DESIGN OF MOBILE AND STATIC SENSOR FABRICS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

Mukundan Sridharan, B.E., M.S.
Graduate Program in Computer Science and Engineering

The Ohio State University

2011

Dissertation Committee:

  Anish Arora, Advisor
  Prasun Sinha
  Rajiv Ramanth
© Copyright by
Mukundan Sridharan
2011
ABSTRACT

Over the last few years, two fundamental changes have happened in the domain of Wireless Sensor Networks (WSN). One, the applications have ceased to be edge-alone and have morphed into a edge-enterprise co-design. Two, sensor networks are beginning to be shared by a number of stakeholders, with varied requirements – an urban sensing network shared by, say a city police agency and a state-wide security agency would be one such example. This translates into three key requirements for today’s WSNs. One, the networks to be designed should be customizable and re-purposeable on short notice. Two, the networks should be sliceable – the ability to deploy and run multiple applications simultaneously. Three, the networks need to be designed in such a way as to enable interaction, collaboration and federation with other networks or agents outside of the network to achieve some common goal. This has led to the re-examination of the application development model for WSNs. One way of designing such a network would be to come up with an architecture, which abstracts functionalities that are common to most programmable sensor networks into services, which can then be reused, ported and standardized.

This dissertation proposes an architecture called the fabric model for designing WSN as generic sensing networks, on which applications can be tailored. The model focuses on standardizing services that are common to most sensor fabrics, so that the common services can be reused across fabrics. The fabric model specifies services and
their APIs, which we believe should be part of all fabrics. The services should be implemented in an energy efficient manner based on the fabric type.

One implication of the model is the clear boundary between the design & management of the fabric on one hand and design & management of the application on the other. While this separation helps in delineating the responsibilities of the two groups, the design of the fabric and the application are not independent processes. A fabric designer needs to be mindful of the requirements of the potential applications and an application designer needs to keep in mind the capabilities of the fabrics that will execute the application. Because of the separation in the roles, an application designer can now focus on realizing the application’s goals, using the services provided by the fabric and leave management of the fabric to the service provider. The model also supports services which are common to a particular domain or type of fabric. Such domain services will help in standardization, federation and portability within a domain.

We illustrate the usefulness of the fabric model by specifying and building three different fabrics, but by reusing a number of common services. The three fabrics, Kansei – a sensor testbed with a wired control channel, PeopleNet – a mobile wireless network and VillageNet – an intermittently-connected mobile network (like those found in village/rural areas of developing countries), use the same fault-tolerant fabric manager and I/O services. At the same time, the divergent requirements on each of these fabrics can be designed and implemented as separate services and plugged into the fabric manager. As one of the key contributions we present the architecture for a fault-tolerant, scalable and autonomous fabric manager that can manage thousands of sensor nodes.
While we reuse a number of services in PeopleNet and VillageNet from Kansei, a few significant challenges remain in designing mobile WSN fabrics. One of the significant challenges is the absence of a reliable messaging service on which the fabric manager can execute its control functions. In an energy constrained multi-hop mobile network, this requirement translates into designing and implementing an efficient and reliable routing protocol. Thus, we make two key contributions in the mobile routing space. We have designed & implemented the Asymmetric Event-driven Routing (AER) service, which provides energy efficient messaging service in a slowly mobile network, for the PeopleNet fabric, and the Reliable Energy Aware Predictive Routing (REAPER) service, which provides end-to-end messaging for intermittently-connected mobile networks, such as the VillageNet fabric.

The fabric model provides a number of advantages in designing customizable fabrics and its services based design lends itself to WSN federations naturally. But, a number of other challenges remain in federating WSN fabrics. For example, one of today’s grander vision is to move towards a global federation of sensor devices that integrate the physical world seamlessly with the cyber world, not only for information collection, but also for command-control. The Global Environment for Network Innovation (GENI) is one such effort, where a number of academic and industrial organizations are pooling their diverse resources to build a global testing facility. Such a federation will require machine readable and interpretable representation of resources and experiment/application representation. Additionally, discovering, acquiring and stitching together resources available at diverse sensor fabrics becomes a challenge. We present KanseiGenie, a GENI-compliant software architecture for federating geographically separated sensor fabrics and to provide the user with a common interface
to program across the federated fabrics. KansiGenie, builds on top of the fabric model and tackles issues related to resource and experiment specification in federating sensor fabrics.

We also demonstrate through WEAVE – a domain specific service – how a federated application can be stitched using multiple sensor fabrics. WEAVE provides a search API, using which diverse applications such as a multi-layered building security system and a social networking application can be built in a federated setting. WEAVE illustrates the merits of standardization of domain specific APIs and serves as one example of the power of the fabric model.

As sensor networks become ubiquitous and federated, we hope and believe that our work in this dissertation will serve as one of the cornerstones, which will encourage further research in the design and integration of sensor networks.
Dedicated to

My Parents

For their Love, Support and Unwavering Faith in Me
ACKNOWLEDGMENTS

First and foremost, I would like to express my sincerest gratitude to my Adviser, Dr. Anish Arora. Working with him has been a rich and rewarding exercise for me in various ways. His vision, application and research methodologies has left a lasting imprint on me. His passion for research, commitment to hard work and strict observance of ethics, will continue to guide me in future. I am truly grateful for his support and guidance during my graduate student life.

Next, I would like to thank my committee members, Dr. Rajiv Ramnath and Dr. Prasun Sinha, and a special thanks to Dr. Emre Ertin. All through my doctoral program they have helped me with inputs, ideas and discussions. I also take this opportunity to thank my Master’s degree advisor Dr. Arjan Durresi who introduced me to the research world. I would also like to acknowledge the National Science Foundation (NSF) and AFRL for funding part of my research.

At Ohio-State, I have been very fortunate to be tutored by many magnificent teachers. These include Dr. Dong Xuan, Dr. Neelam Soundarajan, Dr. Steve Lai, Dr. P. Sadayappan, Dr. Keneth Suppowit, Dr. Ming T. Liu and many others. All these teachers have had tremendously positive influences at various stages of my academics, and I am truly grateful to them.

This work would have never reached completion without help from my colleagues in our research group. While, all of them have contributed to my research in one way
or other, I would like to particularly thank Vinod Kulathamani, Wenjie Zeng and Jing Li, for numerous collaborations, ideas and discussions.

I wish to also express my sincerest gratitude to the always helpful staff members of the Computer Science and Engineering Department at Ohio-State. Catrena, Tom, Carrie and Tamera have all been so helpful and kind to my needs at innumerable times. A special thanks to Dave Kneisly from Computing Services for always obliging my various unique networking and experimentation needs.

My parents have been steadfast in their support for me. They have supported all my decision, not only during my graduate studies, but all through my life and I thank them for that. I also thank my sister and my cousins for all their support and encouragement during the last several years.

Finally, I would like to express my gratitude to my friends for not only their company, support and help, but also for making my stay at Ohio State a memorable and enjoyable experience. They were my family at Ohio State.
VITA

May 14, 1979 .......................... Born - Chennai, India

2000 ............................... Bachelor in Engineering
in Electronics And Communication
University of Madras

2003 ............................... Master of Science
The Ohio-State University

2003-2004 ........................... Research Assistant
OARnet, Columbus, OH - 43212

2005-present ......................... Graduate Research Associate
The Ohio-State University

PUBLICATIONS

Research Publications

M. Sridharan, W. Zeng, W. Leal, X. Ju, R. Ramnath, H. Zhang, and A. Arora,

V. Kulathumani, A. Arora, M. Demirbas, and M. Sridharan, Trail, A Distance Sensitive Network Service for Distributed object tracking, ACM transactions on Sensor Networks, 5(2), 2009, pp.1-40

A. Durresi, P. Kandikuppa, M. Sridharan, S.Chellappan, R. Jain, and L. Barolli,


M. Sridharan, E. Ertin, R. Ramnath and A. Arora, “Mobility Centric Campus Area Sensor Network for Locality Specific Applications”, *demonstration at ACM Sensys*, 2006


FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Computer Networking  Prof. Anish Arora  Prof. Eylem Ekici  Prof. Ten H. Lai  Prof. Ming T. Liu  Prof. Dong Xuan
Distributed Computing and Systems  Prof. P. Gagan Agrawal
High Performance Computing  Prof. P. Sadayappan
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>vi</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>vii</td>
</tr>
<tr>
<td>Vita</td>
<td>ix</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xvii</td>
</tr>
<tr>
<td>List of Acronyms</td>
<td>xxi</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Challenges</td>
<td>6</td>
</tr>
<tr>
<td>1.3 The fabric model</td>
<td>6</td>
</tr>
<tr>
<td>1.3.1 Definition</td>
<td>7</td>
</tr>
<tr>
<td>1.3.2 Discussion</td>
<td>8</td>
</tr>
<tr>
<td>1.4 Other sensor network models</td>
<td>13</td>
</tr>
<tr>
<td>1.4.1 Monolithic application model</td>
<td>14</td>
</tr>
<tr>
<td>1.4.2 Virtual machine model</td>
<td>14</td>
</tr>
<tr>
<td>1.4.3 The database model</td>
<td>15</td>
</tr>
<tr>
<td>1.5 Fabric model in other application domains</td>
<td>17</td>
</tr>
<tr>
<td>1.5.1 The fabric model in grid computing</td>
<td>17</td>
</tr>
<tr>
<td>1.5.2 The fabric model in cloud computing</td>
<td>18</td>
</tr>
<tr>
<td>1.6 Service-oriented architecture (SoA) and the fabric model</td>
<td>19</td>
</tr>
<tr>
<td>1.7 Contributions of this dissertation</td>
<td>20</td>
</tr>
<tr>
<td>1.8 Organization of the dissertation</td>
<td>24</td>
</tr>
</tbody>
</table>

xiii
# 2. Specification of Sensor Fabrics

## 2.1 Overview of generic fabric implementation

## 2.2 Kansei: A sensing testbed fabric

### 2.2.1 Kansei overview

### 2.2.2 Kansei specification

## 2.3 PeopleNet: A mobile fabric

### 2.3.1 PeopleNet overview

### 2.3.2 PeopleNet specification

## 2.4 VillageNet: A Delay-Tolerant mobile fabric

### 2.4.1 VillageNet overview

### 2.4.2 VillageNet specification

# 3. An Autonomous, Scalable and Fault Tolerant Fabric Manager

## 3.1 Introduction

## 3.2 Background and related work

## 3.3 Kansei overview

## 3.4 Fabric director specification

### 3.4.1 Fault model

### 3.4.2 System invariants

## 3.5 A fault-tolerant architecture using detectors and correctors

### 3.5.1 The trusted base

### 3.5.2 Fabric manager architecture

## 3.6 Detectors, correctors and invariant for Kansei

### 3.6.1 Invariants and correctors for job control

### 3.6.2 Invariants and correctors for system health

### 3.6.3 Invariants and correctors for resource monitoring

### 3.6.4 User specified invariants

### 3.6.5 Kansei faults

# 4. Asymmetric Event-Driven Routing: A Routing Service for Mobile Fabrics

## 4.1 Introduction and background

## 4.2 Related work

## 4.3 System model

### 4.3.1 Network model

### 4.3.2 Mobility model

### 4.3.3 Analysis of mobility model
## 4.4 Asymmetric routing

- 4.4.1 Problem definition
- 4.4.2 Intuition for multi-parent routing
- 4.4.3 Asymmetric mobility Event-driven Routing (AER)
- 4.4.4 Analysis of AER

## 4.5 Comparison with classical protocols

- 4.5.1 Multipath Bellman-Ford (MBF) protocol
- 4.5.2 Multipath OSPF (MOSPF) protocol
- 4.5.3 Multipath DSDV (MDSDV) protocol
- 4.5.4 Mobility-aware multipath DSDV (MMDSDV) protocol
- 4.5.5 Analytical comparison of protocols

## 4.6 Simulation results

- 4.6.1 Simulation model and experiment setup
- 4.6.2 Validation of the mobility model
- 4.6.3 Quality of multipath trees
- 4.6.4 Cost of routing

## 5. REAPER: RELIABLE ENERGY AWARE PREDICTIVE ROUTING IN DELAY TOLERANT NETWORKS

- 5.1 Introduction and background
- 5.2 Related work
- 5.3 Semi-deterministic mobility and prediction
- 5.4 Routing in semi-deterministic mobility networks
- 5.5 Validation

## 6. KANSEIGENIE: AN ARCHITECTURE FOR FEDERATED PROGRAMMABLE SENSOR FABRICS
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>146</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Motivation and background</td>
<td>147</td>
</tr>
<tr>
<td>6.2</td>
<td>Requirements of federated WSN fabrics</td>
<td>149</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Clearinghouse requirements</td>
<td>149</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Site requirements</td>
<td>150</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Portal requirements</td>
<td>151</td>
</tr>
<tr>
<td>6.3</td>
<td>Specification languages</td>
<td>151</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Resource ontologies for resource description</td>
<td>152</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Experiment specification and workflow control</td>
<td>154</td>
</tr>
<tr>
<td>6.4</td>
<td>KanseiGenie</td>
<td>155</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Architecture and implementation</td>
<td>155</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Portal-based federation in KanseiGenie</td>
<td>159</td>
</tr>
<tr>
<td>7.</td>
<td>WEAVE: STANDARDIZATION OF INTERFACES FOR INTEROPERABILITY ACROSS SECURITY-FABRICS</td>
<td>163</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction and background</td>
<td>163</td>
</tr>
<tr>
<td>7.2</td>
<td>Example application scenarios</td>
<td>164</td>
</tr>
<tr>
<td>7.3</td>
<td>The WEAVE architecture</td>
<td>168</td>
</tr>
<tr>
<td>7.4</td>
<td>Vertical API for search</td>
<td>171</td>
</tr>
<tr>
<td>7.5</td>
<td>Illustrating search applications composed using WEAVE</td>
<td>175</td>
</tr>
<tr>
<td>7.5.1</td>
<td>Urban surveillance scenario</td>
<td>175</td>
</tr>
<tr>
<td>7.5.2</td>
<td>Social networking scenarios</td>
<td>178</td>
</tr>
<tr>
<td>7.6</td>
<td>Related work</td>
<td>180</td>
</tr>
<tr>
<td>8.</td>
<td>CONCLUSIONS AND FUTURE WORK</td>
<td>184</td>
</tr>
<tr>
<td>8.1</td>
<td>Conclusions</td>
<td>184</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Federation issues</td>
<td>186</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Standardization of APIs and resources</td>
<td>188</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Routing in asymmetric networks</td>
<td>189</td>
</tr>
<tr>
<td>8.1.4</td>
<td>Predictive routing in intermittently connected networks</td>
<td>189</td>
</tr>
<tr>
<td>8.2</td>
<td>Future work</td>
<td>190</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Resource specifications for WSN fabrics</td>
<td>190</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Resource discovery</td>
<td>192</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Data as resource</td>
<td>192</td>
</tr>
<tr>
<td>8.2.4</td>
<td>Network virtualization</td>
<td>193</td>
</tr>
<tr>
<td>8.2.5</td>
<td>Connectivity and capacity in mobile and intermittently connected networks</td>
<td>194</td>
</tr>
</tbody>
</table>

Bibliography | 196
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The sensor fabric model and traditional sensor model</td>
<td>7</td>
</tr>
<tr>
<td>1.2</td>
<td>The database model of sensor networks</td>
<td>15</td>
</tr>
<tr>
<td>2.1</td>
<td>Generic fabric manager architecture</td>
<td>27</td>
</tr>
<tr>
<td>2.2</td>
<td>A Kansei node: XSM, Tmote and Imote2 attached to a Stargate</td>
<td>29</td>
</tr>
<tr>
<td>2.3</td>
<td>Kansei testbed</td>
<td>29</td>
</tr>
<tr>
<td>2.4</td>
<td>PeopleNet deployment in Dreese Laboratory</td>
<td>34</td>
</tr>
<tr>
<td>2.5</td>
<td>Sensor deployment in PeopleNet for elevator localization</td>
<td>35</td>
</tr>
<tr>
<td>2.6</td>
<td>Temperature sensor deployment in PeopleNet</td>
<td>35</td>
</tr>
<tr>
<td>2.7</td>
<td>Cellphone-TelosB mote pair deployment in PeopleNet</td>
<td>36</td>
</tr>
<tr>
<td>2.8</td>
<td>PeopleNet network architecture</td>
<td>36</td>
</tr>
<tr>
<td>2.9</td>
<td>PeopleNet software architecture</td>
<td>37</td>
</tr>
<tr>
<td>2.10</td>
<td>A DTN scenario</td>
<td>39</td>
</tr>
<tr>
<td>3.1</td>
<td>Kansei fabric manager architecture</td>
<td>54</td>
</tr>
<tr>
<td>3.2</td>
<td>Stargate Director architecture</td>
<td>56</td>
</tr>
<tr>
<td>3.3</td>
<td>Autonomic software update architecture</td>
<td>58</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.1</td>
<td>Campus wide mobile network</td>
<td>71</td>
</tr>
<tr>
<td>4.2</td>
<td>Average and minimum time to lose all $K$ neighbors</td>
<td>76</td>
</tr>
<tr>
<td>4.3</td>
<td>Difference between disjoint and braided multipath</td>
<td>79</td>
</tr>
<tr>
<td>4.4</td>
<td>AER protocol actions</td>
<td>80</td>
</tr>
<tr>
<td>4.5</td>
<td>Cost of routing versus probability of movement</td>
<td>92</td>
</tr>
<tr>
<td>4.6</td>
<td>Minimum and average time for node disconnection versus node density</td>
<td>94</td>
</tr>
<tr>
<td>4.7</td>
<td>Comparison of analytical and simulation results for time to lose all parent $t(K)$</td>
<td>95</td>
</tr>
<tr>
<td>4.8</td>
<td>Minimum time for node disconnection versus node density for a $K$-parent random tree and balanced tree</td>
<td>96</td>
</tr>
<tr>
<td>4.9</td>
<td>Quality of the tree versus time period of correction</td>
<td>97</td>
</tr>
<tr>
<td>4.10</td>
<td>Quality of the tree versus probability of movement</td>
<td>98</td>
</tr>
<tr>
<td>4.11</td>
<td>Cost of routing versus time period of correction</td>
<td>99</td>
</tr>
<tr>
<td>4.12</td>
<td>Cost of routing versus probability of movement</td>
<td>100</td>
</tr>
<tr>
<td>5.1</td>
<td>A meeting probability frame between a pair of nodes</td>
<td>111</td>
</tr>
<tr>
<td>5.2</td>
<td>Pseudocode for calculating the path vector exchanged between nodes</td>
<td>117</td>
</tr>
<tr>
<td>5.3</td>
<td>Pseudocode for calculating the best $m$ paths to satisfy a deadline of $dl$</td>
<td>119</td>
</tr>
<tr>
<td>5.4</td>
<td>Schematic for information and packet flow among 3 nodes</td>
<td>120</td>
</tr>
<tr>
<td>5.5</td>
<td>Schematic for flow of path information to source and flow of packets to the destination</td>
<td>121</td>
</tr>
<tr>
<td>5.6</td>
<td>Campus scenario</td>
<td>124</td>
</tr>
<tr>
<td>5.7</td>
<td>Bus network routes</td>
<td>126</td>
</tr>
</tbody>
</table>
5.8 Campus network: number of packet forwards vs. rate of traffic . . . . 131
5.9 Campus network: efficiency vs. rate of traffic . . . . . . . . . . . . . 132
5.10 Campus network: throughput vs. rate of traffic . . . . . . . . . . . . 133
5.11 Campus network: average delay vs. rate of traffic . . . . . . . . . . . 134
5.12 REAPER with different deadlines in campus network scenario: effi-
ciency vs. rate of traffic . . . . . . . . . . . . . . . . . . . . . . . . 135
5.13 REAPER with different deadlines in campus network scenario: number
of packet forwards vs. rate of traffic . . . . . . . . . . . . . . . . . . . 136
5.14 REAPER with different deadlines in campus network scenario: through-
put vs. rate of traffic . . . . . . . . . . . . . . . . . . . . . . . . . . 137
5.15 REAPER with different deadlines in campus network scenario: average
delay vs. rate of traffic . . . . . . . . . . . . . . . . . . . . . . . . . . 138
5.16 Bus network scenario: efficiency vs. rate of traffic . . . . . . . . . . 139
5.17 Bus network scenario: number of packet forwards vs. rate of traffic . 139
5.18 Bus network scenario: throughput vs. rate of traffic . . . . . . . . . . 140
5.19 Bus network scenario: average delay vs rate of traffic . . . . . . . . . 140
5.20 REAPER with different deadlines in bus network scenario: efficiency
vs. rate of traffic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 141
5.21 REAPER with different deadlines in bus network scenario: number of
packet forwards vs. rate of traffic . . . . . . . . . . . . . . . . . . . . 142
5.22 REAPER with different deadlines in bus network scenario: throughput
vs. rate of traffic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 142
5.23 REAPER with different deadlines in bus network scenario: average
delay vs. rate of traffic . . . . . . . . . . . . . . . . . . . . . . . . . . 143
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSN</td>
<td>Wireless Sensor Network</td>
</tr>
<tr>
<td>6LoWPAN</td>
<td>IPV6 for LOw power Wireless Personal Area Networks</td>
</tr>
<tr>
<td>AER</td>
<td>Asymmetric Event-driven Routing</td>
</tr>
<tr>
<td>API</td>
<td>Application Program Interfaces</td>
</tr>
<tr>
<td>DSDV</td>
<td>Destination Sequenced Distance Vector</td>
</tr>
<tr>
<td>DTN</td>
<td>Delay/Disruption Tolerant Network</td>
</tr>
<tr>
<td>FD</td>
<td>Fabric Director</td>
</tr>
<tr>
<td>GENI</td>
<td>Global Environment for Network Innovation</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>IaaS</td>
<td>Infrastructure as a Service</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>KD</td>
<td>Kansei Director</td>
</tr>
<tr>
<td>LSAP</td>
<td>Local Service Access Point</td>
</tr>
<tr>
<td>MBF</td>
<td>Multi-path Bellman-Ford</td>
</tr>
<tr>
<td>MOSPF</td>
<td>Multi-path OSPF</td>
</tr>
<tr>
<td>MDSDV</td>
<td>Multi-path DSDV</td>
</tr>
<tr>
<td>MMMDSDV</td>
<td>Mobility-aware Multi-path DSDV</td>
</tr>
<tr>
<td>NTPD</td>
<td>Network Time Protocol Daemon</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>REAPER</td>
<td>Reliable Energy Aware Predictive Routing</td>
</tr>
<tr>
<td>RWP</td>
<td>Random Way Point</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software as a Service</td>
</tr>
<tr>
<td>SD</td>
<td>Stargate Director</td>
</tr>
<tr>
<td>SoA</td>
<td>Service oriented Architecture</td>
</tr>
<tr>
<td>OGSA</td>
<td>Open Grid Services Architecture</td>
</tr>
<tr>
<td>OGSI</td>
<td>Open Grid Services Infrastructure</td>
</tr>
<tr>
<td>OSPF</td>
<td>Open Shortest Path First</td>
</tr>
<tr>
<td>OWL</td>
<td>Web Ontology Language</td>
</tr>
<tr>
<td>PaaS</td>
<td>Platform as a Service</td>
</tr>
<tr>
<td>WSRF</td>
<td>Web Services Resources Framework</td>
</tr>
<tr>
<td>XSM</td>
<td>eXtreme Scale Mote</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Today the domain of sensor networking is expanding and is no longer constrained to scientific applications like environmental monitoring and intrusion detection. While drivers of technology in the field used to be military deployments, today it is city-scale sensing and social applications, such as campus security networks, vehicle-to-vehicle & vehicle-to-roadside communication, city-wide layered sensing & monitoring, social networking, gaming and entertainment. The hardware, software, and protocol design in the area have matured to the level that system developers are moving towards a network centric view of sensing, from a node centric view.

This expanding scope of WSNs applications have fundamentally shifted nature of network design. Future developers of sensor networks will have to design networks that are not only customized and optimized for a particular sensing application, but also built-in services and APIs that will allow for composability of larger applications, using these individual networks as building blocks. The designers will also build services that allow different levels of re-programming of the networks, for example, depending on the level of client authorization and trust. Thus the traditional view of a sensor network being a simple custom built sensor application is vanishing and is being replaced by a view where a sensor network today is seen a general purpose
programmable sensor network or a sensor fabric, on which custom applications can be deployed. In this dissertation, we take the view that most sensor networks need to be designed as sensor fabrics, and should be built with services that enable easy and stable programming and data access.

1.1 Motivation

The nature of WSN applications have changed in two fundamental ways over the last few years. One, the applications have ceased to be edge-alone and have morphed into a edge-enterprise co-design. Two, sensor networks are beginning to be shared by a number of stake holders, with varied requirements – an urban sensing network shared by, say a city police agency and a state-wide security agency would be one such example. As the nature, scale and deployment strategy of WSN applications change, the application and network design process also needs to keep up. This translates into three key requirements for today’s WSNs. One, the networks to be designed should be customizable and re-purposeable on short notice. Two, the networks should be sliceable – the ability to deploy and run multiple applications simultaneously. Three, the networks need to be designed in such a way as to enable interaction, collaboration and federation with other networks or agents outside of the network to achieve some common goal. This has led to the re-examination of the application development model for WSNs.

This emerging architecture will enable unique applications in the urban sensing domain. One such scenario is the urban-sensing environment where a city can deploy the sensors while providing service APIs to users from state level or national agencies. This architecture combined with a new generation of feature-rich devices are now in
a position to support a number of non-traditional applications like GPS navigation, Internet on-the-go, high quality multimedia broadcast and real-time data delivery on mobile devices using local radios and networks. These mobile sensor networks also enable a number of locality-specific applications, such as buddy-messaging, data collection in sparsely deployed sensor networks, virtual social networks where mobile users can publish profiles and search for others with similar interests, and Alert Services which informs users about important events or emergencies in the vicinity of the user. For the above application scenarios, the composability of fabric architecture is important, since it allows hierarchical composition of data and events across different geographical locations and fabrics, to build applications covering larger and larger geographical area.

Programmability and sliceability go hand-in-hand, i.e, a network that needs in-fields programming usually also needs to share the network resources among number of users. Among the primary categories of in-field programming use cases are:

- **Command and control networks**: Since deeply embedded sensor networks are invariably resource constrained, their use for command and control is typically realized in an on-demand fashion. Programmability in this setting typically consists of (re- or de-)activating services, changing the subscriber to service mapping, specifying quality attributes, controlling parameters of operation, and administrating the access rights and policies.

- **Federated networks**: Applications are increasingly being composed to use multiple, independently managed sensor networks. One motivation is that mobile users in one network can obtain added services from embedded networks in their

---

3
locality. In some of these scenarios, user applications can even make proximal networks aware of each other, so that sensor data can be directly exchanged.

Programmability in the presence of network sharing (or a sliceable network) takes several forms:

1. Users access services provided by individual networks, both for domain specific functions as well as generic functions such as security and trust.

2. Dual- (or multi-) use of sensor networks, which involves allowing users to provide and instantiate signal processing modules.

3. More generally, users refine their signal processing modules from time to time, as their needs evolve or unanticipated uses are discovered. Example of intrusion detection. In complex environments, module changes may be context driven, i.e., new modules are instantiated upon analyzing sensing results obtained from prior modules as part of a planning process to achieve mission goals, for example, in an abductive fashion to validate explanatory hypotheses.

- **Testbeds:** Laboratory research (whether analysis, simulation, or small-scale prototype experiments) has in recent years often failed to predict serious issues in scaling of sensor networks, as a result of uncertainty in sensing platforms and field environments. But at the same time, at-scale in-field development has been inconvenient given its high logistics and time cost. Testbeds have thus emerged as a necessary risk mitigation tool, as they achieve the fidelity of field deployment at a total cost of usage close to that of working in a laboratory environment. Testbed programmability is typically accomplished by enabling
users to allocate and schedule virtualized resources for their exclusive access.

Isolation of user-provided experiments is a key requirement if users can operate concurrently.

These use cases illustrate that user programmability support, which was not a norm in early embedded sensor platforms, is an increasingly important requirement for embedded sensor networks. In an increasing virtualized world, where programmable sensor networks becomes the norm, the ability to federate these networks, increase the scope, opportunity and geographical coverage by many folds. For example, one of today’s grander vision is to move towards a global federation of sensor devices that integrate the physical world with the cyber world, not only for information collection, but also for command-control. The use cases in federated networks include, smart-cities – where everything in a home from lights to microwave oven are integrated into one seem-less network and can be accessed anywhere from the cyber infrastructure securely; city-scale sensing, vehicle-to-vehicle communication to prevent accidents, networked virtual-reality and gaming environments, etc.

The Global Environment for Network Innovation (GENI) \cite{5} is one such effort in which a number of academic and industrial organizations are pooling their diverse resources to build a global testing facility. GENI aims to provide researchers with a common control frame work to access remote virtualized resources of every kind; from sensors to supercomputers, from end-hosts to core Internet routers, and from optical backbone fibers to wireless edge networks. A number of challenges needs to be solved before realizing such a seem-less federation.
1.2 Challenges

There are a number of challenges in designing such sensor fabrics. A design for a sensing fabric, should support heterogeneity of devices, provide scalable, reliable, autonomic and fault tolerant network. It should allow the users (depending on authorization level) to program the fabric at different levels, starting at the hardware level to just providing access to the sensed data. The design should provide tools and APIs for specifying, validating and implementing policy for different user access. The fabric should constantly monitor the health of its hardware and software components and take corrective action (if correctable) or inform the administrator. And finally, if any mobile nodes/components are involved, the fabric should manage the complexities arising out of mobility in each of the above functionalities, while providing a consistent framework across mobile and static fabrics.

1.3 The fabric model

A modern sensor network, has three key requirements as we detailed above, (i) Programmability, (ii) Sliceability, (iii) ability to federate with other networks. Intuitively, these requirements can be met by a certain minimum set of services, which will be common to most sensor networks. One way of designing such a network would be to come up with an architecture, which abstracts functionalities that are common to most programmable sensor networks into services, which can then be reused, ported and standardized. We call such an architecture the fabric model.

The model focuses on standardizing services that are common to most sensor fabrics, so that the common services can be reused across fabrics. The fabric model
specifies services and their APIs that we believe should be part of all fabrics, which can be implemented in an energy efficient manner based on the fabric type.

A fabric manager is responsible for providing these system-level APIs, which could be implemented by the fabric designer in manner that suits the fabric. For example, in WSN fabric with less capable devices, the APIs could be implemented in one powerful central machine, while in more capable fabrics, it could be completely distributed. Figure 1.1 shows such a fabric model.

![Figure 1.1: The sensor fabric model and traditional sensor model](image)

1.3.1 Definition

A fabric \( f \) is an entity, that provides a set of services, through which it exposes a set of resources \( R_f \) to its users. A sensor fabric is a fabric, that offers at least the set of generic services \( S_f \) (listed below). Further, a fabric can offer other services called domain services, which are not specified below, but which are specific to the particular type of resources exposed by the fabric. A fabric manager \( M_f \) is responsible for implementing the fabric services.

The minimal set services \( S_f \), that should be implemented by a sensor fabric are:
1. Resource discovery: Allows users to discover fabric resources

2. Resource status: Provides the fabric manager and users with the status of resources in the fabric.

3. Resource reservation: Allows users to get fabric resources

4. Resource operation: Allows users to use the resources, that have been allocated to them.

5. Operation tracing: Providers users with information about the operations they carried out on their resources.

6. Input and output: Allows users to interact with their fabric resources, either during or after the completion of their operations

7. Authentication: Authenticates the user credentials.

1.3.2 Discussion

The services defined above satisfy the three motivating requirements for todays sensor networks specified in Section 1.1. The programmability requirement is satisfied together by resource reservation and resource operation services. The sliceability requirement is satisfied by the resource discovery, resource reservation and resource operation services. Note that, depending on the fabric, sliceability might require isolation or cooperation between different users slices. In such cases, the resource operation service needs to make sure such requirements are met. The federation and cooperation requirement is met by the all of them put together, except the authentication service. While the above services allows a fabric to operate in a federated mode,
there are other requirement for building a successful WSN federation. We deal with these challenges in Chapter 6. While the authentication service is not strictly needed to qualify as a fabric, we include the service since it is required for the safe-operation of the fabric.

Having introduced the fabric model, next we provide a programmatic definition of the seven APIs and then explain each of them in details. The APIs are:

1. ResourceList = ResourceDiscovery (AuthenticationNonce an, Fabric f, ResourceFilter rf)

2. (ResourceHealthList rhl) = ResourceStatus (AuthenticationNonce an, ResourceList rl)

3. (Success/Failure, ReservationList rl, ReservationProperties rp) = ResourceReservation (AuthenticationNonce an, ResourceList rl, AccessOptions ao)


5. (Success/Failure, Trace t) = OperationTrace (AuthenticationNonce an, ResourceList rl, OperationID oid, TraceOptions to)

6. (Success/Failure, DataLog dl) = IO (AuthenticationNonce an, ResourceList rl, IOMode m, ModeArgs ma). The IOMode should include, ‘input’ and ‘output’.

7. (Success/Failure, AuthenticationNonce) = Authenticate (UserCredential uc, Fabric f)
As the above definition states, a fabric provides a set of services or methods, through which user can access resources, build and deploy applications, and collect data from deployed applications. To qualify as a sensor fabric, the fabric should implement at least the above specified list of services. The fabric is managed by a fabric manager $M_f$ which owns all the resources of the fabric and implements the fabric services. The services can be used by an external user or can be used by the fabric manager itself. For example, the ‘ResourceReservation’ and ‘ResourceOperation’ in ‘program’ mode are services, that can be used by an external user to reserve and program a particular set of nodes in the fabric. Whereas, services such as ‘ResourceStatus’ and the ‘ResourceOperation’ service in the ‘recover’ mode can also be used by the fabric manager itself to keep track the status of resources and to recover them from say a bad state.

**Generic services.** The seven services defined above, must be implemented by all fabrics and are hence called *generic services*. The generic services listed above, correspondingly, provide the following functionalities:

1. **Resource Discovery:** The service lets a user look up the list of resources and/or services offered by a fabric. The user will invoke the service with (an optional) resource filer or resource type, which will be used by the fabric to search its resources and the list will be returned back to the user. The fabric administrator can choose to keep this API open (i.e., anybody can invoke it) or closed (authenticated users only).

2. **Resource Status:** The service returns the status of the list of resources provided by the user. This API can again be open or closed.
3. **Resource Reservation:** The service reserves a list of resources provided by the user for use at a specific time in the future. Apart from the resource list, the user also provides an access option such as ‘read-only’, ‘shared’, or ‘exclusive’. The service returns success or failure for each resource in the list and the properties of the reservation (if any).

4. **Resource Operation:** This service lets the user operate or use the resources that she has already reserved. The implementation and the parameters of this service is expected to vary widely depending on the characteristics of the fabric and the resource type. However, the service provides certain common modes of operation such as ‘program’ - the ability to download a user script or executable to a resource, ‘start’ - starting a resource or a program on a resource (that has been already downloaded to the resource), ‘stop’ - stopping the execution of a program on a resource and ‘recover’ - restoring the resource to the original or a ‘clean’ state after a user job has finished execution or after a failure. It is possible that this API might be split into one or more sub-APIs for implementation purposes. The API returns a unique identification number for the operation, which can be subsequently to retrieve logs or data and the information regarding the success/failure of the operation requested by the user.

5. **Operation Trace:** This service lets a user configure tracing/logging options for an operation on a set of resources. The user provides the operation identification number for which the tracing options needs to be configured/manipulated. The fabric, can not only trace or log information about the user resources, but also information about (common) resources or operations which might have an effect
on the users operations. For example, if the user is say sharing a resource like wireless channel with other users in the fabric, then the user might possibly request the fabric to monitor the channel on her behalf. The API returns success/failure of the configuration and also a trace which is a ID, using which the user can retrieve the trace; for example, it can be a file name, a web URL or a port address (to which a user can connect to).

6. **Input/Output:** This service lets the user input or output data, files or programs to fabric. The service can be used by the user to upload a executable to the fabric, which can subsequently be ‘programmed’ onto the user’s resources. Similarly, a trace or log file created can be retrieved using this service. This service might also provide ‘online’ input and output services which lets a user to interact with a program or resources while an particular user operation is ‘active’. The user files/data might be stored by the fabric in a temporary location such as a web-portal, to facilitate the service.

7. **Authentication:** This service authenticates the user using the credentials provided. This could either be a stand-alone authentication system or a federated authentication system using trust-chaining.

**Domain services.** Apart from the generic services specified above, a fabric can choose to implement other services, depending on the capabilities of resources in the fabric and the domain in which fabric is used. It is entirely possible that a particular domain service is quite useful and common to many fabrics in a particular application domain, such services are called as *domain services*. Examples of such services are, ‘reliable messaging’ and/or ‘routing’ in a wireless/mobile network domain, a target
‘search’ service in an security domain or a ‘data aggregation’ service in the sensing domain.

**Ontologies.** Associated with a fabric is an ontology. An ontology is simply a collection of terms and their definitions used to represent a particular knowledge domain, in our case a fabric. The ontology defines the resources and their relationship. It may be represented in a language such as the OWL Web Ontology Language \cite{16,20} or using any other language that the fabric and its users agree on. Not every fabric need to specify the ontology explicitly. For well known fabrics and resources, the ontology might be implicit, although explicit specification using a standard language such as OWL will enable machine readability and fabric-machine interactions. Use of non-standard ontologies and languages could result in an inter-operability problem. We study the issues related to inter-operability of sensor fabrics and ontology mapping and propose solutions in Chapter 6.

### 1.4 Other sensor network models

Here we discuss other application development models that have been used in the sensor networking domain. After an extensive survey of sensor networking applications we classify the current application models into following 2 categories:

- **Node-level models:** In this class of models the application developer takes a node-level view and develops applications for individual sensor nodes. There are two models in this category, namely, Monolithic application model and the Virtual Machine model.

- **Network-level modes:** In this class of model the application developer takes a network-level view and develops specifications for the network as a whole.
There are two models in this category, namely, the Database model and the Fabric model (that is our model).

We next describe the details of each of the models and point out the differences among them.

1.4.1 Monolithic application model

This is the most common and the most primitive model of application development for sensor networks. The application is vigorously tested in a lab environment and then programmed directly on the hardware and deployed in the application environment. The programs/applications written under this model are generally highly optimized to solve a particular problem on a particular network. An application is written for a specific embedded hardware platform, generally using one of the specialized operating systems [2, 27] that have been designed. The common features of these operating systems include a hardware abstraction layer (to increase portability of application across hardware), modularity and are usually event-driven. Building on this model a number of recent research efforts [7, 47, 84] have focused on horizontal standards for APIs between various layers/modules, to make protocols/applications more reusable. This model has been shown as the traditional model in Figure 1.1.

1.4.2 Virtual machine model

In this model [22, 70], a sensor is viewed as a scaled down virtual machine, with limited capabilities. Applications are written as small ‘byte code’ in a language specified by the virtual machine developer and the application can be injected into the nodes at run time. The virtual machine platform usually has some standard set of networking protocols including a ‘code propagation’ protocol, using which the
application ‘byte code’ can be propagated to the nodes. This provides the model with limited programmability, based on the language and the underlying virtual machine capabilities. The application developed usually cannot modify virtual machine and its associated software.

1.4.3 The database model

![Diagram of the database model of sensor networks]

Figure 1.2: The database model of sensor networks

In this model the sensor network is viewed as database and the application which requests and consumes the data is outside of the network. The network is a black box to the user and the user is only permitted to request data in some query format. Usually the data request or the queries are routed through a gateway, which might transform the query into format that can be understood by the network. The network application itself which implements the queries requested by the user, might be written using the monolithic application model (described above) Usually no changes to the network application are permitted. Or even in case where they are permitted,
the gateway parses the query and generates the ‘code modifications’ that are needed
to satisfy the query and downloads the code to the network nodes. The programming
of the network thus happens transparent to the user and the user has no control over
it. Figure 1.2 shows such a model.

<table>
<thead>
<tr>
<th>Model</th>
<th>Programmability</th>
<th>Sliceability</th>
<th>Federatability</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monolithic</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>The model does not guarantee the three key features</td>
</tr>
<tr>
<td>Virtual Machine</td>
<td>Limited</td>
<td>Limited</td>
<td>No</td>
<td>The programmability and sliceability of the network are restricted by the language and the APIs of the VM</td>
</tr>
<tr>
<td>Database</td>
<td>Limited</td>
<td>Limited</td>
<td>Limited</td>
<td>Can support programmability and sliceability only if the a gateway node can interpret and assembly a program automatically based on user queries. Can support federated queries.</td>
</tr>
<tr>
<td>Fabric</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1: Comparison of sensor networking models

Table 1.1 compares the sensor networking models based on the three key features
that are required for today’s sensor networks.
1.5 Fabric model in other application domains

Some of the application development model in other domains has features which are similar to the fabric model and in this section we briefly review these models as it relates to this dissertation.

1.5.1 The fabric model in grid computing

In Grid Computing resources (such as CPU cycles, memory, bandwidth, etc.) from machines belonging to multiple administrative domains are brought together for a common goal, usually by a grid middleware. Examples of such middleware are BOINC, Globus Toolkit, gLite, UNICORE, etc. The middleware provides functionalities for distributed management of heterogeneous computing resources. Grid applications are run on top of the APIs provided by the middleware. The Grid Computing model resembles the fabric model very closely in the sense the middleware provides APIs to the grid application, similar to the fabric manager which provides APIs sensor application. The main difference between the two models is that in Grid computing, the middleware also abstracts the heterogeneity of the hardware, while in sensor networks the user is aware of the hardware specifications and usually writes programs for the specific hardware.

Grid Computing has adopted a number of standards such as the Open Grid Services Architecture (OGSA), Open Grid Services Infrastructure (OGSI), Web Services Resource Framework (WSRF), etc., and a large number of middleware conform to these standards for the sake of interoperability. This also shows the importance of this dissertation, which could serve as the first step towards standardization of the APIs in WSNs.
1.5.2 The fabric model in cloud computing

The cloud computing is very similar to the Grid Computing model, the only difference being that Grid Computing was invented mainly for using computing resources which are being wasted across the Internet, while Cloud providers usually purpose-built the resources. Being purpose built for commercial purposes Clouds also provide a more versatile set of services according to the user needs. Cloud computing consists of three layers or architectures \[3,29\] (as described below), each providing different level of functionality. Each of these three layers can be viewed as a fabric, with a different set of services and methods.

- Infrastructure as a Service (IaaS): This model delivers computing infrastructure, typically a platform virtualization environment, as a service. Rather than purchasing servers, software, data-center space or network equipment, clients instead buy those resources as a fully outsourced service.

- Platform as a Service (PaaS): Here the Cloud provides an application platform such as a database server, a webservice, a storage service, etc. Usually these platforms run on cloud infrastructure and can be used by cloud applications. Users of such services are freed from the responsibility of managing the hardware and software infrastructure and can concentrate on developing their application.

- Software as a Service (SaaS): In this model, entire software applications are offered as a cloud service. Also called as software-on-demand this is probably the most popular of the three models and a huge variety of applications are already provided under this model. Examples of such applications are Google Docs, Acrobat.com, KnowledgeTree, etc.
Cloud APIs can be classified into three categories [9], which broadly correspond to the three layers of Cloud Computing.

- Control APIs, which allow cloud infrastructure to be added, reconfigured, or removed in real time,
- Data APIs, which are the conduits through which data flows in and out of the cloud.
- Application functionality APIs, which enable the functionality with which end users interact, such as shopping carts, wikis, and widgets.

Of the three Cloud models, the PaaS is probably the closest to the WSN fabric model, since this is the layer that is usually accessed programmatically by the user. While, SaaS is usually accessed by users through a (web-based) GUI, IaaS depending on the user needs could be accessed programmatically through APIs. Once again the need for standardization cannot be over stressed in Cloud Computing, and a bewildering number of standardization efforts [1] in various aspects of Clouds have been proposed.

1.6 Service-oriented architecture (SoA) and the fabric model

“Service-oriented architecture (SOA) is a flexible set of design principles used during the phases of systems development and integration in computing” [31]. While the fabric model broadly follows the SoA principles, there are a couple of differences too. First, while SoA specifies only principles for designing portable and distributed computing software, the fabric model goes one step further and specifies classes of methods that a sensor fabric should implement. The second difference is that the
fabric model violates a core SoA principle of *service abstraction*. In SoA a service provider hides as much implementation detail as possible, specifying only the interaction API. But in the fabric model, it not only provides APIs for interaction, but also provides (in general) as much information as possible about the fabric itself, its resources, its services, its ontology, etc. This difference mainly arises out of the uses cases in the two architectures. In SoA the use cases are fixed and unchanging, that is, the system is designed for a particular application in mind and provides the APIs for it. But the fabric model has been specifically designed to cater to much larger application space, where the application itself can change or evolve, and also where the user is not just a client of data generated in the fabric, but might want to effect changes to the application running on the fabric. Thus the fabric model is much more open in its architecture, compared to SoA.

### 1.7 Contributions of this dissertation

In this dissertation, we make several important contributions and we describe each of them and their significance below.

**The Fabric Model.** Through the fabric model of sensor networks we have defined in this dissertation, we formalize a new model of network design and application development for WSN. Loosely based on Service-oriented-Architecture principles, it retains the strengths of SoA without compromising on the important problem of energy-efficiency in WSNs. Sensor networks today are at a critical point, undergoing a consolidation and standardization phase. Our model will serve as one of the reference architectures for the future WSN designer to compare and contrast the strengths and weaknesses of different network and application development models.
The fabric model makes a conscious effort to separate the functionality of a network designer from that of an application designer. This is an important consideration for integration with enterprise networks, where by the model seamlessly bridges the gap between enterprise-networks and edge-networks and integrates naturally with the bigger federation-architecture (see Chapter 6 and 7) that is evolving today.

**Fault tolerant fabric manager.** Fault tolerant software design and implementation has been one of the challenging research areas, especially large scale systems which can involve hundreds or even thousands of systems. In this dissertation, we based on existing fault-tolerance theory we have designed a generic fabric manager which is autonomous in design and can handle faults in both hardware and software. We have implemented the fabric manager for three real-life sensor fabrics, which manage 100s of sensor devices. We believe that our fault-tolerant Fabric Manger implementation, the lessons learned from it makes important contributions in realizing large scale fault-tolerant software systems.

**Routing in mobile asymmetric networks.** Routing in ad hoc mobile networks has been researched well over the decades. However, most of the work has focused on symmetric networks, where all node have the same kind of capabilities and roles. While this might be true in some networks, most of the current generation sensor networks, mesh networks and delay-tolerant networks are asymmetric in nature, so that there are one or more power-nodes with more responsibilities. To our knowledge this problem has not received attention before. We study routing under such a configuration and make a number of new contributions:

1. We mathematically analyze the rate of change of neighbors under a generic random motion model and conclude that the minimum time for neighborhood
change grows much faster with density, than the average time for neighborhood change. This insight has two implications. One, for a protocol which uses multiple parents the cost gain is going to be substantial over a protocol which uses single a parent. Two, for certain mobility regimes, a proactive routing protocol might be more cost efficient than if the protocol is aware of the mobility in the network.

2. Using the above insights, in Chapter 4 we have proposed a new routing protocol called the Asymmetric Event-driven Routing (AER), a pro-active centralized mobility-aware protocol that is cost effective compared with several standard ad-hoc protocols.

Cost efficient routing in semi-deterministic delay-tolerant networks. Research in Delay-Tolerant Network (DTN) has mostly focused on opportunistic routing with the goal of decreasing the end-to-end delay of message delivery. In this dissertation we focus on a slightly different problem of energy efficient routing in DTN, but within bounded (deadline-driven) delay, since energy is an important constraint in human and remotely deployed DTNs. In this context we make 3 important contributions:

1. We analyze human mobility and present a methodology for predicting the contact time of nodes. In many DTNs the node movements and consequently their meetings with each other is not completely random and certain patterns emerge over a period of time. The algorithm learns these pattern of contacts between pairs of nodes and predicts future contacts, based on the determinism/periodicity present in the contact history. Further, we make the predictions
without making any assumptions on the mobility characteristics of the nodes or on the presence of landmark nodes.

2. We design and implement a routing protocol that is reliable, reduces the energy of the nodes in the DTN and also meets the deadline of packets, based on the contact predictions. Our’s is one of the few protocols that does routes on ‘planned-path’ as against the opportunistic forwarding employed by most DTN protocols.

3. We analyze a number of existing routing protocols using simulation and compare the performance of our protocol with popular existing protocols and show that in most mobility regimes our protocol has the best efficiency.

**Federating sensor fabrics.** A federation of WSN fabric testbeds needs to address two core issues: One, an efficient and flexible method for resource description, discovery and reservation. And two, a convenient, uniform way of tasking and utilizing federation resources. In this dissertation (in Chapter 6) we have proposed a federation called KanseiGenie that is compatible with GENI [5], for federating WSN fabrics. KanseiGenie is based on the position that, on one hand, different WSN fabric aggregates can advertise resources based on different resource ontologies, on the other hand, users can obtain uniform experimentation support from a portal supporting a given federation. Central to the KanseiGenie architecture is a mapping between a uniform experiment specification and a non-uniform resource specification, which is handled by the portal.

Our federation architecture design and implementation will serve as a reference architecture in the ongoing debate regarding federation designs. While one option
would be to standardize resource descriptions, which would simplify addressing two issues (mention above), we find that the diversity of sensor (and in general resource) characteristics and the lack of a compelling standard model for describing (wireless) edge networks complicate the federation design aspects.

Further, in Chapter 7 we illustrate how applications can be tailored using multiple sensor fabrics. We consider the domain of urban sensing applications and design domain specific APIs for intrusion-detection application. This contribution illustrates not only the strength of the fabric model, but also how ‘layered-sensing’ and ‘data-fusion’ can be achieved using the fabric model.

1.8 Organization of the dissertation

The rest of the dissertation is organized as follows: Chapter 2 describes the specifications of three different sensor fabrics that we have designed and implemented for three different sensor networks. The design and implementation of an autonomic fabric manager, which is common to all fabrics is presented in Chapter 3. Next we describe two specialized fabric services the Asynchronous Event-driven Routing (AER) for the mobile sensor fabric in Chapter 4 and the Reliable Energy Aware Predictive Routing (REAPER) Service for the DTN fabric in Chapter 5.

While, the previous chapters discussed issues related to design and implementation of individual fabrics, in subsequent chapter we shift focus and discuss the design of WSN federation. In Chapter 6 we describe an architecture for a sensor federation and also discuss challenges arising out of federating fabrics and provide solutions. In Chapter 7 we look the specific issues of writing a intruder detection application using...
multiple sensor fabrics and provide an API and implementation called WEAVE. We
discuss future work and provide concluding remarks in Chapter 8.
CHAPTER 2

SPECIFICATION OF SENSOR FABRICS

In this chapter we describe the specifications of three different sensor fabrics that we have designed and implemented. The fabrics are, Kansei - a wired sensing testbed fabric, PeopleNet - a wireless mobile fabric and VillageNet - a delay-tolerant mobile fabric. Below, we describe the details of each of these fabrics, their resources, services and implementations.

2.1 Overview of generic fabric implementation

A fabric $f$ provides a set of services, through which it exposes a set of resources $R_f$ to its users. Each fabric is managed by a fabric manager $M_f$. It is the responsibility of the fabric manager not only to provide the services to the user, but also to maintain the status of the resources in the fabric and to implement the operations requested by the user on the resources. In general, the resources held by a fabric need not be homogeneous. There could be several types of resources in a fabric, each with its own resource managers which we call Resource Directors ($D_r$). A subset of resources of the same type is referred to as a resource array or aggregate. The fabric manager is hierarchical in design. At the top is the Fabric Director (FD). The Fabric Director provides the framework wherein various service implementations can be plugged-in
depending on the characteristics of the fabric. The Fabric Director is the first point of contact in fabric. It maintains system status, receives the user request/commands, checks status, makes decisions and implements them using various services available in the system. The Fabric Director could be a single software entity or could be divided into a number of Services, each with a well defined API and functionality. Figure 2.1 shows a generic fabric manager which apart from providing the 7 generic services defined in Chapter 1 also provides a Deployment Service and a Routing Service. The Deployment service can be used by the Resource Operation service to program the nodes and the Deployment service can in turn use the Routing service to deliver the programs to the sensor nodes. This example shows that while there are seven services that a fabric is supposed to implement, internally a fabric manager could consist of many more modules, each one of them could have an implementation, that suits the fabric goals.
While the fabric director is responsible for the overall operation of the fabric, it in
turn could implement these operations by delegating them to the resource director(s).
Each of these resource directors in turn could be comprised of sub-directors, depending
on design of the fabric hardware and/or capabilities of the resources. The physical
location of the resource director are irrelevant in this design, as each resource could
have its own director or there could be one director remotely managing all devices
of a type. The Fabric Manager is a logical entity, which is a union of the Fabric
Director, the Resource Directors and the Service implementations, in other words,
the Fabric Manager is the union of all the software entities in the fabric other than
the user application.

In such a distributed system a large number of faults could occur. Faults could
be both in hardware and software modules. Hence, the design of the fabric director
should be fault-tolerant and robust. Also, to minimize the (human) effort involved in
maintaining the fabric and to automatically recover from faults, the design should be
autonomic. The fabric manager could choose to provide the service APIs to external
users through a web-service layer, for the purpose of standardization and machine-
readability. Such an autonomic fabric director design is presented in Chapter 3.

The fabric manager uses a control network to implement its services and control
functions, while the user applications use experimentation network for its functions.
These two networks could be sharing the same physical network or the control network
could be completely separated from the experimentation network. If they share the
same physical network, it is the responsibility of the fabric manager to make sure that
the traffic on the control network does not affect the traffic on the experimentation
network.
2.2 Kansei: A sensing testbed fabric

2.2.1 Kansei overview

Figure 2.2: A Kansei node: XSM, Tmote and Imote2 attached to a Stargate

Figure 2.3: Kansei testbed
Kansei is a state-of-the-art wireless and sensor testbed that makes sensor and wireless experimentation fast and convenient. Kansei services are offered free of charge for wireless sensor network researchers around the globe. Kansei testbed currently consists of four kinds of hardware devices - 96 eXtreme Scale Motes (XSM) \cite{32}, 384 TelosB \cite{25} and 96 Stargates \cite{23}; an Imote2 \cite{6} array and a SunSPOT \cite{24} array are planned for addition in near future. Figure 2.2 shows the hardware configuration of a Kansei node, where an XSM, TelosB, Imote2 and SunSpot are connected to a Stargate. Figure 2.3 shows a picture of the Stargate-XSM-TelosB-Sunspot-Imote2 devices arranged in a grid in the Kansei Testbed.

The XSM and TelosB are mote platforms which have a 8MHz processor, 4-10KB RAM and a communication radio with a range of tens of meters. The Telos radio is compatible with the IEEE 802.15.4 protocol. The XSM radio could function in the 400MHz or the 900MHz band. The mote platforms, for instance, can run TinyOS \cite{27}, a lightweight, event-based operating system that implements a networking stack and a sensor interface, or other similar lightweight operating systems. Each mote integrates a variety of sensors, including a photo sensor, a temperature sensor, four passive infrared (PIR) sensors, a two-axis magnetometer, and a microphone.

The Stargate is an expandable single-board embedded computer with Intel’s 400-MHz PXA255 CPU running the Linux operating system. It has several interfaces, including RS-232, 10/100 Ethernet, USB, and 802.11a/b. Stargates also serve as an integration point for mote-level devices. Stargates are connected through Ethernet network switches to server which provides the fabric manager interfaces to the external world. The Ethernet network acts as the \textit{control network} for Kansei and all the fabric manager actions and services are implemented over this wired Ethernet network.
Hence, in Kansei, while the experimentation network could be one of the wireless radio technologies (such as 802.11b, 802.15.4, 400MHz, 900MHz), the control network is a wired Ethernet network.

In Kansei, the fabric manager APIs are exposed through a JBOSS web-service layer, which helps portability and programmatic interaction with Kansei. A web-portal which interacts with this web-service layer provides remote and ubiquitous user access to the testbed. Users could develop applications using one or more types of resources available in Kansei. Experiments consisting of a combination of applications developed for the above platforms can be uploaded on the devices and the results downloaded remotely through the web-interface.

### 2.2.2 Kansei specification

The Kansei fabric consists of a set of resources $R$, which are exposed to the user through the fabric APIs. The overall objective of the system, which is satisfied by the fabric manager $M_K$, is to allow the users to run experiments on fabric resources. To this end a user creates an experiment specification $User.E$, which is deployed by the fabric manager on the resources chosen by the user.

#### Resource specification

The Kansei resource specification consists of
• All resources belong to the set \( R \), which has a subset \( RA \), which refers to the set of resources available for user experimentation. The set \( RA \) is further divided into \( RA_{xsm} \), \( RA_{tmote} \), \( RA_{stargate} \) and \( RA_{pc} \) for each type of hardware resource.

• For each resource type array \( RA_X \) Kansei specifies a topology \( RA_X.T \), a resource description \( RA_X.D \), such as the cpu & radio chip, the frequencies and channels it supports, the size of program memory/flash, hardware interfaces like usb/serial port, etc.

• For each user of Kansei, there exist one or more virtual resource containers called \( slices \), referred as \( UserName.slice \). To execute an experiment on Kansei, an user can choose resources from the set \( RA \) and add them to one of the user slices, instantiate the slice and execute programs on the slice.

Experiment specification

To execute a program on the Kansei fabric, a user \( UserName \) creates a experiment specification \( UserName.Exp \). A experiment specification consists of

• A user slice \( Exp.slice \) to be used for the experiment.

• A start time for the experiment \( Exp.StartTime \) and the length of the experiment \( E.Length \)

• A list of all resources \( Exp.ResourceRequest \) requested for the experiment, which must be a subset of Kansei resources \( R \).

• One or more \(<\text{resources list, program}>\) pairs called resource configuration \( Exp.ResourceConfig_i \), where a resources list \( Exp.Rlist \) is a list of resource identifiers, and the program is a file identifier to be executed on the list of resources \( Exp.Program \)

• A list of zero or more support file identifiers \( Exp.SupportList \)

• A list of zero or more output file identifiers \( Exp.Output \)

• A list of Kansei services \( Exp.Services \) that the user wants to use for the experiment. The services include program customization, data logging, data infiltration, data exfiltration, VLAN connection (layer 2 tunneling) to other Kansei sites, visualization and back ground noise measurement. The list of Kansei services will keep evolving and the user can choose one or more of these services.
Fabric Manager $M_K$ specification

The overall objective of the fabric manager $M_K$ is to allow the users to run experiments on fabric resources. To satisfy this objective the fabric manager needs to carry out the following functions: (i) maintain the status of the resources, (ii) maintain user information, such as slice configuration, experiment configuration and user files, (iii) implement a reliable deployment and clean up service, (iv) implement a reliable input/output service, and (v) monitor the hardware and software components of the fabric for faults and recover from faults where possible, else notify a human manager.

The Fabric Manager $M_K$ in Kansei has the following specification:

- The Kansei fabric manager takes as input a user experiment specification $Exp$.
- $M_K$ checks if the resources requested $Exp.ResourceRequest$ are available between the time $Exp.StartTime$ and $Exp.StartTime+Exp.Length$. If yes, then the resources are reserved for the experiment.
- At time $Exp.StartTime$ the resource requested is checked to see if it is a subset of the available list $RA$. If yes, the resources $Exp.Rlist$ part of the one or more resource configuration $Exp.ResourceConfig_i$ are loaded with the corresponding $Exp.Program$ and the programs started.
- At time $Exp.StartTime+Exp.Length$ the files in $Exp.Output$ are collected from all resources in $Exp.ResourceRequest$.
- At time $Exp.StartTime+Exp.Length$ the resources in $Exp.ResourceRequest$ are all restored to the original state.
2.3 PeopleNet: A mobile fabric

2.3.1 PeopleNet overview

PeopleNet [17] is a mobile wireless sensor network infrastructure whose first deployment spans Dreese Laboratories, the home of the Computer Science and Engineering department of The Ohio State University. PeopleNet is based on our open source software and hardware infrastructure for installing and maintaining persistent wireless sensor network applications. Since 2007, it has been populated with an increasing set of people-centric applications. PeopleNet conceptually consists of two sets of resources: static array – a static wireless sensing infrastructure and mobile array – a set of mobile wireless nodes. PeopleNet is deployed across 8 floors (as shown
in Figure 2.4 and provides a realistic environment for network and application developers, which includes weak wireless links across floors and walls, interference with 802.11 and other wireless networks, as compared to the ‘serene’ environment of the Kansei testbed.

The static deployment consists of the TelosB nodes instrumented with sensors to sense the temperature in the rooms, the occupancy of conference rooms, the location of elevators etc. Five TelosB motes deployed in the roof of each floor are connected through USB-over-Ethernet to the central server called the Local Service Access Point (LSAP) form the control channel. The hundreds of sensor motes deployed across each floor communicate wirelessly to the control channel motes, which then send the sensor reading through the wired channel. The sensed information consisting of temperature, elevator location and occupancy of rooms, is collated from
Figure 2.7: Cellphone-TelosB mote pair deployment in PeopleNet

Figure 2.8: PeopleNet network architecture
around the building, which is then subsequently available not only via the web but also via the mobile array. Figure 2.5 shows the deployment of TelosB motes in the elevators of the building for localization and Figure 2.6 shows the temperature sensor which is deployed in each room in the building.

The mobile array consists of 35 Cellphone-TelosB mote pairs is deployed across the eight floors. The mobile devices are carried by people who work in the building on a daily basis and they form a mobile multi-hop network while they are in the network. The hardware consist of a TelosB mote integrated with Motoral Rokr E2 Linux phone through the SDCard slot in the phone. Figure 2.7 shows the mote-Cellphone integration and packaging. The integration of the mote with the phone enables peer-to-peer communication and also communication with the static array devices, using the 802.15.4 mote radio without using the cellphone radio. The phones
multi-hop and connect to a mote connected to the LSAP. The LSAP implements the fabric manager APIs, being connected to the Internet allows users to remotely interact with the network. The LSAP acts as the point of integration between the static and the mobile array. This allows for a flexible setup according to the needs of the experimenter, since PeopleNet can be configured as a standalone static sensor network or a mobile ad-hoc network or a hybrid network of mobile and static devices. It also allows for multiple gateway connections, either using the LSAP to connect to the Internet, or to use the cellphone’s GPRS radio to connect to the Internet. Figure 2.8 shows the network architecture of PeopleNet.

On the mobile devices the users have the option to either implement their protocols and applications on the motes using a TinyOS or other embedded programming languages, or in the cellphone using Perl, Python or C in Linux, or inside the Java sandbox using J2ME. The software architecture of Cellphone-Mote platform is shown in Figure 2.9.

2.3.2 PeopleNet specification

PeopleNet has the same specification as that of Kansei discussed in Section 2.2.2. However the implementation of specifications is much more difficult for the mobile array since we do not have a dedicated wired control channel as in Kansei. The implementation requires a reliable ad hoc routing and messaging protocol using the wireless channel which is discussed in Chapter 4.
2.4 VillageNet: A Delay-Tolerant mobile fabric

2.4.1 VillageNet overview

VillageNet is an experimental Delay/Disruption Tolerant Network (DTN) under development at The Ohio State University. In a DTN network communication links between nodes are available only intermittently and messages are buffered and exchanged whenever links become available. VillageNet is being built with the characteristics of DTN networks, such as networks in villages in developing countries which do not have persistent network connectivity or campus networks where connectivity...
is not available in between buildings. VillageNet will provide a real-life platform for protocol and application developers to test and evaluate.

VillageNet will use the Cellphone-Mote mobile device discussed in Section 2.3. In VillageNet a Local Service Access Point acts as the gateway, which enables interaction with the Internet. Messaging between the LSAP and rest of the nodes in the network is implemented by a DTN protocol. The protocol has been optimized for LSAP-to-mobile nodes and mobile nodes-to-LSAP communication. There is no separate control channel and all communication is carried out using DTN communication.

VillageNet is envisaged as a human DTN, that is, a DTN where the mobile nodes are humans carrying devices such as cellphones; in this case students and university staff carrying cellphone between different classrooms and buildings. The performance of a DTN depends on its mobility characteristics and a DTN based on campus environment would accurately reflect the semi-deterministic or semi-periodic behavior exhibited by human mobility. VillageNet is an experimental network which will be deployed in an intermittent manner to collect data, compared to the permanent deployments of Kansei and PeopleNet. Figure 2.10 shows the architecture and operation of VillageNet in the campus.

2.4.2 VillageNet specification

VillageNet has the same specification as that of Kansei discussed in Section 2.2.2. However the implementation of specifications is much more difficult since neither do we have a dedicated wired control channel as in Kansei, nor do we have an always connected wireless network as in PeopleNet. The implementation requires a DTN routing and messaging protocol which is discussed in Chapter 5.
CHAPTER 3

AN AUTONOMOUS, SCALABLE AND FAULT TOLERANT FABRIC MANAGER

3.1 Introduction

Research on wireless sensor networks having increasingly adopted experimentation with actual state-of-the-art hardware and software platforms. As a result shared large-scale testbeds and programmable sensor fabrics have become the preferred basis for experimentation, testing and deployment. Because of the dynamic nature of the sensor network applications, the difference between a fabric and a deployment have decreased, resulting in the programmable fabric architecture.

However, while sensor fabrics need to be highly available to support user application development and deployment, they are themselves complex systems prone to faults in hardware, software specification and software implementation. Further, since they must support wide-ranging experimentation, fabrics should expose as much capabilities of the underlying hardware and software components as possible, so that users may thereby push the envelope of the state of the field by running experiments at the limits of the underlying system. Thus, any fabric design has the conflicting requirements of allowing maximum control by users, while also making sure that the
Fabric remains a stable platform for experimentation, for other users of this shared infrastructure and returns to a stable state at the finish of an experiment/deployment.

Note also that, fabrics cannot use, for example, simple rebooting of the used portion after every job. Firstly, this could take a considerable amount of time and secondly, for the low-end devices that are typically on a fabric, a soft reboot operation might not be available. Even if available this has been proven to be damaging for some of the devices due to buggy implementation of embedded software.

Also, for a long running fabric it is desirable to minimize the human management for its day-to-day operations. Thus a WSN fabric must be feature-rich and flexible, yet stable and self-managing (i.e. autonomic). In this chapter, we present our design approach in implementing an autonomic Fabric Manager (FM) for large-scale WSN fabrics through the use of detectors for recognizing faults and, on detection of faults, correctors to achieve stabilization.

Specifically, we discuss the design and implementation of the Fabric Director for the Kansei testbed described in Chapter 2.2 of this dissertation. The other two fabrics (PeopleNet and VillageNet) even though mobile fabrics, due to the modular nature of the fabric model, use the same basic fault-tolerant fabric director. PeopleNet and VillageNet differ from Kansei mainly in the Deployment and the Input/Output service, where they in turn use different routing services. Hence, the fault-tolerance aspect of the fabric director applies to all three fabrics, even though we use the Kansei fabric to illustrate specific invariants and faults.

In this chapter we describe in depth the design principles of the fabric director. We begin with a review of literature related to the fault-tolerant software design in Section 3.2. Next, we briefly remind readers of the requirements of a Fabric Manager
and its specifications (already discussed in Chapter 2.2) and then describe the fault model for the fabric that is based on a classification of the actual faults encountered in our implementation. Next, (in Section 3.5), we describe the theory behind our fault-tolerance design and detail a fault-tolerant autonomic architecture and implementation for the system. Finally (in Section 3.6) we detail examples of the detectors and correctors implemented in the Kansei testbed.

3.2 Background and related work

Fault tolerance in a large scale distributed system is a challenging problem, especially in systems such as sensor fabrics, where multiple classes of faults could happen. In sensor fabrics, faults relating to software specification, implementation, state corruption and hardware failure could happen and needs to be handled by the fabric manager. Faults in software specifications are particularly difficult to address since the notion of correctness usually depends on the specification. Faults in specification can happen because of a combination of the following factors: incorrect software engineering approaches, multiple designers without proper coordination (usually over time as system evolves), not enough design documentation and usage of software/scripts from other sources for part of the specification. Implementation bugs result in faults which are almost impossible to avoid in a large distributed system. The other kinds of faults namely state corruption are more prone to happen in embedded systems, because of simpler operating systems which does not check for address violations and hardware failure are also more likely to happen in embedded systems because of a
combination of cheap sensor platforms and harsh environments. Examples of hardware faults include, radio drift, clock drift, FLASH memory failure and bootloader failure, etc.

**Fault-tolerance approaches.** A number different approaches have been used to achieve fault tolerance in distributed systems. Approaches such as N-version programming, originally proposed by Avizenis [36], where N-different versions of the program are executed to guarantee at least one correct execution. This is predominantly applies to software systems and doesn’t suite sensor networks. Software rejuvenation due to Huang and Kintala [60], which assumes no knowledge of component invariants, simply gracefully terminates and restarts a component at a clean state, as one way of proactive compensation for transient faults. The observation by Gray [52], that most faults in complex computer systems are soft/transient/Heisenbugs, in that they will likely not be repeated if the component is immediately reinitialized also points to the usefulness of component restarts as a way of dealing with transient faults. Transient faults is an important class of faults in sensor fabrics, given that sensor fabrics allow user programs to execute, and the correctness of such programs cannot be verified.

The snap stabilization approach guarantees that given a protocol satisfies its specification in the presence of transient faults. Starting from any arbitrary state, the system stabilizes to it specification in 0 rounds, after the fault stops occurring. The approach usually consist of passing a token down a tree and cleaning up the state of the system as it proceeds. While, this is a very useful approach for distributed protocols, our complex systems like sensor networks—where there are heterogeneous systems with implicit specifications—this approach does not work well. In general, an action based stabilization approach is not likely to work well for heterogeneous
systems, since specifications of such systems tend to be complex. Also, in such systems its not just the actions that matters, but the sequence of actions also matters to prove stabilization properties and this entail storing detailed logs of actions across components.

One way to define correctness \cite{35} in a system where state corruption cannot be prevented and system specifications themselves may be inconsistent is to specify the correctness approximately with a set of invariants that the system state should be made to obey. This approach has two advantages: (a) It defines correctness in terms of current system state, which can be less complex than an action based approach. In an action based approach, not only do we need to worry about fault events, but also the order in which they happen, in order to make sure the system stabilizes. (b) It makes inconsistent system specifications visible as irreconcilable invariant violations (this sentence is presented in verbatim from \cite{35}).

We combine the above approach with component based fault-tolerance approach presented in \cite{33} by Arora and Kulkarni. This approach abstracts each source of undependability in the system – represented as a class of faults and the corresponding ability of the system to deal with that undependability source is represented as a type of tolerance. Thus, they use two types of components, detectors to monitor system specification for a fault class (in our case invariant based system specification) and correctors to make corrective action to restore the system to a correct state after the occurrence of the fault. A detector together with its log and timing information can be used to catch incorrigible faults and can be used either to change system specification or correct a software implementation. Multitolerance, thus refers to the ability of the system to tolerate multiple fault-classes, each in a possibly different way. The
invariant based approach is also suitable to incremental correctness, as new invariants, detectors and correctors can be added without change to the base software. \[33, 34\] illustrates and delimits the role of this type of detection and correction in the design of various types of fault-tolerance.

### 3.3 Kansei overview

Faults are different, the overall design approach is the same. We use Kansei as a running case study. The design approach applies equal to the other two fabrics.

While the design and implementation of the fabric manager is common to all three fabrics, we use the Kansei testbed as an example to illustrate our implementation, and provide examples of real-life faults and their classifications, invariants, detectors and correctors. Hence in this section we present a brief overview of the Kansei fabric. We refer to Chapter 2.2 for more details. Kansei is a state-of-the-art wireless and sensor fabric that makes sensor and wireless experimentation fast and convenient. The Kansei fabric consists of 3 different types of hardware devices - 100 XSM motes, 396 TelosB motes and 100 Stargates. The Stargate is an cellphone-class device running a Linux OS with a Ethernet interface, while XSM and TelosB are mote devices. Stargates also serve as an integration point for mote-level devices. Stargates are connected through high-speed network switches to an Ethernet back-channel, which provides high-bandwidth connectivity for management commands, data injection, and extraction.
3.4 Fabric director specification

The fabric director is responsible for implementing the overall specifications of the fabric and acting as the single point of contact for the users. The specification of the fabric manager is described in Chapter 2.2.2, reproduced below for convenience.

The Fabric Manager $M_K$ in Kansei has the following specifications:

- The Kansei fabric manager takes as input a user experiment specification $Exp$.
- $M_K$ checks if the resources requested $Exp.ResourceRequest$ is free between the time $Exp.StartTime$ and $Exp.StartTime+Exp.Length$. If yes the resources are reserved for the experiment.
- At time $Exp.StartTime$ the resource request is checked to see if it is a subset of the available list $RA$. If yes, the resources $Exp.Rlist$ part of one or more resource configurations $Exp.ResourceConfig_i$ are loaded with the corresponding $Exp.Program$ and the program is started.
- At time $Exp.StartTime+Exp.Length$ the files in $Exp.Output$ are collected from all resources in $Exp.ResourceRequest$.
- At time $Exp.StartTime+Exp.Length$ the resources in $Exp.ResourceRequest$ are all restored to the original state.

3.4.1 Fault model

The traditional way of defining a fault model and the fault tolerant behavior for a distributed system is to define the class of known faults, to then show that the system is stabilizing under these faults. This approach is impractical for complex systems such as WSN fabrics where we are forced to consider both hardware and software failures as well as faults created by user programs. Given such a system and its intended use, it is generally impossible to fully specify every possible fault that could occur. Thus we take the alternate invariant-perturbation based approach of specifying the ideal behavior of the system, with every deviation from this specification
a manifestation of a fault action. Given the fabric manager specification $K_{fm}$, we derive specifications and invariants for individual components. Any violation of these invariants is considered a fault state, which the system must be designed to detect and correct, if possible. Note that when such a model for faults is used, the fault tolerance of the system depends on the completeness of the invariant specification. This is, in fact, a strength of the architecture in that it allows for gradual addition of invariants as more faults are discovered – say by a human manager or through other fault analysis methods – without redesign of the system components themselves being required.

Note that achieving the stated objective of an operation under every type of fault in the system is not always possible. For example, when a user program is running on a node and if the node hardware suffers a permanent failure, it becomes impossible for the user program to run to completion. So, depending on what is achievable under a particular fault, we classify the kind of fault-tolerance as:

- **Masking fault tolerance:** In the presence of faults, the system/component never violates the specification and eventually resumes/completes correct operation (i.e. faults in the system are not visible to the users).

- **Non-masking fault tolerance:** In the presence of faults, the system/component violates the specification, but when the fault stops, the system/component eventually resumes correct operation (i.e., the user might observe the fault, but the system corrects itself and completes the job).

- **Fail-Safe fault tolerance:** In the presence of faults, the system/component violates the invariant. When the fault stops, the system/component might not
resume correct operation (i.e. the system detects the fault, but is not able to correct it).

Further, the faults in a WSN fabric can be classified as:

- **Transient** faults: These are not permanent and will stop occurring in finite time. Once a transient fault has terminated, the system should correct itself and run to completion.

- **Fail-stop** (or permanent) faults: In this case the system should be marked as incorrect within finite time. A recoverable sub-class of these faults are known as crash-restart faults. A hardware failure is a non-recoverable fail-stop fault, while a process crash is a crash-restart fault.

### 3.4.2 System invariants

Having adopted a *invariant perturbation* fault model, it is important to define the system invariants carefully to capture the faults. The invariants we define for the system can be grouped into the following categories:

- **Job invariants**: Invariants related to the correct deployment, logging and clean up of jobs

- **Health invariants**: Invariants related to the monitoring the health of hardware and software components

- **Resource invariants**: Invariants related to detecting resource conflicts and overuse.
• **User Program invariants:** Invariants specified by the user as a part of his configuration that the user wants monitored during user program execution.

We discuss in detail each class of invariants in Section 3.6.

### 3.5 A fault-tolerant architecture using detectors and correctors

In this section we first outline the theory of detectors and correctors for designing fault-tolerant software and then describe the Kansei architecture in detail. Our design approach for fault tolerance consists of dividing the system into multiple autonomous components that are self-managing, and implementing ‘detectors and correctors’ for these components, thereby making them autonomic. We draw heavily from the theory for fault-tolerant component design using detector and correctors proposed in [34] and [33]. Our approach is directly based on the framework suggested in [35] for implementing large fault-tolerant software systems. Our architectural approach may be summarized as follows:

1. The fabric manager specification defines the overall expected behavior of the system.

2. Based on the overall specification, the resource specification for the fabric resources and their arrangement/topology, the individual software components are designed to be autonomous and co-operating with each other to achieve the system specification.

3. Each component has a well defined sub-specification and a set of invariants are defined based on the specifications. In particular, our architecture is *hierarchical*, meaning the fabric director consists of a number of resource-directors, each
responsible for managing a sub-set of resources. The resource-directors have the same specification as that of the fabric director and hence more or less the same invariants. This hierarchical design helps in isolating faults and also makes the architecture *scalable*.

4. Fault-tolerance is achieved by implementing detectors and correctors on these invariants, which depend on the services provided by a ‘trusted base’ (Section 3.5.1). Once a ‘corrector’ corrects the system to a ‘legal’ state, any future correct execution will result in a legal system state.

5. Detectors can be scheduled to run periodically or based on specific events (e.g. one fault could trigger a detector for a related fault). However, a knowledge of the fault for which a detector has been designed could help tune the frequency of detector, their by increasing the efficiency of the system.

### 3.5.1 The trusted base

The architecture guarantees that as long as the detectors and correctors themselves are not corrupted, the system stabilizes to a stable state. Thus, in order to avoid the faulty components from corrupting the detectors and correctors, they are isolated and implemented on top of what is called the *trusted base* [35]. The trusted base incorporates a ‘trusted store’ and provides scheduling services so that detectors and correctors are invoked randomly and infinitely; this is known as angelic scheduling [35]. More specifically, the ‘trusted base services’ consist of:

1. **Trusted read and write**: For detectors to detect faults, they have to reliably read the state of components however corrupt that location might be. Similarly
for correctors to correct the state they should be able to correctly write to a memory location, however corrupt that location might be. These services are generally available in any modern operating system.

2. **Trusted schedule:** In order to schedule detectors and correctors in a manner that cannot be affected by, or predicted by, the rest of the system’s components, a way of reliably and randomly scheduling processes is needed. We approximate angelic scheduling using a system-based ‘cron’ daemon (available in any standard Unix system), scheduled independently of the system components.

3. **Trusted store:** Detectors and correctors need reliable storage to store private state that cannot be corrupted by other components. On Kansei server, trusted store is implemented using a relational database. On the Stargates class devices, trusted store is provided by a root level process operating on a protected file area.

This approach guarantees that faults generated locally are handled properly. However, in a distributed system, it is still possible for a component to get corrupted by a remote method invocation; where one faulty component spreads the faults to other components. In [35], the authors prove that if the detectors and correctors are executed in synchronized way across the system, the system will eventually stabilize once the faults stop occurring. In our fabrics, we have observed that fault propagation is minimal. Hence, synchronous execution of the detectors is done only on-demand basis.
3.5.2 Fabric manager architecture

The fabric manager is a hierarchical architecture. We explain the architecture in the context of the Kansei testbed. For Kansei, the fabric manager is a two-level architecture consisting of, a ‘Kansei Director’ (KD) that manages the entire testbed, and a ‘Stargate Director’ (SD) which manages the activities of an individual Stargates. The Stargate Director also manages the mote devices connected to the Stargate. Each of these components has its own detectors and correctors. The KD includes a sub-component ‘Chowkidar’, a distributed autonomic self-stabilizing component that continuously monitors the health of all the devices (See [40][67] for detailed description of Chowkidar).

Figure 3.5.2 shows the overall Kansei fabric manager architecture. Functionally there exists a master-slave relationship between the top-level director and its sub-directors. For example, in Kansei, the Kansei Director (KD) sends the commands and details about jobs/experiments to the Stargate Director (SD), the SD executes them and returns the results. The KD also acts as the user-interface and gets the job inputs through the web (service) interface, while the SD acts as the low-level manager that manages itself and the mote devices.

Autonomic behavior: There are two different levels of autonomic behavior in this fabric manager architecture. The first is conceptually the entire fabric, where the whole fabric manages itself. That is, it deploys a job, cleans up a job, returns results, finds faulty devices, tolerates faults and heals itself. The other level consists of the resource directors and the health monitors, in case of Kansei, the Stargate Directors and the Chowkidar. While the fabric director gets the job input from the users, the
resource directors get their input from the fabric director. In contrast to the master-slave functional view, the components are, in fact, independent autonomic (albeit cooperating) entities, with respect to fault-tolerant behavior. Such a system with hierarchical components each with its own fault tolerance is called Multi-tolerant. We now provide details about the Kansei Director and the Stargate Manager

**Kansei Director**

The Kansei Director consists of (a) the Web Service Layer (b) the Database, (c) the Scheduler, (d) the Chowkidar Health-Monitor and (e) the Detectors and Correctors subsystem. For completeness we will briefly describe all the components of Director here while elaborating on the detectors and correctors (in Section [3.6])

- **Web Service Layer:** The Web Service Layer interacts with a web-portal and enables a user to submit jobs remotely and get results along with debugging
information. The jobs can be submitted for any combination of the devices. It also enables programmatic interaction with the fabric

- **Database**: The database provides a secure storage of state and controlled sharing of job state across the different processes of the Kansei Director.

- **Kansei Scheduler**: The scheduler processes queries from the database periodically for pending jobs, allocates resources and schedules them as needed. For each job that is to be scheduled, it creates a job manifest (i.e. a job configuration, as shown in Figure 3.5.2) and sends it to the appropriate Stargate Director, along with the necessary files, which takes care of deploying the job.

- **Chowkidar (Health-Monitor)**: Chowkidar (Sanskrit for watchman) is the health monitoring component of Kansei. Conceptually the Chowkidar is a detector for all the hardware devices in the fabric. It checks and reports on the hardware status of devices. For more details about the architecture of the Chowkidar see [40][67].

- **Detector/Corrector subsystem**: This ensures the safe running of the Kansei Director and is a separate process that checks the invariants of the Director. Any violation causes triggering of the appropriate corrector(s).

Assuming that the operating system itself is not compromised, the detector/corrector process will restore the state of the system to a legal state. Notifications of job terminations are sent to relevant stakeholders - administrator and testbed users.
Stargate Director

The Stargate Director complements the Kansei Director by autonomically managing the stargate and the attached mote devices. Figure 3.2 shows the architecture of a Stargate Director and its interactions with the Kansei Director. The Stargate Manager is a process that autonomically manages a Stargate based on commands it receives from Kansei Director.

The Stargate Director is similar in design to the Kansei Director, consisting of a scheduler, a stable-storage and a detector/corrector subsystem. A ‘Command Listener’ module receives the experiments to be scheduled (replacing the Web Service Layer on the Kansei Director). A NTPD client running on each Stargate synchronizes the device to the Kansei Director and by extension to the outside world.

In order to schedule a job on a Stargate the Kansei Director creates a ‘Job Manifest’ similar to what is shown in Figure 3.2. The manifest specifies the executable...
files to run on each device, under what user it should run it, how long to run it and what files to return after the job is complete. This job manifest is then zipped with other executables and support files required to run the job and sent to the Stargate as the ‘START’ command from the director.

Once the Stargate Director receives the ‘START’ command with the associated zipped files, it unzips the files, reads the manifest and schedules the job accordingly. The manager also stores the information of jobs such as, user, job end time, files to be returned, of the jobs that are currently running in ‘Job Table’. Once the scheduling is done the SD sends an ‘ACK_START’ message to the director. A separate thread of the manager keeps checking the ‘Job Table’ to see if any of the jobs need to be terminated. If it finds any jobs to be terminated, it kills the processes of that particular user under which the job is scheduled, zips the log files that needs to be returned and sends them to the KD. After this the manager sends the ‘ACK_STOP’ message to the director.

**Automatic software update system**

One of the reasons for faults in a WSN fabric is incomplete software updates, resulting in different resources using different versions of the software. Practical impacts of complex faults in a sensor fabric is that it makes software maintenance and updates difficult. A fabric is made of hundreds of individual devices, each of which is running a small distributed part of the fabric manager. A simple push-based software update process will not work in complex and large scale network such as Kansei testbed fabric, since at any given time, a certain percentage of the network devices may be experiencing some kind of fault. Faults that affect the update process include a number of transient device faults such as network hub faults, network interface fault due
to software failure, temporary node failure due to disk overflow, network time-outs due to busy devices, etc. Since a lot of these faults are transient, while the faults affect the software update process, the node/resource itself might still be available for experimentation, albeit executing the wrong version of the software. Making sure the correct software is being run on the fabric resources, will go a long way in minimizing the faults in the fabrics. Thus, a software update system has the following two design goals. One, maintaining the right version of software on the fabric devices, in-spite of faults in devices and network hardware. Two, the software update system must be autonomic with very minimal or no human involvement.

To overcome the challenges due to transient faults, we use the same detector/corrector architecture that we use to monitor the system invariants, for our software update process too. Moreover, instead of pushing the software updates from the Kansei Director to the rest of the fabric components, we use a pull-based update system.
Figure 3.5.2 shows this architecture of the software update system. The Resource Directors (in the case of Kansei, the Stargate Directors) either periodically or when they boot-up check with the fabric director to see if there are any updates to the software and pull the new software if one is available. The Kansei Director stores the latest version of the software that manages the Stargate Director in a ‘Software Archive’ and these files are served to the Stargates through an ‘RSync server daemon’. During a Stargate’s boot-up process a ‘RSync client daemon’ checks the local copy of the software with the remote copy residing at the Kansei Director and if a newer version of the file(s) is available it automatically downloads the file.

A ‘Software Monitor’ process (which is semantically a detector) keeps track of the version of the each file in the software stack and monitors the files periodically for change. Whenever a change in the detected, it restarts the Stargate Director. Thus, human operator wishing to update hundreds of fabric components, need only to update the ‘current version’ of the software in the Kansei Director’s Software Archive. The RSync based software update process, takes care of automatically updating the Stargate Director software inspite of the faults.

### 3.6 Detectors, correctors and invariant for Kansei

In this section we give examples of each class of invariants introduced in Section 3.4.2 and their corresponding detectors and correctors. Note that this is not an exhaustive list of all the invariants. For every invariant there is an associated detector that deals with the faults associated with that invariant. Thus, an invariant is synonymous with its associated detector. At the end of this section, we provide a list of actual faults from our testbed, together with their corresponding fault type.
and invariant. Table 3.1 summarizes invariants, their expected behavior under faults of the overall system and the component where the faults occur. All detectors are scheduled to run in an ongoing but random manner, with frequencies that depend upon the invariant.

### 3.6.1 Invariants and correctors for job control

This set of invariants ensures that jobs submitted by users are deployed, executed, logged and cleaned up properly. An interesting point to note is that a significant number of faults under these invariant tend to be fail-stop faults for which only fail-safe tolerance can be provided.

- **Pre-deployment Invariant (KD):** All devices selected for a job should be in the “ready” or “failed” state. Less than 10% of the selected devices should be “failed”. The detector for this invariant is scheduled before every job deployment.

  **Corrector action:** Mark Scheduler process as faulty (and start a database consistency check). This fault means that the scheduler accepted a job for a topology which doesn’t have enough “ready” devices.

- **Job table consistency Invariant (KD):** If no job is “running”, then none of the devices should be “busy”.

  **Corrector action:** Send a “CLEAR_ALL_JOBS” message to the devices violating the invariant and set their status to “ready”.
<table>
<thead>
<tr>
<th>Invariant Violated</th>
<th>Overall System Behaviour</th>
<th>Individual Component/Program Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job pre-deployment</td>
<td>Nonmasking tolerance: Notify user about bad configuration</td>
<td>Nonmasking tolerance: Notify user about bad configuration</td>
</tr>
<tr>
<td>Job pre-deployment</td>
<td>Nonmasking tolerance: Kill all jobs violating the invariant</td>
<td>Nonmasking tolerance: Kill all jobs violating the invariant</td>
</tr>
<tr>
<td>Job table consistency</td>
<td>Nonmasking tolerance: Kill all jobs violating the invariant</td>
<td>Nonmasking tolerance: Kill all jobs violating the invariant</td>
</tr>
<tr>
<td>Job status</td>
<td>Fail-safe tolerance: Notify user about failed jobs</td>
<td>Fail-safe tolerance: Kill jobs violating the invariant</td>
</tr>
<tr>
<td>Job termination</td>
<td>Masking tolerance: Clean up job violating invariant</td>
<td>Masking tolerance: Clean up job violating invariant</td>
</tr>
<tr>
<td>802.11 Radio</td>
<td>Nonmasking tolerance: Trigger a leader election process</td>
<td>Nonmasking tolerance: Restart radio</td>
</tr>
<tr>
<td>Hardware Health</td>
<td>Nonmasking tolerance:</td>
<td>Fail-Safe tolerance: Notify Health-Monitor of status within finite time</td>
</tr>
<tr>
<td>Software Monitor</td>
<td>Masking tolerance: Update and restart failed component</td>
<td>Masking tolerance: Update and restart the failed component</td>
</tr>
<tr>
<td>TimeSunc</td>
<td>Masking tolerance</td>
<td>Masking tolerance: Restart NTPD</td>
</tr>
<tr>
<td>Disk space</td>
<td>Nonmasking tolerance: Kill erring jobs</td>
<td>Fail-sage tolerance: Kill erring jobs</td>
</tr>
<tr>
<td>Device access</td>
<td>Nonmasking tolerance: Kill erring job</td>
<td>Fail-safe tolerance: Kill erring job</td>
</tr>
<tr>
<td>Frequency access</td>
<td>Nonmasking tolerance: Kill erring ob</td>
<td>Fail-safe tolerance: Kill erring job</td>
</tr>
<tr>
<td>User specified</td>
<td>Fail-safe tolerance</td>
<td>Fail-safe tolerance</td>
</tr>
</tbody>
</table>

Table 3.1: Kansei invariants and stabilization actions
• **Job status Invariant (SD):** All processes of a job should be alive for the entire period specified in the job configuration.

**Corrector action:** Kill the corresponding job and return a “JOB_ERROR” message to the KD.

• **Job termination Invariant (SD):** For all jobs in the job table, the end time should be greater than the current time.

**Corrector action:** Terminate the job violating the invariant and return “JOB_TERMINATED” to the KD.

### 3.6.2 Invariants and correctors for system health

• **802.11 Radio invariant (SD):** The previous radio cell of any device should be same as the current radio cell.

**Corrector action:** Initiate a radio-leader-election process. After leader election, restart the radio starting with the leader as center and moving outwards in concentric circles until a cell is found.

• **Hardware health invariants (SD):** All hardware devices should be up at all times.

**Corrector action:** Notify the Kansei Director about the failed device.

• **Software Monitor invariant (KD & SD):** All software components of the testbed should be alive and run the same version of software.

**Corrector action:** Update the software components and restart.
• **TimeSync Invariant (SD)**: The time difference between Stargate and Kansei should not be more than a threshold.

  **Corrector action:** Restart NTPD.

### 3.6.3 Invariants and correctors for resource monitoring

• **Disk Space Invariant (SD):** No job should consume more disk space than requested in the job configuration.

  **Corrector action:** Kill the corresponding job. Send a “DISK OVERUSE” error to KD.

• **Device Access Invariant (SD):** No two processes (even from same job) should access a hardware device simultaneously.

  **Corrector action:** Kill all processes accessing the device and send an error message to KD.

• **Frequency Access Invariant (KD):** No two jobs should use the same radio frequency/channel at the same time.

  **Corrector action:** Kill all jobs that violate the invariant and send error to KD.

### 3.6.4 User specified invariants

User specified invariants usually do not have any associated correctors. These invariants are monitored by Kansei and any violations are reported to the user along with the job logs. These invariants are useful in judging the fidelity of output produced by Kansei. Some examples of user-specified invariants are:
• At least 95% of the devices should successfully complete programming and start execution.

• At least 90% of the devices should successfully complete executing the user program.

• Each device must have at least 5 neighbors within 5 feet.

3.6.5 Kansei faults

Table 3.2 classifies actual faults and lists the corresponding invariant. Again, this table is not exhaustive. It should be noted that while some invariants were derived directly from the specification, others were added to the system after analysis of a particular fault. The 802.11 radio invariant, for example, was added after we observed that the switching of node radios switch between two different cells with the same “ESSID” when operating in the Ad-hoc mode. This highlights the iterative addition of invariants and correctors.
<table>
<thead>
<tr>
<th>Observed Fault</th>
<th>Invariant Violated</th>
<th>Fault Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>802.11 switching between two different cells with same ESSID in Ad-hoc mode</td>
<td>802.11 radio</td>
<td>Fail-stop recoverable</td>
</tr>
<tr>
<td>Hardware components fail (motes, hubs, wires, Ethernet cards)</td>
<td>Hardware health</td>
<td>Fail-Stop, non-recoverable</td>
</tr>
<tr>
<td>XSM failure due to pin race conditions (two processes accessing the serial port simultaneously)</td>
<td>Device access</td>
<td>Fail-stop, non-recoverable</td>
</tr>
<tr>
<td>Non-uniform hardware/software in a layer (due to a bad upgrade/network error)</td>
<td>Software health</td>
<td>Fail-stop, recoverable</td>
</tr>
<tr>
<td>Resource conflict between jobs</td>
<td>Device access</td>
<td>Transient</td>
</tr>
<tr>
<td>Disk overflow at Stargate</td>
<td>Disk space</td>
<td>Transient</td>
</tr>
<tr>
<td>Database connection dies away at director</td>
<td>Software</td>
<td>Fail-stop, recoverable</td>
</tr>
<tr>
<td>Unclean cleaning of jobs</td>
<td>Job termination</td>
<td>Fail-Stop, recoverable</td>
</tr>
<tr>
<td>Time mismatch between nodes (NTPD failure)</td>
<td>TimeSync</td>
<td>Fail-stop recoverable</td>
</tr>
<tr>
<td>Health Monitor reports bad status (false positive: Node down, reported alive)</td>
<td>Hardware health</td>
<td>Transient</td>
</tr>
<tr>
<td>State corruption in components</td>
<td>Software health</td>
<td>Transient</td>
</tr>
<tr>
<td>Disk overflow on Kansei Director</td>
<td>Software health</td>
<td>Fail-stop, recoverable</td>
</tr>
</tbody>
</table>

Table 3.2: Kansei faults and invariants violated
CHAPTER 4

ASYMMETRIC EVENT-DRIVEN ROUTING: A ROUTING SERVICE FOR MOBILE FABRICS

4.1 Introduction and background

Convergence in mobility devices has combined the capabilities of a cellphone, PDA, GPS and other sensors, and one or more short range wireless radios (such as Bluetooth, WiFi, and 802.15.4). These feature-rich devices are now in a position to support a number of locality-specific applications, such as Buddy Messaging, data collection in sparse Sensor Networks, Virtual Social Networks where mobile users can publish profiles and search for others with similar interests, and Alert Services which informs users about important events or emergencies in the vicinity. This trend motivates us to explore the desirability and feasibility of implementing locality-specific wireless applications primarily via local mobile-to-mobile multi-hop communications. We consider a network model with dense regions of mobile users in a campus area with office buildings, cafes, terminals, etc. Intra-regionally, the user mobility pattern is typically that users are largely static or they move rarely with low speed. Each region is asymmetric in that one or more non mobile resource-rich nodes – Local Service Access Points – that can take on the burden of providing some basic network services.
In this chapter, we revisit the familiar question of maintaining routes in the context of a constrained-mobility and asymmetric environment, focusing particularly on energy efficiency of the protocols. Communications between devices and the LSAP is a primary requirement here as opposed to direct device to device communications, since the latter can be realized indirectly via the LSAP. Device efficiency, in terms of storage, program size, and energy efficiency is also a primary requirement. This leads to studying whether one of the many routing algorithms used in purely static environments or purely mobile environments is well suited, or an alternative algorithm which exploits the asymmetry between the LSAP and the devices is better suited.

Couple of key finding in this chapter is that, as the density of devices in a mobile network increases, the average time for a device to lose all of its neighbors increases more quickly than the minimum time for a device to lose all of its neighbors and that the average disconnection time increases sublinearly (i.e. grows quickly initially and then saturates) with number of neighbors. For a proactive protocol which tries to maintain paths at all times, a protocol which uses the knowledge of mobility will be much cheaper since the protocol adapts to individual nodes mobility, thereby on average the rate of route fix is much lesser. In contrast, a (mobility unaware) periodic protocol will have to operate at much higher rate, since it has to operate at the time period of the first nodes disconnection time (minimum disconnection time), which is much smaller compared to average disconnection time. The above findings has several implications for maintaining a routing structure: First, awareness of device mobility should be exploited to trigger routing message updates, i.e., routing updates are sent based not on the changes in the ‘state’ of the routing table, but rather on the changes in the actual state of the device, e.g., from mobile to static. Second, synchronized
repair of all devices (which depends on the minimum time for a device to lose all of its neighbors) would need to be more frequent than asynchronous repair of each device (which depends on the average time). And third, initially, increasing the number of parent would substantially decrease the rate of node repair, before it saturates.

We are therefore led to consider a protocol that combines the elements of mobility awareness, asynchronous repair, and multiple parents in an asymmetric setting where the route repairs for each devices are calculated at the LSAP(s). We call this protocol the Asymmetric mobility Event-driven Routing (AER).

Our contributions in this chapter. We make a three fold contribution: One, we propose a variation of the Random Waypoint Model for the constrained mobility scenario which captures well human mobility and does not have the drawbacks of a traditional RWP model. We analytically model the mobility and derive equations for the average and minimum disconnection times of the nodes. We also investigate the stability (node disconnections) of multi-parent trees under mobility. Two, we propose AER routing that exploits the resource rich LSAPs and knowledge of mobility. Third, we compare multi-parent adapted versions of the classic Bellman/Ford, OSPF and DSDV protocols, with AER in the constrained mobility regime. We analyze the message cost and latency constraints of all these protocols, and evaluate their performance in simulations as well, to find that AER outperforms others in terms of routing cost and path maintenance for the constrained mobility region.

Organization of the chapter. We begin with a summarization of existing related work in mobile routing in Sections 4.2. Next, in Section 4.3 we describe our network and mobility model, and analyze the properties of the mobility model. In Section 4.4 we formally define the routing problem, and describe our AER protocol.
In Section 4.5, we describe adaptations of classic protocols and compare them analytically with AER. The protocols are evaluated using simulations and the results are presented in 4.6.

4.2 Related work

Many reactive and proactive MANET protocols have been researched, with Ad-hoc On-demand Distance Vector, and Dynamic Source Routing being most popular reactive protocols and DSDV and its various variants being popular proactive protocols. The Zone Routing protocol \[54\] takes a hybrid approach, where it maintains proactive routes to the nodes in the neighborhood and discovers routes to far away nodes on demand. One might argue that reactive protocols are preferable if all nodes are mobile with relatively high speeds. In contrast, this paper we consider structure maintenance in constrained mobility networks, especially as we can reduce the route maintenance cost by considering the multi-path approach.

Extant work in MANET multipath routing \[68\,74\] has mostly yielded reactive protocols, since it is more attractive for recovery from route failures. Multipath protocols do a controlled flooding of packets to discover multiple paths. Distance-Vector Multipath Routing \[82\] is an exception—it finds all available loop-free shortest paths to the destination—but incurs significant storage and messaging overhead. In \[51\] braided multipath routing is considered to decrease the energy spent in constructing purely disjoint multipaths, while providing approximately same level of path availability. While multipath algorithms are usually used to increase path resilience, load balancing or to optimize delay, we explore it in the context of compensating for mobility of the nodes.
Increasingly, attention is being paid to adapt the work on MANET protocols for resource constrained devices, specifically to incorporate simpler protocols with lesser storage and energy requirements. As one data point, DSDV stores several bytes of information per destination in the network and does not scale well to the thousands-of-nodes networks being deployed currently. TinyAODV \[28\] is one such effort to port the AODV to resource constrained devices such as sensor nodes. It is a minimalist implementation for devices running TinyOS which seems to target static networks, nonetheless the implementation still has thousands of lines of code and requires a significant amount of device space. In comparison, the protocols we considered are considerably more frugal. Each node in our protocols takes 2 bytes (ID and hop count) of information for each of the \( K \) (typically 3) parents, which means reserving 6 bytes per LSAP in the routing table. Previous work related to multipath routing has been already discussed in Section 4.4.2.

4.3 System model

4.3.1 Network model

University, business, and industrial campuses typically have a number of human and vehicles carrying devices. Users mostly frequent certain ‘regions’ (e.g., buildings and warehouses) and more rarely move between these dense regions. More precisely, our campus area network is modeled as follows.

Figure 4.1 shows our campus area network model.

1. It has ‘regions’ which are densely populated with static or constrained-mobility nodes.
2. Each region is *asymmetric*, since they have one or more non-mobile devices called the Local Service Access Points (LSAPs). LSAPs are assumed to not be storage space- or energy- constrained.

3. LSAPs of different regions may be inter-connected via the Internet, wireless point-to-point links, a wireless mesh network, or via the nodes moving between the regions (serving as delay-tolerant data carriers), to form a campus-wide network. The type of connection between the LSAPs is transparent to the nodes.

4. Each node knows its unique ID but not necessarily its geographical location in the network.

Figure 4.1: Campus wide mobile network
5. Each node is assumed to have the same transmission radius. A node is a neighbor of another node if the distance between them is at most the transmission radius and we assume that links are symmetric.

4.3.2 Mobility model

Given our focus on mobility inside campus areas, which is often that of humans walking inside or between buildings or that of vehicles moving with strictly limited speeds, the rate of mobility we are concerned with is slower than what is considered in a typical vehicular network. Also, as suggested by previous work on campus area mobility based on WiFi traces [37], only a small percentage of the network is mobile at any given time. Finally, as observed by [57], human users once in motion move continuously for some time towards some particular destination and, upon reaching there, tend to remain for some time. We refer to this a type of mobility as constrained mobility.

Based on these characteristics of low speed, rare movement, and relocate-and-pause, we model constrained mobility in a fashion similar to the Random Way Point (RWP) model [62], but with important differences. Formally,

1. Each node is in one two states, ‘static’ or ‘mobile’.

2. A probability of movement ($P_m$), dictates the transition from the static to the mobile state.

3. Once a node is in the mobile state, it picks a random destination and moves towards that destination with constant velocity. When it reaches the destination, it enters the static state.
Unlike the RWP model, which uses a velocity that is uniformly distributed between [0, Vmax], we use a constant velocity (humans, for instance, move more or less at a speed of around 1.5 m/s). Also, the pause time in the RWP model is also uniformly distributed, whereas the waiting time in the static state in our model is exponentially distributed with a mean of 1/Pm. Note also that the number of mobile nodes in our model is also decided by the probability of movement. Also, in this chapter we are concerned only with intra-region mobility and routing, thus, the above described mobility model describes only intra-region mobility.

4.3.3 Analysis of mobility model

The RWP model has received the following two criticisms [42,85]: One, the average speed of the nodes in the network become progressively slower with time as nodes become trapped in low speed states. Thus, time averages of simulations in the model can be misleading. Second, the average neighbor density in the network oscillates periodically as nodes converge towards the center of the region and diverge again towards the boundaries.

Our mobility model does not share these shortcomings of RWP. The first one is avoided by our use of constant velocity for all nodes, thus the average speed remains the same. As to the second one, both our simulations and user mobility experiments on our campus show that the average neighbor distribution in our model although varies slowly over time, the pattern is not repetitive and does not cause waves. It also seems intuitively plausible that neighborhood density is neither uniformly distributed nor oscillating with a fixed period. For example, the average traffic in buildings is high in the mornings and evenings, and in cafeterias is high at mealtimes.
If all nodes have \( K \) neighbors, for \( K > 1 \), the minimum and average time for a node to lose all neighbors is a function of \( \lambda_t (\lambda_t = 1/P_m) \), the mean waiting time in the stationary state.

**Property 1:** The average time \( t(K)_{avg} \), for nodes to lose \( K \) neighbors, is given by \( \lambda_t \frac{K}{K+1} \), while the minimum time to lose \( K \) neighbors (defined as the q\% quantile), \( t(K)_{min} \), is upperbound by \( \lambda_t \frac{-\log(1 - q)}{q} \).

**Proof:** When a node is in the static state, its waiting time is distributed exponentially, with mean \( \lambda_t \). Let \( X_1..X_K \) be random variables for neighbors \( 1..K \) that denote their respective times to switch to the mobile state. And let \( X_0 \) denote the time for the node itself to switch to the mobile state. Now, \( X_0..X_K \) are independent of each other and identically distributed exponential random variables.

If we assume that a node will lose all its neighbors as soon as it switches to the mobile state, the time for the node in question to lose all its neighbors is:

\[
t(K) = \min(X_0, \max(X_1..X_K))
\]

where, \( X_i \) are independent and identically distributed such that \( X_i \) is distributed with the mean \( \lambda_t (0 < \lambda_t < \infty) \) and \( i (0 \leq i \leq K) \)

By independency, we have

\[
P(t(K) > x) = P(X_0 > 0 \ and \ \max(X_1,...,X_K))
\]

\[
= P(X_0 > x) \ * \ P(\max(X_1,...,X_K) > x)
\]

We further have

\[
P(\max(X_1,...,X_K) > x) = 1 - P(\max(X_1,...,X_K) \leq x) = 1 - (1 - e^{-x/\lambda_t})^K
\]
It follows that

\[ P(t(K) > x) = e^{-x/\lambda_t} \times (1 - (1 - e^{-x/\lambda_t})^K) \]  

(4.2)

and we have a closed form distribution for \( t(K) \), which gives the distribution of time for a node to disconnect from all \( K \) neighbors. Now the average disconnection time can be easily found by finding the expectation of distribution given by Equation 4.2. Using the telescope formula \( E(t(K)) = \int_0^\infty P(t(K) > x)dx \), we obtain the expectation value of \( t(K) \) as

\[ t(K)_{avg} = E(t(K)) = \lambda_t \times \left( \frac{K}{K+1} \right) \]  

(4.3)

For finding the minimum time for a node to disconnect from all its \( K \) neighbors, we find the \( q \)-quantile of the distribution. Quantiles are points taken at regular intervals from the cumulative distribution function of a random variable. Dividing the distribution into \( q \) essentially equal-sized data subsets is the motivation for \( q \)-quantiles. In our case we define \( t(K)_{min} \) as 10 percentile or 0.1 quantile. The \( q \)-quantile can be obtained by equating Equation 4.2 to \((1-q)\) and solving for \( x \). For large \( K \), the value of \( x \) can be approximated to

\[ t(K)_{min} = x \approx -\lambda_t \times \log(1 - q) \]  

(4.4)

where \( q=0.1 \).

In the above derivation we have assumed that a node loses a link immediately after it starts moving. In order to take into account the time taken by nodes to move out of the transmission range \( R \) of their neighbor, let \( D_{avg} \) be the average distance a node travels to move out of transmission range of a neighbor. If a node moves a distance of \( 2R \) in any direction it would lose all its neighbors and thus \( D_{avg} \) is
upper bound by $2R$. A node which is at the edge of its neighbors range can lose its link by moving infinitesimally small distance which can be approximated to 0. Thus $(0 < D_{avg} < 2R)$. Now let $v$ be the constant node velocity, then, the time to move out of a node’s neighborhood is given by $D_{avg}/v$ and the Equation for $t(K)_{avg}$ and $t(K)_{min}$ becomes:

$$t(K)_{avg} = \lambda_t * \left( \frac{K}{K+1} \right) + \frac{D_{avg}}{v} \quad (4.5)$$

$$t(K)_{min} = -\lambda_t * \log(0.9) + \frac{D_{avg}}{v} \quad (4.6)$$

Figure 4.2 plots the Mean (Equation 4.3) and minimum time by substituting the value of $K$ and $q$ in Equation 4.2 and solving for $x$. From the figure we see that, while the minimum time is fairly constant, the average time increases sublinearly with $K$.

Figure 4.2: Average and minimum time to lose all $K$ neighbors
Property 1 is preserved if nodes have different numbers of neighbors, but each node selects exactly $K$ of its neighbors to be its parents in a multi-parent tree; in this case, the property becomes relative to the node losing all its $K$ parents. Experimental validation of Property 1 is given in Section 4.6 via simulations.

4.4 Asymmetric routing

We begin by defining routing requirements in asymmetric mobile networks, then discuss the intuition for routing based on multi-parent trees, and finally describe the AER protocol.

4.4.1 Problem definition

The goals of routing in an asymmetric mobile network are:

- For each mobile node, to discover and maintain paths from it to the LSAPs of its region, assuming that such paths exists.

- For each LSAP, to maintain paths to each node in its region, assuming that such paths exists.

- Each LSAP should maintain a registry of active nodes by reliably detecting each node ‘join’ or ‘leave’ from its regions network within a reasonable amount of time.

Note that a solution to the problem may require nodes to only maintain (one or more) parents as opposed to the entire paths. And as motivated previously, any-to-any node routing may be accomplished indirectly via LSAPs, which would reduce the program complexity and energy consumption of node routing, and exploit the possibility that most application traffic is bound to and from the LSAP.
4.4.2 Intuition for multi-parent routing

Multipath routing algorithms have better path availability than single-path routing even if the paths from these parents to the LSAP are not necessarily link disjoint or even of shortest length [51]. One implication of Property 1 is that mobility is better tolerated by maintaining more than one parent per node, which can be easily concluded from the sharp increase of $t(K)_{avg}$ in Figure 4.2 for small K. The second implication of Property 1 is that, for a proactive protocol which tries to maintain paths at all times, a protocol which uses the knowledge of mobility will be much cheaper since the protocol adapts to individual nodes mobility, thereby on average the rate of route fix is much small (since $t(K)_{avg}$ is large). In contrast, a (mobility unaware) periodic protocol will have to operate at much higher rate, since it has to operate at the time period of the first nodes disconnection time ($t(K)_{min}$) which is much smaller compared to $t(K)_{avg}$.

On a related note, [51] shows that in a static sensor network setting the braided-multipath approach results in approximately the same path availability as does the disjoint-multipath approach. In the latter, no two paths from a source to a destination share any common links, while in the former for every link on the primary path there exist $K$-alternate links. Thus, there is no single bottleneck link in the braided multipath approach whose disconnection results in the non-availability of a path between the source and the destination. Figure 4.3 illustrates the difference between a disjoint and a braided multipath. A protocol that implements such a tree is also likely to be highly efficient in the usage of storage space. For example, the AER protocol we propose next takes only 2 bytes (ID and hop count) of information for each of the $K$
(typically 3) parents, which means it needs only 6 bytes per LSAP for the routing table.

Our approach to multipaths is to ensure that each node has at least \( K \)-parents, of which at least one is a shortest path parent. We build a spanning tree rooted at each LSAP, such that each node in the region has multiple parents which are of lesser or the same hop-count as that node. If routing through the primary parent (one with the lowest hop count) fails, then other parents are explored for availability. This algorithm is easily implemented on a resource constrained mobile device. Although it is designed for any value of \( K \), in practice, we find that there is not substantial improvement when \( K \) exceeds 3, so all our evaluations in this chapter use 3 parents.

4.4.3 Asymmetric mobility Event-driven Routing (AER)

To maintain parents at nodes that lead to an LSAP as well as paths at an LSAP to lead to each reachable node in its region, protocol AER implements a centralized form of link-state routing at the LSAP. Each LSAP collects and maintains an up-to-date snapshot of the ‘link-states’ of all reachable nodes in its region. Once this region snapshot (‘network-map’) is reliably obtained by the LSAP, it calculates the best paths to each node and then sends via source routing to each mobile node a
message that contains the set of up to $K$ parents that the node should use. An abstract version of the protocol actions is shown in Figure 4.4.

![Figure 4.4: AER protocol actions](image)

To maintain connectivity, the LSAP continuously monitors the network map to check if any of the nodes are in danger of losing all their $K$ parents, i.e., LSAP finds nodes with only one active parent. In this case, the LSAP calculates a new set of parents for each such node and sends to it a parent-set update. Thus the core of the protocol boils down to efficiently collecting a reliable and consistent snapshot.

**Definition 1 (Snapshot):** Snapshot of the network maintained by LSAP at time $t$ is the list of all active links in the network.

**Definition 2 (Consistent snapshot):** A Snapshot maintained by LSAP at time $t$ is consistent if it is a subset of network state at time $t'$ where $t' < t$. 

80
A simple way of maintaining a consistent snapshot would be to make each node report every single event to the LSAP. A more energy-efficient alternative is to exploit the knowledge of mobility at a node, and trigger link updates when the mobility state of a node changes. More precisely, the specific ‘mobility events’ that will trigger an event-report to the LSAP are:

1. If a node changes its state from mobile to static.
2. If a node loses one or more parents.
3. If a node in the mobile state, then after every $D_{\text{avg}}$ distance (the average distance to move out of a node’s neighborhood) that it travels.

For obtaining a snapshot at time $t$, the LSAP considers all the neighbor table update received up to time $t - \Delta$ ($\Delta$ is max one-way delay from mobile node to LSAP). Or in otherwords messages will be buffered till current time $(t)$ - message generation time $= \Delta$. Later in Theorem 1 (in Section 4.4.4) we show that a snapshot obtained by the above procedure is indeed consistent.

AER has two exception cases to handle. First, when a node loses its last parent before its new parent-set arrives, AER correctness remains unaffected since the parent update messages will reach the node, assuming there exists a path from the LSAP to the node. But if the node wishes to send a message to the LSAP in this case, it must execute a ‘node-join’ procedure, where it locally searches for $K$-parents. This procedure is, however, costly and AER is designed to avoid this from occurring in the common case.

The second case is that of region initialization, where none of the nodes have any parents. In this case, the LSAP can initiate a wave computation (similar to the one in MDSDV protocol we will describe shortly), using which the nodes can initialize
their parent-sets. Note that in the scenario where nodes join the region one-by-one, they would end up executing the node-join procedure described above.

Discussion. Recall that we allow nodes with hop-counts equal to that of the node to be its parents while building the multi-path tree. This is motivated for AER by the following simple reason: nodes that are one hop away from the LSAP will have only one path to the LSAP without any sibling-parent (a parent with an equal hop-count) messaging, consequently making the one hop nodes the weakest links in the tree and violating the braided-multipath criteria. But now we have to avoid infinitely long paths on sibling links, so a node is allowed to forward a packet to a sibling-parent only once per hop-count. That is, if a packet travels on sibling link, which is realized by reserving a single bit in the routing header for this purpose, it can be forwarded only to a proper-parent (a parent with a lesser hop-count) in the next hop, thus guaranteeing convergence of routing. This also restricts the maximum number of hops traveled by a packet to twice the length of a shortest path, which helps path reliability.

4.4.4 Analysis of AER

Here, we analyze the correctness of AER and show that its snapshots are consistent. Next we analyze its latency constraints, and finally we derive an expression for the routing message cost.

Before we formally prove the correctness of the protocol, we spell out the requirements of our protocol and how we implement them.

All links are symmetric: The assumption of symmetric links can be achieved by considering only links which are above a certain receiver power level (commonly
called inner-band links), thereby making sure that both the nodes see each other.

**Global ordering of messages at LSAP:** This can be implemented in two ways, one by running implementing a global timesync protocol. In [55], the authors present a global timesync which is robust even in networks with partition and mobile networks. The other way to implement the global ordering is to implement MAC layer timestamping at the transmitting node and every intermediate node adding the time (queuing delay) that the message spent at the node. Thus, the LSAP can calculate the packet origin time by summing up the queuing and transmission delay for each hop.

**Node movement can be detected locally:** This can be accomplished by either by using a sensor, such as an accelerometer (we are in the process of implementing AER or real hardware which have on-board 3D accelerometer), or by continuously monitoring the neighbor table for changes in the neighborhood. [39] suggest a stability parameter based on the number of beacons received from each neighbor and calculating a weighted average to determine the relative motion of the node.

**Message losses will be handled by retransmissions:** Recall that in our protocol each node has K nodes. A Message loss during transmission will be handled by retransmitting the packets through the same or a different parent (if that parent went away). A node which has lost all its parents will execute the ’node-join’ procedure as explained above and retransmit the event.

**Theorem 1 (Snapshot Consistency):** The snapshot maintained by LSAP using the neighbor table updates triggered by mobility events is consistent.

**Proof:**
We start by observing that any node which loses a parent (because of mobility or otherwise) will report the event to the LSAP. The movement of a node $j$ can cause changes in $j$’s neighbor table and tables of $j$’s neighbors. In this case, since links are symmetric, it suffices that only $j$ report the changes.

Now recall that the messages can be globally ordered at the LSAP and let $\Delta$ be the max one-way delay from the mobile nodes to LSAP. Since the LSAP waits upto $\Delta$ time before constructing the snapshot, all messages sent before $t - \Delta$ will be received (message losses will be handled by retransmissions) at the LSAP and thus snapshot constructed will reflect a subset of the network state at time $t - \Delta$ and hence the snapshot is consistent. □

**Lemma 1 (Snapshot Timeliness):** For AER to guarantee that each node has a path to the LSAP, the sum of the maximum times to (1) detect an event ($t_d$), (2) report the event to the LSAP ($\Delta$), (3) compute the snapshot ($\Delta$) and (4) get a fix back from the LSAP ($\delta$) is less than the minimum time for a node to lose all its parents ($t(1)_{\text{min}}$).

In AER a mobile node receives a fix when it has lost $K-1$ parents, that is, when it has just one parent. Hence, to guarantee a path at all times, the time for the node to detect the loss of the last but one parent, $t_d$, the time to report the event, $\Delta$, and the time to receive a fix $\delta$, should be less than the minimum time to lose the last parent $t(1)_{\text{min}}$. That is,

\[ t_d + \delta + 2\Delta < t(1)_{\text{min}} \quad (4.7) \]

$\Delta$ and $\delta$ are network delays in the order of milliseconds and are much smaller than the second term of $t(1)_{\text{min}}$ given by Equation 4.5.
To calculate $t_d$, note that the mobility of node $j$ can cause node $j$ to lose a parent or $j$’s child to lose a parent (i.e., $j$). If $j$ loses a parent it will detect it immediately. If $j$ is the parent of another node and that link is lost, $j$ will detect this within $D_{avg}/v$ time after it starts moving. Thus, the maximum time for detecting an event, $max(t_d)$, is at most $D_{avg}/v$.

**Analysis of routing cost.** We analyze routing cost in terms of the number of messages sent by the entire system per second; we take a system view, rather than a per-node view, since the nodes cooperate in forwarding the routing messages. We assume in the rest of the chapter that apart from the routing messages each mobile node broadcasts a heart beat message periodically (once every $t_{hb}$ seconds), to assist in neighborhood discovery process.

We call the time in which each node will require at least one fix as the correction period $t_c$ and measure the routing cost $C(N)$ for fixing each node once; $C(N)/t_c$ thus gives us the routing cost per unit time. Note that $t_c$ is not a protocol parameter, but a measured time, which depends on the mobility rate of the nodes. In other words, $1/t_c$ gives the frequency at which the protocol collects snapshots.

We observe that the average time for each node to have been fixed at least once by LSAP is upper bounded by $t(K-1)_{avg}$, since the protocol fixes the nodes when they have just parent. Additionally, the nodes would take $\Delta$ to report the event and another $\delta$ to receive the fix. Thus, the total time in which every node receives a fix is

$$t_c = t(K-1)_{avg} + 2\Delta + \delta \quad (4.8)$$

Cost for heartbeat messages: Each of the $N$ nodes send a heartbeat message once every $t_{hb}$, so the total number of heartbeat messages sent in time $t_c$ is at most $\frac{N*t_c}{t_{hb}}$. 

85
Cost for building the network map: In time $t_c$, each node would have lost at least $K - 1$ parents, so each node would have reported $K - 1$ events. On average, each message travels over $H/2$ hops, where $H$ is the maximum hops in the network, thus number of messages sent to build the snapshot is $\frac{(K-1)NH}{2}$.

Cost for fixing the nodes: Each node receives a fix in time $t_c$, where each message travel over a average of $H/2$ hops. So the total number of ‘route fix’ messages are $\frac{NH}{2}$.

Total routing cost per second: Adding these three costs and dividing by $t_c$ we get the total routing cost per second cost to be

$$NH\left(\frac{K}{2t_c} + \frac{1}{t_{hb}H}\right)$$

(4.9)

4.5 Comparison with classical protocols

We have implemented asymmetric multipath versions of three well known proactive routing protocols, which we briefly described here. The reason for choosing these protocols is as follows: The multipath Bellman-Ford is implemented to compare with a simple distributed protocol. Since AER is a centralized protocol, we felt compelled to compare with another centralized protocol, the multipath OSPF. DSDV is by far the most popular adhoc protocol and hence we implemented two versions of DSDV, a pure multipath version and a mobility-adaptive multipath DSDV. The difference among these protocols is primarily in the way they build the node-to-LSAP trees; they use the same approach for building the LSAP-to-node paths, each nodes periodically sends their neighbor table to the LSAP. In this Section we compare them analytically and in Section 4.6 we present simulation results for the comparison.
4.5.1 Multipath Bellman-Ford (MBF) protocol

In order to compare the asymmetric approach with a pure distributed protocol, without focusing attention on optimizations, we implemented a multi-path version of the Bellman-Ford algorithm. In MBF, once every correction period, $t_c$, each node calculates its gradient, which is the minimum of the gradients of its neighbors with a non-zero ‘witness number’, incremented by 1. The ‘witness-number’ of a node is the number of neighbors which have a path to the LSAP and have a gradient less than that of the current node. In a typical connected network, each node should have at least one node with a non-zero witness-number and a gradient less than the current node. After calculating its gradient, the node broadcasts its gradient and witness number. The neighbor with the minimum gradient becomes the primary parent and the node looks for $K - 1$ other neighbors with less than or equal gradients. Our use of the witness-number lets us deal with the problem of network disconnection and the ‘count-to-infinity’ problem and achieves stabilization of the gradient quickly (within max-hop-count rounds).

**Proposition 1:** To guarantee that a path to the LSAP exists at all time from each reachable nodes, the MBF time difference between two consecutive correction cycles (or gradient messages), $t_c$, should be less than half the minimum time for a node to lose all its parents. That is, $t_c < \frac{t(K)_{\text{min}}}{2}$.

**MBF routing cost:** In time $t_c$, MBF expends $N$ messages for building node-to-LSAP paths (which also serve as heartbeat messages) and $\frac{NH}{2}$ messages for building the LSAP-to-node paths, resulting in a total routing cost per second of $NH\left(\frac{1}{2t_c} + \frac{1}{t_c H}\right)$.
4.5.2 Multipath OSPF (MOSPF) protocol

Recall that in OSPF [14] each node distributes its neighbor table to the entire network periodically and each node builds its paths based on the entire link-state topology. We adapt OSPF so that instead of all nodes building their own routes, the LSAP builds the routes (i.e., selects the $K$ best parents) and distributes them to the mobile nodes. This is very similar to the AER, the difference being that while AER builds its network map in an incremental manner through ‘mobility events’, MOSPF builds its network map in a periodic and synchronized manner. Once the network map is built, the LSAP finds nodes that are at risk of losing all parents and proceeds to fix them with a new parent set.

**Proposition 2:** To guarantee that a path to the LSAP exists at all time from each reachable node, the MOSPF time difference between two consecutive correction cycles, $t_c$, should be less than the minimum time for a node to lose all its parents minus the maximum network delay ($\delta$) to fix the node. That is, $t_c < t(K)_{\text{min}} - \delta$

**MOSPF routing cost:** In time $t_c$, MOSPF spends $\frac{N t_c}{t_{hb}}$ for heartbeat messages, $\frac{NH}{2}$ messages for building the network map and $\frac{NH t_c}{2t(K)_{avg}}$ messages for fixing routes of mobile nodes, resulting in a total routing cost per second of $NH\left( \frac{1}{2t(K)_{avg}} + \frac{1}{2t_c} + \frac{1}{t_{hb}H} \right)$.

4.5.3 Multipath DSDV (MDSDV) protocol

The DSDV [76] protocol implements Bellman-Ford algorithm, but tags route entries with sequence numbers to deal with routing loops and the count-to-infinity problem inherent in the approach. To optimize DSDV for a single destination and for multiple parents, we combine DSDV with wave propagation, initiated at the LSAP.
Once every correction period, \( t_c \), the LSAP triggers rebuilding of the tree by broadcasting a special ‘tree-refresh’ message, which consists of the ‘hop-count’ to the LSAP and a sequence number. Each node that receives a new ‘tree-refresh’ message sets the source node as its primary parent and rebroadcasts the message with the ‘hop-count’ incremented by one. Nodes continue to listen for other potential route advertisements with hop counts less than or equal to their current hop count, but with the latest sequence number. Messages with older sequence numbers are discarded. To guarantee that a path exists from every node to the LSAP at all times, the LSAP should trigger the tree construction before any node lose all its parents. To construct the LSAP-to-node paths after the tree construction, the nodes send their ‘neighbor table’ and their parent list to the LSAP, from which the ‘network map and ‘Active Register’ are constructed.

**Proposition 3 (Correction period):** To guarantee that a path to the LSAP exists at all time from each reachable nodes, the MDSDV time difference between the end of two consecutive tree-refresh cycles, \( t_c \), should be less than the minimum time for a node to lose all its parents minus the time taken to build the tree \( t_{RT} \). That is, \( t_c < t(K)_{\text{min}} - t_{RT} \)

**MDSDV routing cost:** Analysis shows that MDSDV uses \( N \frac{t_c}{t_{hb}} \) messages to build node-to-LSAP paths (the tree building messages also serve as heart beat messages) and \( \frac{NH}{2} \) messages to build LSAP-to-node paths, resulting in a total per second cost of \( NH(\frac{1}{2t_c} + \frac{1}{t_{hb}H}) \).
4.5.4 Mobility-aware multipath DSDV (MMDSDV) protocol

The AER protocol uses the knowledge of mobility at the nodes to optimize the routing overhead in the system. Thus in order to compare the mobility-aware AER (centralized routing) with a mobility-aware distributed routing, we improved the Multipath DSDV described in Section 4.5.3. Instead of the sending the routing updates periodically as in MDSDV, in MMDSDV the routing updates are triggered by mobility-event just like in AER, i.e, if a node changes its state from 'mobile' to 'static', loses a parent or if it moves for a distance of $D_{avg}$. Apart from mobility triggered updates, a node send the 'regular' routing updates whenever its hopcount changes due to route updates from its neighbors or if it has not sent any route updates in the last $t_c$ time (guarantees path existence). This is again a multipath protocol where each node chooses $K$ parents based on the route advertisements from its neighbors.

To construct the LSAP-to-node paths after the tree construction, the nodes send their 'neighbor table' and their parent list to the LSAP periodically once every $t_c$, from which the 'network map and 'Active Register' are constructed.

The time constraint for MMDSDV is same as that MDSDV (stated in Proposition 3).

**MMDSDV routing cost:** In time $t_c$ each node would have lost (K-1) parents and would have generated K-1 route advertisements. On average since each node as K parents we can assume that each node also has K children. Hence each route update will be propagated by K nodes. Thus each node would send (K-1)K route advertisements in time $t_c$. Hence MMDSDV uses $NK(K-1)$ messages to build node-to-LSAP paths (the tree building messages also serve as heart beat messages) and
Algorithm | Time Constraint
--- | ---
AER | \( t_c = t(K-1)_{\text{avg}} + 2\Delta + \delta \)
MBF | \( t_c < \frac{t(K)_{\text{min}}}{2} \)
MOSPF | \( t_c < t(K)_{\text{min}} - \delta \)
MDSDV | \( t_c < t(K)_{\text{min}} - t_{\text{RT}} \)
MMDSDV | \( t_c < t(K)_{\text{min}} - t_{\text{RT}} \)

Table 4.1: Summary of correction period for different algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cost</th>
</tr>
</thead>
</table>
| AER | \( NH\left(\frac{K}{2t_c} + \frac{1}{t_{\text{hb}}H}\right) \)
| MBF | \( NH\left(\frac{1}{2t_c} + \frac{1}{t_{\text{hb}}H}\right) \)
| MOSPF | \( NH\left(\frac{1}{2t(K)_{\text{avg}}} + \frac{1}{2t_c} + \frac{1}{t_{\text{hb}}H}\right) \)
| MDSDV | \( NH\left(\frac{1}{2t_c} + \frac{1}{t_{\text{hb}}H}\right) \)
| MMDSDV | \( NH\left(\frac{1}{2t_c} + \frac{K(K-1)}{t_cH}\right) \)

Table 4.2: Summary of cost for different algorithms

\( \frac{NH}{2} \) messages to build LSAP-to-node paths, resulting in a total per second cost of \( NH\left(\frac{1}{2t_c} + \frac{K(K-1)}{t_cH}\right) \).

4.5.5 Analytical comparison of protocols

Table 4.5.5 summarizes the time constraints for the protocols to guarantee a path from all nodes to the LSAP. Table 4.5.5 summarizes the routing cost per second to satisfy the time constraints; i.e., to guarantee path availability. To illustrate the performance of these protocols, consider \( K=3 \) and \( N = 30 \). We can calculate the routing costs for different probabilities of movement, with values for \( t(K)_{\text{min}} \) and \( t(K)_{\text{avg}} \) obtained either via Matlab analysis (as explained in Section 4.3.3) or obtained
via simulation (as explained in Section 4.6); we show later that these two value sets are rather similar. Figure 4.5 shows the latter costs, again assuming there is no routing loss. We see that AER has the least cost under all mobility conditions and also has the least growth rate. MDSDV and MOSPF have similar performance, with MDSDV being simpler to design. MMDSDV begins close to MDSDV but as mobility increases it grows fast (but it has lower routing failure as we will show later). MBF being the simplest and completely periodic the cost grows quickly and has the highest cost.

Figure 4.5: Cost of routing versus probability of movement

4.6 Simulation results

In this section, we first validate the mobility model results of Section 4.3.3 and then evaluate the performance of the protocols with respect to tree quality and routing cost, via simulations.
4.6.1 Simulation model and experiment setup

We have developed a discrete event *Asymmetric Mobile network simulator (AMns)* in Matlab that can:

- Allow different mobility models to be plugged in.
- Simulate message passing, both broadcast and unicast.
- Drop packets based on a link error rate.
- Simulate node failures, both transient and permanent.
- Simulate collision probabilistically using exponential backoff model.
- Accurately reproduced simulation results by using the same random seed(s) in initialization.

Using the simulator, we simulated a 100 x 100 m region with an LSAP situated at its center. The mobility model for nodes is the one described in Section 4.3.2. The communication model for nodes is the unit disk model: the unit represents the transmission radius $R$ and the use of the disk implies link symmetry. The default values of parameters number of nodes(N)=30, radius of transmission(R)=20m, number of parents(K)=3 and velocity of movement(v)=1.5m/s, unless otherwise stated. Each point in our simulation graphs represents the average of 50 different random instances of simulation.

4.6.2 Validation of the mobility model

To validate Property 1, we conducted a simulation where a multi-parent tree was built using MDSDV at the beginning of the simulation and the tree was then allowed
to decay –without ever being rebuilt– as the nodes started moving. We measured the time taken by the first node to disconnect from the tree and the average time taken by node to disconnect, i.e., time taken for 50% of the nodes to disconnect, for different values of $N$ and $K$. The results are shown in Figure 4.6.

Figure 4.6: Minimum and average time for node disconnection versus node density

The figure shows that as $K$ increases the minimum time changes only slightly but the average time increases substantially and particularly for $K = 3$. This is consistent with the analytically derived graph of Figure 4.2. That graph predicts that the average time starts to flatten out around $K=5$, while the simulation finds that the value starts to saturate around $K=3$, which is why our graph is plotted only for $K$ in 1..3 and we use $K = 3$ for comparing different multi-path protocol performance.

We also repeated the same simulations keeping $N$ constant and varying the probability of movement. Figure 4.7 shows the average and minimum time to lose all
parents (for $K=3$), along with the values predicted by Equation 4.5 and 4.6 from Section 4.3.3. From the figure we see that our theoretical and simulation values for $t(K)$ are in agreement, especially for $t(K)_{\text{min}}$.

Figure 4.7: Comparison of analytical and simulation results for time to lose all parent $t(K)$

Next, we question whether the method used for selecting a parent from among the neighbors affects the disconnection time. We used two different methods to select the parents: (i) random parent selection as in MDSDV and (ii) a location based balanced tree construction, which chooses parents to be as geographically separated as much as possible, the logic being that if the parents are well separated that will offset the movement of child in any direction. Figure 4.8 shows a plot of minimum time for disconnection for the two algorithms, the random tree being marked as ‘RT’ and balanced tree being marked as ‘BT’. From the figure we see that the performance of the simpler approach random selection, which eschews hard-to-satisfy assumptions of
localization, is adequate as compared to balanced trees. What matters more is the choice of $K$.

![Stability of Tree Vs Density](image)

**Figure 4.8:** Minimum time for node disconnection versus node density for a $K$-parent random tree and balanced tree

### 4.6.3 Quality of multipath trees

We evaluate the performance of AER in contrast to the multipath versions of Bellman-Ford, OSPF and DSDV algorithms. To compare with a single parent protocol, we also simulate single parent Bellman-Ford (represented with the legend 'BF' in the plots).

Figure 4.9 shows quality of the multi-path tree, in terms of the path non-availability in the network, i.e., the percentage of nodes without a path to the LSAP, with respect to the correction period. From the figure, we see that AER performs fine overall and is independent of correction time as it is not a periodic protocol. MMDSDV also
being mobility-aware performs better than other periodic protocols but has slightly more disconnections as the time period increases. MDSDV, MOSPF, MBF and BF all being purely periodic protocols show a similar trend, getting adversely affected as the correction period increases. BF is affected the worst since it is a single parent protocol.

![Percentage of disconnected nodes Vs Correction Time](image)

**Figure 4.9: Quality of the tree versus time period of correction**

Figure 4.10 shows the percentage of node disconnections versus the probability of movement in the static state. From the figure, we see that MBF and BF, which are completely periodic and distributed, perform poorly as the probability of movement increases. MDSDV and MOSPF also shows a similar trend but grows much slowly than MBF and BF. Once again MMDSDV is the second best and performs well till the probability of mobility is about 0.2, but then slowly start performing worse; a correction time of 10 seconds seems to satisfy their timing requirements (as given in Table I) till the probability of movement is 0.2. AER performance surprisingly
improves slightly as the probability of movement increases. The reason for this result is that, as the movement increases, the rate at which snapshots are taken at the LSAP increases and a better snapshot leads to better route updates; of course this improvement comes with an increase in routing cost.

![Figure 4.10: Quality of the tree versus probability of movement](image)

4.6.4 Cost of routing

Figure 4.11 shows the cost of routing, i.e., the number of messages sent per second in the network for different values of the correction period, \( t_c \). The probability of movement is kept constant at a value of 0.25. From the figure, we see that all periodic protocols are substantially influenced by the time period. AER also shows a similar trend, but saturates quickly. The high cost of AER for low \( t_c \) is because of the very high rate of heartbeat messages (we use same value of \( t_c \) for the heart beat timer too). We also see that the routing cost of the periodic protocols goes down with respect to
correction time, albeit with a decrease in the number of connected nodes, as shown in the performance plot in Figure 4.9. This decrease in cost is a potentially misleading artifact of our metric: as the time period increases, the number of nodes that are disconnected from the LSAP increases and hence the overall cost of the connected nodes sending neighbor table updates over multiple hops to the LSAP goes down.

Figure 4.11: Cost of routing versus time period of correction

Figure 4.12 shows routing cost versus the probability of movement. The correction time, \( t_c \), is kept constant at 10 sec (albeit this is not material for AER). From this figure, we again see the trend of decreasing cost at all protocols other than AER, while AER increases almost linearly from a very low value, as the probability of movement increases. The explanation is again that while AER successively adapts to mobility and maintains (actually slightly increases) the level of connectivity, in the other protocols the number of disconnected nodes increase and so the cost actually decreases. And although the cost of AER increases with probability of movement,
since frequency of AER snapshot and route fixes increases, this increased cost is still less than the cost of all other protocols for most of the constrained mobility region and the increased traffic does not overwhelm the availability of bandwidth.

Figure 4.12: Cost of routing versus probability of movement

In summary, then, we find that AER has rather low cost compared with the other protocols in the constrained mobility region while guaranteeing equal or better connectivity. As the mobility increases, AER maintains high connectivity with a increase in cost proportional to the mobility, but this is still cost-effective in comparison.
CHAPTER 5

REAPER: RELIABLE ENERGY AWARE PREDICTIVE ROUTING IN DELAY TOLERANT NETWORKS

5.1 Introduction and background

Disruption Tolerant Networks (DTNs) are characterized by intermittent connectivity and high end-to-end delays. Packets are transferred whenever a contact becomes available for a node. Examples of such networks vary from satellite networks to mobile human networks to vehicular networks. Some DTNs, such as wild life monitoring networks or people-based mobile networks, have a high constraint on energy and storage because of remoteness or battery operated nodes. Many DTNs may have hot-spots or gateways, where Internet connectivity might exist and where connectivity to outside of network could be established. In such networks, messages can be sent to the hot-spots or even to entities outside the DTN, by identifying and transferring messages to a node, which has a better likelihood of meeting the hot-spot. such multi-hop messaging could decreasing the delay of messaging and improve the system throughput.

Further, it is conceivable that the messages in such a network might be of different priorities and might consequently have different delay/deadline requirements. For
example, a villager who wants an answer for the query “where is the doctor?” will have much higher timing requirements than another villager trying to find out “what are the farmers in the next village sowing?”. It might be even more important to provide message priorities and quality-of-service in a DTN network than in the regular MANET, because of the nature of the network.

Extant research in DTNs have focused mainly on message reliability or message delay, ignoring the energy efficiency for most part. As DTN deployments become a reality, energy efficiency of such networks will become critical. However, message delay and reliability will remain critical given the network characteristics.

In this chapter, we present Reliable Energy Aware Predictive Routing (REAPER) for DTNs which takes into account all the important factors while making forwarding decisions, i) message deadline ii) reliability – the two traditional DTN metrics and (iii) energy efficiency. Our routing protocol calculates low cost paths and decides on which path to forward the message, such that, it not only minimizes cost but also delivers the message within the deadline. The routing achieves energy efficiency by forwarding a single copy of the message, on a predicted path that has very high reliability. The algorithm, first predicts contacts between node pairs, based on their history and then uses this information to build end-to-end paths. The reliability of the prediction depends on the mobility characteristics of the network, but works very well for a number of DTNs scenarios, such as human and animal networks, which are shown to have inherent semi-deterministic mobility. Also the protocol can be tuned to different message deadlines and it achieves the best possible efficiency under a given message deadline.
5.2 Related work

5.2.1 Characterization of mobility in DTN

Grossglauser and Tse in their seminal work [53], analyzing real-life traces, showed that the inter-contact times of nodes follow an exponential distribution and one implication of this result is that, any two-hop forwarding scheme can be successfully used to route packets in DTN networks with bounded finite message delay. But Chaintreau et al [45]—again analyzing traces—claim that the inter-contact times in DTNs followed a heavy tail power law distribution, followed later by an exponential decay. They, however, attributed the later sharp exponential decay to the short experimentation period and hence, argue that it should be ignored. They further showed that for any routing algorithm, with a power law slope lesser than 1, the mean expected delay is infinite, and if the slope is greater than 2, a routing strategy with finite delay is possible. Karagiannis et al [66], in what can be considered another seminal work, made similar observations as Chaintreau et al regarding the dichotomy (initial part is a power law with the latter part being exponential decay) in the inter-contact time distribution. They, however, observed that this dichotomy is more a function of the semi-periodic movement of human and other animal networks. They called the time period after which this switch happens the ‘characteristic time’ of the distribution. This result also clarified that the exponential distribution was important while making predictions about delay characteristics and than finite delay routing is indeed possible in DTNs.

Most extent work on mobility characterization uses aggregate population statistics. Conan et al [46] argue that pairwise inter-contact patterns are a more refined and efficient tool for characterizing DTN’s and show that pair-wise inter-contact time
is more likely to follow the log normal and exponential distributions. Based on our own trace analysis, we agree with this finding and believe that the variation between individual node’s traces is indeed significant and any mobility prediction based on aggregate level statistics is likely to produce sub-par performance.

A number of different mobility models had been proposed for ad hoc and DTN scenarios. Earlier models [19, 43, 62, 78] tended to be based on random distributions, and had a number of drawbacks [10]. A number of recent works have proposed increasing sophisticated mobility models. Particularly, models have been proposed based on the observation that human mobility is not completely random and has various characteristics based on periodicity (day/night, weekday/weekend, etc.), hotspots, communities, social groups, etc. Hong et al proposed Group Mobility [56] where nodes move in groups, to capture the social behavior in campuses. The Weighted Way Point Model [58] is a simple way point model but with weights assigned to each location deciding the node’s movement instead of random mobility. The T++ [69] model is an empirical joint space-time registration model, where space-time correlations are generated via popularity of locations. Hsu et al proposed the Time Varying Community Model [59], which uses communities and the mobility between these communities is dictated by the transition probabilities of Markov-chain process. This model captures both the time and spatial correlations in human mobility and also generates traces which are shown to have similar characteristics to real-life traces.

5.2.2 Routing in DTN

DTN routing algorithms can be classified into two broad categories, forwarding based and replication based. Earlier protocols focused mainly on different variations
of replication or epidemic routing, since achieving minimum delay was the main goal of these protocols. The simplest of them was Epidemic [81] which transfers a copy to every new contact that doesn’t already have a copy of the packet. To minimize the number of packets – and hence conserve bandwidth and energy – algorithms like Spray and Wait [80] limit the number of copies of a packet in the network and a packet is replicated to say, only $L$ contacts. The protocol while being quite simple, shows that there exist a $L$ for each network, above which the decrease in delivery delay is not significant. However, the main drawback of this protocol is that, the protocol is not tunable and finding the right number of copies is non-trivial. MaxProp [41] prioritizes packets based on an estimated probability of delivery, calculated based on a node’s meeting frequency with its peers. While, this protocol could be very effective in networks with short contacts, such as vehicular DTNs, in other DTNs the protocol reduces to flooding. RAPID [38], computes an utility for each packet and starts replicating packets with the highest utility. The protocol presents a general framework where routing metrics like delay, energy, bandwidth and storage can used to create utility functions which is then calculated at every node to optimize the utility. RAPID, again, reduces to flooding when enough network and buffer capacity is available.

The other class forwarding based algorithms, focus on choosing the right node to forward a packet instead of replicating the packet. This methodology is inherently more energy efficient, because of the lack of flooding (and the associated clean up), but it could affect the reliability or delivery delay if the nodes are not chosen properly. In [61], Jain et. al provide a theoretical basis for the expected performance of protocols, based on the amount of knowledge available to them and propose five
different forwarding algorithms each with increasing levels of knowledge about the system. PROPHET [71] calculates a delivery predictability based on the frequency of meeting and forwards a packet if it is above a threshold. Other probabilistic forwarding algorithms [44, 86] use a time homogeneous Semi-Markov model to predict the probability distribution of future contact time to determine the next hop. While, these algorithms learn the mobility of node and assumes semi-deterministic mobility, they also assume the existence of landmarks or communities which provide the basis for the Markov model. [72] uses an optimal probabilistic forwarding metric which is derived by modeling each forwarding as an optimal stopping rule problem. Although it considers regularity between mobility of nodes and it requires global contact information, which can be difficult to obtain.

5.3 Semi-deterministic mobility and prediction

A number of past work (as discussed in Section 5.2.1) in characterizing mobility have shown that, human and animal mobility is not completely random and a has various statistical characteristics such as periodicity, communities, group behavior, etc, along with various degrees of randomness and variations. Of course, which behavioral characteristic dominates a particular network would depend on what the particular network is designed for, and the properties of the mobile nodes. For example, a vehicular DTN based on buses in a campus or cities, would exhibit a very high degree of periodicity and determinism, but very little group behavior. On the other hand a network based on animals, such as Zebranet [64] would have high group behavior, but very little periodicity. Karagiannis et al [66] also showed by analyzing several real-life human mobility traces, that there exist, what they is call the ‘characteristic time’ in
inter-contact time distribution, usually of the order of half-a-day to one-day, which is
the result of periodic behavior. Examples of such behavior are, humans following a
more or less deterministic pattern during the work-week returning to the same place
such as homes, offices etc., and starting over. Another way to look at this result is
that, humans have a bigger cycle of periodic behavior, but within that cycle they ex-
hibit various amounts of randomness. We call such a mobility as semi-deterministic
mobility.

This means there is more information in human mobility and contact patterns
than previous thought and by capturing this information we could predict the future
contacts between nodes. While, a number of previous efforts focus on prediction, they
assume either a uniform network-wide behavior, i.e., all nodes in the network exhibit
the same level of determinism, global information or the existence of landmark nodes
or access points, that could be used to model the mobility of the nodes. Each of these
assumptions have their drawbacks and are impractical in most cases.

A good mobility prediction methodology will have a significant impact on the DTN
routing and energy efficiency. Most of the current DTN routing algorithms focus on
whether or not to make a copy when meeting a node and not on planning a path,
because path planning requires either a priori knowledge of the paths or prediction
of the paths. Path planning is an efficient way of routing, since one can reduce the
number of copies of a packet ( and hence the energy required) and yet achieve high
reliability. But for high performance of path planning algorithm, we need high quality
prediction, which might not be possible in all DTNs. However, it might be possible
in a class of DTN networks which exhibit the semi-deterministic mobility. In order to
build a path based routing, we need a prediction scheme, which can not only predict
the delay for next contact, but also the specific contact node. Using such a prediction we can build a routing protocol which can not only build the shorted path, but also the shortest path that meets its deadline requirement.

Thus, the prediction problem can be stated as below. Given, a history of contacts between a pair of nodes, we are interested in predicted the time by which they are likely to meet next. The prediction methodology should also have the following properties:

1. The prediction should work for varying degrees of randomness and determinism in the meeting patterns.

2. If there is determinism in the meeting pattern, it should take advantage of it to deliver a more accurate prediction.

3. The meeting pattern observed could be a result of multiple periodic or random processes superimposed. The prediction should be able to handle such superposition of multiple random processes.

4. Should be simple enough to be implemented on resource constrained nodes.

Next, we state the only assumption we make for our prediction methodology.

**Assumption 1: Symmetric neighbor discovery:** We assume that when a node $i$ discovers node $j$, simultaneously node $j$ also discovers node $i$. This assumption helps in making symmetric predictions between a pair of nodes, without exchanging any information. In practice this can be implemented using a two-way handshake based neighbor discovery.

We also state the list of common assumptions made by other prediction scheme, that we do not require.
• No landmark or access points: Most contact prediction algorithms, assume the presence of landmark or access point nodes, around which communities are built or which provide geo-location information, based on which a Markov-chain model is built for prediction. We do not assume any such nodes and our prediction scheme is based only on the past contacts. Hence, our scheme can be used in even in DTN network where there is no other infrastructure.

• No exchange of contact information or meta-data: Since the contacts are symmetric and the predictions are made only about direct neighbors or contacts, no contact information is exchanged.

• No global time synchronization is needed: Each node maintains the history of contacts in the local time and since no contact data is exchanged, there is no need for any kind of synchronization.

### 5.3.1 Prediction problem

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_{ij}$</td>
<td>History of meeting between node i and j, an array of tuples $&lt;t_c, l_c&gt;$ time of contact $t_c$ and length of contact $l_c$</td>
</tr>
<tr>
<td>$F_{ij}$</td>
<td>Frame length of meeting times between node i and j, that is the length of time after which the meeting pattern between i and j keep repeating</td>
</tr>
<tr>
<td>$S_n$</td>
<td>Time Slot $n$</td>
</tr>
<tr>
<td>$Ac_{ij}(S_n)$</td>
<td>Average number of contacts at time slot $S_n$ between nodes i and j</td>
</tr>
</tbody>
</table>
The prediction problem can be formally stated as follows: *Given a history of contact* $H_{i,j}$, *which is an array size* $hl$ *of tuples* $(t_c, l_c)$ *of time of contact and length of contact* and time $t$, *predict the delay before the next contact.*

### 5.3.2 Prediction methodology

Our prediction methodology is based on the two key concepts of *frames* and *histograms*. Frames are designed to capture any periodicity or deterministic pattern in the contacts and histograms are used to deal with the randomness inside a frame. The basic idea of our scheme is to exploit the semi-determinism exhibited by most human and animal based networks, as we explain in Section 5.2.1. We are conscious of the fact that the contact pattern could potentially be an output of a number of superposed periodic processes with varying amounts of noise. We identify the largest time period in the contact pattern after which, the pattern repeats.

**Definition 1.** A frame, for each pair of nodes, is the minimum (finite) amount of time after which the contact pattern of the nodes repeats itself. A frame is characterized by a single parameter, the framelength.

We will describe in detail the procedure to learn the framelength in Section 5.3.3, here we will detail the procedure for predicting contacts, assuming we know the framelength for every pair of nodes. Predicting how the nodes behave relies on predicting their behavior in the frame. For a contact pattern to be deterministic means, the frame is finite. After predicting the framelength, we construct a histogram of contacts inside the frame and use it to predict the contacts.

The frame $tf_{i,j}$, between nodes $i$ and $j$ is divided into slots $S_n$ of fixed length $sl$, where $n (1 .. fl/sl)$. Next, we find the average number of a contacts per slot.
$Ac_{i,j}(S_n)$ for each slots in $S_n$, given the history, which is simply the ratio of the number of contacts in time slot $S_n$ and the number of frames of contact history data available, i.e., $hl/fl$. Please note that $Ac_{i,j}(S_n)$ is not a probability density function, but a histogram function, i.e.,

$$\sum_{S_n} Ac(S_n) \neq 1$$

$Ac_{i,j}(S_n)$ is a histogram function, normalized by the number of frames of data available. $\sum Ac_{i,j}(S_n)$ gives the average number of contacts per frame. Given the $Ac_{i,j}(S_n)$, it is very easy to get a predication for having at-least one contact in future. Figure 5.1 show a frame and the histogram function.

![Figure 5.1: A meeting probability frame between a pair of nodes](image)

Let us consider a simple example, where the frame length is one day and the slot length is one hour, and lets say we have contact history data for one week. Let us also assume that there are 3 contacts in between 9 am - 10 am in the history data on various days. In this case, we will have 24 slots in the frame, and the average number
of contact for the time slot $S_{10}$ (9 am - 10 am) will be $3/7$, since we have 3 contacts in that time slot and we have 7 frames of history data.

**Predicting contact delay:** Given the normalized histogram of average contacts $Ac_{i,j}$ and the frame length $tf(i,j)$, we calculate the maximum expected delay for the next $X$ contacts to occur, by summing up the individual contact probabilities in each slot cyclically, starting from the current slot, so that the sum equals the threshold $X$ and finding the corresponding delays for the contact. A natural threshold in for our case is 1.0 or a number very close to 1.0, which means that as the cumulative sum crosses 1.0 you are assured (probabilitically) of at-least one contact before or by that time slot.

Given that the current meeting between nodes $i$ and $j$ is happening in time slot $c$, the delay for next $X$ contact $P_{i,j}(c,X)$ is given by

$$P_{i,j}(c,X) = y, \text{ such that } \sum_{i=c+1}^{y} Ac(S_i) \geq X$$

Thus, a node $i$, for each node $j$ it meets, predicts the maximum contact delays for the next $n$ contacts, by setting the value of $X$ to $1,..,n$, finds the contact slots relative to the frame. This contact vector, represented as $<C_j^1,C_j^2,..,C_j^m>$, that a node $i$ builds for each of its neighbors $j$ after every contact, is the basic structure on which the path routing is built upon.

**5.3.3 Learning the frame length**

The accuracy and the speed of the prediction algorithm rests on large part in finding the right framemength. Intuitively, if we consider the contact pattern to be a periodic process, the frame is the time period of this periodic process.
The framelength for a pair of nodes or a system can be specified in 3 ways. (i) For systems for which the larger periodicity is evident, this can be hard coded into the prediction algorithm. For example, any biological process based network will have a natural day/night cycle and hence, a day should be an appropriate framelength. (ii) It can be specified based on off-line modeling. Systems where the framelength is not evident, but does not change too often, a human operator can analyze the contact patterns and specify it manual. (iii) It can be learned automatically using statistical modeling or machine learning techniques.

In this Section we define the automatic framelength learning problem, and provide some intuition about its learning. However, it is hard problem and we plan on investigating it in the future. The framelength learning problem can be stated as follows:

Given a set of $N$ contact points between a pair of node $i$ and $j$, where a contact point is defined by a tuple \( \text{time of contact, duration of contact} \), find the minimum time period, such that, the contact patterns repeats for every time period, with minimum error.

At the outset this is an optimization problem, in number of contact $N$ and the framelength $t_f$. It is important to note that a number of techniques based on autocorrelation, regression, and spectral analysis exist, to learn the periodicity of processes, that are uniformly sampled, and are either without any holes in the data or at least the location of the holes are known. But, in our case the contact patterns are neither uniformly sampled, could have missing contacts and possibly has large errors in contacts times. Hence, correlation based methods do not work well in this case.
5.4 Routing in semi-deterministic mobility networks

5.4.1 Problem definition

**Informal statement:** Design a routing protocol to optimize the overall cost of converge-cast, under delay and buffer constraints for a disruption tolerant network.

**Formal statement:** Given a set of N nodes, a special Base Station (BS) node, a buffer capacity of B at each node, set of past contacts for every node met $C_{ij}$, find a path from each node i, to the BS, such that, the total cost of the system is minimal and the total end-to-end delay of each path is less than a deadline D.

**Assumptions:**

1. Existence of a semi-deterministic mobility: Mobile nodes have a semi deterministic mobility, which means they travel certain routes or are at certain places often in a somewhat periodic manner.

2. Message Deadlines: Messages need to be delivered to destinations by a deadline. This deadline is specified by the application when the packet is handed over to the routing module.

5.4.2 Reliable Energy Aware Predictive Routing (REAPER)

Create a Routing Protocol for a Delay Tolerant Network which makes routing decisions based on Deadline of messages and takes into account Energy Efficiency. Nodes learn about their mobility using information from landmarks and their own trips and thereby create a Pressure function which signifies the Routing decision for the node. A node then passes on its packet based on that Pressure function which takes into account all the factors and then decides whether to pass the packet or not.
Also along with a deadline there is a failure deadline which is less than the packet deadline and it signifies the time before which a node needs to know if the packet can be delivered or not through pressure functions.

As was explained in Section 5.3 if a prediction algorithm can predict future contact times for pair of nodes, then it is possible to have a routing algorithm that uses that prediction to make a more intelligent decision on which node to forward the message to. Many DTN nodes have a high constraint on energy and hence we try to develop a routing algorithm that optimizes on cost which is the number of transmissions on the network. We also add delay constraints of packets as a metric to consider while optimizing on cost, such that the packet is delivered on the lowest cost path that satisfies a deadline.

In this section we discuss the routing logic and the routing metrics maintained by the protocol. A source node generates a packet and hands it over to the routing layer. The routing layer, after receiving the packet, adds a routing header, and enqueues it in its data buffer.

The logic of the routing consists of finding paths to the destination, its costs and its estimated delay. A number of paths might be available through each of the neighbor, within the next time frame, and all such paths and their corresponding metrics are maintained. When two nodes meet, each of them calculate the list of paths available to them to the destination, that meet a deadline $dl$, through all their neighbors except the node they are currently meeting and send it to the other node. This list of paths through each neighbor is stored in a path table which contains the fields as shown in Table 5.1. The nodes then search through their path table to find the best cost path that meets its deadline $dl$. 

115
<table>
<thead>
<tr>
<th>Neighbor ID</th>
<th>Path Vector (Contact-time</th>
<th>Delay</th>
<th>Cost)</th>
</tr>
</thead>
</table>

Table 5.1: Path table for node $i$

5.4.3 Algorithm

When a node $i$ meets node $j$, they go through the following steps:

1. **Update history and contact predictions**: In this step the contact history of the nodes are updated and the contact prediction algorithm explained in Section 5.3.2 is run to get the expected contact delays.

2. **Compute the available paths vector**: Next, we compute the set of paths available from a node to the destination, through all of its available neighbors (excluding the current contact). Through each of the neighbors, multiple paths could be available,

3. **Exchange available path information**: In this step the nodes exchange the paths with each other.

4. **Update the path table with new information**: In this step the node updates its routing table with the paths it received

5. **Compute the best path for each packet**: Then it computes the lowest cost path that satisfies the deadline.

6. **Forward packets**: It forwards the data packets meant for (current contact) node $j$
**Definition 2.** A path is a sequence of contacts between pairs of nodes, starting at node \( i \) and ending at the base station (BS). A path, say \( l \), starting at \( i \) is quantified, by contact time \( \text{Path}_{i,l}(ct) \) - the time at which the current node \( i \) is likely to meet the next node in the sequence, a delay \( \text{Path}_{i,l}(d) \) - the expected delay for a packet to reach the destination, after it has been handed over to the next node in the sequence (please note that, this delay does not include the waiting time at node \( i \)), a cost \( \text{Path}_{i,l}(c) \) - the cost of delivering a packet using the path from node \( i \) to BS.

Recollect that the contact prediction algorithm, for a node \( i \), for each node \( j \) it meets, predicts the number of contacts \( k \) in time frame \( t_f(i,j) \) and the maximum contact delays \( ct(j,k) \) with respect to a time frame \( t_f(i,j) \), based on the contact history between the two nodes. These delays are stored as vectors for each neighbor \( j \) in the contact time table.

```
1. for each neighbor \( j' \), where \( j' \neq j \)
2. \( \text{ttf}(i,j') := t \% t_f(i,j') \)
3. for each path \( \text{Path}_{j',l} \)
4. \( k_{min}(j',l) := k \) such that, \( \min((\text{Path}_{j',l}(ct) - ct(j',k)) \% t_f(j',k) \forall k) \)
5. \( \text{WD}_{min}(j',l) := (\text{Path}_{j',l}(ct) - ct(j',k_{min})) \% t_f(i,j') \)
6. \( \text{Pathvec}_{j',l}(ct) := ct(j',k_{min}) \% t_f(j') \)
7. \( \text{Pathvec}_{j',l}(d) := \text{Path}_{j',l}(d) + \text{WD}_{min}(j',l) \)
8. \( \text{Pathvec}_{j',l}(c) := \text{Path}_{j',l}(c) + \text{Cost}(i,j') \)
9. end for
10. % Comment: Check if end-to-end delivery delay is acceptable
11. for each path in \( \text{Pathvec}_{j',l} \)
12. if \( \text{Pathvec}_{j',l}(d) \leq dl \)
13. remove path from set \( \text{Pathvec} \)
14. end if
15. end for
16. end for
```

Figure 5.2: Pseudocode for calculating the path vector exchanged between nodes
Computing the available paths: Let $Pathvec$ represent the set of paths available to destination $d$ from node $i$, that meet the packet deadline $dl$. Let $Path_{j,l}$ represent the $l^{th}$ path through node $j$. Each path consist of an estimated contact time, delay to destination and the cost of the path and let $Path_{j,l}(ct)$, $Path_{j,l}(d)$ and $Path_{j,l}(c)$ represent them respectively. In this step we will compute the set of paths $Pathvec$ to the destination, from node $i$, excluding the paths through node $j$, which will be send to node $j$.

To calculate the $Pathvec$ to be send to node $j$ a node $i$ considers each path through each of the neighbors $j'$ (except the current neighbor $j$, to avoid cycles) and finds the contact between $i$ and $j'$ that minimizes the waiting time at $j'$ and adds it to the delay of that path. The cost and probability of delivery are updated appropriately. The subset of paths that do not meet the deadline are eliminated from $Pathvec$.

Calculating the least cost path: In this step, using the path vectors from all neighbors (including the current neighbor) a node calculates $m$ paths such that $rpd$ (the requested probability delivery) is satisfied. Given the current time $curTime$ at node $i$, and the deadline for the packets $dl$, for each neighbor $j$ and for each path $Path_{j,l}$ published by the neighbor, a end-to-end delay is calculated, by the estimating the two-hop handover delay (lets call this $THD(j,l)$), i.e., the handover delay at node $i$ to node $j$ and at node $j$ to the next contact in the path. We want to choose a contact between node $i$, $j$, from $ct(j,k)$ such that $THD(j,l)$ is minimized.
1. for each neighbor $j$
2. $ttf(i,j) := curTime \% tf(i,j')$
3. for each path $Path_{j',l}$
4. $k_{min}(j',l) := k$ such that, $min(\{ (P_{j,l}(ct) - ct(j,k)) \% tf_j \}\forall k)$
5. $THD_{min}(j,l) := \langle (Path_{j,l}(ct) - ct(j,k_{min})) \% tf_j \rangle$
6. $AvaPaths_{j',l}(ct) := ct(j,k_{min})$
7. $AvaPaths_{j',l}(d) := Path_{j,l}(d) + THD_{min}(j,l)$
8. $AvaPaths_{j',l}(c) := Path_{j,l}(c) + Cost(i,j)$
9. end for
10. % Comment: Check if end-to-end delivery delay is acceptable
11. for each path in $AvaPaths_{j,l}$
12. if $AvaPaths_{j,l}(d) > dl$
13. remove path from set $AvaPaths$
14. end if
15. end for
16. end for
17. % Comment: Find the path with the least cost, if multiple paths
18. % of same min cost are available.
19. % choose the minimum delay path
20. nextHop := $j$ such that $min \{ AvaPath_{j,l}(c), AvaPath_{j,l}(d) \}\forall j,l$

Figure 5.3: Pseudocode for calculating the best $m$ paths to satisfy a deadline of $dl$

\[
 k_{min}(j',l) := k, \text{ such that, } min(\{ (Path_{j,l}(ct) - ct(j,k)) mod tf_j \}\forall k)
\]
\[
 + \{ ct(j',k) - (curTime \% tf_j) \}\forall k \tag{5.1}
\]

\[
 THD_{min}(j,l) := \langle (Path_{j,l}(ct) - ct(j,k_{min})) mod tf_j \rangle
\]
\[
 + \{ ct(j,k_{min}) - (curTime \% tf_j) \}\rangle \tag{5.2}
\]

The end-to-end cost for each of these paths are also calculated. For example, if the cost function used a simple hop count, then the hop-count is incremented by one. Once the end-to-end metrics through the neighbors are calculated, paths that do not
meet the deadline $dl$ are eliminated. If this minimum cost path is through node $j$, then the packets are forwarded to node $j$.

5.4.4 Properties of the routing

![Diagram of information and packet flow among 3 nodes]

Figure 5.4: Schematic for information and packet flow among 3 nodes

**Theorem 1.** In a network where all pairs of nodes have the same frame-length $fl$, a path $P$ from a node $s$ to a destination $d$, of $h$ hops and expected delay $P_d$ can be discovered in time $[(h-1)fl] - P_d$

**Proof:** Before we give an outline of the proof, we state our assumptions clearly. We make three assumptions, (i) the framelength $fl$ between all pairs of nodes are the same (i.e., in other words the same periodicity). This is not an unreasonable assumption since all humans usually operate according to the day/night cycle. (ii) There is at
least one contact between neighbors in a frame period. (iii) The predictions in the intermediate nodes do not change for \((h - 1)fl\) time.

Now, let us consider 3 nodes \(L', L'', L'''\), where \(L' \& L''\) and \(L' \& L''\) are neighbors as shown in Figure 5.4. Let us assume without loss of generality that \(L'''\) gets the information about the path through \(L'\) from \(L''\) and choses to forward the packet to \(L'\). At the moment \(L''\) meets \(L'''\), it publishes the path information about \(L'\) which is less than \(fl\) old, let us call this the information flow delay. Again, \(L''\) after receiving the packet from \(L'''\) forwards the packet to \(L'\) at the next opportunity and the wait at \(L''\) is less than \(fl\), which we will call the packet flow delay. It turns out the sum of the information flow delay and the packet flow delay at every intermediate node \(L''\) is exactly \(fl\). Or in other words, the delay between the flow of information and the flow of packet, between two nodes, which communicate transitively through a third
node is $fl$. In an $h$-hop network, there are $h - 1$ intermediate nodes, hence the total delay between the information flow from the destination node up to the source node and the packet flow between source to destination is $(h - 1)fl$. Now, the total packet flow delay between the source and destination is what is estimated as the path delay $P_d$. Hence, the total time for the information flow from destination to source, which is the discovery time is given by $[(h - 1)fl - P_d]$. Figure 5.5 gives the visual intuition for the proof in a 3 hop case.

**Corollary 1.** Let all pairs of nodes in a network have the same frame length $fl$. Let there be a path $P$ from a node $s$ to a destination $d$ of $h$ hops. Let the number of contacts in a frame between the nodes in the path be $c_1, c_2, ..., c_h$. Then, any change in the path can be discovered by the node $s$ within delay $\frac{(h-1)fl}{\min(c_1, c_2, ..., c_h)}$.

**Corollary 2.** Let there be a path $P$ from a node $s$ to a destination $d$ of $h$ hops. Let the frame lengths between the nodes in the path be $fl_1, fl_2, ..., fl_h$. Let the number of contacts in a frame between the nodes in the path be $c_1, c_2, ..., c_h$. Then, any change in the path can be discovered by the node $s$ within delay $\frac{(h-1)\max(fl_1, fl_2, ..., fl_h)}{\min(c_1, c_2, ..., c_h)}$.

### 5.5 Validation

In this section, we focus on validating our prediction methodology along with our routing algorithm. We compare the performance of our protocol with other well known protocols using simulations in the ns-2 [26] simulator. We compare the protocols under two different synthetic mobility scenarios, one called the “campus network scenario”, which simulates a human DTN and the second “bus network scenario”, which as the name suggests simulates a DTN based on mobile buses inside a city. In the following subsection, we give details about the mobility scenarios.
5.5.1 Synthetic traces

Campus network scenario: The synthetic traces for campus scenarios are generated using the Time Variant Community Model of Wei-Jen Hsu et al. [59]. The model is based on communities and time periods. The model takes as input a transition probability between communities for each time period for each node. Also every node has an epoch length and an epoch time which signifies the stationary time of the nodes at the communities. By changing these parameters, one can model a number of different simulation scenarios from highly deterministic and periodic behavior to random mobility behaviors.

We model our campus scenario to approximate a large university or office campus. The campus scenario, as shown in Figure 5.6, consists of 18 communities in a simulation area of 1000 x 1000 meters and are classified into 10 home communities, 4 work communities, 2 food communities and 2 recreation communities. Every community is 100 x 100 meters. Five home communities are located at the north edge of the simulation area representing the ‘North Dorms’ and the remaining 5 are located at the south edge as ‘South Dorms’. The 4 work communities are distributed approximately towards the center of the simulation area. The recreation communities are located at east and west edge of the simulation area respectively. The 2 food communities are distributed randomly in between rest of the communities.

The time periods and the transition probabilities are designed in such a way, as to mimic the life style of a student in a campus environment. There are 7 time periods as shown in Table 5.2 and they occur in a day as follows:

- Time period 1(Travel-1) - 7:00 am to 9:00 am :- Nodes are given higher probability (approximately 0.5) of being in one of the recreation communities for
Table 5.2: Time periods for campus scenario

<table>
<thead>
<tr>
<th>Time period no.</th>
<th>No. of hours</th>
<th>Timing of occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (Travel-1)</td>
<td>2</td>
<td>7:00-9:00</td>
</tr>
<tr>
<td>2 (Work-1)</td>
<td>3</td>
<td>9:00-12:00</td>
</tr>
<tr>
<td>3 (Lunch)</td>
<td>1</td>
<td>12:00-13:00</td>
</tr>
<tr>
<td>4 (Work-2)</td>
<td>4</td>
<td>13:00-17:00</td>
</tr>
<tr>
<td>5 (Recreation)</td>
<td>2</td>
<td>17:00-19:00</td>
</tr>
<tr>
<td>6 (Travel-2)</td>
<td>2</td>
<td>19:00-21:00</td>
</tr>
<tr>
<td>7 (Rest)</td>
<td>10</td>
<td>21:00-24:00 and 0:00-7:00</td>
</tr>
</tbody>
</table>

Figure 5.6: Campus scenario
early morning physical activities, while the other nodes are at their home communities.

- **Time Period 2 (Work-1) - 9:00 am to 12:00 pm**: Nodes have higher probability (approximately 0.9) of being in one of the work communities. Every node has its own work community and it goes to that work community every day.

- **Time Period 3 (Lunch) - 12:00 pm to 1:00 pm**: Nodes pick any 1 of the 3 food communities, each with equal probabilities 0.33.

- **Time Period 4 (Work-2) - 1:00 am to 5:00 pm**: Nodes go back to their work communities with a high probability (approximately 0.9).

- **Time Period 5 (Recreation) - 5:00 pm to 7:00 pm**: Nodes are given higher probability (approximately 0.5) of being in one of the recreation communities for evening recreational activities, while the other nodes are at their work or home communities.

- **Time Period 6 (Travel-2) - 7:00 pm to 9:00 pm**: Some Nodes move between communities, while others are at their home communities.

- **Time Period 7 (Rest) - 9:00 pm to 7:00 am**: Nodes have higher probability (approximately 0.9) of being in one of the home communities. Every node has its own home community and it goes to that home community every day.

**Bus network scenario:** The bus network scenario consist of 10 buses running in the OSU campus environment of 1000 x 1000 simulation area. It is a deterministic mobility model where every node moves along a set path. Different nodes have different paths and different starting and ending times which are given in a parameter
file for each node. With the parameter files, a node starts moving every day at the specified start time and runs along its predefined deterministic path for the whole day at a constant speed till the ending time and this continues each day. The bus routes and their numbers are shown in figure 5.2.

![Bus network routes](image)

**Figure 5.7: Bus network routes**

### 5.5.2 Routing protocols compared

**Epidemic:** The Epidemic routing protocol [81] is a simple flooding protocol which replicates a copy of the packet to every node it encounters. In our implementation of Epidemic we have implemented acknowledgments of delivered packets as well, in order to reduce buffer usage. When a node A receives a beacon (new contact), from
another node B it sends a ‘Summary Vector’ packet of all the packet IDs in the buffer to node B. Node B then checks the packet IDs in the summary vector with the packet IDs in its ACK Buffer and sends a ‘ACK Summary Response’, intimating node A to remove those packets from its buffer, since they have been delivered. Then it sends a ‘Summary Response’ packet suggesting all the packets it does not have, that node A has. After node A receives the summary response packet, it then starts sending a copy of all the packet IDs in the summary response to node B. Similar transaction happens independently in the other direction and thus both nodes now contain copies of all the undelivered packets that they both have.

**Binary Spray and Wait:** The Spray and Wait protocol [80] is a enhancement of the epidemic protocol in the sense that it limits the number of copies per packet that a node can send in the network. In the spray phase nodes replicate packets to ‘L’ contacts it encounters and once all copies have been sprayed all nodes go to wait phase in which only a node that meets the destination delivers the packet. There are various flavors of Spray and Wait and we implement the most effective one called the Binary Spray and Wait in which every node replicates ‘L/2’ packets to the node it encounters until it has only one copy in which case it goes to wait phase to deliver it to destination. In this protocol the efficiency relies heavily on L which is the maximum limit on number of copies and we keep it 8, with the assumption that 8 copies is equal to one-third of nodes having a copy in a network of approximately 25 nodes which is enough to match the performance of complete flooding. In Spray and Wait also we implement the acknowledgment and summary vector mechanism as implemented in the Epidemic protocol and the packet exchanges are exactly similar to the above
epidemic protocol only with an additional cap on the number of allowed copies to flood in the network.

**MEED-DVR:** The Minimum Estimated Expected Delay Distance Vector Routing protocol (MEED-DVR) is a distance vector routing with edge cost as the average inter contact time between nodes termed as the ‘Minimum Estimated Expected Delay [MEED]’ . Every node calculates the average inter-contact time each time it encounters a node, and then sends its routing table to the other node. Then it compares it with its own routing table and updates its routing table according to the distance vector algorithm. It then hands over all the packets that are destined to the other node and then checks all the packets in its buffer and hands over the packets to the other node who is the next hop in its updated routing table entry for that packet’s destination. It is a Single-Copy routing algorithm and thus does not create or replicate any copies of a packet. Hence there exists only one instance of every packet in the network.

**PRoPHET:** Probabilistic Routing Protocol using a History of Encounters and Transitivity (PRoPHET) [71] is also a Single-Copy routing algorithm in which every node has a probability of meeting every other node and that probability changes with respect to three formulas. One formula is used to increase it when it encounters that node, second formula is used to age it gradually with time, and third formula is used to apply transitivity. These three formulas use 3 parameters Alpha, Beta, Gamma respectively which heavily control the performance of the protocol. A node first updates its probability table after encountering a node and then if the other node has a better probability than it own probability towards destination, then the node hands
over the packet to the other node. It also does not create or replicate any copies of a packet and hence there exists only one instance of every packet in the network.

5.5.3 Protocol metrics analyzed

**Protocol efficiency:** This is the most important metric in this study which captures all most the complete behavior of the protocol. It is defined as the total number of packets including data packets and control packets but not beacon packets sent on the network channel divided by total number of data packets received at the base station (Good-put) and the number of packets dropped. We add the packet drops to the denominator in order to penalize a protocol for dropping packets. For example, if we donot add this component, a protocol that drops all packets, except direct delivery will a 100% efficiency, but which is clearly not a desirable outcome.

\[
\text{Protocol efficiency} = \frac{\text{number of data pkts trans.} + \text{number of control pkts trans.}}{\text{pkts rcvd. at base station} + \text{number of pkts dropped}}
\]  

(5.3)

**Transmission cost per packet:** This is also an important metric in this study and is the main metric to optimize. Number of forwards is defined as the total number of times a packet is sent on the network channel and is averaged over total number of packets created in the network.

\[
\text{Cost per packet} = \frac{\text{number of pkt trans.}}{\text{number of pkts created}}
\]  

(5.4)

**Throughput:** Throughput shows the reliability of a protocol which proves how successful the protocol is and hence is defined as total number of packets received at the base station divided by total number of packets created in the network.
Throughput = \frac{\text{pkts rcvd. at base station}}{\text{pkts created}} \quad (5.5)

**Average delay:** Delay is a metric which is optimized in most of the earlier work done in routing in Disruption Tolerant Networks but in our case is a secondary metric with a constraint value named as 'Deadline' that needs to be met. Average Delay is defined as the Total End to End Delay which is the time duration between creation of packet and when packet is received at the base station for all the packets created in the network averaged over number of packets created for all nodes.

\[
\text{Average delay} = \frac{\sum_{i=1}^{n} \text{time packet}_i \text{was received} - \text{time packet}_i \text{was created}}{\text{number of pkts created}}
\quad (5.6)
\]

### 5.5.4 Results

In order to simulate the above mentioned routing protocols and our protocol, we use NS2 network simulator. We use the 802.11 Mac protocol for all our simulations with Two Ray Ground propagation model and Omni-Directional Antenna. The mobility of nodes is set by ns mobility traces files generated for the above mentioned synthetic traces and real life traces. There exists 1 base station and other mobile nodes who generate traffic to be delivered at the base station. At every node, CBR packets are generated with size of 500 bytes at a constant interval and we plot the above important routing metrics with change in interval. We also change the number of nodes in the network and try to analyze the protocols. We set a fixed buffer size of 2000 packets for every protocol. When a node meets it exchanges the routing control information first and then starts sending data packets in order until all the packets
are send or if the node goes out of communication range of the other node. Our protocol is referred as COPR in the validation graphs and results.

Synthetic human mobility trace results

![Graph of No. of Forwards per Packet vs. Traffic Rate]

Figure 5.8: Campus network: number of packet forwards vs. rate of traffic

As we can see from Figure 5.8, our protocol outperforms all the other protocols by a long margin in terms of cost which is number of forwards. Our protocol with deadline 12 achieves the least cost. Modified DVR comes close second, since it also is a single copy forwarding scheme and as it turns out in this case the shorter average delay path is almost close to being the lowest cost path in most cases. Spray and Wait and Epidemic being replication algorithms have more copies in the network, resulting in more number of forwards. Prophet, because of its average metric calculation, results in a high number of transmission thereby increasing cost.
Additionally because of better performance in cost, our protocol has the highest efficiency as can be seen in Figure 5.9. Modified DVR has a the second best efficiency because of the single packet based routing. The other protocols have a very low efficiency because of its high cost and higher protocol overhead.

Figure 5.10 shows us the throughput with respect to rate of traffic. As can be seen Epidemic and Spray and Wait have the best throughput since they use replication. We have also instrumented the algorithm with good buffer management to improve throughput and acknowledgment mechanisms to clean up redundant packets. The decrease in throughput for forwarding based protocols including ours, is due to the buffer overflow, as they await for the best node to forward. A surprising conclusion from our investigation is that, the buffer plays a even more vital role for the single copy protocols than for the replication protocols, since a single buffer drop in single
copy protocols means a drop in throughput. Whereas, for multi copy protocols a buffer drop means a failure probability of $1/k$ where $k$ is the number of replicas of the packet in the network. Thus with enhanced buffer management mechanisms we can improve and match the throughput of replication algorithms. Nonetheless it can be seen that our protocol with any deadlines has better throughput than the other single copy algorithms DVR and Prophet.

Figure 5.11 shows the average delay for packets with respect to rate of traffic. As with throughput, the delay of the replication algorithms is the best as expected. But it can be seen that our algorithm outperforms other single copy algorithms and is also very close in terms of delay as compared to the replication algorithms. One of the design goals for our protocol, is to compete with the delay of replication protocols, while at the same time retaining the cost efficiency of single packet routing protocols.
Figure 5.11: Campus network: average delay vs. rate of traffic

Figure 5.11 taken together with Figure 5.8 clearly shows that we have achieved our design goals.

Figures 5.12 and 5.15 show the variation of our protocol for different values of deadlines. As can be seen, with decrease in the deadline parameter, the protocol decreases in average delay. Also it increases in throughput with decrease in deadline. The protocol trades cost for a decreased delay and a consequence of increase in cost results in decrease in efficiency as the deadlines decrease.

Campus buses synthetic trace

The bus mobility is more deterministic than the campus scenario and hence the prediction algorithm will give much accurate predictions, thereby improving the performance of our protocol considerably as can be seen from Figures 5.16 to 5.19.
Figure 5.12: REAPER with different deadlines in campus network scenario: efficiency vs. rate of traffic

Figure 5.17 shows that the cost of our protocol is 3 times better than the cost of the replication algorithms. Also it is 2 times better than the cost of Prophet which uses a frequency based probability metric and is also better than Modified DVR by a larger margin than in the earlier case of semi-deterministic scenario.

Thus we have almost 3-4 times more efficiency than replication algorithm and 2 times more efficiency than Prophet and almost 25% more efficiency than Modified DVR. Thus our protocol outperforms all the other protocols in terms of cost and efficiency as it exploits the deterministic behavior of the network.

Figure 5.18 shows that Prophet achieves maximum throughput. But most interestingly our protocol matches and slightly outperforms the replication algorithms in terms of throughput. Also Modified DVR gives the least throughput because of its zero utilization of determinism in the network.
Moreover we also achieve better delay than the replication algorithms which are expected to have the least delay. We believe this is due to the large number of packet transmissions in replication protocols, which consume significant network bandwidth and delay at high traffic rates. This is a significant result and shows how prediction combined with single packet routing can significantly decrease the bandwidth usage of the network. Also it can be seen the MEED-DVR gives highest delay because of its average metric calculation and no prediction.

As can be seen from Figures 5.20 to 5.23 all the deadline variation of our protocol show the same trend. As the deadline is decreased, the average delay is reduced and hence the protocol can be refined to suit to the delay needs of the network.
Also decrease in deadline increases throughput as was the case with the earlier semi-deterministic scenario.

Moreover as expected for a lower delay and higher throughput, there is a slight compensation of cost to be paid and this can be seen that as deadline decreases, the number of forwards increase which consequently reduce efficiency. But there is not a vast difference in the various flavors of our protocol is terms of cost and efficiency compared to other protocols and hence accepting slightly higher cost for better delay requirements does not degenerate the performance of our protocol significantly. It actually still outperforms every other protocol in the deterministic scenario case.

To show the importance of buffer and storage constraints, we plot the same above Figures for the bus network but with a high buffer thereby assuming there exists no
storage constraints. Figures 5.24 to 5.27 show that if there are no buffer constraints in the network, our protocol achieves 100 percent throughput for all the different variations or deadlines while the other single copy routing algorithms eventually reduced on throughput due to network capacity. Thus it gives us an insight that our protocol outperforms other protocols in terms of network capacity.

Also with higher throughput it still is extremely low on cost and lower than all the other protocols which shows that it is the most cost efficient protocol and also has high efficiency than all the other protocols. Thus this gives us direction on incorporating the buffer mechanism policy which would make our protocol more efficient in any DTN scenario with any storage and delay constraints.
Figure 5.16: Bus network scenario: efficiency vs. rate of traffic

Figure 5.17: Bus network scenario: number of packet forwards vs. rate of traffic
Figure 5.18: Bus network scenario: throughput vs. rate of traffic

Figure 5.19: Bus network scenario: average delay vs rate of traffic
Thus, REAPER achieves much better efficiency than all other protocols, achieves comparable delay to replication protocols, at a much lower cost than all protocols. The protocol throughput drops a bit in the very high load regime. This is primarily due to buffer drops, which can be rectified by including buffer run out time as part of the packet deadlines in REAPER.
Figure 5.21: REAPER with different deadlines in bus network scenario: number of packet forwards vs. rate of traffic

Figure 5.22: REAPER with different deadlines in bus network scenario: throughput vs. rate of traffic
Figure 5.23: REAPER with different deadlines in bus network scenario: average delay vs. rate of traffic

Figure 5.24: Bus network scenario with high packet buffer: efficiency vs. rate of traffic
Figure 5.25: Bus network scenario with high packet buffer: number of packet forwards vs. rate of traffic

Figure 5.26: Bus network scenario with high packet buffer: throughput vs. rate of traffic
Figure 5.27: Bus network scenario with high packet buffer: average delay vs. rate of traffic
6.1 Introduction

Several edge networking testbeds and deployments have been realized during this decade, in part due to the recognition within the networking community that testbeds enable at-scale development and validation of next generation networks. The role of edge networkS—and edge networking testbeds—is likely to only increase, given the growth of wireless networks of sensors, vehicles, mobile communicators, and the like.

As the number of deployments increase, a federation of these WSN fabrics provide a number of interesting opportunities and possibilities. On one hand, federations should support access to diverse (and potentially provider-specific) wireless sensor resources and, on the other hand, it should enable users to uniformly task these resources. In this chapter, we focus our attention on the requirements and challenges in federating Wireless Sensor Networks (WSN) and propose the KanseiGenie federation architecture.

A federation of WSN fabric testbeds needs to address two core issues: One, an efficient and flexible method for resource description, discovery and reservation. And
two, a convenient, uniform way of tasking and utilizing federation resources. While standardizing resource descriptions would simplify addressing these two issues, we find that the diversity of sensor characteristics and the lack of a compelling standard model for describing wireless networks complicate the federation of WSN fabrics.

KanseiGenie is based on the position that, different WSN fabric aggregates can advertise resources based on different resource ontologies, unified by a Researcher Portal and users can obtain uniform experimentation support also from the Portal. Thus central to the KanseiGenie architecture is a Portal, which acts as the unifying point for the domain users.

6.1.1 Motivation and background

Federated WSN testbeds. Many WSN testbeds are in use today, of which Kansei [8], Orbit [15], NetEye [11], and PeopleNet [17] are but a few to name. Two broad groups of reasons motivate a WSN federation. (i) Recent trend in WSN experimentation, where experiments are being repeated in multiple testbeds to learn about (potentially substantial) variability of performance in different backgrounds, radio types, and size scales. Also, a number of experiments involve long running deployments—that often yield long lived sensing services—which in turn implies that testbeds are increasingly hosting concurrent experiments. (ii) Sensor networks are increasingly being deployed in urban areas. An enterprise user typically requires and has access to more than one sensor fabric, in order to accomplish the task (we provide examples of such tasks in Chapter 7). Also, a single WSN fabric is also being shared by a number of users. These trends motivate the emergent requirement that sensor fabrics be able to collaborate to form federations of programmable fabrics.
The Global Environment for Network Innovation project [5] concretely illustrates an architecture where WSN fabrics are a key component. GENI is a next-generation experimental network research infrastructure currently in its development phase. It includes support for control and programming of resources that span facilities with next-generation fiber optics and switches, high-speed routers, city-wide experimental urban radio networks, high-end computational clusters, and sensor grids. It intends to support large numbers of users and large and simultaneous experiments with extensive instrumentation designed to make it easy to collect, analyze, and share real measurements and to test load conditions that match those of current or projected Internet usage.

Figure 6.1 depicts the GENI architecture from a usage perspective. In a nutshell, GENI consists of three entities: Researchers, Clearinghouses and Sites (aka resource
aggregates). The Clearinghouse keeps track of the authenticated users, resource aggregates, slices, and reservations in the federation. A Researcher (interacting typically via a specially designed Portal) queries a Clearinghouse for the set of available resources at one or more Sites and requests reservations for those resources that she requires. To run an experiment, she configures the resources allocated to her slice, which is a virtual container for the reserved resource, and controls her slice through well-defined interfaces.

The rest of the chapter is organized as follows: In Section 6.2, we detail the requirements of each actor in a WSN federation. In Section 6.3, we discuss the need for Resource and Experiment Specification in federations and their design challenges. Then, we present the KanseiGenie architecture and its implementation in Section 6.4.

6.2 Requirements of federated WSN fabrics

As explained in Section 6.1, the federated WSN fabric model distinguishes three actors: the Site that owns WSN aggregate resources, the Researcher who deploys applications via a Portal, and the Clearinghouse (CH) that enables discovery, management, and allocation of resource. In this section, we analyze actor-specific requirements, to make user experimentation easy, repeatable, and verifiable.

6.2.1 Clearinghouse requirements

Broadly speaking, a Clearinghouse has two functions: One, identification and authentication of various actors in the system (the details of which are out of the scope of this chapter and hence will not be discussed here); And two, resource discovery and allocation using a resource description language. The resource description language should be feature rich and extensible to capture the underlying heterogeneity of WSN
fabrics and their constraints. A CH design should be efficient and robust to manage multiple site resource specifications, large number of resources and their lease states. In a global infrastructure such as GENI, a CH might have work in both master-slave and peer-to-peer modes with others CHs. A Portal/Researcher might request resources from multiple CHs directly or a single one which in turn communicates with other CHs.

6.2.2 Site requirements

To support federated experimentation, a Site needs to support sliceability, programmability, experimentation services for the resources it controls. In the GENI model of experimentation, each Researcher owns a virtual container, aka a Slice, to which resources can be added or removed, and experiments deployed. It follows that a Researcher should have secure intra-slice communication that is isolated from other slices. To enable multiple slices to co-exist on the same fabric, sliceability may require Sites to virtualize resources both at the node and network level. For example, memory, processing time, or communication channels on the same device/network may have to be shared between slices. The challenge in virtualization is to provide as much control to the users (as low in the network stack as is possible) while retaining the ability to share and safely recover the resource. Virtualization in WSN fabrics is nontrivial, sometimes even impossible, given the limited resource on sensor nodes and the interference caused by concurrent wireless communications. Sites are also expected to provide programming services that reliably deploys applications composed by Researchers on the WSN fabric, testbed status and experiment execution, monitoring, logging, trace data injection, and workflow control services.
6.2.3 Portal requirements

A Portal in our architecture is a way to provide the user with a standard set of tools that can be used to exploit the federated resources. We believe that the Portal has an important role in networking sub-domains like WSN, where a high level of domain knowledge is needed to exploit the resources efficiently. A Portal needs to provide an uniform resource utilization framework. The framework should also allow the Researcher to select one or more of the standard User Services provided by the Site/Portal to be instrumented on the slice. This is challenging since the federation may consist of fabrics with a great variety of available platforms, sensors, radios, operating systems and libraries. For instance, while some sensor platforms such as mica family and TelosB are programmed on bare metal, others such as iMote2, Sunspots, and Stargates host their own operating system. The execution environments in these platforms vary from a simple file download and flash reprogramming, to command line interfaces and virtual machines.

6.3 Specification languages

To use WSN fabrics, a Researcher queries the CH to discover what resources are available, selects a set of resources and obtains a lease for them, and configures the resources and User Services needed for the experiment. Each of these steps needs a flexible, feature-rich, and extensible language to convey the goals of the Researcher to the system. We refer to the language used to publish, query, request, and allocate resources as the RSpec language, and the language used to configure resources and script workflow as the ESpec language.
6.3.1 Resource ontologies for resource description

Resources available at multiple sites are usually different from each other. Although describing resources in terms of a taxonomical flat-schema could hide minor differences by shoe-horning similar resources into the same category, building an exhaustive schema for WSNs, which will be conformed to by every Site, Clearinghouse, and Portal, is both difficult and sometimes impossible. The current GENI proposal mitigates this problem somewhat with a partial standardization approach, a uniform core RSpec and multiple domain-specific extensions. Although this solution in principle is better than a single standardized RSpec, agreement on the extensions is difficult in domains like WSNs. Roscoe in lays out the drawbacks of this approach. A key challenge in using a standardized specification is anticipating in advance all of the possible things that must be expressible. He argues that the resource request instead should be viewed as a *constraint satisfaction problem* and not a simple database query. For example, in a wireless network, while a node and channel might be two separate resources, it is usually impossible to allocate just the node but not the associated channel, or vice versa. It is better to explicitly capture such dependencies and relationships in the RSpec which would lead to a better resource allocation.

Thus a RSpec language should be able to specify constraints and handle multiple decentralized specifications. One way of doing this is to represent resources in terms of ontologies, using a resource description language such as NDL, which could describe not just the resources but also the constraints. With this approach, in order to communicate, two entities need only to agree on the language in which resources are specified, but not on the ontology.
Using multiple resource ontologies for WSN resources.

Our primary motivation to support multiple ontologies in WSN federation is the difficulty in specifying things uniquely. We offer two example cases. The first pertains to definitions, or rather the lack of it in the WSN space. There is no agreements on even simple terms like nodes and sensors. Moreover, defining something as either an allocatable entity or as its property could differ significantly between sites, depending on what platforms they provide. For example, is a channel an allocatable resource or simply a property of a radio?

Our second example deals with the difficulty of specifying wireless network topologies. A network topology in a WSN can be specified either implicitly or explicitly. An example of implicit specification would be specifying the transmit power level of each node. An example of explicit specification would be specifying the number and/or the list of neighbors for each node and letting the fabric choose the transmit power level and other radio parameters. A Site provider could choose one or other while defining the Site ontology, depending on the hardware platform. One cannot force a Site provider to adopt an ontology which is not compatible with the hardware.

A question that directly follows from our argument for multiple ontologies is, why not simply use a union of them? This approach could result in RSpecs that cannot be instantiated by the Sites. For example, consider the two ways of (as discussed above) specifying a network topology; Even if a hardware could support both specifications, a single RSpec, which uses both implicit and explicit specification cannot be instantiated by the site. Thus the union approach could lead to the risk of producing inconsistent RSpecs.
Given the intense debate in the WSN community, forcing Sites to use a single ontology is likely to throttle innovation and will result in a needlessly bulky ontology that is not easily extensible. Since most Researchers are expected to interact only through a Portal (or two) of their choice, we envision that Portals will serve as the unifying agent for resource specifications. Since all Sites will use the same language for their descriptions, the Portal can map the different ontologies used by the Sites to a single ontology to be used by Researchers. We note that there are several extant techniques and tools to map and align ontologies [18, 75].

6.3.2 Experiment specification and work flow control

One of the roles of a Portal is to enable uniform experimentation across all federation fabrics. One way of satisfying this requirement is by providing an Experiment Specification (ESpec) language that enables Researchers to configure slices in a generic manner. Intuitively, besides the resource that the experiment is to be run on, an ESpec should also include the selection of User Services that is required by the experiment. WSN applications typically run in multiple well defined phases, with each phase involving a possibly different configuration. In addition to declarative elements, the experiment specification language also includes procedural descriptions (or work-flow elements). This enables iterative experimentation, where a researcher programs repeated experiments, and the configuration of each experiment depends on the outcome of the previous ones. ESpec thus provides Researchers with a flexible and feature-rich way of interacting with resources, rather than just a GUI or a command line interface. They become particularly relevant for future scenarios where applications will primarily involve machine-to-machine, as opposed to human-to-machine,
interaction. The idea then is to standardize the ESpec language and not the format of interaction.

6.4 KanseiGenie

KanseiGenie is a refactoring of the Kansei testbed, to support a GENI compatible federation of geographically separated Sites, each hosting one or more WSN fabric arrays. Each sensor device is represented as a Component that defines a uniform set of interfaces for managing that device. An Aggregate contains a set of Components of the same type and provides control over the set. (In WSN experiments, Researchers normally interact with fabric arrays through the aggregate interface rather than individual component interfaces). An Aggregate also provides other internal APIs needed for inter-component interactions.

6.4.1 Architecture and implementation

In keeping with the GENI architecture, KanseiGenie consists of actors for a Site, a Clearinghouse, and a Portal. The current implementation of federation consists of a Site at The Ohio State University, which has four different sensor fabric arrays, and a Site at Wayne State University, which has two different sensor fabric arrays. The Sites and the Portal (which is hosted at Ohio State) run the KanseiGenie software developed at Ohio State. One of the Clearinghouse functions, namely resource management, is implemented using ORCA [13].
KanseiGenie Site.

A KanseiGenie Site has four components: Aggregate of Aggregate Manager (AAM), the Web Service Layer (WSL), the individual Component Managers (CM) for each device type, and the Orca Site Authority module.

**Fabric Manager.** Given that each fabric array is an aggregate, the KanseiGenie Site Authority (SA) is conceptually an Aggregate of Aggregate Managers that provides access to all the arrays. AAM is responsible for implementing the fabric APIs. AAM provides an AM interface for each sensor array through parameterization. Externally, AAM (i) administers usage of the resource provided by the Site according to local resource management policies, (ii) provides the interface through which the SA advertises its shared resource to one or more authenticated CHs and, (iii) provides a programming interface through which Researcher (via the Portal) can schedule, configure, deploy, monitor and analyze their experiments. Internally, the AAM provides mechanisms for inter-aggregate communications and coordination. The fabric APIs provided by an AM are organized into the four functional planes, namely, Resource/Slice management APIs, Experimentation APIs, Operation & management APIs, and Instrumentation & Measurement APIs.

**Web Service Layer.** WSL provides a wrapper for AAM and acts as a single-point external interface for the KanseiGenie SA. The WSL layer provides a programmatic, standards-based interface to the AAM. We utilize the Enterprise Java Bean framework to wrap the four functional GENI planes.

**Component Directors.** Each sensor device in KanseiGenie has its own Manager (although for some primitive devices such as motes the Manager is itself implemented on other more capable devices). The Component Manager implements the same APIs
as that of AAM and is responsible for executing the APIs on the individual devices. Currently KanseiGenie supports Linux-based PCs/Laptops (Redhat and Ubuntu), Stargates, TelosB, and XSMs. CMs for Imote2 and SunSpots are under development.

![KanseiGenie architecture](image)

**Figure 6.2: KanseiGenie architecture**

**KanseiGenie Portal.**

The Portal contains a suite of tools for the life cycle of an experiment. It provides an easy interface for experiment specification; at present, this is a user-friendly GUI; a user programmable interface is under development. It automates tasks for resource specification creation, requesting, and subsequent deploying of the experiment, for all Sites in the federation. It uses the Orca Slice Manager (explained below), to reserve resources requested by the Researcher. Once the reservation is done, it interacts
with the AAM web services interface to configure and run experiments. Of course, a Researcher could directly program against the AAM web interfaces to gain more fine-grained control of experiments, i.e., write his own portal as need be. The Portal is implemented using the PHP programming language.

KanseiGenie Clearinghouse.

CH has two main functions; (i) Resource management: CHs are responsible for managing the resources on behalf of Sites. They keep track of resources and their leases. We use ORCA \[13\] (see below for more details) Resource Broker to implement our CH. (ii) Identity, Authentication and Trust: CH is also responsible for maintaining the overall security of the system. They authenticate/issue credentials for Sites, Portals and Researchers. In a federation, CHs implement trust-chaining to authenticate Researchers and Brokers from other domains. KanseiGenie, consistent with the GENI/ORCA effort, plans to use Shibboleth \[21\] for this purpose.

ORCA-based resource management system.

ORCA consists of 3 sub-entities, each one correspondingly embedded into the three KanseiGenie actors (Portal, Site and Clearinghouse) respectively, to implement the resource management function. (i) The ORCA Slice Manager interacts with the Researcher and gets the resource request, forwards it to the ORCA Broker and gets the lease for the resources. Once a lease is received, the Slice Manager, forwards it to the Site Authority to redeem the lease. (ii) The ORCA Site Authority keeps inventory of all the resources that need to be managed. It delegates these resources to one or more Brokers, which in turn lease the resources to Researchers through the Slice Manager. (iii) The ORCA Broker keeps track of the resources delegated by
various Site Authorities. It leases resources (if free) to the ORCA Slice Manager on request. A number of different allocation policies can be implemented using a policy plug-in.

6.4.2 Portal-based federation in KanseiGenie

![Diagram](image)

Figure 6.3: RSpec mapping, translation and lease generation

Apart from being the single point of access for a Researcher, the Portal plays an important role in KanseiGenie federation architecture. The Portal has three important functions in federation, namely Resource Specification Mapping, Experiment Specification Mapping, and Federated Slice Stitching. KanseiGenie has thus far chosen the Portal as the main federating agent in the system. This design suits the view that a Portal realizes *application domain specific support*, and that for different domains, different Portals may suit. In other words, we view the KanseiGenie Portal as suitable for WSN experiments (and perhaps only some classes of WSN experiments) as opposed to sufficing for all GENI-Researcher needs.
In this view, Clearinghouses are treated as being generic rather than domain specific. Roles which are less domain specific, e.g. embedding RSpecs or slice stitching, can be moved from Portals to CHs (or even to Sites) assuming the method of stitching desired is communicated to them. Now, should CHs evolve to become domain specific, they may import more roles from Portals. Taken to the extreme, this would suggest that a top level CH be directly or indirectly capable of unifying all resources in GENI.

**Resource specification mapping.**

As explained in Section 6.3 our position is that Sites may use their own Resource Specification dialects (ontologies). To provide a unified experience to the Researchers, we put the onus of interpreting multiple ontologies on the Portal. The Portal discovers resources from multiple sites, understands the Resource Specification ontologies and remaps the different ontologies into a unified ontology at the Portal. The current research [63,75] suggests that this remapping of ontologies can be done automatically with very high probabilities for related ontologies. However, we do the mapping manually since it is a one-time operation per site. After a resource request is created (in the Portal ontology), it first gets *translated* into the individual Site’s ontology and then sent to the CH to obtain the lease.

In the current implementation, the Portal is also responsible for RSpec *embedding*), i.e. the embedding of an abstract RSpec into a concrete RSpec. It is often convenient for a Researcher to request networked resources in an abstract manner. For instance, requesting a 5-by-5 connected grid with 90% link delivery radio is much easier than pinpointing specific sensor devices that match the required topology. Since the resources published at the CHs are specified concretely, a Portal needs to convert
or embed—may be with the help of the Site—the abstract RSpec into an concrete RSpec. The RSpec generation, translation, embedding, ticketing, and lease generation process is shown in Figure 6.3.

**Experiment specification mapping.**

Researchers use the ESpec language to script an ESpec which is fabric/platform neutral. The Portal maps the ESpec created onto Site/fabric specific experiment manager APIs. Thus, a Researcher may “stitch fabric\(_1\) slice and fabric\(_2\) slice”, “inject data on fabric\(_1\) slice from file\(_1\)” , etc., without worrying about the details of fabric\(_1\) slice and fabric\(_2\) slice; The Researchers may likewise repeat the same experiment on different fabric slices easily. In KanseiGenie, we attempt to use the same APIs for different aggregates whenever possible; nevertheless, whenever the notion of a service (say logging on a Virtual Machine fabric versus a Mote fabric) is different between fabrics, the complexity of mapping the configuration onto the corresponding APIs is handled by the Portal.

**Federated slice stitching.**

When conducting a federated experiment, a Researcher expects seamless communication between the resources in the federated slice, which means that sub-slices from different Sites needs to be stitched together to form a federated slice. While individual Sites provide the stitching services as part of the AAM, the Portal possesses the knowledge for implementing stitching (such as VLAN numbers, IP addresses, ports, web URLs, etc.). Note that multiple types of stitching might be needed (and possible) depending on the sensing fabrics involved and their capabilities, e.g., it is easy to stitch a federated slice consisting exclusively of virtual machines connected by wired
virtual LANs, while it is much harder to stitch two wireless network slices to create a single federated wireless slices.
CHAPTER 7

WEAVE: STANDARDIZATION OF INTERFACES FOR INTEROPERABILITY ACROSS SECURITY-FABRICS

7.1 Introduction and background

The domains of urban sensing applications are many and varied, ranging from security to entertainment. Unlike scientific applications where an “optimal” sensor network may be designed and then deployed, the urban environment consists of loosely-coupled (or even un-coupled) sensor fabrics that are already “out there”. We characterize the urban sensing problem as one, where, in response to a specific requirement (such as tracking and subsequent capture of an assailant), an application needs to be developed, for which (a) the operations of varied, independent sensor networks are integrated and (b) the information from these networks suitably retrieved, analyzed and fused to meet the application, either while an event is in progress or post facto after the event has taken place. Also, the exact sequence or pattern of events may not be known a priori, to completely automate a client application. In other words, often a human may (or even, must) exist in the loop to interpret outputs from a given fabric and involve follow up fabric actions based on the outputs. \(^1\)

\(^1\)This human intervention and inferencing requirement is often due to privacy regulations that require that automated inferencing, such as by using data-mining techniques, may not be made without legal approval of probable cause.
In this chapter, we discuss issues relevant to the architecture of urban sensing applications, and present key elements of an architecture, which we call WEAVE. In this architecture, we view individual sensor networks as programmable fabrics that can be leased by multiple high-level applications.

This chapter is organized as follows: In Section 7.2, we describe our concept of operations for example urban sensing applications and use them to motivate architectural requirements. In Section 7.3, we outline the WEAVE architecture. We then, in Section 7.4, describe the standard application programming interfaces (API) for a class of search applications. Next, in Section 7.5, we present example search applications that are composed using the Weave APIs. The applications are chosen from operation scenarios that we have implemented at the Ohio State University (OSU): The first is related to an urban surveillance scenario where multiple sensor fabrics are used to detect and track suspicious persons entering a building. The second and third are location service applications for objects or people of interest within a building in the same fabric or across multiple sensing fabrics. In Section 7.6, we present related work.

7.2 Example application scenarios

To motivate our characterization of urban sensing and identify requirements for our architecture, let us consider concepts of operations and specific application scenarios in two sub-domains of urban sensing: campus surveillance and social networking.

Campus surveillance conops: In collaboration with domain specialists at the Air Force Research Laboratory, the Homeland Security Program at OSU and the Ohio
Association of Chiefs of Police, we identified the following representative scenarios for this domain.

**Scenario:** Suspicious persons need to be tracked across a campus, including through buildings. Monitoring outdoors is via a network of building-mounted video cameras and indoors is via motion sensor and camera network fabrics deployed inside buildings. To support low power operation and mitigate concerns about citizen privacy, the indoor network should not be activated for all moving persons, but be cued instead to operate when the outdoor video camera network tags some activity as suspicious and its tracking of people suggests that they are entering a building.

Figure 7.1(a) shows the view from one such camera installed outside the CSE building at OSU. Figure 7.1(b) shows a motion triggered camera that is part of the indoor camera fabric.

![Figure 7.1: Units in camera fabrics](image1.png)

(a) View from building camera  
(b) Indoor camera
Social networking conops: Motivated by scenarios where users have expressed interest in locating friends and finding out about objects of interest within a locality, we have implemented PeopleNet, a network of mobile stations, as well as multiple static sensor networks at OSU.

Each mobile unit in a PeopleNet fabric consists of a cell phone connected to a (Intel-developed, 802.15.4 radio equipped) psi-mote. Mobile units connect to a central access point in the building that they are in, as well as a set of infrastructure nodes that aid in localizing the mobile units. We are presently deploying an instance of PeopleNet within the CSE building and another instance is planned for the Recreation and Physical Activity Center (RPAC) that is a few blocks away.

Figure 7.2 shows each mobile unit of the PeopleNet fabric that consists of a cell phone integrated with a 802.15.4 equipped psi-mote. Figure 7.3 shows the deployed infrastructure nodes corresponding to PeopleNet and sensors belonging to the indoor PIR fabric.

![Figure 7.2: A mobile unit in PeopleNet](image)

We have also designed and implemented 3 fabrics of static PIR sensors, one deployed within the CSE building, a second one to be deployed in the RPAC to monitor
availability of squash courts, and a third one to be deployed at the nearby Oxley’s Cafe that monitors the queue at the cafe. There is also an Elevator sensing fabric that monitors the position and status of elevators inside the CSE building.

![Figure 7.3: Layout of CSE building with PIR fabric and PeopleNet fabric](image)

**Scenarios:** Representative application scenarios in this case consist of users issuing ad-hoc locality specific queries such as “Is Mukundan already at the RPAC?”, “Is any squash court in RPAC empty?”, “Where is the elevator?”, or “How long is the queue at Oxley?” via their PeopleNet mobile devices. Resolving these queries would typically require routing the query to the central point in the building where the user is and somehow accessing the desired data from the sensor fabric in question.

Common to these conops are the following architecture-relevant elements:

- **No “best” sensor:** Different sensor fabrics may suit for different environments (e.g., indoor and outdoor) even if the objects to be sensed are identical. The right detectors to deploy may not be known until a high-probability threat has been identified or an event occurs. Following the threat or attack and as part of the response, sensor fabrics may be re-targeted and re-purposed accordingly.
“Dual-use” fabrics: Information and intelligence may be needed at multiple layers of the safety response system, i.e., to local, state and federal law and safety enforcement, each investigating situations with different objects and at different scales. In other words, fabrics should have the ability to support multiple clients operating independently.

Resource management: The architecture must support the different fabrics to cope with their resource constraints, while still supporting their ability to federate.

Human-in-the-loop: A primary mode of sensor fusion is human-driven, i.e., sensor streams are typically fed back to a command console and cross-stream inferences made by humans. Thus, support is desirable for letting authorized personnel readily access and search data sensed by the fabric, and conversely, for letting people add human “sensor” information such as “beat cops” reporting on suspicious activity, informants providing tips, etc. to databases that can be fused with data automatically collected via fabrics.

7.3 The WEAVE architecture

Weave has three architectural elements: Sensing fabric, Object and Client.

A fabric offers its services to clients via an application programming interface (API). Some fabric services may be generic, others may be tailored to the application domain(s) that the fabrics support. We refer to the latter set as vertical services. Standardizing vertical services is desirable, so that applications can be readily composed and ported across fabrics geared to support a particular application domain.
In general, a fabric need not make guarantees about its quality of service, delivering its results based only on a “best-effort”.

Later, in Section 7.4, given the context of the “search” application scenarios described in Section 7.2 we detail a proposed standard API for the services offered by sensing fabrics that support search.

Figure 7.4: WEAVE architecture

Object: A fabric enables measurement of physical phenomena to detect and classify physical objects such as humans, vehicles, weapons and explosives. Each type of object is characterized as a predicate on physical phenomena; all objects that satisfy the corresponding predicate belong to an object type. Objects may be identified (or associated) by the fabric or remain anonymous (or non-associated). Some objects may be known a priori to the fabric (for example, “blue force” objects that advertise their presence to the network or permanent elements in the environment such as “Elevator 2” or “Squash court B”). Other objects are not known a priori, but are detected and
then associated by the fabric. Objects have type-specific attributes, such as location
and velocity, associated with them.

Associated with each object type is a detector. Detectors may be implemented by
the fabric designer or by clients. In order to load and run a detector on the fabric, a
client needs to access some fabric services (such as the Instantiate service, which is
discussed in Section 7.4).

Client: A client is an application that can use the services provided by one or more
sensing fabrics. We assume a client can access a fabric, but how it does so (i.e. the
medium for connectivity) is outside the scope of the architecture.

We assume that the client knows the services offered by a fabric through a di-
rectory maintained by an access manager. How the client discovers or looks up this
information is outside the scope of this architecture. An access manager may manage
multiple fabrics if they belong to the same administrative domain. The access man-
ger maintains a directory of object types supported by fabrics and their semantics.
The access manager also grants access permissions to a client for using a fabric.

A client may choose to trust the fabric, or it may not (preferring instead to validate
the fabric-provided information by cross-referencing it with information provided by
other fabrics or sources). The access rights of a client may vary from fabric to fabric.
Moreover, if a client has the rights to instantiate new detector capabilities within a
fabric, it may choose to make these shareable with other clients. We do not elaborate
on how security and trust between clients and fabrics is realized.

To begin use of a fabric, a client registers with the fabric. It may then choose to
instantiate detectors on the fabric. It would then invoke fabric services: In the case of
search services, the client may selectively search for particular object types, specific
objects, or objects in certain segments of the fabric. Operation invocations could be one-time, time-bounded or persistent. A client may therefore terminate time-bounded and persistent invocations. Figure 7.4 shows the WEAVE architecture.

7.4 Vertical API for search

In this section, we describe the standard APIs provided by urban sensing fabrics that support application scenarios stated in Section 7.2. These fabrics support two generic APIs Register and Instantiate, and a vertical API Search specific to the “search” application domain. We propose the following specification for Register, Instantiate and Search APIs.

Register: To use a sensing fabric, a client first registers with the fabric and is provided a network handle which is then used in subsequent invocations of the services offered by the fabric. The register service is a generic fabric service that handles the security and authentication aspects for the fabric.

FabricSession Register(Fabric F, ClientInfo C)

The Fabric data structure contains the name of the fabric being accessed. The ClientInfo data structure contains credentials, such as id and password, assigned to the client by the fabric administrator. A FabricSession is returned to the client. This may contain an expiration date and will encapsulate access rights for the client on the fabric.

Instantiate: The Instantiate service is a generic fabric service that provides the client the capability to program the fabric with its own detector. Instantiate is invoked with a Fabric Session and the Detector to be downloaded to the fabric. Note that in order to create a Detector, the user has to be aware of the internal API of the fabric.
Boolean Instantiate(FabricSession, Detector, Shareable)

A detector that is implemented by a client may be made available to another client through the Shareable parameter.

**Search:** Once a client has registered, it can use Search in order to find objects in the fabric.

SearchSession Search(FabricSession, ObjectType, Object, Locations, Parameters, Persistence, Duration, Periodicity, SearchListener)

The return value of *Search* specifies whether the call is a success or a failure. If the call fails, SearchHandle is NULL. The result of the search is returned through a separate call on the SearchHandle to map to the actual Search call. The result of a search contains the status of all objects that match the search.

We now discuss the parameters of the Search API.

- **ObjectType:** Object type for which the Search is invoked. The fabric uses this to invoke the appropriate detector.
- **Object:** If the client is interested in Searching for a specific object of the type, this can be specified by providing the Object (which must have been returned as part of a prior Search if the object is non-cooperative). If the object is one of the cooperative types, then the id is known a priori to the client.
- **Locations:** If the fabric allows selective execution of the detector on a section of the network, the Locations field specifies those sections
- **Parameters:** These are the Parameters exposed by the fabric for a particular object type (i.e. the corresponding Detector).
Persistence: The persistency parameter specifies whether the search service has been invoked as a one shot or a persistent operation.

Duration: The duration parameter is ignored if Persistence is FALSE. If the search is persistent, Duration specifies the interval over which objects that match the Search should be returned. The duration of a search can be DURATIONPAST, which means that the Search is for objects detected in the past. Note that the fabric may not implement any history capability, in which case, Search will fail if called with DURATIONPAST.

Periodicity: A persistent Search invocation corresponds to tracking the status of an object. The semantics of the persistent Search could be either that a new result is returned every time the status of an object changes or it could be that the status of an object is reported periodically at the interval specified by Periodicity.

SearchListener: The SearchListener specifies the name of the listener function initiated by the Client which the Search service can call back to return the results of the search.

We do not require that a fabric implements all possible parameters. However, the fabric must return a “failed” result if the parameters supplied are not supported.

The status of an object contains the current attributes associated with the object. One example of an object attribute is location. There can also be other attributes associated with objects. For example, if a fabric implements a detector for object type “Elevator”, the attributes of an object could be location and the direction of motion.

Search can be thus invoked in the following modes of operation in combination:
**Find:** One may invoke Search to simply find out if an object exists.

**Search temporally:** Objects could be searched for a period of time that spans the future or the past. Thus Search can be used for tracking an object or a set of objects.

**Search spatially:** Search may be invoked to only look for objects at a specified location or all the locations.

As part of implementation of the sensing fabric, *Search* may internally be supported by services for ‘association’ and ‘power management’. By an association service, we mean identifying new objects of a certain type, assigning an identifier to them and associating subsequent detections of an object with previously existing identifiers in the fabric. Power management services may be used to selectively activate or deactivate detectors within the sensing fabric. For example, a detector may be activated only when the first search for that object type has been invoked. Similarly, if a search is invoked to track a particular object, power management services can selectively activate the detectors in the vicinity of the current location of the object.

**Terminate:** Corresponding to each of the Register, Search and Instantiate operations is a destructor Terminate. A persistent Search that has been invoked or a detector that has been instantiated or a new network handle can be revoked using the Terminate API. Terminating an active detector removes all Search operations invoked on that object type.
7.5 Illustrating search applications composed using WEAVE

In this section, we describe Search applications related to the urban surveillance and social networking scenarios of Section 7.2 that we have composed based on the WEAVE architecture.

7.5.1 Urban surveillance scenario

We monitor suspicious activity using our building-mounted video camera (outdoor) fabric and camera & motion sensor (indoor) fabric. The outdoor video camera fabric is instantiated with detectors to detect unusual events, such as people getting out of illegally stopped cars, atypical individual motion, crowd formation, or repeatedly circulating vehicles. Any person(s) tagged as suspicious and headed towards a building lead to cueing of the indoor sensor fabrics. The following example showcases tracking of people getting out of an illegally stopped car.

The client registers itself to the building camera fabric.

FabricSession_m = Register(OutdoorCameraFabric, ClientInfo)

The client is assumed to be aware of the object types detected by the fabric. In this case, it invokes Search on object type StoppedCar.

Result_m = Search(FabricSession_m, StoppedCar, NULL, LOCATIONALL, Parameters, PERSISTENT, DURATIONFOREVER, 0, SearchListener)

ObjectType refers to the StoppedCar detector, activated in the entire fabric, while Parameters could be settings (such as semantics for a stopped car) exposed by the underlying detector. Search will return every new instance of a stopped car with a
different id. *Periodicity* is set to 0 to indicate that a result should be returned only when a new stopped vehicle is detected.

(a) Person getting out of stopped car, tagged as suspect

(b) Picture of person tagged as suspicious, entering CSE building

Figure 7.5: Campus surveillance scenario

The client also registers itself with the indoor photo camera fabric. Upon detection of a suspicious stopped car, it issues a *Search* to the indoor motion triggered photo camera fabric, as follows. *Duration* is set to the next 30 seconds (during which the object is likely to enter the building) and results are returned at a periodicity of 1 second. *Locations* is set to the region near the entrance of the building.

FabricSession_ca = Register(camera_network, ClientInfo)

Result_ca = Search(FabricSession_ca, NULL, NULL, Locations, PERSISTENT, 30, 1, SearchListener)
Figure 7.5(a) shows a snapshot of a person getting out of a car stopped at a curb outside the CSE building (3 video camera cover this particular area) and heading towards the CSE building with a bag. After this detection, the indoor camera fabric is activated by the client. Figure 7.5(b) shows a photograph from the indoor motion-triggered camera fabric, of the same person tagged as suspicious, entering the CSE building with the bag. (These snapshots were taken during a trial run of the above scenario at OSU staged by AFRL as part of a layered sensing experiment that included airborne sensors in addition to the fabrics described in this example).

In addition, the indoor motion sensor fabric can track any suspicious person entering the building. The Search issued to the motion sensor fabric using a session handle FabricSession_mo is shown below.

\[
\text{Result}_\text{mo} = \text{Search}(\text{FabricSession}_\text{mo}, \text{Human, NULL, Locations, PERSISTENT, 10, 1, SearchListener})
\]

The object type is set to human for which a detector is implemented in the fabric. Object is set to NULL because the object type is non advertised and an identifier has not been assigned. Duration is set to the next 10 seconds when the object is likely to enter the building and the results are returned at a periodicity of 1 second. Locations is set to the region near the entrance of the building.

If the suspicious person enters the building, a new object instance is created by the motion sensing fabric and returned to the client. The client then terminates the previously issued Search operation and invokes a new Search with the returned id. If the fabric cannot clearly associate an id with an object, it tracks all candidate objects.
Result_mo = Search (FabricSession_mo, Human, SuspiciousPerson, LOCATIONALL, PERSISTENT, DURATIONFOREVER, 5, SearchListener)

7.5.2 Social networking scenarios

We now describe WEAVE-based applications associated with two locality specific query scenarios, which are composed to originate from the PeopleNet mobile fabric and to be realized using other sensing fabrics mentioned in Section 7.2.

**Example 1 (Where is Vinod?):** Client Anish in the CSE building, who is scheduled to play squash with Vinod in RPAC, wishes to find out if Vinod is still in the CSE building or has already reached the RPAC. In this case, the supporting application to determine if Vinod is in the CSE building or in RPAC connects to the instances of the PeopleNet networks in CSE and in RPAC, and invokes the Search for the id associated with Vinod’s mobile device (the application is assumed to know the id) in both networks. The PeopleNet fabric supports Search operations on cooperative objects, and can provide a more precise location as well.

The following is the sequence of Search invocations in the client application. (We omit Register operations.)

Result_cse = Search(FabricSession_cse, ADVERTISED, Vinod, LOCATIONALL, ONCE, NOW, NULL)

Result_rpac = Search(FabricSession_rpac, Advertised, Vinod, LOCATIONALL, ONCE, NOW, NULL)

**ObjectType** is set to ADVERTISED and Vinod is the object specified. The search is invoked as a one shot operation, and Periodicity is set to NULL.
Figure 7.6 shows localization of mobile units in PeopleNet in an early trial run, using a combination of static anchor nodes and collaboration among mobile units themselves. This sort of service supports the response to client issued Search on self advertised objects.

Example 2 (Is there an empty squash court?): Anish then wishes to find out if any squash court in RPAC is free. The PIR sensing fabric in the RPAC building implements a detector for object type 'SquashCourt'. The detector returns one of two values, full or empty, for every instance of squash court. These are cooperative objects that belong to the sensing fabric itself. Each squash court itself has an id associated with it and the application to answer Anish’s query can invoke a Search operation on an individual court also.

\[
\text{Result}_\text{pir} = \text{Search(FabricSession\_cse\_rpac, SquashCourt, NULL, LOCATIONALL, ONCE, NOW, NULL )}
\]
Search is invoked in this example to return the status of all squash courts by setting the id to \textit{NULL}.

Note that in both examples, we have abstracted from the client the issue of the underlying communication. In other words, the client is oblivious to communication network that ensures connection with the RPAC PeopleNet fabric and the RPAC PIR fabric and that ensures retrieval of information from these fabrics.

7.6 Related work

Two notable aspects of our proposed WEAVE architecture are (i) it simplifies design of applications across heterogeneous sensor networks that are independently managed and (ii) it views a sensor fabric as providing an extensible database of queryable objects. SenseWeb \cite{65} and GENI \cite{5} are related to (i) in that they propose architectures for heterogeneous networks. TinyDB \cite{73}, SemanticStreams \cite{83} and ASAP \cite{79} are frameworks related to (ii).

\textbf{SenseWeb:} SenseWeb, an architecture proposed by Microsoft Research, allows sensing applications to be developed, which use contributing sensor networks from across the globe. A (presently centralized) engine provides a uniform set of APIs, to which applications and sensor networks, from different domains, connect to subscribe/visualize and publish data, respectively.

WEAVE, in contrast, is distributed, in that the applications directly access individual sensor network fabrics. The fabrics provide applications with the services themselves, as opposed to simply publishing data. This is unlike SenseWeb, where applications process data acquired from the engine. And the fabric APIs need not all be identical.
**GENI:** GENI proposes a framework for experimentation across different types of networks, including wireless subnets and sensor networks. GENI has a notion of user services, services available for researchers or users of a particular network to access its underlying resources. The services that we propose may indeed be viewed as an instance of GENI user service; for instance, they allow users to instantiate their own programs on the underlying network resources and to access the associated results.

Note that WEAVE services also allow users to access data generated by other network services or other user-supplied programs. This contrasts with the default GENI virtualization requirement that different user programs be isolated from each other. We assume that some users may opt-in to share the object data resulting from their programs with other users. (It is up to the latter to decide whether or not to trust the data.) For instance, a user can instantiate a network with a new detector for objects of type “car” and another user can query the network for objects of that type. In this sense multiple users of a network are not isolated from each other.

**TinyDB:** TinyDB is a query processing system for extracting data from within a sensor network. Users specify data requirements as TinyDB queries, the data is then extracted from the network using appropriate aggregation and filtering mechanisms.

WEAVE also views each sensor fabric as a database, but its queries (which can be TinyDB-like) are posed on “objects”, which are semantic values exposed by the fabric potentially via user supplied programs, as opposed to raw sensor data. WEAVE allows composing applications such as tracking across different networks, which are much more complex than those with TinyDB. TinyDB does not focus on extracting data from multiple independent networks.
**SemanticStreams:** SemanticStreams provides a framework for describing and composing applications on semantic values inferred over sensor data, such as “person”, “car” or “truck”. A primary contribution is an interpreter for concurrent high-level applications executing on a single network that optimizes the design of underlying network services while satisfying the requirements of all the applications.

By contrast, we expect that user services be already implemented for the sensor fabric and available to different applications. At run time, upon invocation by an application, a user service is allowed to optimize its operation but this is not a goal of the fabric design. Our goal is to provide a standard set of APIs that allow applications to be tailored across different sensor fabrics.

**ASAP:** ASAP focuses on optimizing urban sensor network applications based on priority and situation awareness. The focus of that paper is on designing an ASAP agent, which implements a query provided by the user on the underlying sensor networks based on the knowledge of underlying network interfaces in an optimized manner. This is complementary to our architecture, where we focus on a standardized interface for sensing fabrics so that composing applications is facilitated.

**Other related work:** Handling location is a basic requirement for the WEAVE architecture (as for most pervasive computing systems). [49] provides a comprehensive taxonomy of location that will be of great use in structuring extensions to the handling of this concept in WEAVE, such as the Locations parameter of Search.

Urban sensing applications are a subset of the pervasive computing applications considered in [50], which presents a taxonomy for characterizing, and providing a
controlled vocabulary for thinking about, such applications. Essentially this taxonomy will provide additional architectural requirements to WEAVE as we develop it further.
CHAPTER 8

CONCLUSIONS AND FUTURE WORK

In this chapter we first present some of our conclusions related to a number of areas we have addressed in this dissertation and then present our plan for continuing the research in future.

8.1 Conclusions

As the number of sensors deployed in the real world increases exponentially, one of the important current debates in the WSN community relates to the architecture of future WSN. Broadly speaking two architectures has been proposed each with its own pros and cons. The two architectures can be summarized as: one where the sensors would be ‘first class citizens (FCC)’, each with an individual identifier, that can addressed and reached individually. In support of this model, the Internet routing protocol IPv6 has been ported to sensor networks, as 6lowpan \[7\]. The alternate architecture is the ‘gateway model’, which takes the position that, individual sensors need not be addressable and reachable directly from the core nodes, but can rather be reached through a proxy or a gateway. Hence, the WSNs can continue to use protocols and services which are unique.
The fabric model takes the latter approach to WSN design. The fabric model fundamentally takes a network-wide view, particularly the services oriented approach, rather than a node-level view. We make the following conclusions from our experience:

- **Collaboration Vs Individuality:** The FCC model is suitable for networks, where there is little network-level collaboration and aggregation, and the information is consumed primarily outside of the network. For example, an inventory tracking system, in which active or passive RFID nodes act as information source for an enterprise database, could benefit from this model. On the other hand information-centric networks, such as an intrusion detection system in which sensor nodes collaborate extensively to track a subject cannot benefit from this model. The gateway/fabric model is more appropriate for such networks.

- **Pragmamibility:** Programmable WSNs testbeds and urban sensor networks will benefit from the ‘fabric model’ since it is based on the services architecture and an enterprise system can programmatically interact with them. Also, the enterprise application design becomes much more simple, when you can program the entire network using a single API. In contract, the FCC model will not be scalable when a large number of nodes need to be programmed.

- **Local Vs Global Mobility:** Sensor networks where mobility is involved poses an interesting challenge. We believe that networks where only local mobility is involved (say within a building or campus) might benefit from the fabric model, where this local mobility can be handled by the fabric manager and ‘hidden’ from the outside world, making access and programming simpler. On
the other hand, networks where non-local mobility is involved (say like cell-
phone or PDAs), could benefit from the FCC model, where a standardization
and global addressability of the protocols like IPV6 (6LowPAN) would be an
important factor.

8.1.1 Federation issues

The KanseiGenie architecture for wireless sensor network fabrics, supports a wide
range of experimentation by enabling slicing, virtualization and federation among
diverse networks. Not restricted to just sensor network fabrics, the architecture can
be readily introduced into a more general programmable network such as GENI. The
architectural model centers on network fabrics rather than on nodes, resulting in a
level of abstraction that supports a wide range of services and enables applications
that were not anticipated by the fabric designer. KanseiGenie can be customized
for domain-specific applications via vertical APIs, allowing researchers to add a rich
functionality that goes well beyond the basic set of services mentioned here. There
are many open areas, including resource specification, discovery and allocation, as
well as issues related to the “data as a resource” paradigm.

What to federate at portals? KanseiGenie has thus far chosen the Portal
as the main federating agent in the system, including roles for unifying ontologies,
embedding resource specifications, providing a uniform Experiment Specification, and
federated slice stitching. This design suits the view that a Portal realizes application-
domain specific support, and that for different domains different Portals may suit. In
other words, we view the KanseiGenie Portal as suitable for WSN experiments (and
perhaps only some classes of WSN experiments) as opposed to sufficing for all GENI-
Researcher needs. In this view, Clearinghouses are treated as being generic rather
than domain specific. Roles which are less domain specific, e.g. embedding resource
specifications or slice stitching, can be moved from Portals to CHs (or even to Sites),
assuming the method of desired stitching is communicated to them. Now, should
CHs evolve to become domain specific, they may import more roles from Portals.
Taken to the extreme, this would suggest that a top level CH be directly or indirectly
capable of unifying all resources in GENI.

Finally, we share here a few of the lessons from our experience with federated
testbed design. (i) RSpec and ESpec design, their completeness and their adaptability
is very important for the growth and research in testbed federations. (ii) A powerful
Portal, such as in our design, is not always a necessity, but definitely a blessing in
highly domain-specific federations such as WSN. The domain knowledge needed to
exploit WSNs is very high and a fully functional Portal is a very useful tool for
Researchers. (iii) The design and implementation of the Aggregate and Component
Managers is a non-trivial aspect of WSN fabric design. The AMs need to make the
least assumptions about platform support, while simultaneously providing non-trivial
guarantees to the User. (iv) A significant amount of work is involved in maintaining
a healthy testbed. A systematic approach to fault-detection and correction, and an
autonomous health monitoring system are an essential part of any long living WSN
infrastructure.
8.1.2 Standardization of APIs and resources

In this dissertation, we have taken two slightly different views towards standardization. We propounded standardization of a minimal set of APIs for the fabric manager, while also arguing against standardization for resource specifications. While standardization clearly has its benefits, it curtails flexibility in specification and implementation. Even for the fabric APIs we strongly support exposing the APIs using a platform-neutral web-service platform, in which case the APIs and their arguments can be learned automatically. In WEAVE we argued for standardizing APIs for domain services, so that a tighter coupling can be achieved between different fabrics. This might be possible since the fabrics are designed for similar purposes and are likely to have similar capabilities.

While a partial standardization approach may suit the traditional core network comprising of homogeneous resources, it is not equally applicable to WSN fabrics. Roscoe in [77] lays out the drawbacks of going down the standardization path. A key challenge in using a standardized specification is anticipating in advance all the possible things that must be expressible. He argues, citing the experience with the erstwhile ANSA trading model [48], that the resource request instead should be viewed as a constraint satisfaction problem and not a simple database query. This calls for a language in which we can specify constraints and the community need to standardize only the specification language, but not the format. Thus a decentralized resource specification scheme better supports the inherent heterogeneity in resource and experiment specification in future federated fabrics.
8.1.3 Routing in asymmetric networks

While wireless ad hoc routing is a very mature area, routing on constrained mobile devices in an asymmetric setting has not received enough attention, which we addressed in this dissertation. AER performs mobility-aware, asymmetric, multi-path proactive routing. It is well suited to constrained mobility networks, where its spontaneous adaptation to increase in mobility preserves the sub-linear growth in cost. Adaptations of well known alternative proactive protocols to accommodate multi-parents do not scale as well for the constrained mobility region. The reason being that, to avoid increasing node disconnections as mobility increases, one has to increase the rate of maintenance significantly. The other way to avoid disconnections is take advantage of the node mobility information. We see this even in one of our adaptations of DSDV, the Mobility-aware Multipath DSDV (MMDSDV), which performs better than other adapted protocols since it is designed to make use of mobility information. Albeit, AER performs even better than MMDSDV, since AER is centralized and adapts to mobility in both direction (LSAP-to-mobile node and Mobile node-to-LSAP).

8.1.4 Predictive routing in intermittently connected networks

As a first step in understanding human mobility, number of recent studies have focused on characterizing the inter-contact time distribution of real-life mobility traces. But, most of these studies have focused only on aggregate level statistics. We believe that human mobility characteristics vary significantly between pairs of nodes and hence making routing decisions based on aggregate level history and statistics might not work well.
In this dissertation we focused on the hard problem of making predictions without making any assumptions about either the mobility model or the presence of landmark nodes, which could aid prediction. Although this is a hard problem, we are very encouraged by the results. Our experiments have shown that not only is such prediction possible, it is even possible to do path planning based on such prediction.

However, the research is far from complete and needs further investigation both in terms of better prediction methodologies and routing algorithms. Also, solving the problem of automatically learning the framelength is important for full-scale deployment of the predictive routing algorithm.

8.2 Future work

8.2.1 Resource specifications for WSN fabrics

The sensor fabric resource specifications act as the language that all the entities in the architecture understands. It is important for the designers to come up with an ontology that is detailed enough for the users of the domain to take advantage of the fabric services and features, and broad enough to enable interaction and joint experimentation with other programmable fabrics (such as other wireless networks and core networks).

Much of the complexity of sensor networks need to be embedded in resource specifications (RSpecs). As new resources and capabilities are added to a fabric, these specifications will inevitably need to be extended. We expect the extensions to leverage a hierarchical name space. This will allow new communities that federate with KanseiGenie to extend the resource specification within their own partition of the name space. Components that offer specialized resources will similarly extend the
resource specification in their own name space. Additionally, we need to consider the granularity of resource specification, which decides the level of details in resource description. In federated resource management, there is no unique solution and the implementation strategy is subject to both technical and administrative constraints. For instance, whether and how much information about resource properties should be maintained by clearinghouses will depend on the trust relations among the entities involved and may be encoded in resource specifications at different levels of granularity.

To enable reliability and predictability in experimentation, resource specification also needs to characterize precisely the reliability and predictability properties of WSN testbeds, including the external interference from 802.11 networks, the stability of link properties, and failure characteristics of nodes in a testbed, so that, an experiment will also use reliability- and predictability-oriented specification to specify its requirements on the allocated resources.

For the same experiment there may be different ways of specifying the actual resources needed. For an experiment requiring two TelosB motes and a link of 90% reliability connecting these two motes, for instance, we may define the resource specification to request two motes six meters away with the necessary power level for ensuring a 90% link reliability between these two motes, or we may define the resource specification to request any two motes connected by a link of 90% reliability. Both methods will give the user the desired resources, but the second method will allow for more flexibility in resource allocation and thus can improve overall system performance.
8.2.2 Resource discovery

For federated resource management, different clearinghouses need to share resource information with one another according to their local resource sharing policies. The two basic models of resource discovery are the push and pull models. In the push model, a clearinghouse periodically announces to its peers or upper-level clearinghouses about the available resources, at its associated fabrics that can be shared. In the pull model, a clearinghouse requests from its peers or upper-level clearinghouses their latest resource availability. We expect the pull model to be mainly used in an on-demand manner when a clearinghouse cannot find enough resources to satisfy a user request. Note that this interaction between clearinghouses also needs to be authenticated using, for example, a Public Key Infrastructure.

8.2.3 Data as resource

One consequence of the fabric model is that the network is hidden behind a collection of interfaces and as long as the interfaces are standardized and known, a User can programmatically access it. In other words, it does not matter if the interface is implemented by a sensor network or by a single PC. Thus, DataHubs – which are data bases that can annotate and store results of experiments and replay the data for similar future queries – can now be viewed as a sensor resource. Alternately, a sensor network can be viewed as a source for a data stream and the user as a data transformation program. Under this unified view a DataHub, which can interpret queries and transform the stored data accordingly can fake a sensor fabric. Hence, in fabric model, data (properly annotate and qualified) and sensing resources are inter-changeable and provides for interesting hybrid experimentation scenarios.
The architecture provides much research opportunities and challenges, as a number of questions need to answered before the architecture can be used beyond the most simplistic scenarios. Challenges include the following:

- How to automatically annotate and tag data coming from sensor networks to create a credible DataHub?

- It is common for the same experiment to produce multiple similar data sets in wireless networks. How does a user decide which dataset to use as a representative of an experiment?

- Does the RSpec ontology need to be extended to represent data?

- What range of queries can be answered with the current data? Should data be pre-processed to decided acceptable queries?

### 8.2.4 Network virtualization

The fundamental aim of the fabric architecture is to virtualize and globalize the resources in a sensor network, so that in principle, a user anywhere in the world can request, reserve and use the resources. However, the more a resource is virtualized, the less control a user has over it. Thus there is a trade-off between the level of access and virtualization. The challenge for modern network designers is to provide as much control (as low in the stack as possible) to the users, while retaining the ability to safely recover the resource and also making sure the resource might be shareable.

In a fabric, multiple researchers will run their experiments concurrently on different subsets of an array of sensors of the same type. Usually, sensors are densely deployed over space. Such density provides the means for different experimenters to
share the same geographical space and sensor array to conduct concurrent experiments that are subject to very similar, if not statistically identical physical phenomenon. In such environments, interference is inherent between users due to the broadcast nature of the wireless communications; its effect is more prominent when the communicating devices are close to one another. The virtualization of wireless networks imposes a further challenge for the sensor fabric providers to ensure isolation between concurrently running experiments. Such interference isolation is usually achieved by careful frequency or time slot allocations. However, these solutions are quite primitive in nature and do not provide optimum network utilization. More importantly, these solutions are not suitable for sensing infrastructures where multiple applications from different users need to be run concurrently in a production mode. Our recent research in this area using statistical multiplexing as the basis of virtualization is promising to provide better solutions, enabling much better network utilization and external noise isolation.

8.2.5 Connectivity and capacity in mobile and intermittently connected networks

As the domain of WSNs also include mobile and disruption tolerant networks, it is important (especially in the low powered WSNs) understand the connectivity and consequently the capacity of a network. While a few theoretical work exists in the case of static wireless network, these mainly tend to be asymptotic analysis and provide bounds on capacity of a network. The capacity of mobile network is not well understood, especially for non-asymtotic scenarios. Another challenge is estimating
and understanding the capacity of a mobile network, under different mobility characteristics. As we start to understand mobility better, we might be able to derive tight capacity bounds under various mobility distributions.

Another related area is the optimal forwarding algorithms in mobile networks. It is known that optimal forwarding/routing algorithms such as the Back Pressure algorithm exist for static wireless networks. However, to our knowledge, such results are not available for mobile and delay tolerant networks. The problem is made complex by the fact that in mobile networks, the routing flows are short lived and its not clear how fast the algorithm needs to converge to achieve optimality. On the other hand, packets are routed per-contact in the DTNs and the notion of a flow is not well defined. Hence, a completely new approach to optimality might be required.


[16] OWL Web Ontology Language. \url{http://www.w3.org/TR/owl-features/}

[17] PeopleNet: A mobility testbed. \url{http://peoplenet.cse.ohio-state.edu}


[19] Random way point model. \url{http://www.netlab.tkk.fi/esa/java/rwp/rwp-model.shtml}

[20] Resource Description Framework. \url{http://www.w3.org/RDF/}


[22] Squawk Java Virtual Machine for Sun SPOT. \url{http://labs.oracle.com/projects/squawk/squawk-sunspot.html}

[23] Stargate gateway devices. \url{http://blog.xbow.com/xblog/stargate_xscale_platform/}

[24] SunSpots: A java based sensor mote. \url{http://www.sunspotworld.com/}


[26] The Network Simulator - ns-2. \url{http://www.isi.edu/nsnam/ns/}

[27] TinyOS: A Operating System of Embedded Sensors. \url{http://www.tinyos.net}

[28] Tinyos implementation of tinyaodv. \url{http://cvs.sourceforge.net/viewcvs.py/tinyos/tinyos-1.x/contrib/hsn/}


