Network Partitioning via Code Mobility

Carmelo Ragusa

UniS

A Thesis submitted for the Degree of Doctor of Philosophy

Centre for Communication System Research
University of Surrey

April 2005

© Carmelo Ragusa 2005
Abstract

Systems relying on increasingly large and dynamic communication networks must find effective ways to optimally localize service facilities. This can be achieved by efficiently partitioning the system and computing the partitions’ centers, solving the classic $p$-median and $p$-center problems. These are \textit{NP-hard} when striving for optimality. Numerous approximate solutions have been proposed during the past 30 years. However, they all fail to address the combined requirements of scalability, optimality and flexibility.

This thesis presents a novel location algorithm that is distributed, does not require any direct knowledge of the network topology, is computed in linear time, and leads to provably near-optimal $p$-medians. The algorithm exploits the key properties of Mobile Agents (MAs), the latter being autonomous software entities capable of roaming the network and cloning or spawning other Mobile Agents. Mobile Agents play the role of service facilities that are capable of iteratively partitioning the network and locating themselves in $p$-median locations.

This thesis motivates the use of Mobile Agents to provide a ‘distributed’ solution to the $p$-median problem that is traditionally tackled only with ‘centralised’ algorithms. Efficiency is achieved through a combination of distribution, code mobility and simple use of network routing information. The proposed near-optimal algorithm does not require the reconstruction of the network distance matrix and is characterized by linear computational complexity. The proposed approach outweighs existing algorithms, which are polynomial of $2^{nd}$ and $3^{rd}$ degree and require a complete knowledge of the network topology.

The evaluation of the proposed algorithm has been carried out through simulations. For this purpose, a simulation environment on top of NS (Network Simulator) was developed, which allows the implementation and evaluation of a Mobile Agent system.

Along with the algorithm a set of applications that shows the flexibility of the algorithm to adapt to different requirements is presented. The range of algorithm applications focused on overlay networks and in particular Application-level Multicast, Content Adaptation Networks and Overlay Resource Management.
To Claudia

who did something priceless for me.
Acknowledgments

I would like to thank my supervisor Antonio Liotta for the chance he gave me to do this doctorate. Its help, guidance and suggestions during the past few years are invaluable. I've learned a lot from him, I couldn't be luckier.

Part of this work has been carried out in the context of M-VCE core 2, Software Based Systems work area. My gratitude goes to the numerous researchers, academics and industrial members who have provided feedback and suggestions.

I would like also to thank a friend of mine, Daniel Chew, who helped me settling down when I first arrived in England. At that time I didn't know anybody and his support was really important to me.

My thanks go also to another friend, Marco Ballette, with whom there has been lot of reciprocal support. Without him, the time spent in the office would have been cheerless.

Finally, I want to thank my family and partner who have been really supporting especially in the tough periods that a PhD put you through.
Table of Contents

List of Figures ......................................................................................................................... iv
List of Tables ........................................................................................................................ vii
List of Publications ............................................................................................................. viii
Chapter 1  Introduction ....................................................................................................... 1
  1.1 Overview ....................................................................................................................... 1
  1.2 Motivation .................................................................................................................... 3
  1.3 Thesis Outline .............................................................................................................. 5
Chapter 2  Background and Related Work ..................................................................... 6
  2.1 Introduction .................................................................................................................. 6
  2.2 The Network Partitioning Problem ............................................................................. 6
    2.2.1 Classification of Location Problems ........................................................................ 6
    2.2.2 Formulation of the p-median Problem ................................................................... 8
    2.2.3 Formulation of our Network partitioning problem .............................................. 9
  2.3 Mobile Agents Technology ....................................................................................... 12
    2.3.1 Code Mobility Mechanisms .................................................................................. 12
    2.3.2 Mobile Agent definition ....................................................................................... 14
    2.3.3 Advantages of MAs ............................................................................................. 15
    2.3.4 MA Applications .................................................................................................. 15
    2.3.5 Issues associated to MAs ..................................................................................... 16
  2.4 MAs assessment tools ................................................................................................ 17
  2.5 Overlay Networks ...................................................................................................... 19
  2.6 Related work .............................................................................................................. 20
    2.6.1 Previous work ........................................................................................................ 20
    2.6.2 Location Problems approaches .......................................................................... 21
    2.6.3 Assessment tools .................................................................................................. 24
    2.6.4 Overlay Networks Examples ............................................................................. 25
  2.7 Summary ..................................................................................................................... 28
### Chapter 3  A Simulator Framework for Distributed Systems and Applications...

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Introduction</td>
<td>29</td>
</tr>
<tr>
<td>3.2 The NS Network Simulator</td>
<td>29</td>
</tr>
<tr>
<td>3.2.1 Software Architecture</td>
<td>31</td>
</tr>
<tr>
<td>3.2.2 C++ - OTcl Linkage</td>
<td>32</td>
</tr>
<tr>
<td>3.2.3 Class Hierarchy</td>
<td>33</td>
</tr>
<tr>
<td>3.3 Design and Implementation of Mobile Agent Extension</td>
<td>34</td>
</tr>
<tr>
<td>3.3.1 Assumptions</td>
<td>34</td>
</tr>
<tr>
<td>3.3.2 MA Class Inheritance</td>
<td>35</td>
</tr>
<tr>
<td>3.3.3 MA Interaction model</td>
<td>36</td>
</tr>
<tr>
<td>3.3.4 Implementation</td>
<td>37</td>
</tr>
<tr>
<td>3.4 Further Extensions to NS</td>
<td>40</td>
</tr>
<tr>
<td>3.4.1 The ResourceAgent class</td>
<td>42</td>
</tr>
<tr>
<td>3.4.2 The ResourceServer class</td>
<td>43</td>
</tr>
<tr>
<td>3.5 Summary</td>
<td>47</td>
</tr>
</tbody>
</table>

### Chapter 4  Network Partitioning Algorithm...

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Introduction</td>
<td>48</td>
</tr>
<tr>
<td>4.2 Algorithm overview</td>
<td>48</td>
</tr>
<tr>
<td>4.3 Parameters and definitions</td>
<td>52</td>
</tr>
<tr>
<td>4.4 Cloning Phase</td>
<td>53</td>
</tr>
<tr>
<td>4.5 Migration Phase</td>
<td>54</td>
</tr>
<tr>
<td>4.6 Aggregation Phase</td>
<td>56</td>
</tr>
<tr>
<td>4.7 The algorithm through an example</td>
<td>58</td>
</tr>
<tr>
<td>4.8 Dynamic Adaptation Phase</td>
<td>60</td>
</tr>
<tr>
<td>4.9 Summary</td>
<td>61</td>
</tr>
</tbody>
</table>

### Chapter 5  System Performance Evaluation...

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Introduction</td>
<td>63</td>
</tr>
<tr>
<td>5.2 Evaluation Methodology</td>
<td>63</td>
</tr>
<tr>
<td>5.2.1 Choice of Methodology</td>
<td>63</td>
</tr>
<tr>
<td>5.2.2 Performance Metrics</td>
<td>64</td>
</tr>
<tr>
<td>5.2.3 Fixed Parameters</td>
<td>64</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>5.2.4 Factors</td>
<td>66</td>
</tr>
<tr>
<td>5.2.5 Simulation Complexity</td>
<td>67</td>
</tr>
<tr>
<td>5.2.6 Hardware Parameters</td>
<td>69</td>
</tr>
<tr>
<td>5.2.7 Simulation Design</td>
<td>70</td>
</tr>
<tr>
<td>5.3 Computational complexity</td>
<td>72</td>
</tr>
<tr>
<td>5.4 Accuracy</td>
<td>74</td>
</tr>
<tr>
<td>5.4.1 Location optimality</td>
<td>74</td>
</tr>
<tr>
<td>5.4.2 Accuracy on number of partitions</td>
<td>75</td>
</tr>
<tr>
<td>5.5 Sensitivity to starting conditions</td>
<td>77</td>
</tr>
<tr>
<td>5.6 Dynamic Adaptation</td>
<td>80</td>
</tr>
<tr>
<td>5.7 Summary</td>
<td>82</td>
</tr>
<tr>
<td>Chapter 6 Discussion and Conclusions</td>
<td>83</td>
</tr>
<tr>
<td>6.1 Thesis summary</td>
<td>83</td>
</tr>
<tr>
<td>6.2 Discussion of Thesis Contributions</td>
<td>84</td>
</tr>
<tr>
<td>6.2.1 Network partitioning algorithm</td>
<td>84</td>
</tr>
<tr>
<td>6.2.2 Adaptability</td>
<td>84</td>
</tr>
<tr>
<td>6.2.3 MA system simulator</td>
<td>84</td>
</tr>
<tr>
<td>6.2.4 Applications</td>
<td>85</td>
</tr>
<tr>
<td>6.3 Practical deployment of the algorithm</td>
<td>85</td>
</tr>
<tr>
<td>6.4 Future Work</td>
<td>86</td>
</tr>
<tr>
<td>6.4.1 Improving the algorithm performance</td>
<td>86</td>
</tr>
<tr>
<td>6.4.2 Implementing an algorithm recovery feature</td>
<td>87</td>
</tr>
<tr>
<td>6.4.3 Prototyping the algorithm</td>
<td>87</td>
</tr>
<tr>
<td>6.4.4 Improving the Overlay Network</td>
<td>87</td>
</tr>
<tr>
<td>Appendix A. Further results</td>
<td>88</td>
</tr>
<tr>
<td>Appendix B. Applications</td>
<td>99</td>
</tr>
</tbody>
</table>
## List of Figures

Figure 2.1 General graph. ................................................................. 7
Figure 2.2. a) Example system topology. b) Partitions and partition centres resulting in minimal Total Hop Distance. .................................................................................. 10
Figure 2.3 Components in a CDN (from [40]). ............................................... 27
Figure 3.1 NS functional architecture. .................................................. 30
Figure 3.2 C++ - OTcl linkage ......................................................... 33
Figure 3.3 NS class hierarchy .......................................................... 34
Figure 3.4 MA model classes inheritance ........................................... 36
Figure 3.5 MA interaction model ...................................................... 37
Figure 3.6 Functional Architecture of NS with MA extensions ........... 38
Figure 3.7 MA execution over a simulated network .......................... 39
Figure 3.8 Resource node architecture for NS .................................. 41
Figure 3.9 ResourceAgent class ..................................................... 42
Figure 3.10 ResourceServer class ................................................... 44
Figure 3.11 Disk class ........................................................................ 45
Figure 3.12 CPU class ....................................................................... 45
Figure 3.13 ServiceAgent class ...................................................... 46
Figure 4.1 Algorithm Overview ...................................................... 49
Figure 4.2 Algorithm example ........................................................ 51
Figure 4.3 Cloning Phase flowchart ................................................ 53
Figure 4.4 Migration phase flow-chart ............................................. 56
Figure 4.5 Aggregation phase flow-chart ......................................... 57
Figure 4.6 Aggregation example ...................................................... 57
Figure 4.7a) Example system topology. b) Partitions and partition centres resulting in minimal Total Hop Distance ................................................................................. 58
Figure 4.8. Partitioning process for the topology of Figure 2.2a, including the MA deployment map resulting in the optimal configuration of Figure 2.2b. ......................... 59
Figure 4.9. Self-healing algorithm .........................................................................................61
Figure 5.1. Transit-stub topology (from [81]). .................................................................65
Figure 5.2 grid-like simulation approach .....................................................................70
Figure 5.3 Simulation environment and tools.................................................................71
Figure 5.4 Partitioning time Vs Diameter for 102-node topologies. ...............................73
Figure 5.5 Partitioning time Vs $p/|N|$ for 102-node topologies .......................................74
Figure 5.6 Location Optimality comparison ..................................................................75
Figure 5.7 Error on number of partitions produced by the algorithm..............................76
Figure 5.8 Sensitivity of partitioning time to starting node; 51-nodes topology, ratio 0.2. .........................................................................................................................................78
Figure 5.9 Sensitivity of number of partitions computed to starting node; 51-nodes topology, ratio 0.2 .................................................................79
Figure 5.10 Sensitivity of THD to starting node; 51-nodes topology, ratio 0.2 ...............80
Figure 5.11 Sample of results on self-healing. .............................................................81
Figure 6.1 Partitioning time vs Network Diameter for topology 147 nodes .................88
Figure 6.2 Partitioning time Vs $p/|N|$ for 147-node topologies .......................................89
Figure 6.3 Partitioning time vs Network Diameter for topology 198 nodes ..................90
Figure 6.4 Partitioning time Vs $p/|N|$ for 198-node topologies .......................................91
Figure 6.5 Deployment time vs Network Diameter for topology 258 nodes ...............92
Figure 6.6 Partitioning time Vs $p/|N|$ for 258-node topologies .......................................93
Figure 6.7 Deployment time vs Network Diameter for topology 303 nodes .......... 94
Figure 6.8 Partitioning time Vs $p/|N|$ for 303-node topologies .......................................95
Figure 6.9 Sensitivity of partitioning time to starting node; 198-nodes topology, ratio 0.2. .........................................................................................................................................96
Figure 6.10 Sensitivity of number of partitions computed to starting node; 198-nodes topology, ratio 0.2 .................................................................97
Figure 6.11 Sensitivity of THD to starting node; 198-nodes topology, ratio 0.2 ...............98
Figure 6.12 Distributed mobile application over IP network. MAs act as end-systems to deliver group data from the source to the users.........................................................103
Figure 6.13 DVMRP Tree build time Vs MA-based Tree build time ............................105
Figure 6.14 StressRatio Vs Topology size.........................................................................106
Figure 6.15 Multicast solution, with stream transcoding at source. ......................... 108
Figure 6.16 Multicast solution, with stream transcoding at receivers. ....................... 109
Figure 6.17 CAN solution, with stream transcoding at intermediate nodes. ............... 110
Figure 6.18 CAN vs PIM-SM performance results. ...................................................... 112
Figure 6.19 Resource Management System ................................................................. 114
Figure 6.20 Overlay resource management system in action. ...................................... 116
Figure 6.21 Performance metrics for Scenario 1. ......................................................... 119
Figure 6.22 Performance metrics for Scenario 2. ......................................................... 120
Figure 6.23 Performance metrics for Scenario 3. ......................................................... 121
Figure 6.24 Performance metrics for Scenario 4. ......................................................... 122
Figure 6.25 Performance metrics for Scenario 5. ......................................................... 123
List of Tables

Table 3-1 Resource Server classes................................................................. 43
Table 3-2 Job classes. ................................................................................. 46
Table 3-3 Job Priorities. ............................................................................... 47
Table 5-1. Network simulator parameters. .................................................. 65
Table 5-2 Agent-based monitoring system fixed parameters ....................... 66
Table 5-3. Simulations factors. .................................................................... 66
Table 5-4 Experiment factors....................................................................... 67
Table 5-5 Illustration of simulation complexity. ........................................... 69
Table 5-6 Hardware and operating system parameters. ............................. 69
Table 6-1 Overlay resource management scenarios. ..................................... 118
List of Publications

Conf. on Simulation, Modelling and Optimization (ICOSMO 2002), Skiathos Island, Greece, September 25-28, 2002.
Chapter 1  Introduction

1.1 Overview

The rapid growth of the Internet over the past decade has pushed the telecommunication world towards the development of large, dynamic distributed systems. Moreover, the integration among wireless and wired networks makes this development even more challenging. Management systems will play a key role within the big picture. In fact, they need to show flexibility to deal with different requirements, adaptability to different conditions/changes and scalability to manage large systems in order to deal with increasing dynamics of such systems.

In this contest, assuming that the most suitable management approach has to be distributed, an important issue arises: how should the network be divided among the different management entities?

Important requirements have to be taken in consideration, such as the optimal number of management entities, which part of the network each entity will manage and how central the management entity is with respect to the managed entities. The problem is similar to the problem of network partitioning, where a network is divided in a number of sub-partitions. More precisely, network-partitioning problems are part of location problems, which involve the optimal placement of service facilities on networks of $N$ nodes. Service facilities are traffic sources, traffic destinations or both and are part of a distributed system, application or protocol. The location problem can be defined as that of 1) finding the optimal number of service facilities; 2) partitioning the network in $p$ sub-partitions; and 3) optimally placing those facilities within their respective partition. Optimal location involves the minimization of an objective function. The location problem is termed '$p$-median problem' when the objective is to minimize the overall traffic incurred by the service facilities [44]. It is termed '$p$-center problem' when the objective is to minimize the maximum response time [45].

Because of their importance, $p$-median and $p$-center problems have been subject to intensive study for over 30 years in a variety of scientific disciplines. In communication
networks, location problems are fundamental to a number of other problems related to distance minimization. It is through network partitioning that large systems are scaled down for the purposes of effective monitoring and control, leading to reductions in traffic and response time. Partitioning problems assume particular relevance in the modern era of mobile ubiquitous computing, ad hoc networking and dynamically re-configurable networks. The challenge is to address the combined requirement of scalability, optimality and dynamicity.

The $p$-median and $p$-center problems are both NP-hard on general networks [44]. Numerous approximate polynomial algorithms have been proposed (see extensive survey presented in [10], [12] and [44]) but none of them suits the aforementioned combined requirements. The main hurdle of existing approaches is their intrinsic centralised nature, i.e. they all require knowledge of the network topology in order the get the distance matrix according to a particular metric as input parameter. While this is less of a problem in offline calculations for medium- to long-term locations over relatively static networks, it becomes a critical problem in the case of dynamic large-scale networked systems. In this case, the task of collecting a real-time snapshot of network topology and traffic profile (if bandwidth is a metric) is indeed ambitious.

This thesis addresses the partitioning problem, presenting a novel approach that has the following provably advantages:

- It is based on a distributed algorithm that has a low (i.e. linear) cost;
- It can satisfy precise constraints on the number of partitions (self-configuration);
- It creates provably near-optimal partitions and partition centres (self-optimization);
- It re-calculates near-optimal partition centres in face of component failures or congestion (self-healing).

To the best of our knowledge, no existing technique satisfactorily addresses the combined requirements of efficiency, scalability, adaptability and optimality. The system exploits the key properties of Mobile Agents (MAs), the latter being autonomous software entities capable of roaming the network and cloning or spawning other MAs [1]. MAs play the role of service facilities that are capable of iteratively partitioning the network and locating themselves in $p$-median locations.
MA systems are complex distributed software entities whose behavior, performance and effectiveness cannot always be anticipated by the designer. Their evaluation often presents various aspects that require a careful, methodological approach as well as the adoption of suitable tools. Therefore, a simulation framework, which is generic – i.e. decoupled from any specific MA system – and addresses the requirements of complex, dynamic, large-scale MA systems, has been designed and developed. The system is built on top of the NS network simulator [77], which supports the most common physical, link, network, transport and application layer protocols for fixed and mobile communication networks. Having extended NS with support for MAs – including agent execution over ‘simulated’ network nodes, agent migration, agent cloning, agent destruction and so forth – an environment in which an arbitrarily complex MA system can be run over arbitrarily complex inter-networks was created. This can be used to evaluate the various aspects of agent systems under a variety of conditions such as network topology, size, communication protocols, etc. As a consequence, the partitioning algorithm was developed using these MAs extensions and has been evaluated through simulations. Finally, some experiences of making use of the algorithm proposed was made in the context of overlay network solutions including Application-level Multicast, Content Adaptation Networks and Overlay Resource Management.

1.2 Motivation

In addition to its theoretical component, this thesis is driven by the following aspects: 1) the need to address the fundamental problem of network partitioning for distributed systems; and 2) the lack of assessment tools for the evaluation of distributed systems.

Beside the theoretical aspect of this thesis work, the algorithm finds its place in distributed systems. The tremendous developments surrounding Internet-based systems are ineluctably pushing towards distribution, mobility, and ubiquity, posing new challenges to the management of such systems. Conventional approaches require a certain degree of centralised control, exerted via either a single management entity or an organised set of co-operating management entities. Hierarchical and distributed management approaches have become increasingly popular in network and systems management.
In practice, and for scalability reasons, those systems must adopt a partitioning approach that involves a certain level of organisation among the various entities (or nodes) of the system. A good example in networking is ad hoc networks, where terminodes are organised in clusters having a node with special functions, termed cluster head [2]. Similarly, peer-to-peer systems are structured in peer groups. Discovery, publishing and inter-node communication are solved, again, via special nodes termed rendezvous or super-peers [3].

Efficient partitioning is a fundamental problem in the area of networking and distributed systems. Because of their peculiarities, distributed systems are particularly sensitive to the way partitions are formed and partition centres are elected. Assuming that distributed systems are composed of entities that move, attach/appear and disconnect/disappear on a relatively frequent basis, those systems must rely on effective means for maintaining optimal partitions and partition centres. This includes the ability to dynamically partition the system into an appropriate number of partitions (depending on the number of involved entities) and elect the best possible partition centres. We exploit the key features of code mobility to achieve a fully distributed clustering algorithm that is also adaptive to network conditions. The mobile code paradigm has been broadly used in the past as an alternative to the client-server model because of its potential to tackle problems where scale and dynamics assume significant importance [1].

The second main motivation that has driven this thesis is the lack of tools for the assessment of distributed networked systems such as MA-based systems. Because of their complexity, MA-based systems are assessed through prototypes, which demonstrate the feasibility of the system, but fail to capture the system scale within realistic network conditions. The MA extensions, developed within the NS network simulator, provide a powerful tool that combines a well-known network simulator with the agent technology,

---

1 In the context of this thesis, code mobility is also refereed as MA (Mobile Agent). See Section 2.3 for more details.

2 We indicate MA-based system as a system that uses weak mobility (see Section 2.3) to perform its tasks, differently from MAs systems, where strong mobility and cooperative intelligent agents are used.
and therefore allow the evaluation of the proposed algorithm under realistic network conditions.

1.3 Thesis Outline

This thesis is organised as follows. Chapter 2 presents background information about the various subjects contemplated by this thesis, this includes an introduction to the network-partitioning problem and a survey of the approaches that deal with this problem. We then provide an overview of MA technology, its applications, pros and cons, and illustrate some of the existing systems used for their analysis. After that, we describe overlay networks as a means to introduce new functionality within the Internet. Finally, the related work on the topics involved in this thesis work is depicted.

Chapter 3 describes the simulation environment adopted to assess the thesis work. First, we explain the reasons that conducted to choose NS. Second, we analyse NS limitations and individuate the requirements needed to enhance the simulator in order to implement the system proposed in this thesis. Then, the NS extensions for MA support are explained. Finally, the extra extensions provided to NS, needed to implement one of the case studies, are described.

The proposed network-partitioning algorithm is examined in Chapter 4. First, a general overview is given, explaining the different phases of the algorithm. Then, the algorithm is discussed in more detail, describing each phase and its interaction with the other ones.

Chapter 5 discusses the performance analysis of the algorithm under different conditions, highlighting different features of the algorithm. Furthermore, another aspect of the algorithm is discussed here, namely the adaptation to dynamic network environment.

Finally, Chapter 6 draws the thesis conclusions, summarising and discussing the whole work, results and achievements. Then, a viability study of the algorithm in the context of IP networks is provided, as well as future directions and developments.
Chapter 2  Background and Related Work

2.1 Introduction

The main areas covered in this thesis are Location problems, code mobility, assessment tools for MAs-based systems and overlay networks. This chapter presents an overview of these areas along with the related work present in the literature.

Location problems are covered in Section 2.2 and concern the study of algorithms aimed at optimally placing $p$ service facilities in a network with $N$ nodes. This is a classic problem, which has been studied in the context of transportation theory and of computer networks. The main focus is on the median problem, which is strictly related to the proposed partitioning algorithm.

In this thesis code mobility is the basis of the proposed system. An overview of this technology along with advantages, disadvantages and applications is presented (Section 2.3). The problem of assessing MAs-based systems is also discussed along with the limitations of available tools and techniques.

Furthermore, the overlay concept is introduced along with claimed potential that this mechanism can offer.

Finally, Section 2.6 analyses the work that is closely related to the thesis proposal.

2.2 The Network Partitioning Problem

The network-partitioning problem that the proposed system intends to solve is analogous to the classic graph-theoretical location problem. Therefore, we introduce below the formulation of the location problem (from [7], [8], [9], [10], [11], [12] and [41]), defining, our network-partitioning problem in terms of this formulation.

2.2.1 Classification of Location Problems

Network location problems aim at placing service facilities in a network. A network can be a road, a city, or a communication network. Therefore, given a set of facilities and a set of demanding points, the objective is to place the facilities in order to minimise a given objective function, as follows [11]:

...
• The potential location of the facilities to be located — either at vertices or anywhere on the network (unrestricted locations);
• The location of demand points — either at vertices or anywhere on the network (unrestricted locations);
• Objective function — either to minimise the total cost to all demand points or to minimise the maximum cost to any demand point.

The combination of the above cases gives four different possibilities. In our case, the network is a communication network, which means that facilities and demand nodes are located at vertices. In practice, in the proposed system a facility is a system entity manager and demand nodes are served nodes. The metric used in general is the distance. Particularly, in our case the distance is the hop-count. Figure 2.1 shows the difference between a restricted location and an unrestricted one in a general graph.

![Figure 2.1 General graph.](image)

In the case in which both facilities and demands occur only at vertices, the **centre** of a graph is any vertex whose farthest vertex is as close as possible. Other terms are used in the case of unrestricted locations. A **general centre** is any vertex whose farthest point in the graph is as close as possible. An **absolute centre** is any point whose farthest vertex is as close as possible. Finally, a **general absolute centre** is any point whose farthest point is as close as possible. For centre problems, the objective function is to minimise the maximum distance from the facility to a demand point.

Similarly to the **centre** for each of these four types of location problems, we can define the **median**, **general median**, **absolute median**, and **general absolute median**. In this case,
the objective function is to minimise the sum of the distances from the facility to all demand points. This thesis focuses on median problems.

Usually, the terms p-centre and p-median are used to indicate the problem of locating p facilities.

**Integer linear programming** (LP) is generally used to formulate these problems. LP problems are usually formulated in terms of inputs (i.e. constants that are given in the problem definition and statement), decision variables (i.e. the quantities that we are trying to find), and surplus variables (i.e. used to convert inequalities constraints into equality constraints). Below, only the formulation for the p-median problem is given, being the one relevant for this thesis work.

### 2.2.2 Formulation of the p-median Problem

This section specifies the vertex p-median problem or minisum problem made by Daskin in [12] (pp.200-203). Defining the cost of serving demands at node i as the product of the demand at node i and the distance between demand node i and the nearest facility to node i, the objective is to find the location of p facilities on a network so that the total cost is minimised. The problem is formulated as follows:

**Inputs**
- \( d_{ij} \) = distance from demand node i to candidate facility j
- \( h_i \) = demand at node i
- \( p \) = number of facilities to locate

**Decision Variables**
- \( X_j = \begin{cases} 1 & \text{if we locate at candidate site } j \\ 0 & \text{if not} \end{cases} \)
- \( Y_{ij} = \begin{cases} 1 & \text{if demands at node } i \text{ are served by a facility at node } j \\ 0 & \text{if not} \end{cases} \)

With this notation the p-median problem can be formulated as follows:

**MINIMISE**
\[
\sum_i \sum_j h_i d_{ij} Y_{ij} \quad (2a)
\]

**SUBJECT TO:**
\[
\sum_j Y_{ij} = 1 \quad \forall i \quad (2b)
\]
\[
\sum_j X_j = p \quad (2c)
\]
\[
Y_{ij} - X_j \leq 0 \quad \forall i, j \quad (2d)
\]
\[
X_j = 0, 1 \quad \forall j \quad (2e)
\]
\[
Y_{ij} = 0, 1 \quad \forall i, j \quad (2f)
\]
The objective function (2a) minimises the total demand-weighted distance between each demand node and the nearest facility. Constraint (2b) requires each demand node \( i \) to be assigned to exactly one facility \( j \). Constraint (2c) states that exactly \( p \) facilities are to be located. Constraint (2d) links the location variables \( X_j \) and the allocation variables \( Y_{ij} \). They state that demands at node \( i \) can only be assigned to a facility at location \( j \) (\( Y_{ij} = 1 \)) if a facility is located at node \( j \) (\( X_j = 1 \)). Constraint (2e) and (2f) are the standard integrality conditions.

For fixed \( p \), the \( p \)-median problem may be solved in polynomial time. Daskin observed that, despite the fact that the problem is \( O(N^p) \), the number of possible solutions that must be enumerated tends to become exceptionally large in practice (see [12] p. 203). In fact, the time required to solve all \( p \)-median problems by enumeration for \( p=1 \) to \( p=N \) for any given value of \( N \) is

\[
\sum_{j=1}^{p} \binom{N}{j} = 2^N - 1 = O(2^N)
\]

which is exponential in \( N \). Therefore, alternative algorithms, e.g. based on heuristics, need to be found in practice to solve problems of realistic size in a reasonable amount of time.

2.2.3 Formulation of our Network partitioning problem

A logical view of the system proposed in this thesis is illustrated in Figure 2.2a, where nodes may represent anything, from the software entities of a distributed system to networked devices. Figure 2.2b shows the resulting near-optimal partitions, as we will describe later (Section 4.2). In this case nodes are considered interconnected through commonplace IP-based networking infrastructure.
The partitioning problem can be formulated as follows:

Given a set of interconnected nodes (N) and an objective function (O), create \( p \) partitions and elect their partition centre nodes (one per partition) so as to minimize the objective function.

The requirements of the proposed system are as follows:

1. The algorithm should be computed in polynomial time with relatively low polynomial degree. The polynomial degree should be relatively low (ideally of first degree or at most of second degree) with maximum number of partitions, number of partition nodes, and network diameter. Some of the algorithms proposed in the literature are not accompanied with any computational analysis or do not satisfy this requirement.

2. The algorithm should admit upper-bounds on computational time. Only those tasks whose duration is reasonably longer than the duration of time needed to compute their location are beneficially implemented with MAs.

3. The algorithm should solve the problem for general network topologies. Some algorithms proposed in the literature solve the problem only for particular classes of topologies, such as tree networks.
4. The algorithm should solve the p-median problem for any given $p \geq 1$ and smaller than the number of network nodes. Some algorithms proposed in the literature can only tackle particular cases such as $p=1$ or $p=2$.

5. The algorithm should provide an indication of the 'goodness' of the computed locations. By having an indication of the distance from optimality of a given facility system it is possible to take the important decision of whether or not it is worth refining the location procedure.

6. The algorithm should not use the network matrix as input parameter. Most of the algorithms proposed in the literature are based on the principle that a complete map of the network (or network topology) – e.g. the network distance matrix – is available at the node where the location algorithm is executed. This is a major limitation towards scalability. As network scale grows the assumption that an up-to-date topological information is available at a central node becomes more and more unreasonable.

7. The algorithm should be computed in a distributed fashion. For scalability reasons it is unreasonable to have to try to solve a highly distributed problem using a centralised approach. This will result in heavy load at the main management entity, which will be the sole responsible entity for collecting and processing topological information. What is envisioned is an algorithm, which can be computed in a distributed fashion to limit the collection overheads and 'parallelise' computational load.

8. The algorithm should re-compute p-medians as conditions change. Partition centres should be continuously re-elected to maintain optimality as system conditions change (link failure, congestion, etc.).

Referring to the $p$-median problem previously introduced, the objective function is the Total Hop Distance (THD) in the partition, i.e. the sum of the distances between partition centres and their respective partition nodes, so that the partition centre is the most "central" node. Hence, the partitioning problem can be formulated as follows:

*Given a set of interconnected nodes (N), compute a near-optimal p-median that self-configures according to system dynamics and self-heals in case of failures.*
Following again the example of Figure 2.2a, the aim is to create near-optimal partitions with dynamically re-configurable partition centres. Assuming that partition centres communicate the same information to all the partition nodes, the configuration depicted in Figure 2.2b results in minimal overall traffic. In the face of topological changes due to failures or congestion, the system re-elects partition centres dynamically to maintain optimality (see Section 4.8).

2.3 Mobile Agents Technology

This section discusses some important features of MA technology. Code mobility mechanisms are analysed along with MAs' main advantages, open issues and applications (from [41]). An MA definition used in this thesis is then given.

2.3.1 Code Mobility Mechanisms

The code mobility mechanisms classification in this section is based on the paper "Understanding Code Mobility" [1], which is also used in [13], [15], [16], [17] and [18].

The idea of code mobility derives from the migration of active processes and objects at operating system level. Examples are in [19] and [20], where transparent process and object migration is supported. However, the transparency philosophy does not tend to be suitable for large-scale distributed systems. In contrast, location-aware programming based on Mobile Code expands this concept for large-scale distributed systems [1]. In this case large-scale distributed systems are intended as system networks composed of heterogeneous hosts, managed by heterogeneous authorities, connected by heterogeneous links. Location-aware applications may take actions based on the knowledge of the locations of the other application components. In contrast to operating systems, where mobility is used just for load balancing, mobile code systems are oriented towards service customisation, dynamic extension of applications functionality, and support for nomadic computing.

Continuing to follow the approach in [1], we shall give some definitions. The main difference between systems that support location-transparent migration at operating system level and MCSs (Mobile Code Systems) is that the former make use of a True Distributed System layer, which implements a platform where components that are in
different locations are perceived as local. An example is CORBA [4], where a programmer interacts only with a broker and is unaware of the network topology. On the other hand, in MCSs programmers know the underlying network composition through CEs (Computational Environments), which are layered at each host of the network. In this way the CE retains the identity of the host, which gives to application the capability to dynamically relocate computation on different hosts.

Fugetta et al. [1] define also the components hosted by the CE as EU (Executing Unit) and resources. EUs represent sequential flows of execution, such as single-threaded processes or individual threads of multi-threaded process. Resources are entities, which can be shared among multiple EUs, such as a file or an object shared by different threads. An EU may be modelled as a composition of a code segment, which provides the static description of a computation behaviour, and a state composed of a data space and an execution state. The data space is the set of references to resources that can be accessed by the EU. The execution state contains private data that cannot be shared, as well as control information related to the EU state, such as call stack and instruction pointer.

Therefore, Fugetta et al. define strong mobility as the ability of an MCS to allow migration of both the code and execution state of an EU between different CEs. Whereas, weak mobility is the ability of an MCS to allow code movement across different CEs. Code may be accompanied by some initialisation data but no migration of execution state is involved in weak mobility.

A mobility mechanism orthogonal to execution state migration relates to data space management. Upon migration of an EU to a new CE, its data space – i.e., the set of bindings to resources accessible by the EU – must be rearranged. This may involve voiding bindings to resources, re-establishing new bindings, or migrating some resources to the destination CE along with the EU. Fugetta et al. describes this particular aspect of code mobility in detail in [1]. This is not a central theme in this thesis and is therefore not discussed in greater detail.

Other forms of mobility have been defined in the literature, but they are not contemplated in this thesis.

To summarise, there are four main code mobility mechanisms or paradigms:
- **Process Migration.** Concerns the transfer of an operating system process from the machine where it is running to a different one. Process migration facilities manage the binding between the process and its execution environment and operate at operating system level. This has been used in loosely coupled, small-scale distributed systems to achieve load balancing across network nodes.

- **Object Migration.** Object migration makes it possible to move objects among address spaces, implementing finer grain mobility with respect to systems providing migration at the process level. In some cases object migration is achieved transparently, that is without user intervention or knowledge.

- **Weak Mobility.** Weak mobility is the ability of an MCS to allow code movement, along with its initialisation data, across different CEs. This is the first type of mobility supported in current MCSs, which is meant to target the requirements of large-scale distributed systems. This is the mobility mechanism exploited in the context of this thesis.

- **Strong Mobility.** Strong mobility is the ability of an MCS to allow migration of both the code and execution state of an EU between different CEs. This mechanism involves significant migration overheads related to the need to save the execution state along with the code. These overheads may be contained with discrete migration, i.e., if migration is triggered only at particular moments when the execution state is relatively small.

### 2.3.2 Mobile Agent definition

A software agent is an autonomous, proactive, and reactive computational entity that exhibits capability to learn, cooperate, and move. A Mobile Agent is a software agent that can move among locations. Typically, MAs are computational entities that act on behalf of some other software entity, exhibit some degree of autonomy, and are particularly featured with migration capability. In the context of this thesis MAs exhibit also cloning capability, that is the ability of agents to create and dispatch copies, or ‘clones’, of themselves.

In the proposed system, MAs (or simply ‘agents’ for brevity) support weak mobility and are used to realise strongly distributed hierarchical management. Agents are not
constrained to particular migration patterns and are not limited in the number of links they can traverse.

### 2.3.3 Advantages of MAs

Some of the advantages of MAs, as claimed almost unanimously in [21], [22], [23], [24], [25], [26] and [27], are as follows:

- *They adapt dynamically.* MAs can sense their execution environment and react autonomously to changes. MAs can distribute themselves among the hosts to maintain the optimal configuration for solving a particular problem.

- *They provide natural support for distributed computation.* MAs are inherently distributed and, as such, can be a fundamental enabler for distributed computation.

- *They potentially result in more scalable distributed applications.* Because they can be dynamically located and can maintain location optimality through migration, MAs can result in increased scalability.

The work described in this thesis intends to exploit the above advantages. In particular, the MA's ability to dynamically adapt to network environment and to increase the scalability of a distributed management application are considered.

### 2.3.4 MA Applications

Despite the advantages claimed for this technology, nobody has yet identified a single killer application for MAs. Nevertheless, many applications can benefit from using the MA paradigm. Examples are in [28], [1], [29], [30], [31], [24] and [26]. The following are used in this thesis:

- *Monitoring and notification.* An MA can locally monitor a given information source and notify specific events without being dependent on the system from which it originates.

- *Parallel processing.* MAs may create a cascade of clones and administer parallel processing tasks.
2.3.5 Issues associated to MAs

The previous two sections discussed advantages and applications claimed for MAs. However, they still pose several problems that require further study. Some of them are discussed in [28], [22] and [32]. This thesis tackles more closely complexity, performance and scalability of an agent-based distributed management system:

- **Complexity.** MA-based management systems are likely to be quite complex to design and debug because it will be difficult to determine their behaviour in a real, dynamic network environment. Whereas their strength (in comparison to the CS design paradigm) resides in easier implementation, deployment, and maintenance. Debugging a distributed system is difficult. An MA distributed system is particularly difficult because it involves mobile autonomous software entities whose behaviour is often determined by the environment in which they sit and by their perception of it. Although some work has been done on agent design [35], on architectural styles for agent distribution [36], and on decomposition patterns for mobile-code based management [37], the application of these design techniques to the field of management has not been investigated so far. Finally, it is not clear how agent-based, delegated management should be used to pursue a real automation of management itself.

- **Limited availability of quantitative performance evaluation.** The ability of the MA paradigm to result in increased performance, scalability and flexibility in comparison to the CS one has been claimed by many authors. Though acceptable in principle, this claim has not corresponded to a widespread application of the MA paradigm to the management arena. One of the reasons is that not enough work has been carried out to assess its strength in a qualitative fashion. Opponents of the MA paradigm suggest that, in fact, an alternative direction could be to enable RPC-based client-server interactions to match the advantages of MAs.

- **Migration overheads.** Agent migration involves overheads that need careful consideration. With today's platforms, migration time between two hosts is in the order seconds (see [38] and [39]); migration traffic depends on agent size and state and on the serialisation mechanism; finally, processing overheads are associated
with the serialisation and de-serialisation process. Further study is required to reduce migration overheads, which limit significantly the MA application domain.

- **Control structures.** Today's MA systems allow the creation and cloning of agents, while efficient mechanisms for controlling agent migration and termination have not been sufficiently investigated. Algorithms for agent location, termination and for orphan detection are discussed in [24]. Agent autonomous migration is a potential source of instability if the triggered mechanisms are not well thought out and fine-tuned. The stability of MA systems is another interesting subject, which requires further investigation. More generally, mechanisms to manage agent mobility in the context of integrated fixed and mobile networks are needed. A mobile networking environment is particularly dynamic as it involves a large number of simple, mobile terminals. Agent control mechanisms are particularly important to prevent instability and require further work.

The proposed system considers the existence of lightweight means for transporting and executing code around the network. Security issues for example do not apply in our case, as we will discuss in our viability study (Section 6.3).

### 2.4 MAs assessment tools

Due to their properties, MA-based systems and MA systems in particular are complex distributed software entities whose behavior, performance and effectiveness cannot always be anticipated by the designer. Their evaluation often presents various aspects that require a careful, methodological approach as well as the adoption of suitable tools. A first important question facing the designer is whether or not the MA approach is suitable to address the given system requirements. This can be assessed by looking at architectural aspects but also requires the identification of critical overheads that may impact the overall system performance, stability and validity. Once the MA approach has been found to suit the system, three alternative roads may be pursued towards system realization and evaluation: prototyping, simulation, or a mixed approach.

Each of these approaches contributes to unveil different, often complementary, aspects of the MA system. Prototyping has the main advantage of demonstrating the feasibility of the proposed approach as well as providing useful hints for improving the final
design/implementation. Measurements of overheads such as delays, code migration time, load incurred by MAs and so forth may be vital to fine tuning the system design and to anticipate the performance of the agent system that is later to be deployed in the large scale. The extent to which this evaluation can be carried out is however limited by the scale of the experimental test-bed.

On the other hand, simulations do provide a better means to assess performance and scalability and have the additional advantage of allowing the assessment of other important aspects such as the correctness, validity, robustness, and stability of the MA system under consideration. Simulations alone, however, often fail to capture important aspects of the real MA system since they must rely on simplified modeling aimed at containing simulation time and trace data.

A more effective MA evaluation method will therefore rely on a mixed experimental and simulation-based approach, which inherits the intrinsic benefits of both providing the necessary insight into the MA system. Such an approach requires the adoption of a suitable ‘hybrid’ framework, which supports the prototyping of agent code through specified Application Programming Interfaces (APIs), as well as facilitating the agent system evaluation via computer simulation.

The current approach to MA systems simulation is to create a simplified model of the system and realize software which simulates its behavior. In contrast, what has been developed for the evaluation of this thesis work is to design a hybrid simulation framework, which is generic – i.e. decoupled from any specific MA system – and addresses the requirements of complex, dynamic, large-scale MA systems. We developed a MA extension on top of the NS network simulator [77], which supports the most common physical, link, network, transport and application layer protocols for fixed and mobile communication networks. Having extended NS with support for MAs – including agent execution over simulated network nodes, agent migration, agent cloning, agent destruction and so forth – we have created an environment in which an arbitrarily complex MA system can be run over arbitrarily complex inter-networks. This can be used to evaluate the various aspects of agent systems under a variety of conditions such as network topology, size, communication protocols, etc.
This has been a major contribution for this thesis work, since to the best of the author’s knowledge only one other work with similar features is present in the literature (see Section 2.6.3).

2.5 Overlay Networks

In the context of this thesis overlay networks were used to experience with the proposed algorithm as will be shown in Appendix B.

Overlay networks are emerging as the mechanism of choice for introducing new functionality into the Internet [40]. An overlay network is a logical network implemented on top of a physical network. The overlay forwards packets in an application-specific way. Multiple overlays can be created on top of the same underlying network or nested one on top of the other.

Overlays are attractive for several reasons. High-level functionality is not typically implemented in the low-level network itself. There are lots of reasons for this: the end-to-end argument, the need/desire to keep the networking hardware and software as simple as possible, and eliciting cooperation from all of the competing interests on the network. Multicast is a good example of all the difficulties encountered for putting high-level functionality in the network. In an overlay, end nodes (or special intermediate nodes) can apply all sorts of very high-level solutions in order to implement new or improved services, hide faults at low levels, or improve the quality delivered by the low level.

Many advantages besides the main one of introducing new capabilities into the Internet can be provided by the use of this mechanism:

- **No need to deploy new equipment or modify existing software/protocols.** Overlays do not require any modification of the underlying network in terms of hardware and software. Usually, new software is needed on top of existing software.
- **Allows bootstrapping of new solutions.** Initially, new solutions are very expensive to develop in terms of hardware/software on large scale. Overlays allow bootstrapping new solutions by gradually deploying new hardware and software on a restricted number of nodes.

A simple classification for overlay networks is given in [40] as follows:
• **Routing Overlays.** This kind of overlay is used to perform alternative routing using IP-tunnels, which means that no application-level processing is made at overlay nodes.

• **Peer-to-Peer Networks.** These networks are made by individual nodes that organize themselves into a network without any centralised coordination. Usually used for file-sharing, their ability to scale to millions of users in a decentralised manner has attracted the interest of the research community, which is investigating different ways to apply this new paradigm.

However, we point out that the possibilities of using end systems that have capabilities far beyond basic operations of storing and forwarding open up the range of applications, which will lead to a wider classification.

Many examples of overlay networks are currently being developed either for research or commercial purposes. Some of them are illustrated in the Section 2.6.4.

### 2.6 Related work

As already stated, this thesis investigates solutions for the network-partitioning problem through the use of MAs, in which the weak mobility approach is exploited. This section focuses on work that is more closely related to the thesis.

#### 2.6.1 Previous work

This thesis was inspired by a previous work, *Towards Flexible and Scalable Distributed Monitoring with Mobile Agents* [41]. The use of code mobility for distributed systems and the employment of network information (i.e. routing tables – see Section 4.2 for more details) were the main ideas from which inspiration was taken to pursue the system' goals.

In [41] Liotta investigates solutions for scalable, distributed, and adaptable system management, which exploit the real essence of MAs. MAs encapsulate management logic delegated by other management entities. This delegated logic actually travels with the MA itself. The thesis worked out a comparative quantitative evaluation of the proposed agent-based monitoring approach against the conventional centralised, static-distributed monitoring solutions.
The differences with the work presented in this thesis are highlighted below:

- The previous work was focused on monitoring while the proposed system is meant for network partitioning for general network overlay applications (see Appendix B).
- Both systems use heuristic approaches but the heuristic functions are completely different.
- This thesis carries out a more accurate and extensive analysis of the system.

We can therefore conclude that the two studies are complementary and differ in scope, applicability and algorithms.

### 2.6.2 Location Problems approaches

This section covers existing location algorithms. In particular, as specified in Section 2.2.3 the proposed system deals with the p-median problem, where we want to divide the network in \( p \) sub-partitions and find their partition centre, which results in minimal incurred traffic by the system as a whole. In addition, being an NP-complete problem, the partition centre seeks a near-optimal solution computable in polynomial time.

Existing p-median algorithms are not able to meet all the requirements of the proposed system – i.e. they do not satisfy at least one of the properties enumerated below. In the following subsections we shall review those algorithms and show why they are not viable solutions to the problem formulated in Section 2.2.3.

In [7], location problems are divided into four different approaches:

- Enumeration
- Graph theoretic
- Heuristic
- Mathematical programming

An extensive survey of location algorithms is presented in [7], [8], [9], [10], [11] and [12]. The following sections discuss the most relevant approaches, analysing their advantages and limitations.
2.6.2.1 Enumeration

Enumerating all possible solutions to determine the optimal solution is a naïve approach, and for large networks the required computational effort is unwieldy. As mentioned in Section 2.2.2, the \( p \)-median problem for fixed \( p \) is \( O(N^p) \) in a network with \( N \) nodes, whereas the time required to solve all \( p \)-median problems for \( p=1 \) to \( p=N \) for any given value of \( N \) is (p. 203 [12])

\[
\sum_{j=1}^{N} \binom{N}{j} = 2^N - 1 = O(2^N)
\]

which is exponential in \( N \). This breaches requirement (1) (from Section 2.2.3) necessary for the \( p \)-median algorithm to be a viable agent location algorithm.

A number of polynomial algorithms characterised by low degree has been proposed to solve the 1-median and the 2-median problems. These meet requirement (1) but breach either requirement (2) or (3). For instance, Hakimi presented a simple enumeration procedure for determining an absolute median in a non-oriented network [42]. This approach is actually based on the distance matrix: the median can be found by summing each row of the weighted-distance matrix and choosing the vertex whose row sum is the minimum. This procedure takes \( O(N^2) \) operations to compute the distance matrix followed by \( O(N^2) \) operations to find the median. Hence this algorithm breaches also requirement (6).

Enumeration algorithms have also been proposed to solve the \( p \)-median problem optimally and in polynomial time on a tree network [44]. However, these do not meet requirement (3).

2.6.2.2 Graph Theoretic

Graph-theoretic approaches take advantage of the underlying network structure to determine the \( p \)-medians. These approaches have been successful only when the underlying network has a non-oriented tree structure [7]. In particular, Goldman has presented an algorithm that solves the 1-median problem on a tree in just \( O(N) \) steps [45]. Matula and Kolde proposed an algorithm which solves the \( p \)-median problem on a tree (where \( p>1 \)) in \( O(N^2 p^2) \) steps [46]; while an \( O(N^2 p^3) \) is presented in [44].
Therefore, graph-theoretic approaches proposed in the literature breach either requirements (2) and (3).

2.6.2.3 Heuristic

Heuristic procedures rely on intuitive trial-and-error methods and cannot guarantee an optimal solution, but they can be applied to any general network structure – i.e., they meet requirement (2). Heuristic procedures are especially useful when ‘good’ rather than ‘optimal’ solution is required. Some of them are based on intuitive notions about the optimal solution of the \( p \)-median problem, while others attempt to generate common-sense approximate solutions to exact mathematical formulations, mostly integer programming formulations. In fact, some of these heuristics are used to generate better initial solutions and/or branching decision rules to obtain faster convergence in the mathematical programming algorithms.

A comprehensive revision of heuristic algorithms is reported in [7], [8], [9], [10], [11] and [12]. Those algorithms belong to the following categories: node partitioning, myopic approach, node substitution, heuristic branch-and-bound, and improvement algorithms. The interested reader may refer to the above references for the details.

The revision of the algorithms proposed in the literature led to some important conclusions. Several algorithms proposed in the literature meet requirement (1), (2), and (3). The major problem of any heuristic approaches is that they do not provide a measure of ‘goodness’ of the solution – i.e., requirement (4) is not met. (See [12] p.221.). Most of them do not satisfy requirement (5) either. Requirement (4) is met in some algorithms. Finally, requirement (7) depends on the implementation. However it should be observed that none of the reviewed algorithms is oriented towards distributed computation, being mostly based on an input network distance matrix.

In conclusion it should be mentioned that the solution proposed in this thesis can be classified within the heuristic procedures.

2.6.2.4 Mathematical Programming

The mathematical programming approaches are based on an integer programming formulation of the \( p \)-median problem. A comprehensive review of mathematical
programming algorithms is reported in [7] and [10]. Because of the availability of several integer programming computer routines and the large theoretical base in mathematical programming, these approaches have attracted wide attention and have proved rather successful for general networks. The major impediment to using this approach for the agent location problem is that it relies on the network distance matrix – i.e. does not meet requirement (6). Moreover, despite the number of authors reporting computational times (see [47], [48], [49], [50] and [51]), there is not much literature on theoretical computational complexity of those algorithms.

Particularly relevant to this thesis are algorithms based on Lagrangian Relaxation (this technique is discussed in [52], [53], [54], [55], [56] and [12]). These are optimisation-based approaches based on relaxations of the integer-programming formulation of the $p$-median problem. When coupled with one or more heuristic algorithms, the Lagrangian approach often gives results that are provably optimal or very close to optimal [12] – i.e., it meets requirement (5). This is an extremely useful metric for establishing the distance from optimality of other heuristic algorithms. For instance, in this thesis we show by means of simulations that the proposed algorithm leads to a near-optimal solution of the $p$-median problem by showing that this solution is upper-bounded by the one achievable with a provably near-optimal Lagrangian algorithm presented in [12].

Nevertheless, Lagrangian algorithms are not viable to the network partitioning problem because they do not meet requirements (1) and (6). Computational times are reported in [55] and [12], which show that this approach is only suitable for moderate-sized networks.

### 2.6.3 Assessment tools

As mentioned in Section 2.4, assessment methodologies used to evaluate MAs systems generally make use of prototypes or simulations. The former demonstrate the feasibility of the approach, but is limited by the level of scalability reachable by the testbed. While the latter can reach higher level of scalability, but are limited by the level of granularity of the model used.

A better solution is to use a mixed mode, where benefits from both approaches are exploited. To the best of the author's knowledge, only one work is present in the
literature that uses this mixed approach, which is made by Shah in [57]. In order to realise this work Shah has built a model for simulating the general behavior of mobile agents by extending NS. Using this model, the performance of the mobile agents can be studied and compared with more traditional client-server approaches.

Shah's extensions could be used to realise this thesis work, since they provide the classical MA features such as creation, cloning, migration, destruction and so on. Unfortunately, those extensions were not present at the time this thesis work started – they have been released only recently.

### 2.6.4 Overlay Networks Examples

The literature presents many examples of Overlay Networks but the purpose here is to give only a brief panorama of this emerging mechanism rather than a detailed survey. The reader interested in an in-depth survey of overlay networks may refer to [40].

One of the first overlays implemented is MBone (Multicast Backbone) [58] [59], which implements IP multicast, allowing other applications to run on top of it. An example is **vic** [60], a video conferencing tool - for instance, IETF (Internet Engineering Task Force) meetings are usually broadcasted through the MBone [61].

Another example is RON (Resilient Overlay Network) [62], a project developed at MIT (Massachusetts Institute of Technology) and funded by DARPA (Defense Advanced Research Projects Agency). The objective is to improve the robustness and availability of Internet paths between hosts of today wide-area Internet routing infrastructure. In practice, RON nodes examine the condition of the Internet between themselves and the other nodes and they should forward packets whether indirectly or via other RON nodes.

PlanetLab is a good example of overlay network testbed for researchers that want to deploy new services [63]. Research groups are able to request a PlanetLab slice in which they can experiment with a variety of planetary-scale services, including file sharing and network-embedded storage, content distribution networks, routing and multicast overlays, QoS overlays, scalable object location, scalable event propagation, anomaly detection mechanisms, and network measurement tools. There are currently over 275 active research projects running on PlanetLab. The advantage to researchers in using PlanetLab
is that they are able to experiment with new services under real-world conditions and at large scale.

Commercial examples of Overlay Network are CDNs (Content Distribution Networks). A CDN is a network used to enhance the performances of Web servers (see Figure 2.3). The general idea of a CDN is to geographically distribute a collection of server surrogates that cache pages normally maintained in some set of backend servers (see Figure 2.3) [64]. Thus, rather than having every client trying to connect to a single server, it is possible to spread request load across many servers. Moreover, if a server surrogate happens to reside close to the client, the client's request could be served without having to cross a long network path (which reduces latency and congestion). Besides providing a large set of servers, CDNs also need to maintain a set of request redirectors, middleware entities that forward client requests to appropriate servers based on various strategies. These strategies will have a profound impact on CDN systems performance and overall robustness. CDNs are designed to improve two performance metrics: response time and system throughput.

Given a sufficiently widespread distribution of servers, CDNs use several, sometimes conflicting, factors to decide how to distribute client requests. For example, to minimise response time, a server might be selected based on its network proximity. In contrast, to improve the overall system throughput, it is desirable to evenly balance the load across a set of servers. Both throughput and response time are improved if the distribution mechanism takes locality into consideration by selecting a server that is likely to already have the page being requested in its cache. Although the exact combination of factors employed by commercial systems is not clearly defined in the literature, evidence suggests that the scale is tipped in favour of reducing response time.
So far the examples given used an overlay approach where the overlay nodes were statically assigned. We now introduce another type of overlay where nodes are dynamic, which means that they can join or leave the overlay network at any time. Examples are P2P systems. These have been used in the past as files exchange networks but their ability to scale to millions of users has inspired researchers to use the same approach for other purposes.

A P2P network example is Gnutella [65]. Gnutella participants arrange themselves in an overlay network, where each node knows only about a subset of other nodes. In order to find the desired object, Gnutella floods the overlay with the request, setting a TTL (Time To Leave) so that the query does not continue indefinitely. The good thing about this protocol is that it is completely decentralised. On the other hand, since it is an unstructured network it has the problem that does not scale well.

In contrast, structured approaches use a DHT (Distributed Hash Table), which allows more reliable and efficient object location. DHT is a distributed approach that maps objects with node IDs [40].

An example of such an approach is Pastry [66], a generic, scalable and efficient substrate for P2P applications. Pastry nodes form a decentralized, self-organizing and
fault-tolerant overlay network within the Internet. Pastry provides efficient request routing, deterministic object location, and load balancing in an application-independent manner. Furthermore, Pastry provides mechanisms that support and facilitate application-specific object replication, caching, and fault recovery. Pastry provides the following capabilities. P2P applications built upon Pastry can utilise its capabilities in many ways, such as mapping application objects to Pastry nodes, inserting objects, accessing objects, load balancing, object caching, scalable information dissemination etc. Similar projects are Chord [67], Freenet [68] and OceanStore [69].

2.7 Summary

This chapter examined the background work of the various topics involved in the thesis. Location problems were introduced along with the formulation of network partitioning problem that is addressed in the thesis. Then, code mobility was examined along with advantages, applications and issues related. Finally, overlay networks completed the background work section, showing their use as means for introducing disruptive technologies over the Internet.

The chapter then reviewed related work. The literature contains a number of approaches to solve the location problem, but none of them satisfy the combined requirements of scalability, optimality and flexibility in general large networks. This thesis argues that this is because most of them rely on the knowledge of the network matrix and are centralised in nature. The remainder presents a novel distributed algorithm that scales up linearly and is near-optimal on general networks.

We used our approach to experiment with some application that makes use of overlay networks. Therefore, we have also given relevant examples, showing various projects that apply this mechanism to introduce new services to the Internet. In addition, P2P systems were introduced as further examples of overlay networks.
Chapter 3 A Simulator Framework for Distributed Systems and Applications

3.1 Introduction

This Chapter describes the simulator framework used for the assessment of the thesis work. The simulator used is NS Network Simulator [77], which supports the most common physical, link, network, transport and application layer protocols for fixed and mobile communication networks. Unfortunately, NS does not support simulations that include mobile agent or mobile code. Therefore, an extension was required to include the necessary features.

Having extended NS with support for MAs — including agent execution over 'simulated' network nodes, agent migration, agent cloning, agent destruction and so forth — an environment, in which an arbitrarily complex MA system can be run over arbitrarily complex inter-networks, has been created. This can be used to evaluate the various aspects of agent systems under a variety of conditions such as network topology, size, communication protocols, etc. Consequently, the network-partitioning algorithm has been developed using the MA extensions provided to NS.

This chapter will introduce the network simulator and its main characteristics. Following, the MA extensions implemented are discussed in detail along with further extensions provided to the simulator. Finally, discussion and conclusions complete the chapter.

3.2 The NS-Network Simulator

NS is a network, discrete-event simulator that runs in a non-real-time fashion and supports the most common physical, link, network, transport and application layer protocols for fixed and mobile communication networks.
Being an open source, over the last few years its extensive use in the networking research community has contributed to make it a valuable tool for studying, improving, and introducing new protocols.

In addition to fixed networking, NS supports wireless and satellite networking, unicast and multicast routing, centralized and hierarchical routing, static and dynamic routing. Extensions to the core NS have been provided by researchers worldwide, including support to GPRS, Mobile IPv6, Bluetooth, RSVP, Differentiated Services, MPLS, active networking, IEEE 802.11 for WLAN, multi-hop wireless ad-hoc networking, Cellular IP and UMTS [77]. NS allows the characterising of point-to-point bi-directional links through bandwidth, delay, and queue type. It also allows modelling of packet scheduling (i.e. the decision process used to choose which packet should be serviced or dropped) and buffer management (i.e. any particular discipline used to regulate the occupancy of a particular queue). NS includes support for several algorithms such as drop-tail (FIFO) queuing, RED buffer management, and different variants of fair queuing. There is also literature on the experimentation of various queuing disciplines for Quality of Service..

Figure 3.1 NS functional architecture.
management, active queue management, and stochastic queue management (refer to [77] for documentation).

In addition to the extensive networking support, as summarised above, NS enjoys the advantages of an open software tool (the source code is freely available for experimentation) and of an architecture that can be easily extended. The user specifies the simulation scenarios in OTcl, an object oriented extension to Tcl [78]. These are interpreted in the NS Kernel that includes the above networking functionalities. For increased efficiency, the NS core functionality is implemented and executed in C++. In order to ease the use of the simulator, however, most C++ classes are mirrored into OTcl and can be directly invoked by OTcl scripts (see Figure 3.1). The simulator functionality may be simply enhanced by creating new C++ classes, adding new functions to the NS library, and mirroring the new code into OTcl classes.

Two important tools come with NS, the Network Animator (NAM) [79] and the GT-ITM topology generator [80]. NAM generates graphical animations of trace files generated by NS simulations including the relevant network events. The tool is extremely useful for verifying the correctness of the simulations.

Finally, the topology generator is used to automate the process of generating networks for the purpose of assessing the protocols and systems under scrutiny. GT-ITM generates realistic Internet-like topologies as well as random topologies following a well-established methodology [81] [82].

3.2.1 Software Architecture

NS is a discrete event-driven network simulator written in C++ and OTcl. In order to provide flexibility and performance, NS uses the split-programming model, which is distributed between two languages (C++ and OTcl). Specifically, the low-level event processing or packet forwarding through simulated router requires high performance and does not require frequent modifications. Therefore, it is implemented in C++. In contrast, tasks such as the dynamic configuration of protocol objects to experiment with different scenarios go through frequent changes as the simulation proceeds. Thus, they are realised in a flexible and interactive scripting language, namely OTcl. As a consequence, the core
is implemented in C++, to give high performance, while configuration and control of the simulation are made through script language Otcl, which gives the needed flexibility.

### 3.2.2 C++ - Otcl Linkage

NS provides a two-class hierarchy: a compiled one in C++ and an interpreted one in Otcl. The user sees a correspondence between a class in the interpreted hierarchy and a class in the compiled hierarchy as illustrated in Figure 3.2 [83]. *TclObject* is the root of this class hierarchy. Simulator objects are created through the interpreter, which in turn are mirrored by a corresponding object in the compiled hierarchy. Methods defined within the class *TclClass* are responsible to create the interpreted class hierarchy while methods defined in class *TclObject* are responsible to mirror user instantiated objects. C++ and Otcl linkage is maintained by the following classes:

- **Class Tcl.** The Otcl interpreter instance is encapsulated within this class, which provides methods to access and communicate with that interpreter, such as reference to Tcl instance, invoking Otcl procedures through the interpreter, getting or passing results to the interpreter, storing and looking up "TclObjects" etc.

- **Class TclObject.** It is the base class for most classes in the interpreted and compiled hierarchies. The user from within the interpreter creates every object within the class *TclObject* and an equivalent shadow object is created in the compiled hierarchy, through the class *TclClass*.

- **Class TclClass.** This is a pure virtual compiled class. Derived classes provide two functions: construct the interpreted class hierarchy to mirror the compiled class hierarchy, and provide methods to instantiate new *TclObjects*.

- **Class EmbeddedTcl.** The objects of this class are responsible for loading and evaluating some NS scripts that are required at initialization.

- **Class InstVar.** This class defines the methods and mechanisms to bind a C++ member variable in the compiled shadow object to a specified Otcl instance variable in the equivalent interpreted object. This binding allows using the variable at interpreted or compiled level.
3.2.3 Class Hierarchy

This section gives an overview of the NS class hierarchy rather than a detailed explanation of all the classes, which would be out of scope for this thesis. The NS class hierarchy model is shown in Figure 3.3, including the main components [83]. The TclObject class is the root of the hierarchy and is the super class of all OTcl library objects. NsObject derives from TclObject and is the superclass of all basic network objects handling packets. It can be used to create complex network objects such as nodes and links.
The basic network components are further divided into two subclasses, Connector and Classifier. The former contains the basic network objects that have only one output data path, while the latter contains the switching objects that have multiple possible output data paths.

3.3 Design and Implementation of Mobile Agent Extension

3.3.1 Assumptions

The main purpose of the MA extensions is to provide an environment that allows the implementation and assessment of the algorithm developed in this thesis (Chapter 4). In this respect, the MA model was developed with basic MAs functionalities. The following assumptions were made:

- The MA is sent through the network as a packet of whose size is configurable – this has been set to 100 Kbytes, a value obtained by implementing the MAs in OTcl.
- Since the agent carries parameters that are subject to small variations, the agent size is assumed not to vary during the simulation.
- It is assumed that the agent-partitioning task involves a processing time that is negligible compared to the MA serialization and deserialization times – this
assumption derives from practical measurements performance on real MA platforms [84].

- **UDP** is used as the underlying transport layer protocol.
- No contemplation is made to security issues. Thus, no security overheads are considered.
- *Only weak migration* is considered. Therefore, agent status is not taken into account in the migration model.

It may be worth mentioning that the way the model was implemented allows easy extensibility, which may be useful to further generalise the framework.

### 3.3.2 MA Class Inheritance

In NS, Agents are entities that represent endpoints where network-layer packets are constructed or consumed and are used in the implementation of protocols at various layers. They can be considered as network interfaces, whose task is sending and receiving packets. For example TCP and UDP transport protocols are implemented using agents. In this case, a separate object representing the demands of an application may control the distribution of packet sizes and/or inter-departure times. Therefore, agents are equipped with an application programming interface (API) that allows interactions with the application. Instead, for agents used in the implementation of lower-layer protocols (e.g., routing agents), protocol messages within agents determine size and departure timing. Considering this approach, the MA model was implemented in two different levels: a **MAgent** derived from the **UDP Agent** used as network interface and a **MAgentApp** derived from the class **Application** to control the distribution of packets and to implement the MA logic. Figure 3.4 shows the MA model class inheritance.
3.3.3 MA Interaction model

Figure 3.5 shows the interaction model. The MAgent, as said, is the interface with the network. When the MAgent is created an MAgentApp is created and attached to the MAgent. After that, the MAgent is attached to a node, which acts as its host. Thus, the MAgent can send and receive packets using the methods send() and recv(). These are inherited from the NS Agent class. While send() did not need overwriting, because the format of the packet to be sent is always the same, recv() needed to be overwritten. This was done to allow the recv() method to receive the packet, extract the data, interpret the data based on the different packet header flag and pass it to the MAgentApp. The method Process_data() is used to pass the data to the MAgentApp, while Send_data() is used to pass the data with different instructions to the agent, after which the MAgent creates a packet, inserts the data and sends it to the actual destination.
3.3.4 Implementation

MA extensions are depicted in bold in Figure 3.6. The new MAgent class added at NS kernel level was then mirrored into an equivalent OTcl class. MAgent has all the necessary methods to create, run, stop, clone, migrate and destroy an MA. These methods are exposed at user level as an Application Programming Interface (API), which is used to write the necessary MA code using the OTcl scripting language. In this way, one can easily write a script that creates one or more agents, each of which incorporates a specific task, building an MA distributed system.

Agent creation involves the instantiation and initialization of an MA object and its association to a network node. The agent code is executed in the network simulation environment through the following stratagem, as shown in Figure 3.7. Agents are allowed to execute their code in real time as far as they do not involve any communication with other distributed entities or agents. In other words agents can process data that is available locally and take decisions that do not incur any network load. However, as soon as agent processing involves network resources – e.g. the agent needs to send or receive packets or wishes to migrate to a different node – the relevant network events are triggered into the network simulator that handles them accordingly.
Figure 3.6 Functional Architecture of NS with MA extensions.

Figure 3.7 depicts an example of how agent migration is handled by the simulation environment. As the agent resides at node $A$, the local resources accessible by the agent are those of node $A$. If the agent wishes to migrate to node $F$, agent execution is suspended and the agent code is serialized for transmission. At this point a packet having size equal to the serialized agent is scheduled for transmission from $A$ to $F$ – i.e. a network event is added to the event queue of the simulator. The simulator according to the networking protocols set – i.e. layers 1 to 3 protocols, handles this event. Finally, as soon as the packet is received by node $F$ the simulator generates an appropriate packet_received event. This event instantiates the MA in the execution environment, setting up the NodeID parameter to $F$. This means that when the agent execution is resumed, its local resources will be those of node $F$. 
3.3.4.1 The MAgent class

In this section the main methods implemented within the MAgent class are discussed. When an MA is created a MagentApp is created as well and attached to it. Then the MA is attached to a node. After that, the agent performs setup and activation actions as follows:

- **set_up.** This method is used to set-up all the agent parameters.
- **activate.** This method invokes the process_data method of the MAgentApp attached to the MA, passing the parameters set within the set_up phase in order to start the MA tasks.

Cloning and migration are the two main actions performed by an MA within the proposed system:

- **create_clone.** This method creates a new MA, attaches it to the destination node (where the new MA is supposed to start its computation), connects the actual MA to its clone and sends a packet of the agent dimension that carries the data needed. The cloned MA receives the packet, extracts data and, based
on the content of the packet header flag, activates a new computation at its node.

- **migrate.** In this case the migrating MA creates a *support_agent* to the destination node. The two agents get connected and a packet reflecting the agent size is sent. At the destination, the support agent detaches the first agent from the previous node, attaches it to the new node and forwards the data to the first agent. The *support_agent* is also used for further migrations.

- **recv.** The receive function handles all the packets depending on the header flag used and passes them to the *process_data* method of the *MAgentApp*.

### 3.3.4.2 MAgentApp

This class implements the actual logic of the MA, that is the algorithm described in Chapter 4. General methods are as follows:

- **run.** This method is used to start the MA task.
- **stop.** This method is used to stop the MA task.
- **destroy.** This method is used to destroy an MA. The actual NS object is deleted.

### 3.4 Further Extensions to NS

In this section further extensions provided to NS are explained. The explanation regards just what they can do and how they have been implemented, while it is not explained at the moment for what and how they have been used (this is the subject of Appendix B).
Figure 3.8 Resource node architecture for NS.

Figure 3.8 shows the model used to simulate system resources, which allows executing a job request locally or remotely. A job request specifies the amount of disk, cpu cycles and a priority needed to execute a particular task. This is intended to simulate a task execution.

The blocks within the resource manager represent the entities that address the resource management algorithm tasks that will be explained in Section 0. The admission control block uses the computational resources demand and checks for resource availability at the node. In case of positive answer it accepts to execute the request locally otherwise the request may be refused and a resource discovery procedure may be triggered.

The Disk is a buffer managed as a FIFO (First In First Out) discipline. However, the CPU, after getting a job request from the Disk, will simulate the job execution by waiting for a time equal to the one resulting from the job request.

The same approach used to implement the MA was used to implement the resource node. The resource node was split in two different classes, a resource agent called ResourceAgent deriving from the NS Agent class and an application called ResourceServer deriving from the NS Application class. The ResourceAgent was used as the network interface, while admission control, cpu, disk and allocation strategies were implemented within the ResourceServer. When a ResourceAgent is created a ResourceServer is created automatically and attached to the ResourceAgent. Then, the ResourceAgent is attached to a node making this node resource-capable and therefore able to run job requests.
In order to create and send job requests a further entity was developed, which is called ServiceAgent. Its task is to create a job request and send it to a resource node for execution.

At user level an Application Programming Interface (API) is also provided in order to instantiate resource nodes and ServiceAgents.

In the next subsections further details on the classes are given.

### 3.4.1 The ResourceAgent class

The ResourceAgent manages sending and receiving of all packets, while the ResourceServer carries out all computations. This agent uses a class, ResData, which extends the NS AppData. ResData embeds an instantiation of the struct resource_data, which contains the information about current requests (service agent owner of the request, disk required, cpu required etc.).

```python
# color_ : int
# comprdata_ : ResData
# comprsdata_ : ResData
# disrdata_ : ResData
# monag_ : name_node
# odata_ : OfferData
# offerdata_ : OfferData
# offersdata_ : ResData
# rdata_ : ResData
# rdatal_ : ResData
# remote_comprdata_ : ResData
# remote_jserver_ : ResData
# rdatal_ : ResData
# up : UpdateOverlay *
# verbose_ : int
# wireless_ : int

+ ResourceAgent()
+ command(argc : int, argv : const char * const *): int
+ recv (Packet * : Handler *): void
+ recv_data(rdata : res_data): void
+ res_discovery(rdata : res_data): void
+ send_offer(ra_off : name_node, rdata : res_data): void
+ send_reply(rdata : res_data, disk_offer : double, wait_time : float): void
+ ~ResourceAgent():
```

Figure 3.9 ResourceAgent class.

Some of the methods within the ResourceAgent are as follow:

- `recv`. This method receives packets, extracts the data within the packet and passes the data to the ResourceServer.
• *res_discovery*. This method initiates a resource discovery procedure when job requests are refused by the admission control module.

• *send_reply*. This method is used to send its offer for a request received during a resource discovery procedure.

• *send_offer*. This method is used to send the job request to the chosen server after the choosing strategy has been applied.

### 3.4.2 The ResourceServer class

This class provides the functionality to perform admission control, jobs scheduling and jobs processing. Five different *ResourceServer* classes have been created, which give different processing capabilities (see Table 3-1).

<table>
<thead>
<tr>
<th>Server Class</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU clock</td>
<td>1</td>
<td>1.5</td>
<td>2</td>
<td>2.5</td>
<td>3</td>
</tr>
<tr>
<td>Number Disk Units</td>
<td>100</td>
<td>150</td>
<td>200</td>
<td>250</td>
<td>300</td>
</tr>
</tbody>
</table>

*Table 3-1 Resource Server classes.*

As will be explained later (Section 3.4.2.3), since a job can specify the number of cpu cycles needed, the same job may require a different execution time on different resource nodes. The main methods of the *ResourceServer* are illustrated below:

• *process_data*. This method performs admission control when jobs request processing resources. It checks if the request of disk can be satisfied according to the local disk availability. Also it checks job priority (which is a fraction of the time needed by the job to be executed) against the maximum waiting time that the job has to wait for before being processed. The latter means that the *ResourceServer* estimates the time necessary by the *CPU* to process all the jobs stored in the *Disk* before the new request can be processed.

• *check_priority*. This method is used during admission control to check that the job priority is met against the waiting time needed before the CPU can process the job. Five different priorities can be specified by a job (see Table 3-3).
• **get_cpu_time.** This method allows getting the time needed by a job requiring a given number of cpu cycles. The cpu time is calculated as:

\[
\text{CpuTime} = \frac{\text{CpuCycles}}{\text{CpuClock}}
\]

where the CpuClock depends by the ResourceServer class as illustrated in Table 3-1.

• **get_disk_wait_time.** This method is used during the admission control procedure to check the waiting time before a job can be processed by the CPU. It basically checks all the jobs in the disk queue, summing up all the individual execution times required.

• **res_enq.** This method is used to query the ResourceServer during the resource discovery procedure.

```cpp
+ agent_ : ResourceAgent *
+ ch : char []
+ color_ : int
- cpu_ : CPU *
- disk_ : Disk *
- out : char []
+ server_class : int
+ verbose_ : int
+ wait_queue : vector
- wait_time : float
+ wireless_ : int

+ ResourceServer()
+ add_offer(iod : offer_data *) : void
+ check_offer(id : const char *) : void
+ check_priority(priority : int, cpu_time_required : float, estimated_disk_wait_time : float) : bool
+ command(argc : int, argv : const char *) : int
+ end_req(id : const char *) : void
+ exec_job(argc : int) : void
+ exec_offer(index : int) : void
+ first_offer(iod : offer_data *) : void
+ get_cpu_time(cpu_cycles : float) : float
+ get_disk_wait_time() : float
# print(str : char *) : void
+ print_data(id : ResData *) : void
+ res_enq(id : ResData *) : void
# reverse_bind() : void
+ set_cpu_status(flag : int) : void
+ set_server_class(s_class : int) : void
+ update_cpu_size(size : int) : void
+ update_disk_size(size : int) : void
+ update_wait_queue(id : ResData *) : void
+ ~ResourceServer() ;
```

**Figure 3.10 ResourceServer class.**
3.4.2.1 The Disk class

This class simulates the resource Disk and it is modelled as a vector that accepts jobs up to its maximum size. Therefore a job accepted by the ResourceServer is saved into the vector and the disk size is updated.

![Disk class diagram](image)

While when a job is executed it is removed from the vector and the disk size is updated accordingly.

3.4.2.2 The CPU class

The CPU class simulates the resource CPU. The execution of a request consists of scheduling an NS event that calls the method that actually performs the job.

![CPU class diagram](image)
3.4.2.3 The ServiceAgent class

The ServiceAgent class extends the NS UdpAgent class. The latter creates jobs and sends them to resource nodes that execute them locally according to job requirements and local resource availability. Otherwise a resource discovery process is initiated.

![Figure 3.13 ServiceAgent class.](image)

A job is composed by a class, which specifies the number of CpuCycles and DiskUnits, and a Priority (see Table 3-3). The latter gives a percentage of CpuTime that a job can wait for before being processed. For example if the CpuTime obtained is 1 sec and the priority is 1, the job can wait up to 10% of CpuTime which means 0.1 sec. This value is therefore compared with the waiting time needed before being processed by the cpu. In case the ResourceServer cannot meet that specific priority, the job is not accepted for execution.

Five different job classes can be created, depending on the job requirements. Table 3-2 shows these job classes:

<table>
<thead>
<tr>
<th>Job Class</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>CpuCycles</td>
<td>0.5±1</td>
<td>1.01±1.5</td>
<td>1.51±2</td>
<td>2.01±2.5</td>
<td>2.51±3</td>
</tr>
<tr>
<td>DiskUnits</td>
<td>1±10</td>
<td>1±15</td>
<td>1±20</td>
<td>1±25</td>
<td>1±30</td>
</tr>
</tbody>
</table>

Table 3-2 Job classes.
In this way, jobs of same classes require different execution times in *ResourceServers* of different classes.

Table 3-3 shows the job priority values, which are from 1 that gives the highest priority to 5, which gives the lowest.

<table>
<thead>
<tr>
<th>Job Priority</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>% of cpu time</td>
<td>10</td>
<td>25</td>
<td>50</td>
<td>75</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 3-3 Job Priorities.

### 3.5 Summary

This chapter described the NS network simulator used as the bases for the thesis assessment and the extensions realised by the author. Despite the wide range of functionalities intended to address general networking demands, NS does not provide any support to Mobile Agent systems and does not include models accounting for resource node capabilities. The MA model implemented has been explained along with the assumptions made, class inheritance and methods. The NS extensions can be used to develop and study generic MA-based systems on top of NS. Exploiting the NS networking capabilities the system was assessed under different scenarios, capturing diverse aspects that influence the system performances.

Regarding the resource node extensions, the general use was given, but no description regarding their employment was specified since this is the subject of Appendix B.
Chapter 4  Network Partitioning Algorithm

4.1 Introduction

This chapter presents the core contribution of the thesis, that is a novel network-partitioning algorithm for dynamic, large-scale systems. As dynamic we intend topology changes, for example links faults.

Our approach relies effectively on information produced by distributed IP routing algorithms, i.e. Dijkstra or Bellman-Ford, which reflects the current state of the network in terms of connectivity and specifies the minimum distance from any node to remote destinations based on a given route metric. The latter may be link usage, in which case congestion results in nodes appearing as further away because of increased “distance” from the current node. Nodes continuously compute routing tables using relatively simple operations, resulting in a global distributed database that drives packet delivery from sources to destinations. The information required by our algorithm is knowledge of the next-hop neighbors at any node and knowledge of the minimum distance, i.e. shortest path, to remote destinations; this information is available at every node through the SNMP IP routing MIB (Management Information Base) [84].

In the remainder of this chapter we shall demonstrate how this information can be used by relatively simple heuristic procedures in order to solve the partitioning problem formulated in Section 2.2.3.

4.2 Algorithm overview

Our approach is to use a Mobile Agent (MA) system that actually computes the partitions following the 5-phase process depicted in Figure 4.1. The process can start at any node (see Section 5.5 for our study on its independence from starting node) by “injecting” a single MA. The starting node is also the initial partition centre of the whole system to be partitioned, so effectively there is only one partition initially that contains a list of \( N \) nodes. The agent is set-up with a goal in the form of a simple heuristic procedure that drives the partitioning process. The input parameters are:
- The list of nodes, $N$ to be partitioned (this is a list of node identifiers, e.g. IP addresses);
- The number of target partitions, $p$;
- The tolerance margin admissible for $p$, i.e. $\varepsilon_p = \left(\frac{Ap}{p}\right) \times 100$;
- The initial location of the partition centre, $i$, i.e. the starting node.

Figure 4.1 Algorithm Overview.

Starting from its initial location (i.e. the root node), the MA initiates an iterative, distributed procedure that subsequently creates new partitions and clones new agents (one per new partition). Every MA has exactly the same logic, i.e. they all go through the process depicted in Figure 4.1 and they all behave in the same fashion. We provide below an overview of the 5-phase procedure performed by each agent: set-up, cloning, migration, aggregation and self-healing. Further details of the latter four phases are given in subsections 4.4, 4.5, 4.6 and 4.8, respectively. Figure 4.1 shows that cloning, migration and aggregation result in the algorithm starting again, with new parameters.

Our approach is to obtain the required partitions through a clone-and-migrate process. The agent performs a "matching operation" between the local routing table at the current
node and the list of nodes included in its partition. This operation identifies first the next-hop neighbours and for each of them finds which of the rest of the partition nodes are reachable through it. The MA can take five different decisions:

1. **Set-up.** Initialise the MA with the initial parameters.

2. **Cloning/Partitioning.** If the partition has a size considerably larger than \(|N|/p\) and the MA is not in the edge of the partition, the MA splits the partition into two or more partitions; clones one agent per partition; initializes them with their respective node list; sets up their respective initial partition centre location; and spawns them. Each new agent initiates the whole process from the beginning, the only difference being the list of partition nodes and the initial partition centre location.

3. **Migration.** If the partition has size larger than \(|N|/p\) and the agent is at the edge of the partition it migrates within a more central location (this is to help the cloning phase as will be explained in Section 4.5). If the partition has a size comparable to \(|N|/p\) it starts migrating within the partition until the central point is found. The metric for centrality may vary – i.e. herein we consider the median.

4. **Aggregation.** In some cases, the cloning/partitioning process may not succeed in creating partitions of size comparable to \(|N|/p\). This happens for instance when partition nodes are sparsely distributed. In order to meet the requirements on the target partition size, the agent creates an appropriate number of sub-partitions by subsequently aggregating nodes on the basis of their distance from the partition centre.

5. **Self-healing.** Once the partitioning process is complete and the partition centre has been found, the MA gets into self-healing mode. It periodically re-checks the network conditions to re-enter the migration mode when these change, e.g., due to link failure or congestion, which will result in different values obtained from the routing table at the partition centre node.

---

5 The operator || returns the cardinality of a list of nodes, e.g. \(|N|\) is the number of elements of \(N\)
A simple example showing the algorithm in operation is in Figure 4.2. After the first agent is created, it gets as initial parameters, the list of nodes to be partitioned (6, 7, 9, 12, 15, 17), the number of partitions wanted (3) and the initial starting node (0) (Figure 4.2 - A).

At this point, the agent reads the local information available at node 0, performs a heuristic and decides to create 2 new clones at node 2 and 13 respectively and split the list of nodes with the new clones. It also decides to migrate itself to node 16 (Figure 4.2 - B).
- B). From now on the agents continue the partitioning process in parallel and independently from each other. This is the key property of the algorithm, the ability to parallelise the partitioning process. Now, the agents by reading local information and performing a heuristic decide that there is no need to create new clones and therefore migrate towards a central location within their respective partitions (Figure 4.2 – C). Eventually, they reach the optimal location and settle down (Figure 4.2 – D). At this stage the agents keep sensing the network changes and in case some link fault occurs (Figure 4.2 – E) they are able to relocate themselves to keep the optimal location within their respective partition (Figure 4.2 – F).

The following subsections describe the agent behaviour in greater detail, including parameters, thresholds and heuristics used in the MA decision process, aiming to offer insight into the algorithm operation in the 5 different phases.

### 4.3 Parameters and definitions

In this Section we specify the notation used in the remainder of the thesis.

Starting from the aforementioned input parameters $N, p$ and $e_p$, the agent computes the following variables during the set-up phase:

- $n = \text{round} \left( \frac{|N|}{p} \right)$ is the average number of nodes per partition.
- $\Delta p = \left( \frac{p \times e_p}{100} \right)$ the absolute variation admitted on $p$.
- $n_{\min} = \text{round} \left( \frac{|N|}{p + \Delta p / 2} \right)$ is the min value of $n$.

The following parameters are used in the cloning and migration phases:

- $N_i$ is the list of partition nodes “managed” by an MA. At bootstrapping time $N_i = N$.
- $N_i$ is the list of partition nodes reachable through the neighbor node having identifier $i$. 

---

52
4.4 Cloning Phase

The cloning process that results in the generation of new partitions is depicted in Figure 4.3. At this point the MA resides at a certain node (the tentative partition centre), holds the list of partition nodes, $N_i$, and has to decide on how to partition it. By "matching" $N_i$ against the local routing table as described, the MA discovers the next-hop neighbor nodes and partition nodes reachable through each of them.

The decision on whether or not and how to partition $N_i$ is controlled by the three conditions specified in Figure 4.3. The first condition makes sure that the algorithm considers all next-hop neighbor nodes.

The second condition ($|N_i| \geq 2 \times n_{min}$) prevents partition fragmentation. In fact, we do not want the partitioning process to result in partitions that do not have at least $n_{min}$ elements. We also need to prevent the situation in which the original partition is depleted excessively by the partitioning process.

Having gone through the previous conditions, we know if the partition under scrutiny is too big. What we have to decide is whether or not the sub-partition $N_i$ is large enough
but we also have to make sure that, should \( N_i \) form a new partition, the remaining nodes in \( N_i \) – i.e. \( \{ N_i / N_i \} \) – would still constitute a sufficiently large partition. The latter holds when \( |N_j| - |N_i| \geq n_{min} \), hence the inequality in the third conditional box.

If the third condition holds, \( N_i \) is partitioned into two different partitions, \( \{ N_i / N_i \} \) and \( N_j \). The main thread of execution continues the process using the former list and clones a new MA whose task is to take care of the new sub-partition \( N_j \).

Therefore, there are two types of exit conditions from the cloning phase:

- One or more new MA clones/partitions are created, depending on the cloning conditions. Each MA is spawned to its initial location that is set to the relevant neighbor node, \( i \).
- Once all the branches of the routing tree rooted at the agent node have been assessed, the MA leaves the cloning phase and enters the migration phase, as explained in the next section.

### 4.5 Migration Phase

This phase has two main goals. The first is to help the cloning phase. This is executed when the second condition of the cloning phase is verified (\(|N_i| \geq 2 \times n_{min} \), the agent has a partition big enough to be split), but the agent resides at the edge of the partition (\(|N_i| - |N_e| < n_{max} \), the third condition of the cloning phase is not verified). In fact in this case the agent will consider the migration and upon verifying the migration conditions migrates toward a more central position in which the third cloning condition could be verified (\(|N_i| - |N_j| \geq n_{min} \)). The second goal of the migration phase is to reach a \( p \)-median location within the partition. This occurs when the second condition of the cloning phase is not verified (\(|N_i| < 2 \times n_{min} \)).

---

\(^4\) The operator / returns the list of nodes resulting when removing list \( N_i \) from list \( N_e \).
Starting from its current location, the principle that drives an agent towards its central location (within the partition) is based on the calculation of weighed routing costs ($E_i$) for each of the neighbor nodes ($i$), defined as:

$$E_i = \frac{C_i \times |N_i|}{C_i \times |N_i|}$$

where $N_i$ is the list of partition nodes; $N_i$ is the subset of partition nodes reachable through neighbor $i$; $C_i$ is obtained by adding the individual routing costs of $N_i$; and $C_i$ is the sum of all $C_i$. $E_i$ provides an estimate of the global distance (in terms of routing costs) associated to each neighbor. The aim is to migrate towards the location with the highest associated cost. So the agent elects as new candidate location the neighbor $i$ that has the maximum value of $E_i$ (first box in Figure 4.4).

Agents avoid migrating in loops by retaining the highest value of $E_i$ for all previously visited nodes. So, as soon as the agent is presented with a candidate node it has previously visited, it detects a potential looping condition. In that case, it simply elects the node having the highest value of $E_i$ (among all historical values, including that of the new candidate node).

At this point we have two alternatives. If the newly elected node is in fact the node where the agent resides, the agent goes to the aggregation phase (described below). Otherwise, it first disables the migration and cloning flags before migrating to the target node, since it “knows where is going”. In this way it makes sure that upon reaching the new location (and re-starting the whole procedure) these phases are skipped, getting directly into the aggregation phase.
4.6 Aggregation Phase

We have seen how the cloning process is based on simple heuristics, which means that despite having a relatively simple and computationally efficient algorithm, we also have to cater for exceptional conditions. In practice, we have found rare occasions in networks with highly sparse nodes that lead to a small number of partitions that are considerably larger than $|N|/p$ but still do not satisfy the cloning conditions of Figure 4.3.

In order to meet the requirements on the target partition size, the agent creates an appropriate number of sub-partitions by subsequently aggregating nodes on the basis of their distance from the partition centre (Figure 4.5). Each newly formed partition sparks the cloning of a new agent whose cloning flag will be set to ‘disabled’ since the size of partition already meets the requirements. On the other hand, upon exiting the partitioning loop, the main agent now has a minimal partition but should again get into migration mode to consider better locations.
Figure 4.5. Aggregation phase flow-chart.

Figure 4.6 shows an example that illustrates a situation where the aggregation phase is needed. In case an agent reached node 7 with the list of nodes to be partitioned comprising all the nodes in the figure and $n_{\text{min}}$ is 7, the agent meets the cloning condition $|N| \geq 2 \cdot n_{\text{min}}$, but for each neighbour we have the condition $|N| \leq n_{\text{min}}$ which does not allow to create a new agent.

Figure 4.6 Aggregation example.
Therefore the agent is in a situation where the total list of nodes is big enough to create new partitions but the cloning conditions are not met. In this case the aggregation phase allows creating new clones by aggregating the nodes on the base of their distance from the agent node sitting.

4.7 The algorithm through an example

To further illustrate the operation of the partitioning algorithm, let us consider again the topology depicted in Figure 2.2a which for simplicity we repeat below.

Figure 4.7a) Example system topology. b) Partitions and partition centres resulting in minimal Total Hop Distance.

Figure 4.8 illustrates the MA cloning/deployment map that finally results in the near-optimal partitions of Figure 2.2b.
Let us assume that the process is initiated at node 0, which is a relatively "central" node so that there are fewer steps involved to convergence. On the other hand, starting at any other "remote" node, e.g. 46, will result in fairly similar partitions as our algorithm exhibits relative independence from the starting conditions, as shown later in Section 5.5. The first (unique) agent at node 0 starts by assessing the cloning conditions of Figure 4.3 for each of the neighboring nodes (nodes 1, 2, 6, 13, and 23). Given the input parameters $N=\{0, 1, 2, \ldots, 50\}$, $p=10$ and $e_p=20\%$, the cloning conditions are found to be true for all neighbor nodes except for node 23. Hence four new agents are cloned, while the original agent migrates to node 23. At this point each agent proceeds independently from
each other. Agents have their own list of nodes – for instance, the agent residing at node 1 has \( N = \{1, 27, 28, 29, 30, 31, 32, 33, 34\} \).

Let us now see what happens to this agent (other agents will behave in a similar fashion). The cloning conditions are not satisfied at this point because the agent resides towards the periphery of the partition. So the agent subsequently migrates to 34, 30 and 32. At 32, the cloning conditions are met, leading to the creation of two sub-partitions, \( \{1, 27, 29, 30, 34\} \) and \( \{28, 31, 32, 33\} \). The newly cloned agent subsequently migrates to 30 and 34. At this point loop avoidance comes into play, leading to a final migration to 30. The other agent subsequently migrates to 28 and 31 and, following loop avoidance, settles at node 28.

The next section will discuss dynamic adaptation of the algorithm when network changes occur.

### 4.8 Dynamic Adaptation Phase

Upon completing the whole partitioning process, the agent system still has to continuously adapt to network dynamics. All agents get into self-healing phase, consisting of a periodic re-computation of the weighed routing costs \( (E_i) \) – as described in Section 4.5. As network conditions change, so does \( E_{\text{max}} \) – i.e. the maximum value of all \( E_i \). As previously explained, \( E_{\text{max}} \) grows as a result of network degradation or failure; so by comparing the new value of \( E_{\text{max}} \) \( (E_{\text{max, new}}) \) with its current counterpart \( (E_{\text{max, old}}) \), the agent can decide whether or not to trigger migration.
The periodic re-computation of $E_{\text{max}}$ assumes the choice of an 'appropriate' period, $\tau$. We recall that agents are not only meant to maintain location optimality but also to perform the tasks of the partition centre. Consequently, there is no point in setting $\tau$ to be smaller than the typical agent migration time, risking otherwise the occurrence of MA system instability. Per contra, excessively high values of $\tau$ would compromise prompt adaptation.

$\tau$ determines system stability, controllability and duration of transient conditions. However, the choice of $\tau$ is a trade-off decision driven by the laws of classic control theory, so we do not delve into this aspect any further. More pragmatically, we have set $\tau$ to be twice the average migration time, obtaining good overall controllability. In general $\tau$ should at least greater than the migration time.

4.9 Summary

This chapter introduced the network partitioning algorithm, dividing it into 5 different phases: set-up, cloning phase, migration phase, aggregation phase and self-healing. Through these phases the algorithm solves the network-partitioning problem formulated in Section 2.2.3.

In the set-up phase the agent gets all the initial parameters needed to perform the partitioning process.
The cloning phase allows the agent matching the list of nodes to be partitioned against the local routing table in order to get the subset of nodes reachable through each neighbour and decide if new clones are needed or not. This phase permits to parallelise the partitioning process.

In the migration phase the agent instead use the routing table to evaluate the distance of the nodes within the partition and decide if a migration is needed or not. It is through this phase that the agent reaches an optimal location within its partition.

The network fragmentation is tackled by the aggregation phase, where the agent aggregates the nodes upon considering their distance from the agent node sitting.

Finally, the self-healing phase allows the agent keeping an optimal location within the partition by comparing the information within the local routing table at different instants of time.

A comprehensive performance evaluation of the propose algorithm is presented in the next chapter.
Chapter 5  System Performance Evaluation

5.1 Introduction  
This chapter evaluates the algorithm presented in Chapter 4. First, the methodology adopted is discussed giving motivations, assumptions, performance metrics and simulation design. Then, the experimental results are presented.

5.2 Evaluation Methodology  
5.2.1 Choice of Methodology 
The three broad different techniques used for performance evaluation are analytical modelling, simulation and measurement [86]. In order to decide the right technique to use, few considerations have to be done:

- Level of accuracy desired. Usually analytical models require many assumptions and simplifications so that results can be relatively accurate ([86], page 31). Simulations need fewer assumptions and therefore can give more reliable and results closer to reality. Measurements, instead, might provide inaccurate results depending on the specific parameters setting of the experiment, which may be unique and not representing typical real world scenario. In this case the results could be from very accurate to inexact.

- Performance Metrics wanted. The distributed nature of the proposed system must take into account metrics such as scale, system precision, consistency etc (more details in section 5.2.2). In this regard, simulations can offer more confidence in modelling aspects that are difficult to accurately model or analyse with real networks. On the other hand, measurements cannot scale over the testbed dimension and therefore their results are limited.

- Tools available. The impact of the proposed system on realistic IP networks is also an important aspect to be taken into account. The availability of ready network simulators such as NS, J-Sim (formerly known as JavaSim) [87],
OPNET [88] etc. able to supply powerful tools rich of functionalities that resemble the real systems make this choice easier. In this respect, analytical models are not able to delivery the same degree of accuracy, while measurements still fail to meet the scale requirement.

As a result of the above considerations, simulation was preferred as the most suitable technique to evaluate the proposed system.

5.2.2 Performance Metrics

The system performance evaluation aimed to show various characteristics, such as system scalability, system precision and consistency. The aspects studied for the system evaluation were the following ones:

- Computational Complexity
- Sensitivity to the starting conditions
- Accuracy
- Dynamic adaptation

Computational Complexity is studied in order to assess the scalability of the algorithm, while Accuracy assesses the optimality of MAs locations within their partitions and the goodness of the resulting partitions in terms of desired number of partitions. Sensitivity to starting conditions studies how regular and consistent the algorithm performs when a different root node is chosen. Finally, the algorithm adaptability to changing network conditions is considered.

5.2.3 Fixed Parameters

Fixed parameters are ones that are the same for all experiments. These parameters regard the network simulator as well as the MA based partitioning algorithm. The most significant parameters of the network simulator are reported in Table 5-1.

In order to reflect real inter-networks transit-stub topologies have been used [81][82]. This class of topology can be viewed as a collection of interconnected routing domains, which are groups of nodes that are under a common administration and share routing information. Each routing domain can be classified as either a stub domain or a transit domain (Figure 5.1).
Transit domains purpose is to interconnect stub domains efficiently. Stub domains can be further classified as single- or multi-homed. Multi-homed stub domains have connections to more than one transit domain. Single-homed stubs connect to only one transit domain. Some stubs have also links to other stubs.

The most significant parameters of the agent-based partitioning system are reported in Table 5-2.
Values in Table 5-2 depend upon the following assumptions:

- Since the agent carries parameters that are subject to small size variations, the agent size is assumed not to vary during the simulation.
- It is assumed that the partitioning processing time for an agent is negligible with respect to the serialization and deserialization time and therefore it is not taken in consideration.
- The serialization and deserialization time are assumed to be in the order of hundreds of milliseconds [89]. They are set as simulation parameters that don't change during the simulation, since agent size is kept constant as previously mentioned.

### 5.2.4 Factors

Factors are parameters that vary in each simulation and have an influence on the metrics under study [86]. The most significant simulation factors are reported in Table 5-3.

<table>
<thead>
<tr>
<th>Simulation Factor</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of nodes per network topology</td>
<td>51 – 102 – 147 – 198 – 258 – 303</td>
</tr>
<tr>
<td>Network diameter</td>
<td>11 to 17</td>
</tr>
<tr>
<td>Network diameter for dynamic adaptation</td>
<td>8 to 12</td>
</tr>
<tr>
<td>Ratio p/N</td>
<td>0.05 – 0.1 – 0.15 – 0.2 – 0.25 – 0.3</td>
</tr>
<tr>
<td>% Number of link failures (dynamic adaptation</td>
<td>5 – 10 – 15 – 20 – 25 – 30</td>
</tr>
<tr>
<td>only)</td>
<td></td>
</tr>
</tbody>
</table>

Table 5-3. Simulations factors.
5.2.5 Simulation Complexity

The combination of factors under analysis is shown in Table 5-4, which gives an indication of the complexity of our simulation based study.

<table>
<thead>
<tr>
<th>Experiments</th>
<th>Topologies $N$ (10 topologies per family)</th>
<th>$r = p/[N] $</th>
<th>Tolerance on $p$: $\varepsilon_p$ (%)</th>
<th>Diameter $D$</th>
<th>Root Node</th>
<th>% of Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computational complexity</td>
<td>102 147 198</td>
<td>0.05 0.1 0.15</td>
<td>15</td>
<td>11 12 13 14</td>
<td>0</td>
<td>/</td>
</tr>
<tr>
<td></td>
<td>258 303</td>
<td>0.2 0.25 0.3</td>
<td></td>
<td>15 16 17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Starting Node</td>
<td>51 198</td>
<td>0.05 0.1 0.2</td>
<td>15</td>
<td>/</td>
<td>0 to 50, 0 to 198</td>
<td>/</td>
</tr>
<tr>
<td>Optimality &amp; Partitioning</td>
<td>51 102 147</td>
<td>0.05 0.1 0.15</td>
<td>0.5 10 15 20</td>
<td>/</td>
<td>0</td>
<td>/</td>
</tr>
<tr>
<td></td>
<td>198 258 303</td>
<td>0.2 25</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic adaptation</td>
<td>147</td>
<td>0.1 0.15 0.2</td>
<td>15</td>
<td>8 9 10 11 12</td>
<td>0</td>
<td>0 5 10 15 20 25</td>
</tr>
</tbody>
</table>

Table 5-4 Experiment factors.

The simulation of the partitioning process involved relatively long computational times. For each experiment the number of levels and combinations of the relevant simulation factors are calculated. Factors involved from Table 5-4 are: 1) the number of network nodes to be partitioned, $|N|$; 2) the ratio $p/[N]$; 3) tolerance on $p$, $\varepsilon_p$; 4) network diameter, $D$; 5) Root node (for Starting node experiments only); and 6) percentage of faults (for Dynamic Adaptation only).

Hence, since that the average computation time of a single simulation depends on the experiment, it is possible to calculate the computational time for each experiment. The resulting overall computational time is in the order of 11,400 hours. The calculations (see Table 5-5) were made considering the standard school PC, whose specs are in Table 5-6.

Simulation traces were very large and required a total of few hours to process and analyse with specific Tel scripts. Minor time was required to process the results and convert them in a format that could be used by the Microcal Origin [90] statistical tool. Significant effort was spent also on the production of diagrams, which required significant manual intervention.
Linearity Experiment

Randomise 10 times {
  For each \( N \) \{102, 147, 198, 258, 303\} // 5 *
  For each \( p/N \) \{0.05, 0.1, 0.15, 0.2, 0.25, 0.3\} // 6 *
    For Each \( \varepsilon_p(\%) \) \{15\} // 1 *
      For each \( D \) \{11, 12, 13, 14, 15, 16, 17\} // 7 *
        Compute partitions // 80 (mins)
    }
  }
  2800 hours

Starting Node Experiment 1

Randomise 10 times {
  For each \( N \) \{51\} // 1 *
    For each \( p/N \) \{0.05, 0.1, 0.2\} // 3 *
      For Each \( \varepsilon_p(\%) \) \{15\} // 1 *
        For each RootNode \{0->50\} // 51 *
          Compute partitions // 3 (mins)
    }
  76.5 hours

Starting Node Experiment 2

Randomise 10 times {
  For each \( N \) \{198\} // 1 *
    For each \( p/N \) \{0.05, 0.1, 0.2\} // 3 *
      For Each \( \varepsilon_p(\%) \) \{15\} // 1 *
        For each RootNode \{0->197\} // 198 *
          Compute partitions // 1 (hour)
    }
  5940 hours

Optimality and Error Experiments

Randomise 10 times {
  For each \( N \) \{51, 102, 147, 198, 258, 303\} // 6 *
    For each \( p/N \) \{0.05, 0.1, 0.15, 0.2, 0.25, 0.3\} // 6 *
      For Each \( \varepsilon_p(\%) \) \{5, 1, 15, 2, 25\} // 5 *
        Compute partitions // 70 (mins)
    }
  2100 hours
5.2.6 Hardware Parameters

The previous section identifies the enormous amount of time required to perform all the experiments required to assess the proposed algorithm. It should also be taken into account the debug time spent in order to reach the final algorithm version, which increments the overall time by at least 5 times. The available machines to run the simulations were 4 servers and a standard school PC (see Table 5-6). The servers had lower specs than the standard school PC and they were also shared with other users. The problem was addressed using a computational Grid. The concept behind grid is the use of idle resources that can be considered otherwise wasted. In this regard, the SEPS (School of Electronic and Physical Science) school of Surrey University provides a number of computer labs with a total number of 165 computers. Considering the total simulation runs given in the previous section and adding the debugging time, the total simulation time would be of 56832.5 hours \( \sim 6.5 \) years (in a single computer). Dividing this estimation time by the number of machine available in the grid, the estimation drops to \( \sim 345 \) hours \( \sim 15 \) days. Considering that the machines were not used continuously, but mainly over night, which means \( 1/3 \) of day, and not all of them were available at the
same time, the estimation is ~ 60 days. Therefore, it is possible to see the enormous benefit of using the grid approach, which has considerably reduced the total amount of time required from 6.5 years to 2 months.

In order to use the grid-like approach (Figure 5.2), several mechanisms were implemented to avoid disruption. Two conditions were taken into account:

1. A pc was used only if nobody was logged in.
2. A pc was used only if it had at least 90% of its CPU free.

Figure 5.2 illustrates this simple but effective mechanism to allocate tasks to the Grid.

5.2.7 Simulation Design

The simulation environment and the tools used for our study are depicted in Figure 5.3, where shaded boxes indicate new software tools developed by us.
The different conditions, or scenarios, such as network topology type, network size, communication protocols are specified as Tcl scripts.

Another important aspect concerns the generation of network topologies that resemble real networks. Realistic Internet-like topologies were generated with the GT-ITM topology generator [80], following a well-established methodology by Calvert and Zegura [81][82]. Transit-stub topologies having 51, 102, 147, 198, 258 and 303 nodes were generated in order to assess the algorithm sensitivity to network size. For statistical significance, we produced families of at least 10 topologies each.

Since NS is an event-driven simulator, the analysis is carried out on the output traces. Given the large amount of simulations, we had to automate the conversion of NS traces to a format that could be fed to statistical tools. We produced scripts in Tcl, Perl and Matlab for this purpose. The data was analysed using Microcal Origin [90]. The NAM network animator [79] was also used for debugging, verification and validation.

Another important aspect was the study of the optimality of the presented approach. This was done by a comparison with the well-known Lagrangian algorithm [12], which computes provably near-optimal $p$-medians. Despite computing $p$-medians in polynomial time, the Lagrangian algorithm relies on the whole network topology (i.e. the distance matrix), so it is not a viable solution in practice. It can only be used as a benchmark for other algorithms, as in this case. We used the SITATION simulation package [91] to compute Lagrangian $p$-medians and their total hop distance.
5.3 Computational complexity

An important requirement of the partitioning problem defined in Section 2.2.3 is scalability, which is the ability to increase the size of the problem domain with a small or negligible increase in the solution's time and space complexity. In our case, the main scalability factors are the size of the initial partition, \(|V|\), the partition diameter (i.e. the maximum distance between any two nodes in the topology), and the number of target partitions \(p\).

Our initial hypothesis was that, because of its distributed nature, the complexity of our algorithm would be affected mainly by the network diameter. The cloning process of Figure 4.3 guarantees a good level of parallelism, while the migration process of Figure 4.4 stops, in most cases, well before visiting the whole network. In order to assess our hypothesis and determine the computational complexity of our algorithm, we have computed the partitioning time (i.e., the MA deployment time) in all combinations of the parameters depicted in the relative experiment of Table 5-4.

Figure 5.4 depicts the results obtained for 102-nodes topologies. The most important result is linearity with the network diameter, which makes the algorithm suitable for large-scale systems.

It may be worth mentioning that in all cases, the slope of the partitioning line is relatively small in comparison to the intercept between the line and the y axis. The reason for that is that MA migration time dominates the partitioning process. In most general purpose MA platforms, the time to hop between two nodes is of the order of hundreds of milliseconds [89]. This means that our algorithm would not be able to compete with conventional partitioning algorithms in the case of small-scale networks. On the other hand, the linearity and relatively small slope ensure that the algorithm is indeed the best possible option for large-scale systems.
It is worth mentioning that in typical Internet-like topologies, a small increase in network diameter results in a relatively large increase of $|\mathcal{N}|$, especially for large networks. This is why when we studied partitioning time versus $|\mathcal{N}|$ we found that the former increases when an increase of $|\mathcal{N}|$ is also accompanied by an increase in the network diameter. In other words, an increase in $|\mathcal{N}|$ with constant diameter, does not significantly affect partitioning time. We also point out that the algorithm is not affected by the average node degree. In fact, if there are more links per node when the agent migrates towards optimality it most likely end up in a previous visited node. In this case the loop resolution is performed and the agent takes a final decision.

A second important set of simulation results is depicted in Figure 5.5, capturing the effect the number of target partitions has on computational time. In this case, we notice an even better behaviour. Computational time gradually increases only in the lower range of $p/|\mathcal{N}|$ but plateaus when the percentage of number of target partitions becomes greater.
than about 15% of the initial partition. This result shows, once more, the high level of parallelism and distribution of our algorithm. In fact, the more partitions we want to create, the larger the number of MAs operating in parallel.

![Graph](image)

Figure 5.5 Partitioning time Vs \( p/|N| \) for 102-node topologies.

The results showed are a sample of the overall results computed. For the rest of the results refer to the Appendix A.

5.4 Accuracy

5.4.1 Location optimality

Having demonstrated the viability (scalability) of our partitioning algorithm, we then studied its ability to produce ‘good’ partitions and partition centres, according to the problem formulated in Section 2.2.3. We recall that our objective function is to minimize Total Hop Distance (THD) – this is equivalent to saying that we need to assess the ‘distance from optimality’ of the \( p \) partitions and partition centres found by the algorithm.
As mentioned in Section 5.2.2, we have carried out a comparative study using the Lagrangian algorithm as benchmark. We have computed the THD of both algorithms in all combinations of the parameters depicted in the relative experiments of Table 5-4.

Figure 5.6 depicts just one of the cases, with $|N|= 51$ nodes. The most interesting and, to a great extent unexpected, result is that the THD of the presented algorithm has been found to be always smaller than the THD of the Lagrangian algorithm, which proves that proposed solution is at least near-optimal too.

![Figure 5.6 Location Optimality comparison.](image)

5.4.2 **Accuracy on number of partitions**

Being based on simple heuristics, the algorithm will generally work better under certain conditions. The algorithm has produced near-optimal $p$-medians under all test cases (as discussed in the previous section). We also assessed its accuracy in the generation of the required number of partitions. We have seen that one of the input parameters is $e_p$, which is the tolerance on $p$ expressed in percentage. We have assessed

---

5 We recall that the Lagrangian algorithm has been mathematically proven to lead to near-optimal $p$-medians on general networks [12].
the error on $\varepsilon_p$ in all combinations of the parameters depicted in Table 5-4 for the relative experiment, using the following formula:

$$
Err = \begin{cases} 
\frac{(p - \frac{\Delta p}{2}) - p_o}{(p - \frac{\Delta p}{2})} & \text{if } p_o < p - \frac{\Delta p}{2} \\
\frac{p_o - (p + \frac{\Delta p}{2})}{(p + \frac{\Delta p}{2})} & \text{if } p_o > p + \frac{\Delta p}{2} \\
0 & \text{if } p - \Delta p < p_o < p + \Delta p 
\end{cases}
$$

where $p$ is the target number of partitions, $\Delta p = \left(\frac{p \times \varepsilon_p}{100}\right)$ is the absolute variation admitted on $p$, and $p_o$ is the actual number of partitions computed by the algorithm. A representative sample of all cases of Table 5-4 is depicted in Figure 5.7. We can see that the accuracy increases considerably (i.e. $Err$ diminishes) as we relax the constraints on the tolerance on $p$ (i.e., when $\varepsilon_p$ increases).

![Figure 5.7 Error on number of partitions produced by the algorithm.](image-url)
The algorithm is not particularly accurate in the generation of the exact number of partitions. It does compute near-optimal partitions but it can only meet relatively relaxed constraints on the exact number of partitions. Nevertheless, in practical applications, the latter property is far less important than the former. One is most commonly interested in creating optimal partitions rather than determining their exact number a priori. What is typically needed in the applications described in Appendix B is the ability to partition a large system into smaller systems for efficiency reasons. Given the results of Figure 5.7 (typical errors are of the order of 10%), we can conclude that the algorithm provides a good control of the order of magnitude of $p$.

**5.5 Sensitivity to starting conditions**

Another important requirement of the algorithm is its independence from the input parameters specified in Section 4.3. The results reported below demonstrate that the algorithm provides comparable results in all the cases of the relative experiment of Table 5-4 — we have computed partitioning time, number of computed partitions and THD, which are illustrated in Figure 5.8, Figure 5.9 and Figure 5.10 respectively for 51 nodes topology. The graphs depict mean values along with the percentiles (internal ticks) and min/max values (externals ticks). Noticeably, all mean values are comparable, demonstrating that the algorithm works equally well irrespectively of the starting node.
Figure 5.8 shows for the first three nodes lower values of partitioning time than the rest of the nodes. This is due to GT-ITM, which creates those three nodes centrally within the topology. While being the other nodes not central, the first agent needs to migrate closer to the topology centre in order to meet the cloning conditions in those cases. This results in higher values of partitioning time.
Figure 5.9 Sensitivity of number of partitions computed to starting node; 51-nodes topology, ratio 0.2.
The same experiments but for 198-nodes topologies are shown in Appendix A.

### 5.6 Dynamic Adaptation

Communication distributed systems should be adaptive to congestion and faults conditions without the need for external intervention. Results of the study of adaptation to network link failure are here reported. These are also indicative of the algorithm ability to adapt to network congestion. We have simulated all combinations of the parameters of Table 5-4 for the relevant experiment. For each family of topologies we have randomly created an increasing number of link failures (from 0 to 25%) – for statistical significance, faulty conditions were randomized 10 times per topology.

Link failure does result in a change of topology – it is the responsibility of the routing protocol to update the routing tables accordingly. As the faults percentage increases, the
probability of obtaining unconnected nodes increases too. We have created a very large number of scenarios, selecting only the cases in which full connectivity is maintained.

To benchmark the ability to self-heal, we have first measured the Total Hop Distance (THD) in the absence of any failure. Link failure results into new paths between partition centres and partition nodes that are generally equal or longer than the paths established before the failure. As a result, THD generally increases with failure and the partition centre locations may become non-optimal. To quantify the degradation in THD we have measured its value immediately after link failure but before triggering the self-healing procedure. We have also re-calculated the theoretically near-optimal set of new partition centres (using the Lagrangian algorithm) and, correspondingly, the new near-optimal THD. Finally, we have re-calculated the THD after self-healing, i.e. after MAs migrate to the newly calculated partition centres. An indicative sample of the results is depicted in Figure 5.11. The key result is that self-healing always succeeds to bring the system back to optimality.

Figure 5.11. Sample of results on self-healing.
5.7 Summary

This chapter illustrated the evaluation methodology used to assess the proposed system and introducing performance metrics, fixed parameters and factors involved in the algorithm assessment. Furthermore, the simulation complexity was examined. Due to the enormous amount of simulations needed for the system assessment, a grid-like approach was used.

The system findings were then analysed, showing computational efficiency (linearity with network diameter), configurability (independence from initial conditions), optimisation (near-optimal partitions and partition centres) and self-healing (adaptation to component failures or congestion).

In the next chapter the system applicability is discussed along with some case studies.
Chapter 6  Discussion and Conclusions

6.1 Thesis summary

This thesis addresses one of the fundamental theoretical problems in communication networks, partitioning and centre identification. Systems relying on large and dynamic communication networks must face the problem of partitioning the network and finding optimal partition centres. We have formulated that in terms of the classic \( p \)-median problem, which is \( NP \)-hard when striving for optimality. Over the past 30 years numerous approximate solutions have been proposed, but they all fail to address the combined requirements of scalability, optimality and adaptability.

This thesis motivates the use of code mobility to provide a ‘distributed’ solution to the \( p \)-median problem that is traditionally tackled only with ‘centralised’ algorithms. Efficiency is achieved through a combination of distribution, code mobility and simple use of network routing information. We propose a near-optimal algorithm that does not require the reconstruction of the network distance matrix and is characterized by linear computational complexity. Our approach outperforms existing algorithms which are polynomial of 2\(^{nd}\) and 3\(^{rd}\) degree and do require a complete knowledge of the network topology.

We have carried out an extensive simulation-based analysis that allows us to draw conclusions on computational efficiency (linearity with network diameter), self-configurability (independence from initial conditions), self-optimization (near-optimal partitions and partitions’ centres) and self-healing (adaptation to component failures or congestion).

On the other hand, network partitioning obtained through code mobility poses new problems in terms of controllability, stability, and predictability of system behaviour. Systems such as the one presented herein are difficult to assess. We have developed a novel simulation framework starting from a well known network simulator and we have designed a comprehensive set of experiments based on that.
The work presented here has given us ample opportunities to learn a number of lessons. Code mobility is a potential paradigm that, by its very nature, suits a wide variety of system requirements. Emerging systems are increasingly mobile, dynamic, ubiquitous and highly distributed. Mobile code offers powerful means for addressing classic NP-complete problems in a relatively simple fashion; the work presented here provides an example of such a system.

6.2 Discussion of Thesis Contributions

6.2.1 Network partitioning algorithm

The use of code mobility as a powerful means to address a fundamental problem in large distributed networks/systems has been largely assessed. Existing approaches available in the literature present various problems: most of them are meant for centralised systems; others have high computational complexity; and others do not guarantee near-optimality. The proposed algorithm can be computed in a distributed fashion, is characterised by linear computational complexity and guarantees near-optimality. Moreover, the algorithm showed independency from starting conditions, which ensures the same level of performances with any initial conditions.

6.2.2 Adaptability

The system adaptability has been assessed over a large set of simulations, which showed the system ability to keep near-optimality upon network changes. In particular the system has been evaluated in case of link failure, which is also indicative in case of network congestions.

MAs perform adaptation through simple migration. Since the migration time has a relatively high cost, the adaptation may be effective when network changes occur in timescales larger than the migration time. In fact, if the network dynamic is faster than the migration time the system may fail to keep near-optimality.

6.2.3 MA system simulator

The MA extensions provided on top of NS are a practical contribution to this thesis. The absence of an MA system simulator combined with a network simulator, led us to
develop such extensions. It is interesting to note that only recently a similar work has been made public. This proves the need for such a framework in a time where systems are becoming largely distributed, complex and dynamic. Our simulator is certainly a tool that can be employed by other researchers.

6.2.4 Applications

The proposed algorithm, because of its intrinsic distributed nature, is prone to building overlay networks. These are gaining momentum as means to introduce new technologies within the Internet. However, other applications can benefit from using this algorithm, including most large-scale distributed systems - characterised by dynamic and frequently changing environments - which need to be scaled down in order to address the combined requirement of scalability, optimality and flexibility.

Three case studies are illustrated in the Appendix B in the context of Application-level Multicast, Content Adaptation Networks and Overlay Resource Management.

6.3 Practical deployment of the algorithm

Our simulation-based study provides an insight into the viability of the partitioning algorithm in the context of IP-based networked systems - the simulation design described in Section 5.2 covers a broad range of cases. We discuss here other practical issues towards a full deployment of the algorithm, focusing on the necessary infrastructure.

The first important question concerns the maturity level of code mobility frameworks. Our system assumes the presence of lightweight means for transporting and executing code throughout the network. However, it does not dictate code mobility support in every node - that assumption would be unrealistic. On the other hand, code transport and execution in routers is not problematic nowadays. The IETF Distributed Management (DISMAN) [94] working group has been studying mechanisms for distributing scripts that can perform arbitrary tasks in managed nodes [94]. The Script-MIB defines an SNMP-compliant MIB and a standard interface for code pushing to managed nodes [95]. The MIB supports scripts in arbitrary programming languages and makes no assumption about code formats.
Code execution in network devices such as IP routers is also viable today. For example, Cisco routers provide the ability to run Tcl (Tool command language) scripts that can have SNMP MIB object access [96]. Public access to MIB objects is read-only but there are also security mechanisms allowing full manipulation.

In our simulations, we have assumed the functionality and typical performance figures of Script-MIB and Tcl. The resulting size of our mobile code in Tcl is in the order of 100Kbytes, while the 1-hop migration time is in the order of 800msec. We have not specifically tackled the security and safety issues related to code mobility, given our relaxed requirements on security\(^6\) and the ample literature available in the subject (the interested reader may refer to [97] and the references therein).

Finally, our algorithm assumes initial knowledge of the network size \(|V|\) and of the identifiers, i.e. addresses, of the nodes. This information can be obtained through a discovery process that precedes the algorithm operation. The discovery mechanism is specific to the application domain and, as such, outside the scope of this thesis. For example, in fixed networks node discovery can be achieved through SNMP, in ad hoc networks flooding with responses to the source node may be used while peer-to-peer overlay networks have their own specific mechanisms.

### 6.4 Future Work

This section discusses possible improvements to the network partitioning algorithm as well as future research areas.

#### 6.4.1 Improving the algorithm performance

The algorithm showed a linear computational complexity, which mainly dependents on agent migration time. This in turn depends upon the agent serialisation/deserialisation time. Reducing the latter will indeed improve the performances of the system as a whole. First the computational complexity would have a smaller slope and second the adaptation would benefit too. The system is able to adapt to networks whose dynamics is within timescales greater than the agent one-hop migration time. Hence, by reducing migration

\(^6\) We recall that our clustering algorithm merely needs read-only access to routing tables.
time the system would adapt to networks with faster dynamics. Therefore, new migration techniques could be studied to reduce this problem.

6.4.2 Implementing an algorithm recovery feature

During the partition process, the agents can exchange a number of packets to create an overlay network. However, if an agent fails to reach its final location no mechanism is implemented to tackle the loss of the partition. In this case the father agent can setup a time after which it re-spawns a new agent. If the first clone suddenly replies, the father can force the first clone to kill itself because not needed anymore.

6.4.3 Prototyping the algorithm

In Section 6.3 the viability study of the algorithm in a real network was discussed. The next step of this work would be to realise a prototype of the proposed algorithm based on the considerations made. The prototype should show the real feasibility of the system and pointing out the practical problems that cannot be foreseen otherwise.

6.4.4 Improving the Overlay Network

In section 2.6.4, P2P approaches were classified as unstructured and structured. The overlay constructed by the algorithm for the experiment applications presented in Appendix B can be classified as unstructured. In fact, as the P2P Gnutella approach, each member of the overlay has a partial knowledge of the network and therefore is completely decentralised. On the other hand, this overlay uses flooding to contact the other members of the overlay but being an unstructured network, it has the problem that does not scale well.

It would be interesting to use a structured approach such as DHT (Distributed Hash Table) to build the overlay, which allows more reliable and efficient object location. Using this approach all applications should have benefits in terms of performance.
Appendix A. Further results

This Appendix shows other results obtained for the evaluation of the proposed system. Figures 6.1 to 6.8 show a larger sample of results. The behavior of these diagrams is not qualitatively different from the results of figures 5.4 and 5.5, demonstrating that the aforementioned results on computational complexity have general validity.

![Graphs showing partitioning time vs network diameter for topology 147 nodes.](image)

**Figure 6.1 Partitioning time vs Network Diameter for topology 147 nodes.**
Figure 6.2 Partitioning time Vs p/[N] for 147-node topologies.
Figure 6.3 Partitioning time vs Network Diameter for topology 198 nodes.
Figure 6.4 Partitioning time Vs $p/|N|$ for 198-node topologies.
Figure 6.5 Deployment time vs Network Diameter for topology 258 nodes.
Figure 6.6 Partitioning time Vs $\frac{p}{|N|}$ for 258-node topologies.
Figure 6.7 Deployment time vs Network Diameter for topology 303 nodes.
Figure 6.8 Partitioning time Vs $p/|N|$ for 303-node topologies.

Figure 6.9, Figure 6.10 and Figure 6.11 show results on the independency of the algorithm from the starting conditions for a 198 nodes-topology. Results are qualitatively equivalent to those of Figures 5.8 – 5.9 – 5.10.
Figure 6.9 Sensitivity of partitioning time to starting node; 198-nodes topology, ratio 0.2.
Figure 6.10 Sensitivity of number of partitions computed to starting node; 198-nodes topology, ratio 0.2.
Mean Values

198 nodes; \( p/|N| = 0.2 \); \( \varepsilon_p = 15\% \)

Figure 6.11 Sensitivity of THD to starting node; 198-nodes topology, ratio 0.2.
Appendix B. Applications

This Appendix illustrates some experiments that make use of the algorithm proposed in this thesis.

The adaptive partitioning method proposed in this thesis has been formulated and addressed in a general way. Because of that, it is easy to think of a range of applications that may significantly benefit from its features. Our algorithm solves one of the most controversial graph theoretical problems — i.e. how to efficiently partition a dynamic, large-scale network and find the central location of each sub-partition. Thus, the algorithm may be directly applied to any of the classic network optimization problems such as the optimal service facility location problem [12]. In this respect, this problem is fundamental in overlay networks. Overlay networks are gaining significant momentum because of their ability to complement existing networking protocols with new application-level solutions, as previously discussed in Section 2.5. These kinds of systems, which can be offered by the simple provision of libraries, produce fast deployable architectures where the price paid is in term of increased latency and reduced network performance. The former is caused by the deployment time required by the additional overlay whereas the latter stems from the transmission of duplicate packets across the same links in the network (duplication is not carried out at optimal nodes but only at end hosts). Therefore, a key aim of overlay network research is to reduce the overheads associated with the overlay deployment and operation. Our MA deployment system can be considered as a way to create adaptive overlays, with the agent bearing also the logic of the application.

An example is overlay multicast or Application-level Multicast. We take the view of Application-level Multicast provisioning and demonstrate the applicability and advantages of our MA-based approach — this is an alternative approach to native IP multicast. Application-level multicast implements its own distribution tree where each node is provided with all communication functionalities (e.g., replication and membership management). Clearly, placing the replication functionality in end hosts such as dedicated servers or border routers, as opposed to core routers for IP multicast, allows the provision of such functionalities without any further assistance. The emphasis is
therefore on providing replication, forwarding, and other functionalities in these nodes. The idea is to use end-systems connected in an overlay to serve groups of users. The problem in this case is still the creation of user groups and the allocation of users to groups, which is equivalent to the network-partitioning problem addressed in this thesis. Relevant examples are reported in [74], [75] and [73].

To demonstrate the wide applicability of the proposed algorithm, we also illustrate an example in the context of service management, namely the emerging area of content adaptation networks. The problem in this case is, again, the creation of application-level forwarding trees via network overlays. However, nodes have also media transcoding capability, allowing the delivery of a given media stream to a set of users having different requirements on the media format.

A third application tackles the problem of overlay resource management, which is similar to that of Grid resource management. Optimally placed agents can monitor a large amount of servers (or unused PCs) and redirect job execution requests in order to exploit unutilised resources.

Getting into more pragmatic applications, adaptive partitioning can also be seen as the basis of ad hoc networks. These are self-managed and, as such, they have to deal with a small, as well as large, number of terminodes. Partitioning is one way to give structure to the network, addressing scalability issues. Partition centre selection is another fundamental problem when dynamic conditions cause frequent changes in the network topology. Our MA-based system can be potentially used as a self-organizing management ‘fabric’ for ad hoc networks, possibly with extensions taking into account node mobility prediction in the objective function. Proactive routing is assumed so that terminodes have pre-computed routing tables.

A similar self-management infrastructure is also needed in peer-to-peer networks [40], which may scale up to millions of nodes but are meant to be used by a virtually unlimited number of users. Some of the existing peer-to-peer frameworks such as JXTA [70][71] are open source and aim at becoming de facto standards. However, they do lack effective mechanisms for peer group management (the equivalent of our partitioning problem) and super-peer election (the equivalent of our partition centre selection). Also, given the rapid
shift towards user mobility, self-organization and self-healing are bound to become major issues in peer-to-peer systems.

In the following section we focus more closely on three applications, namely Application-level Multicast, a Content Adaptation Network and an Overlay Resource Management.

**Application-level Multicast**

The simultaneous provision of services to many different users is realised through the multicast solution. This approach performs data group distribution at network layer adding more capabilities to the routers. In this way multicast is able to reduce packet replication to the minimum necessary. Although various multicast protocols have been proposed over the last decade, this solution performs well for LAN networks but does not achieve the same performance for inter-domain networks. In [76] a number of problems are identified and discussed. Existing network-level multicast protocols incur significant overheads in the routers; they require specific hardware support in place; they cause sensitive issues behind protocol standardisation; and finally they are not suitable for extremely dynamic environments. These reasons have induced some researchers to find new solutions working above the network level, which are commonly referred to as application-level multicast or overlay multicast. Unlike network-layer multicast, in application-level multicast data is replicated at the end hosts rather than at routers. Those end-systems form, therefore, an overlay network that is used to deliver data packets to end-users. The purpose of application-level multicast is to build and maintain this overlay network.

In this case study we present an Application-level Multicast solution in which end-systems are realised as Mobile Agents (MAs) that are in charge of creating the overlay network and of maintaining it as network conditions evolve. In this context, the task of constructing the overlay network is equivalent to that of optimally placing the end-systems within the network. We divide this task in two different phases: 1) partitioning the multicast group in \( p \) sub-groups; 2) near-optimally placing those end-systems within their respective partition. To demonstrate and assess our MA Application-level Multicast
approach, we carried out a simulation-based comparison with the Distance Vector Multicast Routing Protocol (DVMRP) [92], which is used as a representative network-level solution. We focus on two performance metrics: 1) The DVRMP multicast tree build-up time and 2) the stress ratio\textsuperscript{7} of the overall network. We have prototyped and evaluated this MA-based Application-level Multicast on top of the NS network simulator.

*Application-level Multicast* is aimed at supporting end-systems that are interconnected through networks not supporting native multicast (i.e. layer 3 or network layer multicast). They create overlay networks, which employ the end systems for data forwarding. In contrast, conventional multicast protocols deliver data streams from source to destination through a store-duplicate-and-forward mechanism performed at the routers.

Using the partitioning algorithm presented in Chapter 4, an overlay network is constructed by the MA-based system, as illustrated in Figure 6.12.

\textsuperscript{7} Stress is one of the metrics employed to compare the performance of an application-level multicast solution with the traditional one. It is defined as the number of identical packets that a physical link carries. The optimal value for native multicast is 1.
Figure 6.12 Distributed mobile application over IP network. MAs act as end-systems to deliver group data from the source to the users.

Upon being deployed, starting from the data source, MAs act as end-systems delivering the group data to the users (dotted line). While deploying, agents relay data among themselves, constructing the overlay network. This can be interpreted as a source-rooted distribution tree where MAs represent the tree leafs. In particular each agent will know the position of its father agent (if it has any), its clones (if it has any) and the users it has to serve. The multicast source sends the data to the first MA, which in turn redirects the data to the overlay. Within their competition zone, MAs deliver data to the users using the unicast model. In this phase of data delivery our approach pays off in terms of adaptivity in comparison to native multicasting.

**System assessment metrics**

Without loosing generality and in order to achieve reasonable simulation times, we assume that all the nodes in the network are group members. The metrics used for the comparison are the `TreeBuildTime` and the `StressRatio` that are plotted against topology.
size. The former is defined for the DVRMP as the period of time elapsed between the initiation of the members' joining procedure and the construction of the data distribution tree. In the MA system, the tree build time is the time to finish the MA partitioning phase.

StressRatio is defined as:

$$\text{StressRatio} = \frac{\text{StressOverlay}}{\text{StressMcast}}$$

which gives the variation of the overlay network from the optimal value. Since we assume that all nodes are group members, the optimal stress value is 1 packet per link, corresponding to 1 packet per member. For example, a topology of 25 nodes may have a stress of 24 (the data source node is not counted) for the DVMRP algorithm, and 33 (calculated as the total number of packets generated) for the MA-based one. In that case, the stress ratio would be 1.32, which means that the overlay network generates on average 1.32 packets per node.

StressOverlay is defined as:

$$\text{StressOverlay} = \text{Stress}_{\text{source-end systems}} + \text{Stress}_{\text{partitions}}$$

where

$$\text{Stress}_{\text{source-end systems}} = \begin{cases} C_{\text{source-father}} + \sum_{i} C_{\text{father-clone}}, & \text{for } p > 1 \\ 0, & \text{for } p = 1 \end{cases}$$

and

$$\text{Stress}_{\text{partitions}} = \sum_{i} \text{THD}_{i}$$

The Stresssource-end systems is given by the sum of the cost of the first MA (we call it father) from the source plus the costs of all the MAs (except the first MA) from the their respective father. The Stresspartitions is the sum of the THD of each partition. The StressOverlay elements are inversely proportional to the ratio $p/|N|$, i.e., to the number of partitions created (this is explained later in this section).

---

3 DVMRP is a multicast routing protocol for dense topologies, which means that group members are densely distributed across the network. It forwards packets using a multicast tree that is built by applying the Reverse Path Multicasting (RPM) algorithm.

4 Note that the leaf members receive only 1 packet, but the end-systems can receive more than 1.
In order to carry out this comparison, extensive simulations under random network environments have been performed. However, for brevity we report here only the most representative results. Simulations included topologies of 102, 147, 198, 258 and 303 nodes\(^\text{10}\). To ensure statistical significance, simulations have been repeated 10 times per topology. That is, we have randomised the simulation process, generating families of 10 topologies at a time.

**Performance results**

Figure 6.13 shows the *TreeBuildTime* for each of the two approaches under evaluation. We can see that the MA-based approach performs worse than DVMRP. This was expected due to the difference in the very nature of the two approaches. Being an application-based solution, our MA approach is bound to incur larger overheads which are, however, countered by a larger flexibility and by its ability to work without layer 3 multicasting support.

![Figure 6.13 DVMRP Tree build time Vs MA-based Tree build time](image)

---

\(^{10}\) The network diameter was not considered as a variable; therefore the topologies generated could have different diameters.
Our findings on the StressRatio are depicted in Figure 6.14. The StressRatio has an exponential trend, regardless of network size. In fact, considering that

\[ Stress_{\text{overlay}} = \begin{cases} 
Stress_{\text{partition}} & \text{for } p = 1 \\
Stress_{\text{source-end systems}} & \text{for } p > 1 
\end{cases} \]

when \( p = 1 \) the StressOverlay is equal to the THD of the only partition created. While, when \( p/N = \) the number of partitions created is equal to the number of nodes, which means one agent per node. In this case, the overlay behaves as native multicast and therefore the StressRatio tends to 1.

Figure 6.14 shows this trend for each family of topology. It is possible to identify a range of \( p/N = [0.1, 0.3] \), where the StressRatio = [1.35, 1.95], which are acceptable values of the variation of the overlay performance from the optimal theoretical value. Therefore by choosing \( p/N \), it is possible to control the order of magnitude of the overlay degradation with respect to the optimal value.

![StressRatio Vs Topology size.](image-url)
In conclusion, we should say that, although the MA algorithm has not been developed specifically for tackling application-level multicasting, the final results are surprisingly good. Clearly, the system is not optimal and there is ample space for improvement but this was not the original intent so is left for future work. The purpose of system such as the one presented here is not to replace but rather to complement native, layer 3 multicast and we believe that our initial findings show the viability of the MA approach.

The case study presented in the following section can be though as a further evolution of Application-level Multicast and shows a situation in which overlay networks are functionally superior to native solutions.

**Content Adaptation Networks**

The previous case study discussed one of the most controversial areas of research over the past years, IP multicast. As we have highlighted previously, native multicast is struggling to take off on a global scale (i.e. across multiple domains), with part of the research community arguing that it belongs to the application-level rather than to level three networking. This aspect is even more accentuated when multicast is intended to users with different Quality of Service requirements. Think for instance of the scenario where users connect through different access networks and use terminals having different capability. In fact, different standards and encoding technologies allow users to customise data streams according to their media devices. For instance, Internet entertainment sites as well as user choices for multimedia devices are growing rapidly. Therefore reliable and efficient services offering different QoS should be available.

One way to pursue that is by transcoding data streams, obtaining a multilayered stream that can be then delivered according to the user QoS requirements. Transcoding is the operation that allows converting a media object from one format to another. New software standards and encoding techniques, e.g. AVI and MPEG4, are able to convert stream format in real time [98]. In practice, technologies that can accomplish such a task use two different approaches. The first is to realise the transcoding at the source (Figure 6.15). Say that a group of users having different media requirements registers to receive
the same content. The source converts the media format in different streams and sends them independently. This solution has the drawback to generate high traffic around the source, which also means more traffic injected into the network that can, in turn, result in bottlenecks and network instability. Another problem is that the server is loaded considerably more than what would happen for instance in a normal web server because transcoding is a resource-intensive operation.

![Transcoding capability](image)

**Figure 6.15 Multicast solution, with stream transcoding at source.**

On the other hand, the advantage of this approach is that it relieves the end user from complicated, resource-intensive operations. This is a crucial requirement nowadays with the advent of small, mobile devices.

A different approach would be to move the transcoding capability to the user devices (Figure 6.16). In this way, we relieve the server and the network around it but we have to rely on powerful user terminals.

We propose a solution, which can be seen as intermediate between the two approaches presented above. Again we use network overlays based on MAs creating a Content Adaptation Network (CAN). We use our partitioning algorithm to create user groups.
MAs act as end systems with transcoding capability (Figure 6.17). This allows a reduction in traffic from the source to the MAs, because only one type of stream is needed, since that the MAs convert the stream into different formats. Also, this solution relieves the user terminals from the transcoding operation.

The CAN is formed exactly the same way as the multicast overlay of the previous section. The algorithm performs the partitioning operation on the set of users and during the partitioning MAs exchange packets to update their position. At the end, each agent knows the position of its father agent, its clones and the users it has to serve.

In order to study the system performance, we have implemented a CAN, comparing it with a solution based on native multicast PIM-SM [99][100]. In the next subsections, we explain the metrics used to carry out this comparison and show the results achieved. The CAN solution gives promising results, which open new areas for future research.
System assessment metrics

The system has been assessed on a topology of 147 nodes, where 103 non-central nodes formed the set of users. Users are classified according to two factors: device capability and QoS preferences. The combination of these two factors generates many different requirements/combinations in terms of stream format. We have considered a range of different data streams going from 0 (i.e., only one type of stream like in simple multicast) to 100. The simulation involved 5 seconds of data streaming.

The performance comparison was studied for three different metrics: Streaming time, \( N^o \) of packet lost and Traffic around source. The former is the time needed to deliver all 5 seconds of streaming, while the other two metrics are self-explanatory.
Performance results

Figure 6.18 shows the performance results for the three metrics, respectively. All results show a sensible performance degradation of PIM-SM when the number of terminals/user-preferences (i.e. the number of different data stream formats) grows more than 35. This is due to the fact that native multicast becomes less efficient because it has to create a number of multicast trees that increases proportionally to the number of data formats.

The CAN was tested for three different values of the $p/N$ ratio (0.05, 0.1 and 0.2) in order to show the impact of the number of transcoders on performances. Figure 6.18 A shows that the CAN is not significantly affected by the number of terminals/user preferences, since only one stream is sent from the source to the MAs, while PIM-SM heavily depends on the number of terminals/user-preferences as explained before.

However, CAN is affected by $p/N$, which determines the number of transcoders distributed on the network. When the ratio varies from 0.05 to 0.1 the streaming time improves, but from 0.1 to 0.2 CAN worsens. This is because increasing the number of MAs means a larger overlay, which means more time for the stream to travel through the overlay and therefore higher streaming time. Therefore, incrementing the $p/N$ ratio does not necessarily imply a performance improvement. In fact, after a certain value the performance degrades.

Figure 6.18 B shows our results on packet loss. The sudden degradation suffered by native multicast corresponds to network saturation. Clearly, our approach saves bandwidth, which leads to negligible packet loss for a larger range.

Finally in Figure 6.18 C, the traffic around the source is shown and, again, for the same reasons PIM-SM has a worse performance due to the duplication of streams at the server side.
Figure 6.18 CAN vs PIM-SM performance results.
This case study has shown another application of the algorithm in an interesting area, which promises significant developments in the near future. The results achieved are encouraging, showing a better performance compared to other solutions. The approach proposed is able to preserve network resources, meeting also the important requirement of mobile and pervasive computing (e.g. lightweight terminals, terminal heterogeneity, user customisability). However, further work is needed, looking at all the aspects and comparing all factors in a more comprehensive way. In the next section we apply the MA algorithm to another area, resource management.

**Overlay Resource Management**

This case study aims to show the algorithm applicability in the context of resource management. By resource management we mean the management of distributed servers (or idle PCs) across the Internet. An example of such a system is represented by Grid computing [93], where idle resources across the Internet can be used to run user processes.

Like in a computational Grid, in our system each resource server\(^1\) needs to have an execution environment and a resource manager. The resource manager uses relevant information about the availability of local and distributed resources to trigger a variety of load-balancing actions.

In this context, the proposed system acts as a monitoring system. The set of resource servers are the nodes to be partitioned and the MAs constitute the monitoring system. As for the previous case studies, during the partitioning phase agents exchange a number of packets in order to create an overlay network. In particular each agent will know the position of its father agent, its clones and the resource servers to be monitored. This means that each agent has a partial knowledge of the overlay network but guarantees that each resource server within the system is reachable through the monitoring system.

When requests to run processes are sent to a resource server, the resource manager plays the key task of matching the job resource requirements with the run-time resource

---

\(^1\) Subsequently all PCs and servers that offer their resource to run user processes will be referred to as resource servers.
availability. This operation is aimed at load-balancing the requests across the resource servers used. The MA-based monitoring system gets resource information from the resource servers.

Each resource server needs to have functionality to expose local resources. The extensions explained in Section 3.4 are used in this case study. The resource manager performs the important tasks of admission control, resource allocation, and resource discovery, while ServiceAgents act as users that send job execution requests. Figure 6.19 illustrates the various steps starting from the submission of a job request until job execution. It depicts the logic executed at two different resource servers. The 'local' resource server is the one that receives the job request in first place. The 'remote' resource server represents a generic candidate for the execution of the job.

The execution of jobs within resource servers has to be endorsed by their respective resource managers. When a resource server receives a job request, this goes through an admission control procedure aimed at establishing whether sufficient resources are locally available.
Job requests that pass the admission control test are instantiated and executed locally, following an appropriate resource allocation procedure. Instead, any job that is not admitted locally triggers a resource discovery procedure through which the local resource server finds out about alternative resource servers that have the potential to admit those jobs. Again, the monitoring system realised through local and remote monitoring agents (MA<sub>1</sub>, MA<sub>2</sub>, ..., MA<sub>j</sub>) assists and provides information during this phase. In such a way the local manager eventually delegates each of the pending jobs to the appropriate remote resource servers that will, again, put the job through the same admission/allocation/discovery procedure (right hand side of Figure 6.19). Job request delegation will continue until a suitable, unloaded resource server is found.

The discovery procedure will generally result in one or more candidate resource servers for each of the candidate jobs. The local manager can then employ different strategies aimed at selecting the best match between jobs and resource servers. Allocation strategies are as follows:

1. **Proximity.** In this case each request is matched with the resource server that responds first. The fastest reply may be received by the entity that is logically closer to the requestor. This may therefore be useful when trying to minimise latency between local and remote resource servers.

2. **Load Balance.** In this case jobs are matched with resource servers holding the largest availability of resources. This may assure an even utilisation of resources across the system.

3. **Mixed (Load Balance + Proximity).** This solution compromises the above approaches, seeking a good load balance as well as latency minimisation. The highest value of the ratio availability/discovery time is chosen, where availability is the resource server candidate resources available and discovery time is the reply time of the candidate.

**Overlay resource management system logical view**

In this section a logical view of the resource management systems previously explained is given. Figure 6.20 depicts the MA system after the partitioning process, where each monitoring agent monitors a certain number of resource servers and is
logically part of the overlay network. In each resource servers there is the resource manager previously described.

![Diagram of overlay resource management system in action.](image)

**Figure 6.20 Overlay resource management system in action.**

Suppose that a job request is sent to a resource server by a user, the resource manager performs the admission control procedure, which accepts the job request to be executed locally according to the local resources availability. Alternately, in case the admission control procedure rejects the job request, the resource manager within the first resource server triggers the resource discovery in order to execute the job request in a different location. Since each resource agent knows its monitoring agent position the request is sent first to this monitoring agent, which then redirects the request to the overlay network (i.e. the peer entities, see Figure 6.20). The peer entities will subsequently enquire the relative resource servers that will reply with their availability. All the information is sent directly to the first resource server. The resource manager within the first resource server depending on the strategy used selects the resource server. Consequently, the job request is sent to the chosen resource server where is executed. After the execution the job is sent back to the first resource server and then to the user.
The next section introduces the metrics used to assess the overlay resource management system.

**System assessment metrics**

In order to evaluate how the system performs using the strategies previously introduced, both the network and the server aspects have to be considered.

Existing simulators either focus on the network, assuming a constant processing cost at the server, or they accurately model server processing, but use a static estimate for the network transfer time. Therefore, the NS simulator has been enhanced with computation capabilities as explained in Section 3.4. In this way the simulator is able to provide both network and server load functionalities. It is worth mentioning that the aim of this study is to show the algorithm applicability. Therefore, the extension developed does not model a server processing behavior accurately, but it provides a model that is sufficiently reliable for the purpose of this study. The same applies to the choice of the strategies, which are relatively simple in this study. However the system is general enough to easily realise a more elaborate behaviour.

The system performance can be evaluated in terms of system capacity and execution time. The former evaluates the number of requests per seconds that the system is able to manage. The latter gives the amount of time spent by the system to accomplish all the requests. However, since these two metrics are often inversely proportional, a third metric that gives a trade-off between them is defined as the ratio between the normalised values of system capacity and response time. The normalisation is made by dividing the relative metric by the smallest value.

Simulations were run for different request rates, from a low rate where all the requests can be satisfied to higher rates where the system is overwhelmed by requests. The number of job requests the system is able to serve gives the system capacity; while the amount of time spent by the system to serve all the requests gives the system response time.

Table 6-1 shows the scenarios used for this study. Considering the job request definition defined in Section 3.4.2.3, two degrees of heterogeneity are considered: 1) Null, which means that all servers, jobs and jobs priorities are the same; and 2) High, the
parameters are highly heterogeneous, e.g. servers can be of class 1 or class 5. The set of scenarios considers a case in which all the parameters have Null heterogeneity, three cases in which only one parameter is highly heterogeneous and a final case where all parameters are heterogeneous.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Servers</td>
<td>Null</td>
<td>High</td>
<td>Null</td>
<td>Null</td>
<td>High</td>
</tr>
<tr>
<td>Jobs Priorities</td>
<td>Null</td>
<td>Null</td>
<td>High</td>
<td>Null</td>
<td>High</td>
</tr>
<tr>
<td>Jobs</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 6-1 Overlay resource management scenarios.

Performance results

Our findings show the benefit to have in place a resource management strategy. When the system has no strategy, the servers are quickly loaded and the system is not able to serve all the requests, despite having several unutilised resources.

The three metrics are plotted against the job requests rate. For low rate, each strategy is able to perform all the requests, while for increasing rates some of the jobs are not performed.

Figure 6.21 illustrates scenario 1, where there is no heterogeneity in any of the parameters. From the trade-off ratio graph, the proximity strategy on average performs better than the other ones (i.e. it is on average the highest for different rates).

In scenario 2, Figure 6.22, the load balance strategy has the best trade-off ratio (on average). This is due to the resource servers’ heterogeneity, which advantages this strategy.

Figure 6.23 shows scenario 3, where the jobs priority is highly heterogeneous. The best trade-off ratio is given by the proximity strategy, which supports jobs with high priority.

Scenario 4 in Figure 6.24 again is best performed by the proximity strategy, which in this case benefits jobs heterogeneity.

Finally, Figure 6.25 shows scenario 5 that is characterised by high heterogeneity of all parameters. Here, there is not a best strategy. All strategies perform on average the same.
Figure 6.21 Performance metrics for Scenario 1.
Figure 6.22 Performance metrics for Scenario 2.
Figure 6.23 Performance metrics for Scenario 3.
Figure 6.24 Performance metrics for Scenario 4.
Our analysis shows that in the system some strategies perform better in some case and worse in some others, i.e. there is no particular strategy that is always preferable. These initial results can be used as hints to improve the strategies. Here, as said, the aim was to show the algorithm applicability and not to investigate the best redirection strategy.

Figure 6.25 Performance metrics for Scenario 5.
Summary

We have discussed some experiments of our MA system to three different applications. In the Application-level Multicast case, MAs act as end systems after partitioning the set of users. A comparison was made with DVMRP, representative of a native multicast algorithm, in terms of TreeBuildTime and StressRatio. Despite the expected degradation in performance the MA algorithm offers greater flexibility and can be thought of a complementary solution to the native approach for cases where a native layer 3 infrastructure is not in place. Further studies may lead to improvements that may make our solution even more competitive.

The second case study brings us a step forward in the area of adaptable content delivery. In this case the system can deal with the heterogeneity of users’ devices and preferences. The results show that the native solution is unable to cope when the number of stream formats required increases. Our CAN approach show encouraging results worth of further research in the future.

The final application was the overlay resource management system. In this case the algorithm was used to monitor a set of resource servers distributed within the network. Various job allocation strategies were performed, which showed the benefits of having a resource management in place.

The above cases studies show the algorithm flexibility to adapt to different requirements. Other applications may benefit from our algorithm. In general, all large-scale network applications that need to be scaled down for the purpose of distribution and management can benefit. We believe that with the proper adaptations the algorithm can embrace a large range of applications including mobile, ad hoc and P2P systems.
References


[61] Internet Engineering Task Force (IETF), www.ietf.org


[90] ORIGIN, www.microcal.com


[92] RFC 1075 (rfc1075) - Distance Vector Multicast Routing Protocol.


