should realistically model the mobility of students on a campus.

In each of the scenarios, every node starts with the initial version of the content item, which is a list with 100 elements. The nodes will periodically modify their copy of the content item by inserting five elements in random locations in the list. In the first two scenarios the nodes modify the content
Finding and Manipulating Content

Figure 4.13. Content growth due to information flow in the adoption strategy.

on average every 1000 seconds, and in the last scenario every 8.4 hours—on average this results in 20 modifications per node over the simulation duration for every scenario. We do not consider deletions of elements from the list for two reasons: First, without deletions the content length will directly reflect the amount of information that has accumulated in the content item, which gives us a simple metric for analysis. Second, deletions cannot create conflicts (although they can remove conflicts), so leaving them out gives us a worst-case bound on the system behavior. Note that in all the results in the figures, we subtract the local modifications made to the content and thus only report the elements that have been received from others, which makes the analysis clearer.

**Adoption strategy:** We consider two heuristics with the adoption strategy, *adopt longest* and *adopt latest*. In the former, when two nodes with different content versions meet, they will both adopt the version with more elements—in the latter, they will both adopt the version with the latest changes. The resulting growth in content length over time is shown in fig. 4.13 (again note that the figure does not include growth due to local modifications, only due to new versions received from others).

On the left of the figure, we can see that both longest and latest heuristics perform identically in the static topology. This is because messages propagate quickly in the fully connected topology, and as a result other nodes receive the new version of the content item shortly after it has been changed. At that point the received version will be both the longer than the old version and contain later changes, so it will be adopted by both heuristics. However, we can also see that the adoption strategy does not reach the maximum possible performance (blue line in the figure). This demonstrates the information loss caused by the adoption strategy: Even in near ideal conditions, there is still a propagation delay for the new
content versions. If another node modifies the old version during this propagation delay, those modifications will be lost as only one of the versions will be adopted and the changes in the other will be lost.

In the random free-space mobility, in the middle of the figure, we can see the different behavior of the two heuristics. Adopting the longest version during every contact greedily maximizes the length of the content item, while the adopting the version with the latest changes will sometimes pick the shorter version. As a result, the content growth (and information flow) when using the longest adoption strategy is almost twice as fast as when adopting the latest version: Longest adoption grows at the speed of 39 elements per hour, while latest adoption grows by 20 elements per hour in the MRD scenario. This rate of 20–40 elements received per hour, however, is far below the editing rate in the system, which is 1062 elements per hour on average—i.e., 2–4% of the created information will flow to the peers via the opportunistic network. This shows the weakness of the adoption approach when connectivity is poor and large amounts of content edits will be discarded during contacts.

On the right of the figure, we can see that the behavior of the realistic scenario is between the two extremes. Adopting longest has higher content growth rate than adopting the latest changes, but the difference is not as large as in the fully opportunistic case. The difference is 35% (vs. 100% in MRD) with longest adoption strategy receiving 8.6 elements per hour and the latest adoption strategy 6.4 elements per hour. Since the content modification rate is also lower in the realistic scenario (35 elements per hour), we reach much higher information flows than in the MRD case—25% and 18% for the two heuristics (vs. 4% and 2% in MRD). This behavior reflects the nature of mobility in the SMOOTH-KAIST scenario: Nodes spend most of their time in clusters (which approximate the static topology) with infrequent flights between them (which is closer to the MRD scenario).

**Merging strategy:** We consider pure merging where the operation fails if there are any conflicts in the versions, as well as automatic resolution of various sizes (size being the number of conflicting locations in the list that can be automatically resolved). For the automatic resolution we consider *destructive* and *preserving* strategies, which differ in the way they deal with the conflicting elements.

These approaches work as follows: When nodes A and B carrying versions $V_a$ and $V_b$ meet, they will run the merging algorithm described in
Figure 4.14. Content growth due to information flow in the merging strategy.

The previous section to generate change sets $O_a$ and $O_b$ against the LCA version $V_{LCA}$ containing (position, value)-tuples for each addition operation. We then have a conflict set $C = \{((a_p, a_v), (b_p, b_v)) | (a_p, a_v) \in O_a, (b_p, b_v) \in O_b, a_p = b_p, a_v \neq b_v\}$. I.e., pairs of operations, one from each change set, such that they add a different element to the same location in $V_{LCA}$. The pure merging operation will succeed only if there are no conflicts ($|C| = 0$), in which the new version $V_{ab}$ is created by applying all operations from $O_a$ and $O_b$ against $V_{LCA}$. The destructive automatic resolution will pick one side of each conflicting element to apply, i.e., for each element $(o_a, o_b) \in C$ we pick either $o_a$ or $o_b$ and apply that. If the conflict set has more distinct values for the same location in one change set, we pick all the values from that change set. Otherwise, we pick arbitrarily but consistently (each node will always make the same choice for the same location). In the preserving automatic resolution we apply both operations for every conflict in an arbitrary but consistent order. Finally, we consider various limits $n$ on the size of the conflict set such that the merge fails if $|C| > n$. We will denote a merging strategy that applies all conflicting operations as long as there are less than three conflicting positions as “destructive 3”.

Figure 4.14 shows the growth and rate of growth for content length for the different strategies in the three scenarios. In the static scenario, on the left of the figure, we can see that merging strategies with any automatic resolution mechanism reach the maximum possible performance—higher than adoption strategies in the same scenario. This is because the few concurrent modifications that occur during the message propagation can
be merged together, rather than discarded as in the adoption strategy. However, if these concurrent modifications introduce a conflict, the merging strategy with no conflict resolution fails. As we can see in the figure, this occurs frequently enough in the static scenario that a pure merging strategy will perform worse than the adoption strategies. This is because with the pure merging strategy, two conflicting versions will permanently split the node population in two—with each half modifying a different version—which means that the changes received by each node will also be cut in half. This splitting can occur multiple times, with each split causing a corresponding fall in information flow (the steps in the bottom-left graph in the figure might indicate such splitting events).

When we introduce a high degree of mobility and disconnections in the MRD scenario (middle of the figure), we can see that the behavior of the merging strategies changes. Initially the merging strategies reach significantly higher rates than the adoption strategies (up to 172 elements/hour, 16% of max rate vs. 30 elements/hour, 3% of max rate), but the rate quickly falls to close to zero. This is due to unsolvable conflicts accumulating in the different versions over time. Since in the MRD scenario the message propagation time is large, it is likely that two versions get modified in a conflicting way before they get merged together. As a result, conflicts accumulate until eventually the version in every node conflicts with every other node, causing the information flow to halt. How long before this happens depends on the merging strategy: With no ability to automatically solve conflicts, the content size will only increase by 15% before becoming conflicting. This increases up to 160% when using the preserving strategy that tolerates up to ten conflicting positions, with the remaining strategies falling between these two extremes.

The picture once again changes in the more realistic SMOOTH-KAIST scenario (right of the figure). The content growth rates are still falling, but so slowly that the content growth is almost linear. The best performing strategy, Preserving 10, reaches a rate of around 13–14 elements per hour, which is 40% of the maximum possible rate. Other strategies perform worse, but even a small amount of automatic conflict solving ability achieves good performance—unlike in the MRD scenario where it does not improve performance much. This is as expected considering the characteristics of the scenario: Most of the edits occur while the node is in a cluster resembling the static topology and the merging strategies are effective. However, conflicts can build up between the clusters if nodes do not travel
between them frequently enough, and as a consequence the system cannot reach the maximum possible information flow rate.

**Hybrid strategy:** As we have seen, merging strategies have a potential of high information flow rates but suffer from conflicts, while adoption strategies have low information flow rates but do not have conflicts. The natural way to combine these strategies into a hybrid strategy is to first attempt a merge and if it fails then apply the adoption strategy. We can combine the different versions of merging and adoption in any way we like, so we test both latest and longest version adoption in combination with both destructive and preserving merging with a maximum conflict size of three. Since the merging strategy already achieves optimal performance in the static scenario, we only consider the MRD and SMOOTH-KAIST scenarios. The results are shown in fig. 4.15.

On the left of the figure, we can see that in the more challenging MRD scenario the hybrid strategies significantly outperform both merging and adoption strategies, and the preserving variants outperform destructive variants. There are two important characteristics visible: First, the hybrid strategy does not exhibit the falling rate of content growth seen in the merging strategy. It means that information will continue to flow in the system indefinitely. That is expected, since the hybrid strategy will always revert to adoption rather than failing when there are conflicts. Second, the rate of growth of the hybrid strategies is larger than in the adoption strategy. This means that the strategy does not simply degenerate into adoption, and the content merges keep occurring throughout the process. As a re-
sult, the content growth rate reaches around 230 elements/second (hybrid with preserving merging and longest version adoption), which is 22% of the maximum possible rate—this is almost 2.5 times the rate reached by equivalent merging alone (94 elements/second, 9% of the max), and almost 7.5 times the rate of the adoption strategy alone (31 elements/second, 3% of the max). Applying a merge strategy with more conflict tolerance would naturally lead to even better performance.

In the more realistic SMOOTH-KAIST scenario, the behavior is similar although the performance improvement of the hybrid strategy over the pure strategies is not as large. This is because the pure strategies by themselves already achieve high performance. Importantly, however, we can reach over 60% of the maximum information flow with the hybrid strategy, with even higher rates achievable with more conflict tolerance in the merging component. This shows that in realistic scenarios the hybrid strategy can propagate most of the information generated by the users, indicating that the proposed approach to shared content editing may be feasible in practice.

Overall the simulation results reveal the fundamental trade-offs in the proposed strategies: Merging can quickly propagate changes, but will eventually be overcome with conflicts that cause the system to not be able to make further progress. In contrast, adoption can always make progress, but has a very high rate of information loss, meaning most of the user contributions to the content will be lost in the process. The hybrid strategy appears to be able to combine the benefits without the drawbacks—it can always make progress while having a high information flow rate.

Further, the behavior is highly dependent on the underlying mobility. Within well-connected segments (static scenario or clusters of SMOOTH-KAIST), a merging strategy with a small amount of automatic merging ability can reach optimal or close to optimal performance. In contrast, in fully opportunistic scenario (MRD) the merging strategy is not viable and will stop making progress due to conflicts even with high amount of automatic merging ability.

In any case, the ability to automatically merge some conflicts is crucial to the performance. For a practical system, this means that the implementations should employ heuristics to perform merges even though this will inevitably occasionally break the content semantics. This does lead to the user being presented with content that has weird issues, but the nature of
the system means that the user can simply edit the content and correct the semantics. Whether this is viable in practice or leads to too poor user experience will require a real implementation and user studies.

4.3.3 Summary

In this section we set out to answer whether opportunistic networks can support shared content editing in addition to dissemination immutable content. Conceptually this means that each node in the network may have a different version of the same content item, and these versions must somehow be reconciled during opportunistic contacts. Multiple approaches can be taken to achieve this: Adoption of one version, merging of the versions, or using CRDTs. The simulation study shows that neither adoption nor merging alone is adequate for a practical system. However, by combining the merging and adoption strategies, we can reach information flow of over 60% in a realistic scenario. This provides initial evidence to answering the research question in the affirmative—it seems possible to support shared content editing applications, such as wikis and Google Docs, in opportunistic networks.
4.4 Conclusions

This chapter develops mechanisms for content interactions in opportunistic networks that go beyond simple send/receive and publish/subscribe interactions. The request-response interactions with origin servers on the Internet, searching for specific content within the opportunistic network, and shared editing of content provide building blocks needed by many modern networked applications. The results show that each of these mechanisms is viable, but implementing them in an opportunistic networking context has challenges and limitations. In most cases, these limitations mean that the applications must be designed with new approaches to user experience, rather than serving as drop-in replacements to existing applications—users should not be expected to busy-wait for a website request or content search to complete, but rather the application should be continuously searching and requesting in the background, so that content is ready immediately when the user opens the application.

The first mechanism, request-response, uses websites as the driving use case. It is unlikely that every website in existence has a copy in the opportunistic network, or is directly reachable via it. So, to support the Web, we need a way to bridge website requests made by clients in the opportunistic network and the web servers on the Internet. We can do this by using the Liberouters or opportunistic networking enabled Wi-Fi access points to serve as bridges that know how to locate and fetch the requested websites from the Internet. The simulation study shows that we can likely reach high enough success rates to make the approach feasible, and the prototype implementation shows that the system is realizable in practice. While we target the Web, the approach is applicable more widely to any opportunistic networking application that needs to reach for content stored in infrastructure servers.

The second mechanism, query-response, enables applications to search for content matching some criteria within the caches of their peers in the opportunistic network. This leverages the fact that opportunistic networking by its store-carry-forward nature results in many message copies being stored around in the network. The search mechanism makes this wealth of content available to any node within the network. Our simulation results show that it is feasible to build such a system, but with significant limitations. It is not, for example, feasible to do any kind of exhaustive search or reliably find rare content items. Nevertheless, applications can run such
a query mechanism in the background to continuously find interesting content around them.

The final mechanism, shared content editing, enables more interactive applications, such as wikipages and Google Docs, to be built on top of opportunistic networking. This has the fundamental problem of conflicts being introduced when different users modify the same part of the content at the same time. The long delays of opportunistic networking mean that the nodes learn about each other's edits infrequently and as a result are likely to make conflicting modifications. However, our simulation study shows that it is possible to construct strategies for combining different content versions in a way that provides high success rates in realistic mobility scenarios. While it will not be possible to build a real-time experience like, e.g., Google Docs provides, it is possible for opportunistic networking users to collaborate asynchronously similarly to the way wikipedians collaborate when editing Wikipedia.

The set of three mechanisms explored in this chapter is only a small subset of the possible approaches to content interactions in opportunistic networks. Future work could include advanced optimizations to the underlying content distribution system—which in the Scampi router is publish/subscribe-based—and their interoperation with these more complex mechanisms, such as the shared content editing. For example, in the request/response and query/response systems, content could be proactively placed and indexed to make it more likely to be found; and in the shared content editing system, some nodes could be designated as authoritative merging points rather than having every node do merging.

In the next chapter we will move beyond content-based mechanism and consider how to move and compose computations when building opportunistic networking applications and services.
5. Moving and Composing Computations

In the previous chapter, we developed advanced mechanisms for handling content in opportunistic networks. The content or data, however, is only one part of the story of distributed systems—the other part being the computations that operate on the data. So far we have assumed that those computations are somehow statically installed as applications on the devices that make up the opportunistic network. In this chapter, we explore another idea: Can we dynamically distribute and execute the computations in the opportunistic networks the same way that we distribute the data?

In a sense, this generalizes the concept of an opportunistic contact to not only be an opportunity to forward data, but to also instantiate and invoke computations on the peer devices. A concrete benefit of such a capability would be to remove the requirement of applications being pre-installed in the devices. This is an important problem in opportunistic networks where a global app store is not always accessible via an Internet connection, and which may have devices such as Liberouters that are disconnected and not actively managed—solving it has the potential to improve the adoption of opportunistic networking. More generally, it could create a new, local execution environment for distributed services, which could be used instead of (or in addition to) the modern cloud computing backends. Being able to dynamically instantiate distributed services in the local environment could enable new types of applications and services with better privacy and latency characteristics, and which can make better use of the computational and other resources that exist in our increasingly smart environments.

We start in Publication VIII by extending the opportunistic web access work of section 4.1 to support web applications in addition to just static websites. The major problem with web apps in opportunistic networks is the need for frequent interactions between the browser and the backend
logic, which is not feasible when the latencies are unbounded. We solve this by bringing the backend logic into the opportunistic network so that it is directly accessible via opportunistic contacts. Concretely, we attach small pieces of computation to content messages, which a web app framework running in a Liberouter can use to automatically synthesize a backend to serve web apps accessible via standard web browsers in directly connected clients. The framework also enables interoperability between the pre-installed native opportunistic networking apps and the web apps. We present a system design, a prototype implementation of the framework for Liberouters, and create web app adaptations of all the native Liberouter apps described in section 3.3.3. Simulations and testbed evaluation of the prototype implementations provide evidence that the approach is feasible and can support web apps in opportunistic networks.

We then extend the dynamic execution capabilities from web apps to a more general opportunistic service composition system in Publication IX. The goal is to provide a dynamic execution environment where interconnected computations can be instantiated and executed opportunistically—this is somewhat analogous to modern backend systems that allow microservice containers to be dynamically instantiated and executed in data centers. We want this execution environment to encompass not only infrastructure devices such as Liberouters, but also the mobile devices carried by the users—it should also encompass graphical user interface elements in addition to computational service elements. We present a system design for achieving this, along with a prototype framework that implements it on Liberouters and Android devices. Further, we develop an example classroom polling service using the system, which can be dynamically instantiated into the local infrastructure during a lecture. A testbed evaluation of the polling service shows the small-scale feasibility of our approach for dynamically instantiating and operating such services, and also demonstrates its robustness against the types of disruptions that are frequent in opportunistic environments.
5.1 Opportunistic Web Apps: Publication VIII

In this section we extend the work on Liberouters (section 3.3) and opportunistic web access (section 4.1) to go beyond fetching static websites to support interactive Web applications. We do this by introducing opportunistic computation alongside the opportunistic content distribution mechanisms. In this section, our model of computation is simple and focused on supporting the use case of Web applications, while in the next section we will develop a system for more general opportunistic computing.

In the previous chapter, we saw that it is possible to design a system for clients in an opportunistic network to fetch websites from the Internet. The major limitation in that design is that it only works with static, self-contained websites. While many types of websites fit this constraint (e.g., news sites), increasingly many popular websites are interactive web applications that cannot be delivered as static bundles. These websites—such as Instagram, Twitter or Facebook—have constant interactions between the web browser and the backend servers driven by user actions such as posting, liking, and commenting on the content. Given the arbitrarily long latencies of opportunistic networking, we cannot support these types of websites with the mechanisms we have studied so far.

Furthermore, these dynamic services are often not purely web-based, but also offer native mobile apps to access them. In section 3.3.3 we saw how to build native opportunistic networking applications for Instagram-style photo sharing, Twitter-style broadcast messaging, and Facebook-style social networking. However, we have no means to enable web clients access to the content of these native applications similarly to the Internet-based services, instead requiring native apps to be installed on all the devices. Enabling web browser based access to the native opportunistic networking applications would significantly lower the barrier of entry and potentially promote adoption of these apps and technologies.

To tackle these limitations of our previous designs, we formulate the research questions for this section:

- Can we support interactive web applications in opportunistic networks?

- Can we provide interoperability between native and web browser based opportunistic networking applications?
We answer these questions by designing, implementing and evaluating a web-based framework for opportunistic networking applications. The key idea that we leverage is to attach custom computations to the messages, which can be executed by nodes to provide interactive application features. In particular, we extend the Liberouters (or throwboxes [278] in general) to serve web apps to connected clients by using the interaction scripts. The Liberouters then essentially become opportunistic web server infrastructure, which loads and serves web apps directly from the messages circulating in the network. Unlike the system in section 4.1, the Liberouters do not need to be Internet connected and can act fully autonomously, and the content can come directly from native apps and web browsers rather than being fetched from origin servers on the Internet.

5.1.1 System Design

On a high level, our goal is to turn Liberouters in an opportunistic network into web servers from which we can serve interactive web apps to standard web browsers of directly connected clients—fig. 5.1 shows this idea visually. On the right of the figure, we have an opportunistic network with an underlying router (such as Scampi) and native apps running on mobile devices. As we saw in section 3.3, light-weight infrastructure (such as the Liberouters) may be added to improve the opportunistic connectivity and to provide additional services. The figure shows such a Liberouter device on the left. The design we saw previously had the Liberouter run
an opportunistic router and a web server serving static content (such as the native apps) whose services nearby devices can use. Here, however, we want to extend the web server on the Liberouter to serve interactive web apps. These web apps will consume and produce the same underlying messages as the native apps, allowing for seamless interoperability similarly to modern web services such as Instagram or TikTok that provide both a website and a native mobile app (shown as a green arrow in the figure).

This immediately presents two problems: How can developers build web apps for this opportunistic web server infrastructure that differs completely from the current well-connected cloud infrastructure? And how are these apps distributed and loaded into the disconnected Liberouter devices?

The solution that we propose is to attach the web app logic directly to the content messages. Whenever a Liberouter receives one of these content messages, it can extract the executable code needed to instantiate a web app to interact with that content. For example, in the case of a photo sharing application such as Instagram or Guerrilla Pics (from section 3.3.3), each photo message includes executable code to render it into an HTML page. The message also carries code to interact with the content from the web browser, such as adding comments or posting new photos. The framework then wires these functions to HTTP endpoints to make them available to web browsers. Essentially the Liberouter synthesizes web app backends for the applications whose content it has received.

While the approach may be conceptually simple, we now face the problem of actually defining a practical model for these computations. Given that all the executable code is attached to all the messages, we must minimize

![Diagram showing the opportunistic network and web app framework.](image)
their overhead. This means that we cannot follow the typical approaches of virtualizing web app backends, such as containers or virtual machines.

As a solution, we propose a generic web app framework that is pre-installed into the Liberouters, with pre-defined hooks for custom logic, such as for rendering the message contents as an HTML page. Only these minimal custom functions need to be written by the app developer and attached to the messages—we call these transformations.

Figure 5.2 shows this idea. In the lower left of the figure, we can see the generic application model of the framework, which is based on the Model-View-Controller paradigm [157] and has previously been applied to web development [167]. The model holds the state of the application, which we assume to comprise content items (e.g., photos of a photo sharing app), the view presents the contents to the user (e.g., creates an HTML grid of photo thumbnails), and the controller handles user input (e.g., commenting on a photo). We define transformation functions for each one of these three components (shown in the middle of the figure):

- **Model.AddToState**: Updates the local state by adding an incoming message. E.g., adds an incoming photo to the local photo collection.

- **Model.DeleteFromState**: Deletes a previously added content item from the local state. E.g., a user deletes a photo from the local collection.

- **View.AggregateView**: Renders an aggregate view of all the items in the current model. E.g., a grid of photo thumbnails.

- **View.MessageDetailView**: Renders a detail view of a single content item. E.g., a high resolution photo and its metadata.

- **Controller.NewPost**: Creates a message that contains a new content item from user input. E.g., a photo that the user wants to publish.

- **Controller.Reply**: Creates a message that contains a reply to an existing content item. E.g., a comment on a photo.

The transformation functions that we have chosen above implement a specific application control flow structure—in this case an Aggregate-Detail structure as shown in fig. 5.3. The figure shows how the different trans-
formations drive the different interactions within the abstract application. The flow has two main views, the aggregate view and the detail view. The former displays an overview of all the content in the local state and allows the user to choose a specific item whose details are then shown by the latter view. This fits applications such as photo sharing where the basic control flow is a gallery view from which users choose photos to view.

Figure 5.4 shows an example of how a native app could be mapped to the Aggregate-Detail model\(^1\). The content items in the example are “cards” that include an image, a text description and some metadata (e.g., the creator). Users of the native application can create these cards from their mobile phones, as shown on the left of the figure. When a Liberouter receives a card, it executes the \textit{AddToState} transaction to add the card into the state storage (shown in blue in the figure). When a web client accesses the app, the framework executes the \textit{AggregateView} transformation to generate a list of all the cards (middle of the figure). If a user wants to reply to a card, they can add a comment at the bottom of the aggregate view, which executes the \textit{Reply} transformation. The detail view of the card, i.e., the full resolution image is created by the \textit{MessageDetailView} if the user clicks on a card in the aggregate view.

We have chosen the Aggregate-Detail model due to its simplicity and apparent generality as we find that it can represent all the native Liberouter applications we saw in section 3.3.3. It does, however, limit the type and complexity of applications that can be supported, since they must fit the constraints of the model and transformations—e.g., map-based apps, or apps with special functions like Corona trackers cannot be supported. To support further types of applications, new control flow models can be defined, implemented and installed in the Liberouters. To reduce the

\(^{1}\)We use Here & Now as the example here, although our actual implementation of it is different, as described in the next section.
additional complexity, these new models can reuse the transformations from other models. For example, the detail view transformation does not need to be redefined even if a new model replaces the aggregate view with a different approach. This reduces the burden on the developers, who only need to write one detail view transformation even when supporting multiple application models.

As explained above, the transformation functions are distributed to the Liberouters by attaching them to the content messages. This, however, poses a bootstrapping problem since there must be content sent into the opportunistic before the web app will be available. We propose to alleviate this problem in two ways: First, we use the app store function of the Liberouters to distribute native apps that attach the transformation functions to all the content that they send out. Second, we send these native apps as messages into the opportunistic network, and attach the transformation scripts to these messages as well. This means that both the native and the web versions of the applications will be distributed automatically and concurrently into the Liberouters by the opportunistic networking.

### 5.1.2 Prototype Implementation

To show the practical realizability of the system design, we implement a prototype of the framework on the Liberouter platform. Figure 5.5 shows the components and high level design of the implementation. At the bottom of the figure is the router or middleware that implements opportunistic networking and contains the message cache. Above, the

---

**Figure 5.4.** Example of defining a web app version of Here and Now (see section 3.3.3) with the framework.
DTN Communicator component interfaces with the underlying router to read and write messages—this allows rest of the framework to be reused with different opportunistic routers. Next, the two components in purple, Message Generator and Message Processor, translate between the data model within the framework and network messages. The State Storage component, shown in red in the figure, maintains all the runtime state of the framework. At the top of the figure, the Web App Server implements the application logic and serves the web browser clients. The transformation function execution is done inside an isolated Sandbox for security reasons, as shown on the right of the figure.

When a new message is received by the opportunistic router, it is picked up by the DTN Communicator and passed to the Message Processor. The Message Processor parses the message, potentially executing some transformation functions, and adds the new content (and possibly functions) to the State Storage. Later, when a web browser does a REST call to, e.g., get the aggregate view, the Web App Server reads the content item from the storage and executes the transformation script to generate HTML from the contents, and returns it to the browser.

In the opposite direction, a web browser makes a REST call to the Web App Server endpoint to, e.g., post a new content item. The server then creates a new item from the received parameters, potentially executing transformation functions, and writes it into the State Storage. Immediately
after, the server invokes the *Message Generator*, which pulls the item from the storage and generates a message from it (including adding the transformation functions). The generator then passes the message to the *DTN Communicator*, which passes it on to the underlying router.

As we target the Liberouter framework, we use Scampi as the underlying opportunistic router. We have, however, validated the design by also interfacing with the popular IBR-DTN [89] Bundle Protocol router. We implement the *DTN Communicator* that interfaces with Scampi as a standalone Java application. To receive messages from the router, we implement direct parsing of the Bundle Protocol [233] bundles from the cache files of the router. While the more orthodox approach would be to use the Scampi API to receive messages, directly parsing the bundle files allows us to reuse the same code for any router implementation that directly serializes Bundle Protocol bundles to disk. To send messages, we use the Scampi API, as writing bundles directly into files does not work due to Scampi not expecting and processing new files written into their cache directories by third parties. The *Message Generator* and *Processor* are integrated with the *DTN Communicator*. The processor parses the Scampi message format from the bundle payload of the Liberouter apps, and generates content items that are passed to the *State Storage*. The generator works in the opposite direction, generating Scampi messages using the Scampi API library from content items received from the storage.

We use the *Redis* [27] in-memory data structure store to implement the *State Storage*. Redis runs as a separate process, and acts as a central hub to which we interface rest of the components of the system. Since the entire state of the framework, including content items and transformation functions, is contained within the messages of the opportunistic router, we can always rebuild the complete state by re-processing all the messages. This means that the state storage can be held entirely in memory and persistence to disk is not needed—although it is provided by Redis and can be used if desired. The state is stored in collections of application specific data structures within the Redis instance. The framework does not need to understand these application specifics since the custom transformation scripts will translate between messages, data structures and HTML views.

The transformation functions are written as Python scripts in our prototype implementation, and executed by a Python interpreter running on the system. Scripts that output HTML can use the Bootstrap [5] front end framework to generate rich HTML without needing additional CSS style
files. Since Python scripts can potentially interact with the underlying system in arbitrary ways—e.g., reading and writing any accessible files on the file system—we need to execute them in a heavily constrained sandbox. We construct the sandbox by limiting the scripts to a single directory with the data relevant to each execution instance using the UNIX chroot system call. We further limit the system calls that the executing script can make by using the Linux kernel seccomp-bpf facility that allows us to filter out dangerous system calls such as those related to file manipulation.

The Web App Server is implemented as a Node.js [22] application, which exposes the necessary REST endpoints for the web clients. It implements the generic application model described in the previous section and executes the transformation functions as necessary. For example, the server has EJS [13] templates for the aggregate and detail views with empty div elements that are then filled in with the output from the AggregateView and MessageDetailView transformation functions (the Python scripts executed in the sandbox). In addition, we expose a generic REST API for querying the data in the State Storage, which can be used by modern frameworks such as React [26] (which we use for the more advanced example application below). Finally, the same web server implements the app store, from which the users can download the native web enabled Liberouter apps using a web browser.

To validate the approach, we have created web adaptations of all the Liberouter apps described in section 3.3.3 using the framework implementation. Figure 5.6 shows example views from three adapted Liberouter applications along with their native versions. The top left of the figure shows an example of a detail view transformation from the People Finder application. The content items for this app are people-records and notes attached to those records. The AddToState transformation collects the records and their associated notes as entries in the state storage, and the MessageDetailView transformation renders the HTML view for the record. The bottom of the figure shows an example of an aggregate view for the Guerrilla Pics application where the content items are photos. The AddToState transformation collects these photos into a collection in the State Storage, out of which the AggregateView transformation generates an HTML list showing the photo thumbnails and creation dates. Finally, on the right of the figure, we can see the adaptation of Here and Now, which we use as a showcase for more complex app design. Unlike the
other apps, we implement it as a single-page React app, which is similar to how many modern web apps are implemented. Instead of relying on the Aggregate-Detail model, it uses a custom transformation to continuously poll the State Storage for new data. When new data is received, the app updates the contents without requiring the web page to be reloaded. This demonstrates the escape hatch from the restrictive Aggregate-Detail application model, but comes at the cost of significantly higher overhead as the entire custom React app must be included in every message.

5.1.3 Evaluation

To evaluate the feasibility of the design, we conduct simulation and testbed experiments. As we have previously established the feasibility and behavior of the Liberouter apps (see section 3.3), in this evaluation, we want to establish an initial understanding of the impact of the overheads caused by attaching transformation functions to all their content. Further, study the content reach to understand how well the system can serve web-only users given different numbers of Liberouters in the system. And finally, we want to establish the practicality of the system by experimenting with the prototype implementation in a small but realistic testbed environment.
Similarly to the previous sections, we use The ONE [153] and the downtown Helsinki map as the basis for our simulations. We create three scenarios for the three experiments—overhead, content reach, and testbed. In all scenarios the mobile nodes in the simulation follow shortest path movement between randomly selected points on the Helsinki map with uniform random speed in $[0.5, 1.5]$, which corresponds to pedestrians. Both the overhead and content reach scenarios have 50 nodes with full opportunistic networking capability, while the testbed has 20 and 30. In the content reach scenario, we add 50 and 500 web-only nodes (for 1:1 and 10:1 ratio of web nodes to full nodes) and 10, 65 and 325 randomly placed Liberouters. The web-only nodes have the same mobility characteristics but can only communicate with the Liberouter devices, not with the full nodes or each other. Note that the web-only nodes can carry messages between the Liberouters, which assumes that the user keeps the web app loaded when moving between Liberouters, and that the browser can upload the web content to other Liberouters—we have demonstrated such a mechanism in previous work [189]. In the testbed scenario we add 10 Liberouters and no web-only nodes (instead we connect a real web client to each Liberouter in the testbed to send and receive live test traffic). In all scenarios we assume a radio range of 50 m and transmission rate of 2 Mbps. Messages are generated by a randomly picked node every 12, 60, and 300 seconds—corresponding to high, medium, and low load—they expire after 5400 seconds, and are spread epidemically between the nodes.
To go beyond simulations into practical evaluation, we construct a testbed to run the prototype implementation (section 5.1.2). The design is shown in fig. 5.7. The testbed comprises an emulation server with Intel Xeon E5-2640 v4 2.2 GHz CPUs with a total of 40 threads and 756 GB of RAM available to virtual machines (VMs) running on OpenNebula. On this server we run 10 VMs with the full web framework stack, and 20 and 30 VMs with only the Scampi router acting as opportunistic carriers. Each web framework VM has 6 GB of RAM and one vCPU running Debian 9.7, while the carrier VMs have 4 GB of RAM and 0.5 vCPUs running Debian 9.6. The link creation between the Scampi routers in the carriers is controlled by an external topology controller, which is fed the simulation trace from the testbed simulation scenario. The topology controller runs on a separate test controller server, which has Intel Xeon E5-2640 v3 2.6 GHz CPUs with 32 threads and 132 GB RAM. The controller server also runs web clients that send and receive the test traffic. We use Selenium [28] scripts connected to each web framework instance, which generate new Here and Now web app messages every 10 minutes and poll for new content every 5 seconds—Selenium uses a real web browser (Chromium) as a driver. The two servers are in the same rack and connected by 10 Gbps wired link.

**Overhead:** The first characteristic of the system design that we want to understand is the impact of the overhead. For this we measure the overhead generated by the prototype implementation for the Guerrilla Tags text messaging (very small content size) and Guerrilla Pics photo sharing.
Moving and Composing Computations

The measured overhead for Guerrilla Tags is roughly 16 KB per message, which corresponds to nearly 50 times overhead for the small text messages. For photo sharing with larger content messages, even for small photo sizes of 65–120 KB the measured overhead is a more reasonable 5–10%. This indicates that to support applications with very small message sizes, more compact representations are needed to reach a reasonable overhead level—for example, highly optimized byte code could be used instead of Python code encoded as strings.

To understand what the impact of this overhead is on the network performance, we parametrize the message sizes in the overhead simulation scenario with those measured from the implementation. As our goal is to simply understand if the additional message size impacts message spreading, we do not simulate the framework nodes in this scenario. Instead, we compare native (non-overhead) case to the framework augmented (overhead) case for three different load levels for the two different applications.

Figure 5.8 shows the results of the simulations as a cumulative distribution function of the message spread over time. We can see that for all cases there is no significant difference in message spreading between the native and framework-augmented behavior. This indicates that the overhead added by the framework is not significant enough to impact the message spreading. In the right figure, we can see that the highest load case with the largest messages has worse performance than the other cases. This is likely due to the network being congested under this load, and indicates that even in a congested network there does not seem to be a performance difference due to the framework overheads. While these results indicate the feasibility of the design from the overhead perspective, given the simplicity of the simulation model, a more detailed study is needed to understand the impact in detail.

Content reach: Next, we want to understand how well the system can serve web-only users. To study this we add Liberouters and web-only nodes to the simulation, as explained above for the content reach scenario. The web-only nodes can only receive content directly from a Liberouter and cannot exchange messages between each other or with the carrier nodes. We are interested in how well these web-only nodes receive content.

The simulation results are shown in fig. 5.9. The figure shows the cumulative distribution function for the fraction of nodes reached by messages over time after their creation. We show results for three cases with low (10), medium (65) and high (325) number of Liberouters added to the simulation.
Moving and Composing Computations

We also test the difference between having an equal mix of web-only and carrier nodes, and having 10 times more web-only nodes than carriers.

The figure shows that we can reach significant coverage (over 40%) of the web-only nodes even with just ten Liberouters running that web framework—comparison to fig. 5.8 shows that this reach is similar to native nodes. Increasing the number of Liberouters in the system increases the spread of the content, ultimately reaching over 60% with both 65 and 325 added Liberouters. The content reach for both small text messages and larger photos is qualitatively similar, with photo sharing performing slightly worse, likely due to the additional congestion caused by the larger message sizes. Finally, we can also see that adding more web-only nodes also improves the performance significantly. This is due to the web-only nodes acting as carriers between the Liberouters, improving the content spreading in the system.

As before, these results are based on a limited set of simulations, but provide a preliminary indication that the approach has potential to serve a significant fraction of users via web browsers without the need for native apps. They also hint at the importance of developing the offline capabilities of web browsers: Since the user is unlikely to be in range of a Liberouter the moment they want to use an app, it should be possible to keep the app loaded and functional in a browser tab even when the user doesn’t actively interact with the site. Further, using the web browsers as carriers between Liberouters significantly improves the performance of the system.

**Testbed:** Finally, we validate the prototype implementation via testbed
experiments. As described before, we run ten virtualized instances of Liberouters with the full web framework prototype, and 20 or 30 virtualized mobile nodes that carry messages between the Liberouters. We then send and receive Here and Now web app messages with a test client driven by a Chromium browser engine. Since the test traffic is generated and consumed by test clients permanently connected to the Liberouters, these experiments give the upper bound of the performance. In reality the performance is degraded due to the nodes having to reach a Liberouter before being able to send or receive messages. We are also limited in the number of virtualized carriers we can run in the testbed, so the results are not directly comparable to the simulations where we can run significantly more nodes. We record the end-to-end latency between the test web clients, and the coverage over time. Figure 5.10 shows the results of the evaluation.

The left of the figure shows the latency and coverage in the experiments. From the coverage graphs we can see that the content published to one Liberouter reaches 60–70% of all the Liberouters within 4–7 hours. The latencies are as expected based on the mobility trace, with the median end-to-end latency of roughly two hours, and 90th percentile of 5.5 hours. Interestingly, adding more carriers improves the coverage but not the latency. Overall, when a web client connected to a Liberouter receives new content, it is typically two hours old, and only a fraction of all messages will be available from any given Liberouter. For applications such as Here and Now and Guerrilla Pics this is likely feasible performance, but is of
Moving and Composing Computations

course dependent on the underlying mobility characteristics.

The right side of the figure shows two example propagation processes. We show the fastest (blue circles) and the slowest (red diamonds) full-reach propagations in the case of 20 carriers, with receptions by Liberouters shown separately (green triangles). Both example processes seem to be linear, except for the last few carrier receptions in the slowest case, but with a significant difference in speed: The faster example propagates to 6.1 Liberouters per hour ($R^2 = 0.98$), while the slowest reaches 1.7 Liberouters per hour ($R^2 = 0.98$) during the linear phase. These speeds are naturally determined by the connectivity trace used for the experiments.

Overall these findings are preliminary, but point to the feasibility of our proposed approach. The overheads due to attaching transformation functions, even when large in relation to the content size (e.g., in applications such as text messaging) do not appear to have a significant impact on the message propagation performance. The reach of the content is also significant, even with as few as ten Liberouters in a downtown Helsinki sized area. And the testbed experiments show that the design is realizable with standard technologies and works with existing web browsers. Ultimately, as always with opportunistic networking, the performance is determined by the underlying mobility of the nodes. In a scenario with mobile devices carried by pedestrians in a downtown environment, the latencies and spread of messages are suitable for web adaptations of the types of applications in the Liberouter system (see 3.3.3).

5.1.4 Summary

In this section we set out to answer whether it is possible to support web applications in opportunistic networks, and to provide interoperability between them and native applications similar to modern web apps on the Internet. We saw that such a system can be designed based on the idea of providing a generic application framework in Liberouters and attaching small, custom transformation functions to the content messages, which the framework calls to serve interactive web apps for web clients. We also saw that such a framework along with web adaptations of existing native applications can be implemented in practice based on the technologies, in particular Scampi and the Liberouter, described in the earlier chapters. Interestingly, a very simple Aggregate-Detail model with a handful of transformation functions was enough to implement web app versions of all the Liberouter apps from section 3.3.3. The simulation and testbed
experiments provide preliminary evidence that the approach is feasible in terms of additional overheads and the achievable reach for the content. In the future, a more extensive simulation study could be conducted to understand the performance and behavior of the system in more detail.

In the next section we will extend this idea of moving and instantiating computations within opportunistic networks, and provide a generic framework for composing distributed systems that goes beyond what is possible with the transformation functions we used in this section.
5.2 Opportunistic Distributed Services: Publication IX

In the previous section we saw how attaching computations to messages in an opportunistic network can be used to automatically instantiate web apps in the Liberouters. However, this approach is limited in expressiveness due to the constraints placed by the desire to minimize the overheads from attaching computations to every message—the applications must follow the generic Aggregate-Detail structure pre-installed in the Liberouters and can provide only limited customization via the transformation functions.

In this section, we will extend the idea of distributing and dynamically instantiating computations in opportunistic networks. The goal is to be able to develop arbitrary distributed applications and services, and to opportunistically execute them in the devices in the local environment (e.g., Liberouters and mobile devices).

The motivation for this work comes from the ever-increasing number of technological advances in the everyday environments: More local computational capacity in diverse “smart” devices; more wireless communication capacity between co-located devices and to the Internet; more ways to interact with the physical world via sensors and actuators; and new immersive ways for interfacing with the users. In our particular case, Liberouters deployed as a part of an opportunistic network have more resources than are needed for just message forwarding, and could potentially be augmented with, e.g., environmental sensors. These resources could be used to provide distributed services to nearby users, turning the Liberouters into dynamically customizable local servers similarly to the idea of cloudlets [229]. We formulate this as a research question for this section:

- Can an opportunistic network be used as a dynamic execution environment for distributed systems?

To answer this question, we create a design for a system based on Liberouters and mobile nodes that can be used to deploy and execute distributed services. The basic idea is the same as in the previous section: We distribute computations in the opportunistic network and load them into Liberouters to provide services to nearby nodes. However, instead of the very restricted model of computation based on transformation functions, we introduce a more general model of computation and service composition that allows arbitrary services to be developed and deployed. We also extend
this execution capability to the mobile devices in addition to the Liberouters. To prove the feasibility and practical realizability of the design, we implement a prototype of the execution platform for Liberouters and Android phones, using Scampi for the underlying opportunistic networking. We also implement an example distributed service that demonstrates the capabilities of the approach. The example is a classroom voting system, where the entire system is deployed from the lecturer's mobile device into the classroom Liberouter and into the students’ devices, and disappears after the lecture. Finally, we present an evaluation of the prototype.

This idea of seamlessly using the computational resources in the environment has a long history. The seminal description was given in 1991 by Mark Weiser of PARC as *Ubiquitous Computing*, with a vision that the computing technology increasingly surrounding us should integrate itself seamlessly into the world at large [265]. As the consumer technology caught up with this vision a decade later in early 2000s, the research evolved into *Pervasive Computing* with research thrusts on effective use of smart spaces, invisibility to the user, localized scalability, and masking of uneven conditions [228]. Another decade later in 2010, Marco Conti and others proposed to combine the ideas of pervasive computing with opportunistic networking into *Opportunistic Computing* [80]. This effectively generalizes the idea of opportunistic networking by using the opportunistic contacts for accessing *any* resources of the peer—CPU, sensors, cameras, databases, functions, content—not just message forwarding [79]. The work here is in this direct lineage, but where the previous work has had a heavy user-centric focus, we are inspired by the modern web backend technologies. Technologies such Google’s Borg [261], Omega [232] and Kubernetes [64] allow heterogeneous applications to be developed for—and their execution dynamically managed on—a datacenter infrastructure made up of thousands of machines with no explicit knowledge of the details of the execution environment. We want to move towards providing this type of service deployment capability in the local opportunistic environment, starting with Liberouters and Android phones.

### 5.2.1 Dynamically Composable Opportunistic Services

Our goal in this section is to enable the opportunistic creation of distributed services that can execute in the local environment—particularly in Liberouters and mobile devices. To achieve this, on the high level the system needs three basic capabilities: First, we need to *decompose* the full service
into individual elements that can be executed independently. Second, the elements must be instantiated—i.e., loaded and executed—in the Liberouters and mobile devices. Finally, we need to compose the instantiated elements by wiring the inputs and outputs together. The first of these requires a framework that developers can use when creating the services, while the last two require a runtime that can be installed in the devices.

Figure 5.11 shows the overall concept. As in the previous sections, the system is composed of stationary devices (e.g., Liberouters) and mobile devices carried by the users, each with an opportunistic router that enables the networking between them. We augment this basic system with an execution environment in each node (shown in light blue in the figure). This execution environment can dynamically load, unload and execute the components that make up the distributed services. We define two types of components: services (circle with a gear) and widgets (square with an eye). The services are purely computational components that consume input and produce outputs (e.g., an image recognition algorithm), and may be stateful or stateless. The widgets provide the user interface in devices that support user interactions, by both displaying information to the users and by collecting inputs from them. Events transmitted between components compose them together into a distributed service—they can be either local (solid lines) or remote (dashed lines), with remote events transmitted by the opportunistic network.

**Composable Components:** Our model of computation is based on components—services and widgets—that can be composed inside and between execution environments in the nodes. This model goes beyond the expressiveness of the transformation function model we used in section 5.1, allowing developers to create arbitrarily complex distributed services as
component suites without needing to understand the details of the environment within which they will execute. This also gives the flexibility needed at runtime to schedule and orchestrate the services using dynamic sets of devices in the environment.

The two types of components allow us to model various types of distributed systems in familiar ways. For example, the widget components can be seen the frontend and the service components as the microservice-based backend, similar to many current web service architectures. Alternatively, the Model-View-Controller structure, used in section 5.1, can be implemented by using a widget component as the view, a stateful service component as the model, and a stateless service component as the controller. The component model also gives us flexibility in representing various types of hardware resources and capabilities—e.g., an environmental sensor, or a camera, in a phone could be represented as a service component, and an LED or a button could be represented as a widget.

Developers have full freedom on how they design these components, but there are some characteristics which we believe to be desirable: Each component should be self-contained, have a single purpose, and interact via a well-defined interface. This promotes modularity, which in turn allows a high degree of flexibility in how the components are orchestrated for execution at runtime. The components should also avoid hard dependencies and instead interact opportunistically with other components—e.g., it should be possible to combine one of many photo gallery widgets with one of many photo providing services. Interoperability can be achieved by commonly defined service-oriented interfaces. Avoiding hard dependencies also allows the components to be loaded, unloaded and replaced independently without affecting other components.

**Event-based Interactions:** In order to compose a distributed system, the various components need a way to interact with each other. Further, our component-based design needs a way to connect dynamic sets of loosely coupled services and widgets. To achieve this, we propose an event bus approach, which has been used to solve a similar problem in modern serverless backend systems by services such as AWS EventBridge [1], Azure Event Grid [2], and Google Cloud Pub/Sub [8].

In the event bus pattern, each component can publish events into the bus and other components can subscribe to receive them. This way the interacting components do not need to be explicitly aware of each other, and components can be independently loaded and unloaded from the system.
The events transmitted by the bus are \((\text{topic, data})\)-tuples, where the \text{topic} is an arbitrary identifier used to match the publishers and subscribers—i.e., subscribers to a topic receive all the events published to that topic. This allows 1:1, 1:N, N:1, and N:M relationships between the components.

We further differentiate between \textit{local events} and \textit{remote events}, and define a different event bus for each. This distinction is important since the components executing in the same runtime can communicate reliably with close to zero latency, while components executing in different devices are connected by an opportunistic network with arbitrarily long latencies. We further leverage the local events by defining a remote procedure call (RPC) mechanism on top of the local event bus by extending the events into \((\text{procedure, reply-id, query})\)-tuples that invoke a specific procedure in a local component, which then replies with a \((\text{reply-id, result})\)-tuple. The remote events get mapped directly to opportunistic networking messages and passed to the underlying router—this requires a pub/sub capable opportunistic routers, such as the Scampi router used in the Liberouters.

\textbf{Runtime Service Distribution:} To instantiate a distributed service on the platform, we need to distribute the components into the various devices in the environment. The core idea that we started this section with, was to use the opportunistic network to distribute the computations. Because we have defined the system in terms of loosely coupled components interacting over an event bus, we can take (sets of) these components and bundle them into messages to be distributed by the opportunistic network. In addition to the component code, we also bundle any necessary data and metadata in the messages. In particular, we use metadata to describe any constraints for the components, such as resource requirements (e.g., memory or storage) and security parameters (e.g., the publisher's signature). The bundled messages are then published as events into the remote event bus, which maps to the underlying opportunistic network, to be distributed to framework instances. When a framework instance receives such a message, it loads and executes the components, subject to the constraints in the metadata.

One of the limitations of our work in this section is that we do not define any explicit \textit{orchestration} mechanism for managing system-wide constraints on the service composition (e.g., how many components should be executed and where). Instead, our approach is opportunistic, allowing each individual runtime instance to make independent decisions on what components to load and execute. This is likely to work when a user just
wants to load a few components into nearby devices, but will likely lead to problems when trying to instantiate large, complex, or long-lived services. For this, advanced orchestration mechanisms will be needed, but such are outside the scope of our work here.

**Security:** A major non-functional concern for the system is security. There are at least two major sets of security issues: Deciding and controlling which components are executed (an admission control problem); and enforcing limits on what the components can do when executing (a runtime control problem). The design of security mechanisms is outside our scope, but we foresee some directions for solving these problems.

For admission control, we could mandate that the components are digitally signed by the publisher. A long-lived, trusted root certificate could be included with the framework installations to validate components and only load those from trusted sources. This can be combined with admission control policies to either accept or deny certain publishers, or to prevent known malicious components from being loaded again. The major limitation of this approach is the difficulty of managing the certificates and policies in the disconnected devices such as the Liberouters.

Since it is very likely that this type of system would rely on a virtual machine (such as the JVM that we use for our implementation described in the next section) to enable dynamic loading of the components, it can also be augmented with runtime control mechanisms. At a minimum, the virtual machine should be able to pre-empt an executing component, which prevents a rogue component from blocking the processor. It could also place limits on the amount of resources (CPU, memory, storage, networking) that can be consumed by any individual component, perhaps tied to the admission control policies. Inside the runtime, fine-grained sandboxing (such as V8 Isolates) can be used to prevent components from interfering with each other. In addition, operating system level sandboxing techniques (such as that we used in section 5.1.2) can be used to limit the access that components have to the underlying system services.

### 5.2.2 Framework Design and Implementation

To realize the system described in the previous section, we need to design a runtime framework that implements the concepts of services, widgets, local and remote event busses, and the associate control logic. This section develops the necessary framework design and an example distributed service, which we have implemented for the Liberouter system. In the next section,
we will evaluate the performance of these prototype implementations to show the practical feasibility of the design.

Figure 5.12 shows the high level structure of our prototype framework design. It can be divided into three main layers: User interface (UI) layer made of widgets (top of the figure), services layer made of service components (middle), and an opportunistic network layer that in our prototype is implemented by the Scampi router (bottom). Depending on the device, the user interface layer may or may not be present, e.g., the headless Liberouters have no hardware for user input and output and thus do not run any widgets (as shown on the right of the figure). The layers are connected by the event busses—the user interface connects to the services via the local event bus, and the services to the opportunistic network via the remove event bus. The UI input and output events flow over the local event bus, as do events between local services, while remote events between service components in different devices flow through the remote event bus into the Scampi router. The figure shows an example distributed service in purple with a backend running as two services in a Liberouter and a user interface running on an Android device.

Figure 5.13 shows the framework structure in more detail as a UML diagram. The figure shows the four major subsystems and their relationships that we have described above (widgets, services, event busses), but adds two further subsystems that implement the control logic: Component registry, and component controller. The component registry is a database
that maintains information about the components that the framework instance has received from the network. The component controller contains the logic for managing the component lifecycle by loading, unloading, and executing the widgets and services that are stored by the registry.

We can see the details of example widget and service components as a UML diagram in fig. 5.14. On the left of the figure we can see a widget for reading sensor values. The widget is internally composed of a Sensor driver that can read values from the sensor hardware (possibly via some system services). The values are passed to a SensorController, which bundles them into events and publishes the events into the local event bus with some well-known topic. Other components that want to receive the sensor values can then subscribe to the topic to receive the events. This internal
structure is arbitrary and up to the developer, and the framework and the other components interact with the widget only via the events.

The middle of the figure shows another widget that provides a user interface with input and output. We can use a similar internal structure as before, with a ViewController that sends and receives events via the local event bus, and a View that contains the user interface components. To provide output to the user, the controller subscribes to a local event bus topic to get the desired output data—e.g., it could subscribe to a sensor widget to receive sensor readings. When receiving an event, the controller unpacks the data from the event updates the view accordingly. Similarly, to receive input from the user, the controller listens to user actions on the view and packages them as events to be published into the local event bus. How the views are implemented are naturally platform dependent, and will typically use the native UI libraries. The interactions with the rest of the system, however, are again via the events and the rest of the system does not need to know the implementation details of the user interface.

The right of the figure shows the internal structure of a stateful service component. The service is divided into business logic in the Domain Model and state in the Storage components. The state could be held in persistent or in-memory storage, which the business logic uses via insertion and deletion operations. In this design the storage also acts as mediator for the events, which it passes from the local event bus to the business logic for handling. A stateless service component would omit the storage component and handle the events directly.

We implement a prototype of this design in both pure Java (Liberouters) and for Android (mobile devices). The core framework is developed as a Java library, which contains a local bus interface, generic component interface, service component with storage for state, and a remote bus interface. The framework compiles into a JAR and can be used in both the Liberouter and Android versions, augmented by platform-specific implementations of some subsystems. For the event busses, we use the Google Guava EventBus [16] as the local bus in pure Java and the GreenRobot EventBus [15] on Android—we use Scampi as the remote event bus on both platforms. Widget components are implemented as full Activities on Android, and using JavaFX [19] on pure Java. We use a simple Component Controller implementation that schedules components to be executed by a thread pool—the JVM does not have a facility to pre-empt running threads, so we rely on cooperation from the running components to stop and unload them.
Since the purpose of the framework is to enable the creation of distributed services, we will examine an example service built using the platform. The example service is a classroom polling application that runs on both Android phones and in a Liberouter connected to a smart screen. We also use this application for the evaluation in the next section.

The concept for the classroom polling application is shown in fig. 5.15. Our scenario has a classroom with a lecturer and students with Android phones, and a Liberouter connected to a screen or a projector. The polling service allows the lecturer to create poll questions on their mobile device which the students can then answer on their own mobile devices. The Liberouter collects the results and shows them on the attached screen.

Importantly, we want to instantiate the service dynamically from components carried in the lecturer’s device, without anything being pre-installed in the other devices—we also want the service to disappear from all the devices after the lecture is over. As shown in the figure, the teacher enters the classroom at time $t_1$ carrying all the components. The lecturer’s device connects to the classroom Liberouter at time $t_2$ and transmits copies of the components intended for the smart screen and the students’ devices. The Liberouter then loads and executes the smart screen components. As students enter the room, their phones connect to the Liberouter and get copies of the poll participant components. At time $t_3$ when the lecture starts all the components of the service have been distributed to the correct devices, and the lecturer can use the service to poll the students during...
Moving and Composing Computations

Once the lecture is over, the components are unloaded from the Liberouter and the students’ phones.

We can use our framework to design the polling service as a handful of components as shown in fig. 5.16. The figure shows three different node types, PollCreator, PollManager, PollParticipant, corresponding to the lecturer, the classroom Liberouter, and the students respectively. Each node type has its own widgets to provide the appropriate user interface—poll creation for the lecturer, result display for the Liberouter, and poll responding for the students. The application logic is implemented by service components interacting via remote events in each of the nodes.

The lecturer’s application, on the left of the figure, is composed of the Poll Creator Widget and the New Poll Service. The widget provides a user interface for creating new polls and publishes them as local events. These events are picked up by the service component, which publishes them as remote events that in turn result in Scampi messages being created by the remote event bus. In addition, the user interface allows the lecturer to explicitly deploy the other components to the other devices. This too is mediated by the service component, which creates a special external event that gets picked up by the framework in the nearby devices and results in the components being loaded and executed.

The remote new poll events created by the lecturer’s service component are received by the Poll Management Service running on the Liberouter, as shown in the middle of the figure. In response to the event, the service displays the new poll using the Published Polls Widget and creates a new remote event for the students. The students’ Poll Participant Service, shown on the right of the figure, picks up these remote events and displays...
the **Poll Answer Widget** that allows the student to respond to the poll. This response is published as a remote response event by the students’ service component, and gets picked up and added to the response set by the Liberouter’s service component.

Once the poll manager in the Liberouter decides that the polling has finished, it calculates the results based on the received response events. The result is then displayed on the attached screen by the **Poll Result Widget**. It is also published as a result event into the remote event bus, which allows the **Poll Result Widget** to display the result on the students’ devices. If we wanted to extend the system to also display the results on the lecturer’s device, we could do it simply by loading the **Poll Result Widget** also on the lecturer’s device.

Figure 5.17 shows how the resulting system appears to the users. The top left of the figure shows the lecturer’s poll creation view, generated by the **Poll Creator Widget**. Filling in the fields and clicking the “publish” button will result in the poll being set up as described in the previous sections. The top middle of the figure shows how the polls appear as a list on the students’ devices. When the student selects a poll, it either shows a voting view or the result view depending on whether the polling has concluded—the top right of the figure shows how these appear in the GUI. The bottom of the figure shows how the poll results appear on the classroom Liberouter’s screen. Unlike the mobile interfaces, which are
native Android applications, the Liberouter uses a JavaFX application for the widgets, and can thus run on a standard Java runtime.

5.2.3 Evaluation

To establish the feasibility of the design, we conduct a small-scale testbed experiment using the prototype implementations of the framework and the polling application described in the previous section. We are interested in three core behaviors of the system: The dynamic instantiation of new distributed services, the operation of an instantiated service, and the operation under disruptions of the underlying network. Since this is only a small-scale experiment, we will not be able to say anything about the scalability of the system to large numbers of nodes. However, the scenarios targeted by this initial design (e.g., the polling application) are naturally small and local in scale, for which we can show feasibility. Scaling these up to perhaps 50 students would only impact the messaging volumes, which we showed to be practically feasible in section 3.2.

The evaluation scenarios are based on the polling application, with one mobile device in a lecturer role and one in a student role, both connected to a laptop acting as the classroom Liberouter. The mobile phones are Android 4.4.2 devices with 1.4/1.6 GHz A7/A9 and 2/1.5 GB RAM, while the laptop is a Windows 8.1 device with 1.7 GHz i5 and 8 GB RAM. We evaluate the entire process as it would happen in a classroom, starting with the lecturer distributing the service components to the other devices, and then running the full polling process from poll creation to answer to result tabulation and publication. Where possible, the test interactions are automated to perform the actions a human user would take, but in a repeatable manner without the delays of human intervention.

We construct two evaluation scenarios: static and dynamic. In the static scenario the devices are permanently connected via a Wi-Fi network created by the Liberouter node. Within this scenario, we define two phases to evaluate: instantiation phase and operational phase. In the instantiation phase the lecturer device publishes the polling app services and widgets to the other two devices, while in the operational phase the lecturer device publishes new polls to which the student device answers.

In the dynamic scenario, we have the same devices and roles, but introduce disruptions to the underlying network. We do this in two different test cases: temporary node absence and infrastructure failure. The first case simulates a situation where the student device is temporarily un-
available during the polling process, and evaluate whether the service can recover when the node returns. We achieve this by toggling the Wi-Fi on the student device between on and off every two minutes. The second case simulates a situation where the Liberouter device fails and is replaced by another device, and evaluate whether another device in the environment can opportunistically take over from a failed device to keep the distributed service operational. We achieve this by having two independent framework instances in the Liberouter role, which we toggle on and off every two minutes such that at any given time only one of the instances is running.

The detailed sequence of events in the evaluation testbed is shown by fig. 5.18. The left side of the figure shows the lecturer node, which acts as the initiator for the sequence. The student device is on the right of the figure and responds to events received from the system. The middle of the figure shows the Liberouter node, which runs the poll result creation process described in the previous section.

The top half of the figure shows the *instantiation phase* of the test case. It consists of the lecturer node publishing three components into the network: *Poll Management Service* (PMS.jar, 24 KB) in step 1, *Poll Participant Service* (PPS.jar, 24 KB) in step 3, and *Poll Participant Widgets* (PPW.jar,
5.4 MB) in step 5. These correspond to three remote events in the framework, and map to three separate Scampi messages in the opportunistic networking layer. In each case the framework in the appropriate device dynamically loads and executes the received components (steps 2, 4, 6).

The bottom half of the figure shows the operational phase of the test case. This phase only starts after the initiation phase has finished and all the components are executing in the framework instances in the nodes. In this phase the lecturer publishes a new poll as a remote event every 30 seconds (step A). The Liberouter node listens for these remote events and, after receiving an event, creates a new poll state (step B) and activates the poll by publishing a new remote event (step C). The student node responds to these new polls by creating poll answers from the widget component (step D), which the service component turns into remote answer events (step E). These answer events update the poll state in the Liberouter (step F), which in turn creates external events with the poll results that the student device picks up (step G).

**Static instantiation phase:** To characterize the instantiation phase behavior, we measure the latency from the lecturer publishing the component and the Liberouter or student instantiating it: \( t_{PMS} = t_2 - t_0 \), \( t_{PPS} = t_4 - t_1 \), and \( t_{PPW} = t_5 - t_3 \) (top of fig. 5.18). We find that the instantiation of the poll management service component into the Liberouter node takes a mean of 14.6 seconds—the 95% confidence interval (CI) is 8.0–21.2 seconds. The time to instantiate the poll participant service component in the student device is 11.2 seconds (6.2–16.2 seconds 95% CI).

This process is fully automated and represents the full end-to-end latency from the component distribution being triggered by the lecturer’s widget to the service components starting execution on remote devices. Since the transmission time for the 24 KB components over the direct Wi-Fi connection is negligible, most of the latency comes from the framework processing. A large part of this latency comes from the Java Virtual Machine’s (JVM) dynamic class loading procedures and is therefore unavoidable without modifying the JVM. This indicates that virtual machines targeting these types of use cases should focus on the speed of dynamic code loading—as is often done by JavaScript runtimes that must quickly load and execute code embedded in websites. Further, there are still significant inefficiencies in our prototype framework implementation, causing us to believe a production quality implementation to improve on these times significantly.
The latency for instantiating the widget component on the student device is significantly larger: 54.1 second mean and 37.5–70.6 second 95% CI. This is partly because of the larger message size (5.4 MB), but mainly due to the need for a manual installation process on Android. This still within feasible range for our scenario, since after manually accepting the components, rest of the installation happens in the background as the students enter the classroom. The manual installation is necessary since we ship the participant widgets as an Android application package (APK), which requires explicit user authorization to install. This indicates that a more performant approach would be to have a single Android application to act as a GUI dashboard framework that dynamically loads only the widgets similarly to how service components are loaded. Such an app would act similarly to an app store, except the widgets would execute within it instead of being separately installed. Another alternative would be to install an HTTP widget in the classroom Liberouter to serve a web app interface to the students instead.

**Static operational phase:** In the operational phase, we want to understand the performance of the interactions within the distributed service. We measure three interactions latencies within the polling process (bottom of fig. 5.18): Poll creation from the lecturer via the Liberouter to the student \( t_{\text{new}} = t_B - t_A \), local poll answering in the student device \( t_{\text{answer}} = t_C - t_B \), and poll answer and result between the student device and the Liberouter \( t_{\text{result}} = t_D - t_C \). These give us measures of an end-to-end latency through the three devices, the local latency within a device, and a round-trip latency between two devices.

We measure the end-to-end delay \( t_{\text{new}} \) to be 1.5 seconds (1.49–1.51 second 95% CI), which includes two separate remote events and the local poll state creation in the Liberouter. This is an order of magnitude smaller latency than the service instantiation, showing how costly the dynamic class loading is in comparison to the other operations. The time taken from an incoming remote event, through the GUI widget, and back to a new remote event in the student device \( t_{\text{answer}} \) is 0.78 seconds (0.75–0.81 seconds 95% CI). This includes all the local processing in the service and widget components, as well as two trips through the local event bus. The round-trip between the student device and the Liberouter \( t_{\text{result}} \) is 0.61 seconds (0.6–0.62 second 95% CI), which again includes two remote events and local state update in the Liberouter. In other words, the round-trip between two separate devices takes as long as a local round-trip between
a service component and a widget, which shows the freedom we have in placing the different components either locally or in remote devices via an opportunistic contact.

**Disrupted node absence:** Since we target opportunistic environments, the network is not always static. For example, the students may enter and exit the range of the system during the lecture. Next we want to understand what happens to the end-to-end polling process if the student node is not statically connected. We achieve this by toggling the Wi-Fi in the student device every 120 seconds, while the lecturer node is publishing new polls every 30 seconds. The resulting end-to-end latency ($t_{\text{new}}$) over time is shown in the left of fig. 5.19.

In the figure we can see a sawtooth pattern with a period of 240 seconds (twice the Wi-Fi toggling period). The pattern shows that all the new polls are received by the student device, but some are delayed. This is exactly as expected, since the opportunistic networking layer is buffering the remote events while the student device is not connected to Wi-Fi and delivering them once the connection returns. The delay peaks just after the Wi-Fi has been turned off and the remote events must wait up to 120 seconds for the connectivity to return (which matches the highest peak in the figure). While the Wi-Fi is turned on, the latencies are close to zero and the system is operating at close to real time. This demonstrates that the system can tolerate the absence of nodes and to rebuild the full service state when the node returns.

**Disrupted infrastructure failure:** Another form of possible disruption
in the polling service is the failure of the Liberouter device that manages the polling process. In such cases the system should be able to opportunistically switch to use another device for the lost functionality. We evaluate this ability by stopping the Liberouter instance every 120 seconds and starting another one to replace it. We again measure the end-to-end latency ($t_{\text{new}}$) over time with the lecturer device creating new polls every 30 seconds. The resulting behavior is shown on the right side of fig. 5.19.

In the figure, we can see that the behavior of each individual Liberouter is identical to the previous case, for the same reasons (messages getting buffered while the Liberouter node is stopped). We can also see that the events are now delivered twice due to the epidemic routing used in the underlying opportunistic network creating two paths for each remote event (one via each Liberouter)—these duplicate events will be filtered out by the framework. Importantly, we can see that more of the interactions have near zero latency when compared to the previous case. When one Liberouter is off, the other one is processing the remote events instead—the work is being dynamically moved between the service component instances in the different framework instances. This demonstrates the ability of the system to respond to infrastructure failures by opportunistically shifting the components to different devices.

Overall the results of the experiments demonstrate the feasibility of our approach in small scenarios. The service component instantiation latencies of around 10 seconds are low enough for a use case such our polling application, but might be too long in a more dynamic environment. Our approach to Android widget distribution as APKs is problematic due to the long latency and the need for manual user interaction. It should be replaced with an approach that instead loads the widgets inside a dashboard-style application, which would prevent the need for application installation and allows for smaller binary size. Once the service has been instantiated, the experiments show that it can operate with very low latencies and tolerates disruptions due to nodes entering and leaving.

The main limitation of the results is the small size of the testbed. To evaluate the scalability of the system, further studies with larger emulated testbeds or simulations are needed. However, the results do indicate that the approach is practical in small local scenarios, such as our classroom polling application.
5.2.4 Summary

In the beginning of this section we asked whether an opportunistic network could be used as a dynamic execution environment for local distributed services. Based on the work presented here, it appears to be both feasible and practical—albeit so far we can only claim that on a small, local scale. The proposed approach of service and widget components interacting via local and remote events appears to result in building blocks that are both intuitive for the developers and easy to support with an opportunistic runtime framework. The testbed experiments show that the dynamic instantiation of components is somewhat slow, but still acceptable in mostly static scenarios such as classrooms. Once instantiated, the event latencies between the components are so low, that we can freely compose them within the immediate environment. A particularly nice property of the resulting system is the automatic tolerance of disruptions and changes in the environment.

The main limitation of the work so far is the limited scale reachable due to our opportunistic approach to service composition. Each node decides independently which components to load and execute, which appears to work in the types of small-scale environments that we experimented with, but which is unlikely to work in larger and more complex environments. What is needed are more intelligent opportunistic orchestration mechanisms that can decide which components to instantiate in which nodes in order to achieve larger scale systems.
5.3 Conclusions

This chapter explores the idea of dynamically distributing and instantiating computations in opportunistic networks.

We start with the goal of supporting web apps in opportunistic networks, and allowing them to interoperate with native apps. This is not possible with the opportunistic networking mechanisms that only move data (such as those in section 4.1), since the latencies between the client and the server are too large for the frequent interactions necessary. To achieve this goal, we design a restricted model of computation, where the Liberouters run a generic web app framework, which calls small transformation functions to provide the custom, interactive application specific behavior. These transformation functions can then be attached to the messages that carry the content in the opportunistic network, and therefore become available everywhere that the content is available. Essentially, the Liberouters automatically synthesize the necessary web app backend and make it available to their directly connected clients. Since we only augment the existing content messages, the approach is fully compatible with the native applications. Interestingly, the work shows that a very simple generic Aggregate-Detail model in the Liberouter with a handful of transformation functions attached to the content messages is capable of representing all the Liberouter applications from section 3.3.3. This indicates that the generic model and the design patterns and semantics of the native Liberouter apps are a good match. The simulation and testbed experiments indicate that the approach is feasible in practice, and the overheads due to the computations attached to all the content messages are small enough to not impact the performance significantly.

It further seems that using the Liberouters, along with mobile devices, to dynamically execute computations distributed by the opportunistic network can serve as a platform to develop local distributed services. Our design allows distributed services to be developed as small service components and user interface widgets that interact by exchanging events over both local and remote event busses. Since these components are self-contained and highly decoupled, it is possible to flexibly distribute, instantiate and execute them in runtime frameworks installed both in the infrastructure nodes and the mobile devices. The resulting services are also robust against disruptions due to the underlying opportunistic networking layer. This enables interesting local distributed services to
be designed and developed, such as our classroom polling application that gets automatically installed into the students’ devices and classroom infrastructure when the lecturer enters the room and disappears after the lecture ends. While this is only an initial step—scaling beyond small local deployments will require more work—it shows the feasibility and potential of the approach. And ultimately, many hyper local opportunistic services may not need to scale beyond small deployments.

The work here shows that opportunistic networking can go beyond just moving data, to dynamically distribute and execute computations. It gives us another important degree of freedom in opportunistic networking: Not only can we move the data (as established in chapter 3 and 4), but we can also move the computations that operate on the data. This new degree of freedom allows us to build services, such as web apps, that would not be feasible otherwise. In the future, these techniques could be useful in conjunction of edge computing to enable richer collaborative computing at the edge.
6. Conclusion

The goal we set for this thesis is to advance the availability and capability of opportunistic networking. In particular, we propose to increase the availability of opportunistic connectivity by using both existing open infrastructure and by creating new opportunistic infrastructure. And, on top of this basic connectivity, we propose more advanced capabilities for both opportunistic content interaction and opportunistic computing. Together, these contributions create a set of mechanisms for opportunistic networking that both build on and complement each other.

For creating connectivity, we find that at least three approaches seem viable: First, we can use existing open Wi-Fi access points to create opportunistic contacts. However, as these access points are intended for Internet traffic, they block opportunistic traffic roughly half the time in various ways, and thus complex protocols will be needed to create the contacts in practice. Second, we can automatically move opportunistic traffic from an infrastructure access point to access points dynamically created by the mobile devices themselves. This has the benefit of being able to use the entire ISM spectrum rather than just the single channel of the infrastructure access point. Finally, opportunistic infrastructure created by low-cost single-board computers provides an attractive alternative to infrastructure networks. In all cases, opportunistic connectivity would benefit from flexible support for virtual Wi-Fi interfaces in the mobile devices—the devices could connect to each other and the Liberouters without impacting the primary Wi-Fi interface. Most chipsets and drivers already have this capability, so it might become an important enabler for opportunistic networking.

We propose three content-based mechanisms for opportunistic networking: Request/response, query/response, and mutable content. All of these mechanisms appear viable, but each has challenges and limitations. The
request/response mechanism can reach out to infrastructure networks via
bridge nodes to fetch content such as websites. However, the success rates
and latencies are far too long to support the usual active web browsing
experiences. The query/response mechanism, in turn, can reach out to the
message stores of other nodes in the opportunistic network and return
content of interest. Similar limitations apply as to request/response, mak-
ing active or exhaustive searching infeasible. And finally, it is possible
for opportunistic networking users to collaboratively edit documents, but
various strategies are needed to manage the inevitable conflicts.

Finally, we propose mechanisms for opportunistically distributing and
instantiating computations. It is possible to extend opportunistic infra-
structure, such as the Liberouters, to automatically synthesize backends in
order to serve interactive web apps to connected clients. This mechanism
relies on directly attaching computations to the content in the messages,
and also enables interoperability between native opportunistic apps and
web apps. Further, the opportunistic infrastructure can be extended into a
dynamic execution environment for deploying distributed services. This
could potentially turn the local environment into a distributed computing
platform that resembles modern cloud infrastructure. However, much more
work is needed particularly in orchestrating such opportunistic services.

The recurring theme in our results has been that trying to provide the
types of active experiences that classical networking provides is unlikely
to work. Having the user actively wait while the opportunistic network
fetches a website or receives responses to a search query would result in a
completely unsatisfactory user experience. However, these mechanisms do
eventually produce useful results. This indicates that opportunistic net-
working is most useful when running in the background and continuously
accumulating content and services for the user.

More generally, opportunistic networking is not simply a curiosity or a
corner case of classical networking. Instead, it is another fundamental
modality of networking, building on a distinct set of concepts and mecha-
nisms. It also has a distinct set of strengths and weaknesses as compared
to classical networking. The most obvious strength is the ability to trans-
mit messages without infrastructure or instantaneous end-to-end paths,
while the most obvious weaknesses are the long latencies and the limited
reach (at least in smartphone-based scenarios). Classical networking is
clearly superior for reaching global cloud services, while opportunistic
networking is better at reaching content and services provided by other nearby devices. Opportunistic networking incurs an additional resource cost (e.g., battery use in the smartphones) due to the cooperation needed to create the network, while classical networking often incurs a monetary cost to pay for the required infrastructure (e.g., cellular network subscription). Neither one should be seen as a complete replacement for the other.

The logical question, then, is if we can combine these two modalities of networking in the same devices? Currently, the balance is completely tilted towards cloud-based services reached via classical networking. This has enabled rich, interactive, global services, but has also led to a high degree of centralization and homogenization—a few large services carry the majority of the communications and interactions between people. Opportunistic networking could provide a counterbalancing force for this direction of progression. Instead of global homogeneous services, it can provide hyperlocal and diverse services. When the user opens their smartphone, they would have access to the interactive experiences in the cloud, but also to the local content that the device has opportunistically picked up in the background. Interactions between colocated users would not need to transit over the Internet, but could instead happen directly between the devices.

The next obvious question—given that the basic capabilities for opportunistic networking already exist in most smartphones—is why this form of networking has not been widely adopted? Some basic technical challenges remain, such as increased energy use and inefficiencies in device-to-device communication technologies, but these are likely not showstoppers and can be solved by engineering. Some less technical challenges also remain, particularly around moderation of content in a fully distributed messaging model, but these too are likely solvable in a distributed and collaborative manner. Ultimately, however, everything comes down to the incentives of the decision makers. Network operators have an incentive to have people use their network infrastructure rather than communicate directly. They are also often the largest buyers of smartphones, which pushes the same incentives to the device manufacturers, and network equipment, which pushes the same incentives to the standardization work. Similarly, the large Internet-based services have an incentive to have their users tied to their centralized platforms, as this allows them to create an accurate profile of their users and thus reach high performance for their advertisers—they would have no visibility into the direct interactions between the smartphones. It seems possible, however, that the opportunistic networking
technology is in a critical state, where it could quickly find adoption if a single major player (e.g., Apple or Google) made it easily available to developers on their platform.

Outside smartphone-based scenarios, a potential field where opportunistic techniques might find adoption is the Internet of things and machine-to-machine networking. Both of these are bringing more devices in more places in our environments. As the data generated by these devices increases, and more resource intensive processing is desired (e.g., by machine learning), it becomes less attractive to send everything to the cloud. Instead, it becomes more attractive to opportunistically leverage the communication and computational capabilities of the nearby devices—many of the techniques discussed in this work would be directly applicable.

Today, the technological basis for opportunistic networking is quite advanced, due to the efforts of many researchers over the years. Perhaps these ideas will eventually find their ways into our smartphones to create new opportunistic applications and services. In the meanwhile, the ideas from opportunistic and delay-tolerant networking seem to be resurfacing in new fields of networking research, such as Information Centric Networking and Edge Computing, whenever they run into the basic concepts of mobility, device-to-device communications, and the lack of stable infrastructure. Hopefully these ideas will continue to evolve both in theoretical research and in practical systems for the years to come.
References


References


architecture for mobile devices. Department of Information Technology,

[193] M. Nottingham and R. Fielding. Additional http status codes. RFC 6585,
RFC Editor, April 2012.

[194] M. Nottingham and R. Sayre. The atom syndication format. RFC 4287,
December 2005.


[196] K. Oku. An http status code for indicating hints. RFC 8297, RFC Editor,
December 2017.

[197] G. Oster, P. Urso, P. Molli, and A. Imine. Data consistency for p2p collabora-
tive editing. In Proceedings of the 2006 20th anniversary conference on

for probabilistic information sharing. Pervasive and Mobile Computing,

Information sharing in urban areas. In 2011 IEEE International Conference
on Pervasive Computing and Communications (PerCom), pages 136–146.
IEEE, 2011.

WNEPT, 2006.

documents, such as html (mhtml). RFC 2557, RFC Editor, March 1999.


1905.

1905.

Thesis, Helsinki University of Technology, Department of Communications
and Networking, 2008.

forwarding in disconnected mobile ad hoc networks. IEEE communications


References


[264] K. Wei, X. Liang, and K. Xu. A survey of social-aware routing protocols in
delay tolerant networks: Applications, taxonomy and design-related issues. 


[266] H. Wirtz, T. Heer, R. Backhaus, and K. Wehrle. Establishing mobile ad-hoc
networks in 802.11 infrastructure mode. In *Proceedings of the 6th ACM 

of opportunistic offloading. *IEEE Communications Surveys & Tutorials*, 

[268] B. B. Xuan, A. Ferreira, and A. Jarry. Computing shortest, fastest, and fore-

[269] G. Xylomenos, C. N. Ververidis, V. A. Siris, N. Fotiou, C. Tsiropoulos, X. Vasi-
lakos, K. V. Katsaros, and G. C. Polyzos. A survey of information-centric net-
working research. *IEEE communications surveys & tutorials*, 16(2):1024–
1049, 2013.

[270] E. Yoneki, P. Hui, S. Chan, and J. Crowcroft. A socio-aware overlay for pub-
lish/subscribe communication in delay tolerant networks. In *Proceedings of 
the 10th ACM Symposium on Modeling, analysis, and simulation of wireless 

In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the 
IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*, 

[272] H. Zang and J. C. Bolot. Mining call and mobility data to improve paging 
international conference on Mobile computing and networking*, pages 123–

Trust and 2011 IEEE Third International Conference on Social Computing*, 

bus-based disruption-tolerant network: mobility modeling and impact on 

[275] X. Zhang, G. Neglia, J. Kurose, and D. Towsley. Performance modeling of 

[276] Z. Zhang. Routing in intermittently connected mobile ad hoc networks and 
delay tolerant networks: overview and challenges. *IEEE Communications 


