An Investigative Study of The INFANT-Haggle

Benjamin Odiyo
Abstract

An Investigative Study of The INFANT Haggle

Benjamin Odiyo

Recently there has been a lot of interest in research on performance-challenged networks, which are typically characterized by intermittent connectivity, high mobility, and frequent network partitioning. Attempts to overcome the difficulties in this environment include opportunistic networking and the associated applications that support it. Routing and routing algorithms has been the focus of most research. A number of simulation projects on opportunistic networks routing algorithms have been produced using data from real mobility traces. These simulations are thought to be more realistic than simulations based on generic random mobility models. One of these simulation projects is the Haggle project, which is an implementation of the Pocket Switched Network for mobile computing devices.

In this thesis, we study the first implementation of the Haggle architecture, referred to as INFANT-Haggle. We present our results of profiling INFANT-Haggle, whose main objectives were to increase understanding of INFANT-Haggle, find out the performance bottleneck, and attempt to port the engine on Nokia 770 using SQLite as a database engine. The tools used were the Eclipse Test & Performance Tools Platform for profiling, the Scratchbox cross compiler, and the Maemo SDK platform for porting on Nokia 770. We discuss the methods used and present our findings.
This work is dedicated to my late parents ...
Acknowledgements

I would like to express my gratitude to all persons who have supported me throughout this work. Special thanks to my supervisor Fredrik Bjurefors and examiner Dr Christian Rohner for their continuous effort and input given to see this work complete, the members of Communication Research Group for their support and to Erica Shane Hamilton of Saybrook Graduate School for her advice into writing this document. Finally to God our father, for having given me faith, strength and will throughout the whole work.
# Contents

Abstract ii

Acknowledgements v

List of Figures viii

List of Tables ix

1 Introduction 1

2 Background and Related Work 4

2.1 Opportunistic Networking 4

2.1.1 Algorithm for infrastructure-less network 5

2.1.1.1 Dissemination algorithms 5

2.1.1.2 Context-Based Algorithms 7

2.1.2 Algorithm for infrastructure network 8

2.2 Epidemic Communication 8

2.3 Pocket Switched Network (PSN) 10

2.4 Related Work 11

3 Haggle 12

3.1 Introduction 12

3.2 Haggle architecture 14

3.3 Prototype Implementation 17

4 Profiling 19

4.1 Introduction and Aims 19

4.2 Eclipse TPTP 21

4.2.1 TPTP Architecture 22

5 Experimental Setup 25

5.1 Simulation Methodology and Environment 25

5.1.1 Haggle node simulation 25

5.1.2 Nokia Simulation 26

6 Analysis and Evaluation 28

6.1 Profiling a Haggle node 28

6.2 Testing on Nokia 770 35

6.3 SQLite Testing 35
## Table of Contents

7 Conclusion and Future Work 37  
7.1 Conclusion ........................................... 37  
7.2 Future Work ......................................... 38

A Appendix 39  
A.1 Sample Data ......................................... 39

Bibliography 43
# List of Figures

2.1 Taxonomy of routing/forwarding techniques for opportunistic networks according to Pelusi et al. ........................................... 5
2.2 Example of network coding efficiency ........................................ 7
2.3 A source S transmit message to destination D using carriers C₂ and C₃ . 9
3.1 Overview of Haggle Architecture .............................................. 14
4.1 TPTP architecture as according to Yaffe ...................................... 22
5.1 Nokia Internet Tablet OS block. .................................................. 27
6.1 Heap memory usage. ................................................................. 29
6.2 Heap usage during sending data. ............................................... 30
6.3 Multiple nodes sending data. .................................................... 31
6.4 Objects surviving garbage collection. ......................................... 32
6.5 Thread generation trend for an isolated node. ................................. 33
6.6 Thread generation trend for a data transmitting node. ..................... 33
6.7 Thread trend of data transmitting node for ≃ 5 hrs. ......................... 34
6.8 Heap memory utilisation trend of data transmitting for ≃ 5 hrs. ........ 34
6.9 Object generation trend of data transmitting node for ≃ 5 hrs. ........... 34
List of Tables

A.1 10 methods with highest processing time for a Haggle node in isolation . 40
A.2 10 methods with highest processing time for a Haggle node sending data . 41
A.3 Number of class objects Allocated and maximum heap memory utilisation 42
Chapter 1

Introduction

The field of mobile ad hoc networks (MANETs) has gained considerable interest and popularity in research over the recent years. Interest in MANETs started with heavy application in military use and then spread to the business world. This interest, together with advances in ubiquitous computing and wireless communication technology, have led to more attention being directed towards the use of MANETs in devices such laptops, PDAs, multimedia players, cell phones, and all sorts of small devices with short range wireless interface.

The exponential growth of the Internet has also increased human needs to have virtual connections, anywhere and anyhow, with or without a fixed infrastructure, leading to an increase in the need for a mobility in networking connections. While the traditional network had fixed nodes with wired connections, an ad hoc network is a multi-hop wireless network with mobile nodes. The implication of the mobility factor is that its elimination would lead to defining an ad hoc network as simply a wireless network with no central or dominant node, thus all nodes are at the same hierarchical level. MANETs are a special case of ad hoc networks where all nodes are mobile in nature, which makes them interesting to study. Their mobility and infrastructure-less factors has led to their popularity but these factors also pose challenges in the design of their communication systems, with the least being that one has to consider the incorporation of both fixed and mobile nodes.
Chapter 1. Introduction

In most designs for mobile ad hoc networks the assumption is that the physical layer of the OSI model of computer networking has the capability to cope with channel impairments, hence there is a focus mainly on the layers above the physical, such as the network and the medium access control layers. Routing has been of great interest considering that MANETs are characterised by: (a) the arbitrary topological changes of the network caused primarily by node mobility, (b) the bandwidth-constrained nature of nodes, and (c) the possibility of a few nodes crashing or leaving the network unannounced. Research in routing has led to numerous routing algorithms, some of which we will discuss, and has led to the evolution of opportunistic networks. In opportunistic networks, mobile nodes are able to communicate with each other even if the route connecting them did not exist prior to the start of communication. Furthermore, nodes do not need to possess the knowledge of the network topology. Haggle, which we study in this thesis, is an implementation of a special type of opportunistic network, one in which the mobility model is based on a study of Pocket Switched Networks [1].

In this thesis, we focus on testing and profiling Haggle. Profiling refers to several different abilities including the ability to monitor and trace events that occur during run time, the ability to track the cost of these events, as well as the ability to attribute the cost of the events to specific parts of the program [2]. In order to accomplish profiling, one needs a profiler tool that will monitor the execution of a program, gather statistics on the program execution, and log the results in an understandable form. The units of a program that can be profiled are functions and source lines that provide information such as which function has a higher frequency for execution and the amount of memory used by a given source line. For Java applications, the profiler has the ability to leverage information provided by the JVM and provides a comprehensive and correlated analysis of the profiling output. One is able to not only get a picture of the application during run time in terms of CPU consumption and memory profile, but lock contention and garbage-collector activity too. Profiling helps in exposing scalability, performance, and reliability problems. It is with these interests in mind that we profiled INFANT-Haggle implementation.

The remainder of this thesis is organised as follows. In the next chapter, we look at the concepts in opportunistic network, routing algorithm, and Pocket Switched Network. Chapter 3 discusses the Haggle architecture/protocol and its implementation, specifically Haggle 0.5 referred to as INFANT-Haggle. Chapter 4 discusses profiling and looks at
Chapter 1. *Introduction*

the Eclipse Test and Performance Tool Platform that was used in the analysis of the implementation. Chapter 5 describes the simulation environment in which we consider both the experimental setting for profiling and test run in Nokia 770. In chapter 6, we present our analysis and discussion from the profiling exercise and draw our conclusions in chapter 7.
Chapter 2

Background and Related Work

2.1 Opportunistic Networking

As we mentioned in the introduction chapter, the application area for MANETs has increased over time. The two key areas emerging from MANETs research are Mesh Networks and Opportunistic Networks [3]. In opportunistic networking, there may not be a path connecting two communicating nodes and there is no requirement for the source and destination nodes to be connected in the network at the same time. The consequence of these are longer delays in messaging, and these delays inhibit functioning for applications requiring synchronisation, but they do allow for applications such as wireless sensor networks, underwater sensor networks, Pocket Switched Networks, transportation networks, and other networks that can withstand longer delays. The current focus in our research is mainly on routing and forwarding issues.

Opportunistic networking has grown from the results of studies on Delay Tolerant Networks (DTN) [4] conducted by Delay-Tolerant Networking Research Group, a research group chartered as part of the Internet Task Force. This task force is responsible for developing specifications on the DTN architecture, the latest being RFC 4838 [5]. DTN architecture networks consist of independent internets called regions, and a system of DTN gateways connects these regions. Each region has an Internet-like connectivity within and has only occasional communication opportunities among them. These opportunities are either scheduled over a period of time or completely at random. With
DTN, it is possible to know points of possible disconnections and isolate them at gateways. Each internet relies on its own protocol stack that best suits its infrastructure, communication means, and technologies.

Routing is the most challenging task in opportunistic networking since one cannot exploit the network topology knowledge as in DTN in order to improve on packet forwarding. According to Pelusi et al [3], the taxonomy of routing/forwarding algorithms for opportunistic networks looks as follows (see Figure 2.1):

The two main categories are algorithms that target ad hoc networks without infrastructure and those algorithms that target ad hoc networks with infrastructure support.

2.1.1 Algorithm for infrastructure-less network

The algorithms targeting infrastructure-less support are further divided into context based and dissemination based.

2.1.1.1 Dissemination algorithms

Dissemination-based algorithms are simply controlled flooding algorithms. They distribute application messages to every neighbour but only apply a policy to determine
when to stop the distribution. They work on the idea that since there is neither a known path towards the destination node nor appropriate next-hop, simply forward the application messages to every node within range. They are suitable in a network where nodes are highly mobile. Their main advantage is that packet delays are limited, and the disadvantage is that they are resource hungry and often suffer high contention that may lead to network congestion. To counter network congestion problem, a policy on maximum number of a message’s relay hops to travel or maximum number of duplicates can be applied.

From Figure 2.1, the examples of dissemination algorithms are Epidemic, MV and Network Coding routing algorithms. Epidemic [6] algorithm, also sometimes called probabilistic or gossiping routing protocol, is implemented in the Haggle engine. It simply distributes message to hosts, known as carriers within a section of ad hoc network, which in turn distributes to other carriers, thereby getting the message to be distributed to the connected portion of the network. During pair-wise exchanges, the nodes examine their buffers and transfer only the missing messages. This algorithm uses message hop count to control further distribution of a message. We will look at this algorithm in much more details in later in this chapter.

Meetings and Visits(MV) [7] routing protocol extends the nodes’ capability by making them able to remember their meetings with the previous nodes. It uses the same principles as Epidemic except that the messages’ forwards are only to nodes that have a higher probability of successful delivery. Successful delivery is achieved by checking the node’s history of previous meetings with other nodes and the geographical locations visited by the node. It is a direct optimisation of Epidemic algorithm. One key assumption here is that the nodes have the capability to return to their previous geographical location.

Network-coding-based algorithms derived from network coding field in information theory are also applicable to this category. It is similar to Epidemic routing but instead of simply forwarding the messages, the nodes process the messages and send out messages with linear combinations of previously received messages. An example borrowed from Widmer et al [8] may clarify how it works. In a three-node A, B and C, topology suppose A and C want to exchange packets via an intermediate node B. A sends a packet “a” to B and C respectively sends a packet “c” to B, which then broadcasts “m” which is
“a” xor “c” instead of “a” and “c” in sequence. Both A and C can recover the packet of interest, while the number of transmissions is reduced.

2.1.1.2 Context-Based Algorithms

Context-Based routing makes use of information about the context in which the node is operating in when choosing the best next hop. While these algorithms minimise the number of duplicates for a given message in the network significantly, they can easily suffer from long delays because of errors and inaccuracy when selecting the best relay. In addition, they utilise more resources in terms of storage and processor time since they have to compute the usefulness (utility) of the neighbours and they have to keep track of that value, for use in selecting the next best hop.

Two examples from the Figure 2.1 are Context-Aware Routing (CAR) and MobySpace Routing. In CAR, each node periodically sends delivery probability towards a known destination host. This information together with other attributes such as the residual battery level, the rate of change of connectivity, the probability of being within reach of the destination, and the degree of mobility are used to determine the best carriers. The best carriers buffer the message to the destination host or forward it to a more suitable career.

MobySpace routing uses a high dimension Euclidean space, MobySpace, to determine the probability that a given next hop will lead to successful delivery. Heuristic knowledge of node mobility is used to enhance MobySpace routing. The logic here is that two people having similar mobility patterns are more likely to meet each other, and will then be more able to communicate. Using the same argument for nodes, a Euclidean virtual space is formalised and used to aid routing decision. Messages are forwarded to nodes that have a mobility pattern similar to that of the destination node. The mobility
pattern of a node provides its coordinates, referred to as its MobyPoint. Routing is done by forwarding messages toward nodes that have their MobyPoint as close as possible to the MobyPoint of the destination.

2.1.2 Algorithm for infrastructure network

Infrastructure-based algorithms depend on the message being buffered by a node until the base station, which is essentially a gateway to a less challenged network, is reached. Since the focus of Haggle is application in non-infrastructure environment, we will not dwell on these algorithms. However, it is worthy to note that there are a couple of interesting algorithms like Infostation, Share Wireless Infostation Model (SWIM), Snake Protocol, Runner’s Protocol, Virtual Mobile Nodes etcetera. More in-depth discussion of these and others can be found in Pelusi et al. [9]

2.2 Epidemic Communication

Introduced by Vahdat and Becker [6], the goals of Epidemic Routing are to maximize message delivery rate, minimize message latency, and minimize the total resources consumed in message delivery in partially connected ad hoc networks. Its main focus is to enable message delivery even when there is no path existing between the sender and destination node at the time of sending.

Epidemic routing makes use of intermediate nodes referred to as carriers to buffer the message to the destination. When the carriers are exposed to another portion of the network, it exchanges its buffer content with the nodes within its reach in that network and hence with such transitive transmission, the probability of data getting to the destination is increased. Figure 2.3 illustrates how Epidemic routing occurs.
Chapter 2. Background and Related Work

Figure 2.3 shows Epidemic Routing at a high level, with mobile nodes represented as dark circles and their wireless communication range shown as a dotted circle extending from the source. In (a), a source, S, wishes to send a message to a destination, D, but no connected path is available from S to D. S transmits its messages to its two neighbours, C_1 and C_2, that are within direct communication range. Later, as shown in (b), C_2 comes into direct communication range with another host, C_3, and transmits the message to it. C_3 is in direct range of D and finally sends the message to its destination.

Each node in the network has a buffer, whose content includes both its original messages and the messages buffered as a result of exchange with other nodes that it meets. These are stored in a hash table with a bit vector called a summary vector, for retrieval. When two nodes come into contact, they exchange the buffer content by making use of summary vectors. The first node sends its summary vector to the second node. The second node identifies the records missing from its buffer by performing a logical AND operation and the negation of its summary vector and the vector from the first node. Then it transmits the vector resulting from AND operation to the first node, which consequently forward messages whose values correspond to that vector. The same procedure is repeated for forwarding messages from the second node to the first node.

Epidemic routing uses hop count to control flooding. When the maximum hop count is reached, a message can only be forwarded if the recipient node is the destination node otherwise it will be buffered but dropped when there is need to buffer new messages. A larger hop count value will lead to a quicker distribution, leading to reduced delay time but also more total resource consumption for every message delivery. Thus, high priority messages might be marked with a high hop count, and most messages can be marked with a value close to the expected number of hops for the given network configuration to minimize resource consumption.
Each node has a buffer size and it will drop older messages to create room for new messages. It is estimated that the amount messages that can be transmitted cannot exceed the total maximum buffer size available within the network [6]. As per buffering policy, Fair Queuing algorithm [10], including Weighted Fair Queuing is argued as usable for logical distribution of buffers among competing host.

### 2.3 Pocket Switched Network (PSN)

Pocket Switched Network (PSN) [11] is the environment in which Haggle is to operate in and reflects the current mobile information user environment. Today’s mobile information users probably have one or more devices, some/all of which may be with them at any time as they move around. In so moving, they find themselves in “islands of connectivity” that is time and again in places where they have access to infrastructure and such access points (APs) allow them to communicate with other nodes via the Internet followed by regions where infrastructure connection is absent. Occasionally they find themselves within a wireless range of other devices (either stationary or carried by other users) and are able to exchange data directly with those devices.

In PSN, there are three possible methods by which data can be transferred, namely (a) neighbourhood connectivity to other local devices, (b) infrastructure connectivity to the global Internet, and (c) user mobility, which can physically carry data from place to place. In the former two methods one has to factor in issues like bandwidth, latency, congestion, synchronicity, the duration of the transfer opportunity, and monetary cost. In the later method mobile users acting as “data mules” can transfer significant amounts of data, and based on measurements and patterns in these movements, can be exploited. Therefore PSN applications can take advantage of all types of connectivity (neighbourhood, infrastructure, mobility) without having to specifically code for each circumstance.

In terms of usage model, the two classes in PSN are known-destination and known-sender. Known-sender needs to transfer data to a user-defined destination. The destination may be another user, all users in a certain place, users with a certain role, etc. Thus, the destination is not a single node but is instead a set of nodes with some relationship. Known-recipient is a device that requires data of some sort. The source for this data can
be any node that is reachable using any of the three connectivity types. Therefore, the endpoints of a network operation are no longer described by network-layer addresses, but are instead a set of desirable properties. As a result, general network operations no longer have single source and destination nodes but many endpoints.

Another important issue in PSN is resource management. Resources in terms of storage, network bandwidth, processing power, memory, and battery are very scarce. This situation is compounded by the fact that devices may need to use storage and network bandwidth to help forward messages for other devices. PSN allows efficient use of these limited resources by taking into account user-level priorities of tasks.

2.4 Related Work

A number of other applications make use of opportunistic networking to transport data. Typical examples are the sensor network used by biologists to track wildlife for studying their habits, mobility patterns, and migratory phenomena, and underwater sensor networks (UWSN) [12] that are used for oceanographic data collection, pollution monitoring, offshore exploration, disaster prevention, assisted navigation, and tactical surveillance applications. Using multiple Unmanned or Autonomous Underwater Vehicles (UUVs, AUVs) the information is transferable using opportunistic networking. Vehicular networks [13, 14] used in rural India provide very low cost connectivity. Communities of colleges and conferences can also make use of opportunistic networking.

Other projects that also include mobility model study are the ZebraNet [15] and Shared Wireless Infostation Model (SWIM) [16]. In these two projects, attempts have been made to develop a mobility pattern that can be used later to develop opportunistic networking routing/forwarding algorithms.
Chapter 3

Haggle

In this chapter, we will summarise the core ideas in the design and implementation of Haggle, which are detailed mainly in [17, 18].

3.1 Introduction

The Haggle project is funded under the Situated and Autonomic Communications (SAC) research area within the Future and Emerging Technologies (FET) part of the IST research program by the European Commission [19]. Situated and Autonomic Communication was created out of a need to increase more understanding of the recent advances in communication and networking and their integration in human working and socio-economic areas through long-term research. The objective is to promote research in the area of new paradigms for communication/networking systems that can be characterised as situated (i.e. reacting locally on environment and context changes), autonomously controlled, self-organising, radically distributed, technology independent and scale-free. Consequently, communication/networking should become fully scalable, task- and knowledge- driven.

Haggle is a new autonomic networking architecture designed to enable communication in the presence of intermittent network connectivity, which exploits autonomic opportunistic communication [3, 20]. This autonomic communication is necessitated by the need for ubiquitous access to applications such as email and web browsing in an always-on always-available fashion. The current OSI-layered IP-based network, used also in mobile
communication, assumes fixed network design and its performance is sometimes badly or not at all available in mobile environments such as Pocket Switched Network or Delay Tolerant Networking.

OSI-layered IP-based network presents applications with a synchronous end-to-end connectivity model using numeric addresses for endpoints. This addressing force applications to be built around the structure that includes the reliance on the networking infrastructure. The problem with synchronisation is that applications must be aware of and be able to handle changes in connectivity status or assume an always-on connectivity state. End-to-end connectivity prevents applications from using network routes that may involve inconsistent connectivity and the numeric addressing force has a reliance on a Domain Naming System (DNS), which limits the use of ad hoc connectivity in situations where fast Internet is not available or is expensive.

The main idea in Haggle is to have a data-centric architecture in which applications are not concerned with the mechanism of transporting data to the right place because Haggle can handle the task of propagating data. According to Su, J., et al [18] the four main design decisions for Haggle are:

1. Data persists inside Haggle. That is, Haggle must manage persistent data storage for applications rather than the applications themselves.

2. Inclusion of networking protocols support in Haggle thus allowing data-centric abstraction to applications rather than connection-centric.

3. Haggle makes use of name graphs that support late binding thus allowing mapping from user-level to protocol-specific names for specifying the various ways to get user level names. The name graphs have to be protocol independent.

4. A centralised resource management that allows Haggle to make decisions based on cost/benefit assessments of how resources are utilised. These assessments mainly concern tasks that will be performed on each network interface at any given moment.
3.2 Haggle architecture.

From a higher level, Haggle consists of six managers namely data, name, connectivity, protocol, forwarding, and resource manager. Figure 3.1 shows an overview of the Haggle architecture.

Haggle is layer-less, in that data and control signals are not passed up and down between layers, but all managers provide abstract interfaces through which other managers can communicate. Since the managers are modular and are replaceable individually, Haggle is able to maintain the key value of layering without having a layer structure. A brief description of the managers is as follows:

1. The Data Manager

   The Data Manager provides an interface for defining and manipulating data objects. Haggle data format is designed to be both structured and searchable. A data object is composed of many attributes, each of which is a pair of a type and value. Related data objects are linked into a directed graph, hence enabling retrieval of
Chapter 3. *Haggle*

embedded objects and attachments. Data can be queried, either by using unique ID or by the use of data filters. A data filter is a set of regular expressions-like queries about the attributes of a data object. They can be persistent or non-persistent.

2. The Name Manager

The Name Manager is responsible for providing the endpoint description for data transmission. The naming notations used allow late-binding of user-level names similar to the approach in internet indirection infrastructure [21]. These notations are achieved by using name graphs, which are hierarchical descriptions of all known mappings from a user-level endpoint to lower-level names and using the whole name graph as recipient of the message, both at source node and at intermediate nodes. Special data objects contain that name graph as a string that is represented in name graphs. The name on the graph is mapped to a transmission method if there is a protocol that understands the name; hence, there are different methods of mapping names to a transmission method. Before transmission, a name object is constructed. This could be from message-id mappings stored in application, names gathered at connectivity, names discovered via the existence of other neighbours, names from messages received or from trusted internet servers.

3. The Connectivity Manager

The Connectivity Manager encapsulates connectivity objects and initializes them at the start. A connectivity object supports a physical network interface by defining functionality for neighbour discovery, opening/using/closing communication channels and estimating the cost (in terms of money, time and energy) of performing a network operation. Each network interface on a node is represented by a connectivity object, which is regarded by the Resource Manager as a schedulable resource. A connectivity object also interacts with protocols by providing them with a list of neighbours discovered during neighbour discovery. A neighbour is a potential next-hope defined by protocol, data type and named object. There are two categories of neighbours, non-Internet and Internet. The non-Internet neighbours are direct next hops running a Haggle engine whereas Internet neighbours are next hops supporting IP for accessing the Internet.

4. Protocol Manager

The Protocol Manager encapsulates a set of protocols. A protocol is a method
by which data objects can be forwarded to a name such as SMPT, HTTP, direct peer-to-peer, and simply monitors neighbours to determine which name object to deliver. Protocols accepting incoming connections must provide enough information for data to be directed to that protocol.

5. Forwarding Manager

The Forwarding Manager encapsulates forwarding algorithms, sends forwarding task to the resource manager and provides API to applications for sending data remotely. When applications make requests for data transfer, the forwarding manager constructs a forwarding object, which is a data object linked to destination name object graph, the set of data objects to be transferred, and source name object, with its metadata containing forwarding operation. The forwarding algorithm determines the next suitable hop by using information on forwarding object and an estimate factor of the probability that sending it that way would result in successful end-to-end delivery. The current implementation include direct forwarding, which proposes to send the forwarding object if it can directly reach a name object within its name graph and epidemic forwarding algorithm. This latter algorithm can send data to every reachable node (see 2.2). Haggle has the capability of using other MANET algorithms too. For each forwarding object that is to be sent, the Forwarding Manager creates a forwarding task to be executed by the Resource Manager.

6. The Resource Manager

The Resource Manager’s mainly responsibility is to decide upon the task to be executed. A task is defined by the method of achieving it, the benefit, and cost of executing it. A task cost is expressed in terms of energy, money (the cost of using the infrastructure), and time-on-network (connectivity-specific nature). A task benefit refers to the estimated utility to the end user of executing it, which in itself consists of forwarding, application and user benefits.

A task is either an asynchronous (one that is called asynchronously by the Resource Manager when it wants to execute the task) or an immediate (one that requires immediate execution such as making connection to incoming network). Some asynchronous tasks can be persistent; and therefore are recallable for a later execution. For these kinds of tasks, the benefits and the costs vary with time and are recalculated with the assistance of the connectivity object. In addition, the
Resource Manager can continue executing a task if the scope of work that is being done by the task increases over the initial cost/benefits. The continuous execution is achieved by synchronously polling for permission to authorise continuation of the task.

The security aspect has not been implemented and the proposal is to use standard security techniques such as encryption, access control, and data signing [22]. For connections that require certificate of authority, the proposal is to accept but mark them as “untrusted” until an infrastructure access is available for validation. In addition, since Haggle exposes name graphs, which might contain sensitive information, a trust restriction to a particular group could be used until data has been encrypted to the level where intermediate nodes are no longer able to decipher information from the received name graph.

### 3.3 Prototype Implementation

The implementation is based on the Java programming language. At the beginning, MySQL is used as SQL back-end with the database containing only two tables, attributes (DOID, type, value), and links (head-DOID, tail-DOID). We later modified this in order to investigate the performance when using SQLite.

In task execution, the Resource Manager only considered time-on-network in calculating a score for cost/benefit. Only the top scoring asynchronous task is executed at a time and for immediate tasks, and only those resulting from Peer-to-Peer (P2P) protocol and receiving an incoming network connection were executed.

For connectivity, at the beginning it only focused on IEEE 802.11 with the drivers for Windows XP, but this has been modified to include Linux Wifi drivers and protocol for Ethernet connection (PlainIP). The IEEE 802.11 is used to discover neighbours in ad hoc mode and access points (AP) while IP is used to discover neighbours in which there is infrastructure.

The implemented Peer-to-Peer protocol enables forwarding of objects between neighbouring Haggle nodes. The name graph for a Haggle node is constructed with the root name object being global unique identifier GUID [23] followed by name objects of the
connectivity’s MAC addresses as children. Similarly, for a new neighbour discovered, the name manager generates a name object for the neighbours MAC address and checks if there is a similar name object on it name graph. If there is no similar name object present, the node initiates a name graph exchange to the new node by sending a forwarding object containing its name graph. Upon receiving the forwarding object from the new node, it reconciles its knowledge base by applying data object filters.
Chapter 4

Profiling

4.1 Introduction and Aims

As mentioned in chapter 1, profiling is a very important part of system development. In this chapter, we discuss profiling and look at the tool we used in profiling the INFANT-Haggle implementation.

Program profiles are collected to identify system resource utilisation by that particular program. The collected information may be fed into a profile-based optimization system to be used in performance tuning or to aid in program understanding. Traditionally, program profiling was performed offline, that is, a program profile is collected in a separate preparatory run of the program and the information is then consumed afterwards, for example, during a re-compilation of the program.

With the current use of dynamic compilation systems such as just-in-time compilers, there is a need to have more profile information to be collected and consumed online, that is, within the same run. Online profiling requires a low profiling overhead and gives predictions of the program behaviour instead of summaries as in offline profiling. It mainly focus on hot paths [24] rather than establishing precise relative frequency variations between the profiled units. Attempts have been made to use hardware schemes for online profiling but the main challenge is that there are difficulties resulting from dynamic input sources, thus online profiling remains dominantly a software prediction scheme.
Another way to categorise profilers according to Viswanathan et al [2] is to group them into those that provide information to the programmers and those that give feedback to the compiler or run-time system. This categorisation leads to the difference in system requirement even though the aim is the same. For example, a profiler that sends feedback to the run-time system must incur as little overhead as possible so that it does not slow down the program execution while a profiler that constructs the complete call graph may be permitted to slow down the program execution significantly.

During profiling, a software prediction scheme can collect frequency information through instrumentation or emulation. In java programs, instrumentation inserts extra byte-codes to the class methods for gathering data to be utilized by the profiling tool. This is required in order to obtain information on its execution behaviour. Although there are different levels of instrumentation, it is important for the instrumentation overhead to be at minimum level for accurate measurements. It is difficult to manually instrument Java programs for every performance aspects needed to be analysed. The alternative used is an automated instrumentation based on a description of the behavioural aspects that the performance analyst is interested in.

The profiling tool that is used monitors data from instrumentation process. There are two types of monitoring: time-driven monitoring and event-driven monitoring. Time-driven monitoring, also known as sampling, observes the state of the monitored system at certain time intervals. For instance, by observing the call-stack at every millisecond a list of methods using the most processing time can be obtained. Time-driven monitoring does not provide complete behavioural information, only the snapshots. Event-driven monitoring is a monitoring technique where events in the system are observed. An event represents a unit of behaviour, for example the creation of a new thread.

There are several factors that often contribute to the footprint of a given program. In java applications RAM footprint are mainly due to Objects, Classes, Threads, Native data structures, and Native code [25]. The relative memory consumption associated with each item varies across applications, runtime environments, and platforms.

The Eclipse Test and Performance Tool Platform (TPTP) used to profile the INFANT-Haggle implementation, uses JVMPi [2] and JVMTI [26] for instrumentation and collect data through the TPTP agent. In the following section we discuss the Eclipse TPTP in detail.
4.2 Eclipse TPTP

The Eclipse Test and Performance Tool Platform (TPTP) started as an Eclipse tool subproject Hyades, in 2001 and was promoted to a Eclipse top-level project in 2004. It is an open source platform that supplies frameworks for Automated Software Quality (ASQ) tools, services and data models to enable the development of integrated testing, tracing, profiling and monitoring tools. As a top-level project within the Eclipse project, its primary goal is to provide a common base upon which additional open source or in-house test and performance tools can be built, thereby reducing the cost and complexity of implementing effective automated software quality control processes. It brings together four subprojects that are also the definers of its functionality. A brief summary subprojects as provided by Eclipse Foundation [27] is:

- The TPTP Platform Project provides a common infrastructure and the capability for other TPTP projects to expand and specialize. It contains a common user interface, a standard data models, a data collection and communications control, as well as a remote execution environments.

- The TPTP Monitoring Project addresses the monitoring and logging phases of the application lifecycle and provides a framework for building monitoring tools by extending the TPTP Platform. The framework has the capability of collecting and analyzing system and application resources.

- The TPTP Testing Project addresses the testing phase of the application lifecycle, provides a framework for building testing tools by extending the TPTP Platform. The framework contains testing editors, deployment, and execution of tests, execution environments and associated execution history analysis and reporting.

- The TPTP Trace and Profiling addresses the tracing and profiling phases of the application lifecycle, provides a framework for building tracing and profiling tools by extending the TPTP Platform. The framework has the capability of collecting and analyzing application performance information.
4.2.1 TPTP Architecture

Figure 4.1 is an overview of the TPTP architecture. The architecture includes two sub-frameworks, a test control framework and a data collection framework. The test control framework handles tests by use of the test agent control interface, test engine and testability interface. The test engine is responsible for generating stimuli for the software under test. The data collection framework includes functionality for collecting and importing trace and log data generated into the Eclipse Modelling Framework (EMF).

The architecture also defines four EMF-based data models, that is, Test, Statistical, Logging, and Trace Model. The data models are abstract descriptions in UML of the types of assets (tests, traces, logs etc.) that the TPTP project handles and their concrete implementations are provided through the EMF. This implementation allows tools inside the Eclipse Workbench to manipulate objects, which correspond to these abstract descriptions that are managed and persisted inside the eclipse asset model. The models are stored on the file system as XML Metadata Interchange (XMI) format and accessed through EMF generated API, hence individual tool vendors do not have to go through the effort of manually constructing code to map into XMI format, but they simply make use of the EMF generated API. Furthermore, if two tools both speak to that particular API they can interoperate through shared use of the stored assets inside the asset
management model provided by Eclipse [27].

The data models are standard. The test data model is based on an early draft of the UML 2 Test Profile (U2TP) defined by the Object Management Group [28]. The log model is derived directly from a standard currently proposed at Organization for the Advancement of Structured Information Standards (OASIS) [29] known as Common Base Event. The trace model is ad hoc, but maps quite closely to the data formats of Java Virtual Machine Profiling Interface (JVMPI) and Java Virtual Machine Tool Interface. The statistical model is also ad-hoc, but maps well onto the formats used by Java Management Extension (JMX) and the Microsoft Performance Monitor (PerfMon) [30] counters.

There is a common framework for accessing data collection and test-execution agents support the data models. The agents can be running either locally or remotely. There are classes to discover and manage agents and the data they provide. An agent produces data in one of the formats recognised by TPTP after which a standard data loader reads the data and populates the data model. The other alternative is where the agent developer provides a custom data loader in cases in which the data format is not recognisable by TPTP.

An agent is a logical object that exposes services through the TPTP Agent Controller. The Agent Controller manages access to and control of agent running on the target system. A dynamically loaded transport-layer manages the actual communication between components, although additional mechanisms can be supported through custom library written to the TPTP transport layer interface. The transport-layer interface classes are independent of the client and the agents hence can easily be extended to reflect the users need.

An external component establishing a connection to the Agent Controller is assigned a unique ID and treated as a logical object. Should the object send the Agent Controller request via a RegisterAgent command, it is declared as an agent. The clients can then access it by specifying its name or through a query that is based on the command interface that they support. The Agent Controller uses a standard but extensible XML-based protocol for sending asynchronous command back and forth between agents and clients.
Chapter 4. Profiling

TPTP command elements are XML fragments and have attributes such as source, destination, and context that they use for routing them. Command-specific information that is intended for the recipient is implemented through a sub-element.
Chapter 5

Experimental Setup

5.1 Simulation Methodology and Environment

5.1.1 Haggle node simulation

The main purpose for this profiling was to investigate reliability, scalability and performance bottleneck in the implementation of the INFANT Haggle on hand held devices characterised by limited computing resources in terms processor capacity and memory. Reliability in terms of the Haggle node being able to perform data transmission effectively, scalability; the ability for the same node performance characteristic when deployed in higher amount and performance bottleneck; what parts of the implemented system caused poor performance both on resource utilisation and fulfilment of requirement specification. We examine a single node in absence of the other virtual nodes in order to examine utilisation of system resources, which are primarily the processor time and the memory there by being able to establish performance bottleneck. We further compared the node behaviour in presence of the other virtual nodes to establish consistency. This Haggle implementation was released under GNU GPL License and therefore we used mainly open source tools to perform the analysis. We choose Ubuntu Linux as the operating system and kernel-based virtual machines (KVM) \[31\] virtualisation. KVM takes advantage of the hardware built in virtualisation, but it also requires a software based virtualisation solution, such as QEMU, in order to function.
In implementing virtual system we had five virtual more nodes that could run on the same computer. In having virtual machines we could easily evaluate performance of one Haggle node in presence of other nodes thereby analysing possible scalability issues. All the nodes including the virtual machines were running on laptop with 2.0GHz Intel Core 2 Duo T7300 processor, Mobile Intel PM965 chipset, 3GB DDR2 memory, Intel Wireless LAN 802.11a/b/g MOW mini-pci card and Intel Gigabit Network Connection (10/100/1000 NIC). The virtual machines where each Haggle engine was running were allocated 3 Gigabytes of hard disk space and 512 MB memory with the virtual machines network interfaces were bridged through the Ethernet card interface. During profiling, the initial we initially set the JVM to use 5 MB with of memory with an allowance of increment unto 7 MB in size.

The main disadvantage of running Haggle node in this manner is that changes on resource utilisation and performance characteristics due to node movement cannot be accountable as the setting assumes a setting of stationary node. Yet the advantage is that that we can measure the consumption of system resource with a relatively higher precision since the nodes are having similar hardware platform as compared to having the different nodes running on different machines.

5.1.2 Nokia Simulation

We used a Maemo Software Development Kit and a Scratchbox cross compiler running on Ubuntu Linux as the setup for simulation. Scratchbox is a cross compilation toolkit for Linux application development. It allows one processor architecture (host) to compile software and another processor (target) that uses different architecture. This set-up implies that the machine on which the software is compiled cannot natively execute the software since it is intended for another processor. The motivation for cross compilation is code reuse or in this case code development.

In order to cross compile and at the same time overcome problems associated with the absence of availability of native hardware for running and testing, the host computer uses a processor emulator. Scratchbox uses QEMU, a generic open source virtual machine program for emulating processor and system. It was originally developed by Fabrice Bellard [32] and released under the GNU license. One of the architecture supports is
Advanced RISC Machine (ARM). The Nokia 770 Internet Tablet used for the experiment has 252–MHz ARM-Based StateplaceTexas Instruments OMAP 1710 processor. Its operating system, the latest as at the point of writing is OS 2006 edition version 3.2006.49-2, is built on a Debian file system Linux Kernel 2.6 source [33, 34]. We used version 2.2006 for our experiments. The block diagram for the operating system is as follows:

Figure 5.1: Nokia Internet Tablet OS block.

Figure 5.1 show how the Maemo SDK maps into the Nokia Internet Tablet OS architecture. The Nokia software is proprietary to Nokia and is available in the form of binaries. The third party software is mainly commercial software developed to support applications.

The Maemo platform is built around the Debian Linux distribution and was developed by Nokia as part of its development process for the Internet Tablet. It was subsequently open sourced and offered to community [35]. It mainly targets open source developers and innovation houses developing applications, as well as new technologies for Linux based internet connected handheld devices. The development environment is based around Scratchbox. Maemo provides Scratchbox compatible root- straps, which are software packages containing all the development libraries and header files required for application development.
Chapter 6

Analysis and Evaluation

6.1 Profiling a Haggle node.

In this section we look at the analysis of the profiling results performed on INFANT-Haggle engine implementation in order to understand resource utilisation, which is mainly CPU speed and availability and System memory. We look at principally two aspects, the behaviour of a single node when not in the presence of other nodes.

The approach was to look at both class objects and their member functions. Looking at class objects gives a view of the number of instances created, how they are referenced, how many of those instances are collected by the garbage collector (GC), the number of objects in the surviving generations and hence exposing any possibility of high usage and memory leak. The execution flow exposes resource intensive methods that may require optimisation. Candidate methods for optimisation were the ones that had higher call frequency than expected or had long execution time.

In searching for performance bottlenecks, we started by selecting packages that had higher execution time. The choice was according to the base time, which is the time taken to execute the method invocation, excluding the time spent in other methods that were called during the invocation. Starting with a package, we went into individual classes and then into member functions of those classes. Having identified a member function, we traced the method’s invocation details. These details give a summary of the call frequency and the execution time. In addition, they give details of the other methods that the current method invokes and the ones that invoke it.
From the method invocation details, the process of identifying the resource intensive is more of a recursive exercise that looks at the methods being invoked by the current method, selects the one with the highest base time and repeats the same procedure. The process ends when the current method belongs to a non-haggle package class, that is, it is either from an external library or java core foundation classes

The results from the profiling were as follows:

**Heap memory usage**

The following were our results obtained from examining heap memory usage.

When looking at Figure 6.1, the trends in heap memory usage remain the same whether a node is in the neighbourhood of others or is in isolation for a short period. We observed a similar pattern of the heap usages even when there was variation of the heap memory allocation.
Figure 6.2 shows the heap memory utilisation for a node sending data. The usage trend curve shows a pattern depicting a continuous increase in heap memory usage when the node transmits data. A further analysis, as seen in Figure 6.4, showed an increase in the number of objects surviving garbage collection. There was an increase in class objects generations but there was no equally corresponding removal of these class objects through garbage collection. This implies that there were some objects created but not removed after from one instance of garbage collection to the next. The normal condition would be a situation where after a given, the class objects generation would reach an optimal level where the number of objects generated would have a corresponding equal number in garbage collection.
Figure 6.3 shows the effects on heap memory utilisation as the number of nodes increases. We observed that working with limited memory set to 68 MB the nodes were able to use up all the allocated memory. The trend was that with every extra node introduced in the network, it took lesser time for the heap maximum to be reached. This trend is a clear indication that the performance would deteriorate significantly, especially in hand-held devices that may not have a similar amount for heap memory available and that the current implementation is not scalable at after just a few nodes.
Figure 6.4 shows the number of objects surviving garbage collection. The instrumentation was based on 10 percent of the class objects created, that is, only 10 percent of the class objects generated were being monitored. Table A.3 show the the classes with the highest number of objects allocated at the point when the node run out of heap memory. We observed that haggle.data.DO and haggle.run.LogHandler$Entry had the highest number of objects. The number of heap allocations to DO increased with time and was not dependent on the number of nodes in the network. This shows that the more DO are generated and held in the memory and not influenced by external events in the network. Similar observations are made on haggle.run.LogHandler$Entry.

**Processing time**

In respect to processing time spent in executing methods, there was consistent pattern whether executing a single node in absence of other nodes or in presence of other nodes. Table A.1 and Table A.2 in appendix A show the top 10 class methods that had the highest processor time.
Thread generation

There was a consistent pattern in the number of thread generated whether a node is in isolation or in presence of other nodes as shown in Figure 6.5 and Figure 6.6.

In Figure 6.5 and Figure 6.6, we observed stabilisation at a thread count of 14, whether the node is in the presence of other nodes or not for a short period. One exception that we observed was a situation in which the nodes were involved in active transmission...
over a long period of time (in this case about 5 hours). In this case, the number of threads fluctuates and the heap memory usage continuously increased as shown in the Figure 6.7.

**Figure 6.7**: Thread trend of data transmitting node for $\approx 5$ hrs.

**Figure 6.8**: Heap memory utilisation trend of data transmitting for $\approx 5$ hrs.

**Figure 6.9**: Object generation trend of data transmitting node for $\approx 5$ hrs.
Figure 6.7 shows the thread generation trend fluctuating even at point of time when the node was expected to have stabilised. The fluctuation was mainly due to redundancy in thread execution designed to support fault tolerance. The redundant threads in turn generated more class objects that survived garbage collection as seen in Figure 6.9 and increased in the heap memory used. It also visible from Figure 6.9 and Figure 6.7 that the thread count decrease drastically when more objects are forced for garbage collection. Forced garbage collection takes a longer time to collect objects, as shown in Figure 6.9. In Figure 6.8, we observed that the java virtual machine increases the heap memory allocation in order to accommodate more demands for the node. This trend shows that the node will eventually run out of memory since the demand will be more than virtual machine can allocate.

6.2 Testing on Nokia 770

When testing for Nokia 770 on scratch box, we used the JamVM [36] virtual machine to execute the application. While we were able to do so, it was difficult to verify the validity of performance mainly because of two reasons. The first reason being that JamVM designed to use the GNU Classpath Java class, which in itself is very limited, compared to the Sun JDK in which the application was developed. For example, the GNU Classpath is missing about 50 methods that are available in Sun’s java.sql package. Secondly, JamVM only works in Maemo’s developer SDK, which currently supports only USB networking. Dependence on USB networking made it difficult to test the ability of a Haggle node to detect the presence of a neighbour within its range. The sample applications that were used during the testing required an Ethernet or 802.11 network interface. Accurate testing with Nokia 770 would require an OS that is compatible with Sun Java virtual machine and offers support for either Ethernet or 802.11 networking protocol.

6.3 SQLite Testing

SQLite as DBM testing was done using JDBC drivers with wrappers to connect to a temporary database. Our findings were that current implementation of Haggle engine can be engine support SQLite DBMS. We observed that the required two tables (objects
and attributes tables) have been created and correct data updated. Listing 6.1 shows a SQLite dump from the database.

```
PRAGMA foreign_keys = OFF;
BEGIN TRANSACTION;
CREATE TABLE object (id integer primary key autoincrement,
owner integer default NULL, owned integer default NULL);
INSERT INTO "object" VALUES (-256, NULL, NULL);
INSERT INTO "object" VALUES (-9, NULL, NULL);
INSERT INTO "object" VALUES (-8, NULL, NULL);
INSERT INTO "object" VALUES (-7, NULL, NULL);
INSERT INTO "object" VALUES (-6, NULL, NULL);
INSERT INTO "object" VALUES (-5, NULL, NULL);
INSERT INTO "object" VALUES (-1, NULL, NULL);
INSERT INTO "object" VALUES (1, NULL, NULL);
INSERT INTO "object" VALUES (2, NULL, NULL);
CREATE TABLE attribute (id integer primary key autoincrement,
object integer, type varchar(64), value blob,
foreign key (object) references object(id));
INSERT INTO "attribute" VALUES (1, 1, 'DataObject(UUID',
X'313236333136333735393430332D2D2333935337373832323039393833333130373836');
INSERT INTO "attribute" VALUES (2, 1, 'SolicitationHandler.NODE',
X'536F6C69636974617468696F6E2E48414E444C54522E48414E44');
DELETE FROM sqlite_sequence;
INSERT INTO "sqlite_sequence" VALUES ('object', 2);
INSERT INTO "sqlite_sequence" VALUES ('attribute', 2);
COMMIT;
```

Listing 6.1: Haggle SQLite dump
Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we have looked at the performance of a Haggle node by simulating interaction between Haggle nodes using virtual machines. We have seen that a single isolated Haggle node performed both in terms of processor time and memory utilisation as expected. When a Haggle node is exposed to the network of other nodes, it runs into out-of-heap-memory error. The time it takes for a node to experience out-of-heap-memory error depends on the number of active nodes within the network. An increase in the number of active nodes leads to a reduction in the time it takes for a node to exhaust heap memory allocation. This relationship shows that the current implementation of Haggle is not scalable in just a few nodes.

A further analysis showed that there were objects being generated randomly and held in the memory irrespective of any activity on the network. The main ones were the haggle.data.DO and haggle.run.LogHandler$Entry objects. A corresponding relationship is also observed on number of threads generated. While threads should not have impact on RAM footprint, but their invocations do require space to store their stack state and the system-specific data structures do consume memory. The number of objects continued to increase even after the loaded class count remain relative stable during transmission. This increase in number of objects in heap memory is due to thread random invocations. These thread invocations are not response to network activity but internal design. We
propose that these threads are redesign so that they are event driven in response to network activity rather than randomly executed.

We have observed also that the Haggle engine support SQLite DBMS but the engine has limitations in porting Haggle on Nokia 770. This limitation is basically due to the implementation of Java virtual machine in Nokia 770 rather than the engine itself. Our proposal is a reimplementation of Haggle in a language that is compilable into machine native such as C rather than dependence on Sun’s Java virtual machine for use in hand held devices.

7.2 Future Work

There is need to investigate how a Haggle node performs in mobile devices that support other Java virtual machines, especially Sun’s J2ME or K virtual machines. This investigation would give a wider view about its performance on hand held devices that support virtual machines that are closer to the Sun’s Java virtual machine.

Secondly, a redesign of fault tolerance approach is needed, such that the threads are executed based on events occurrence rather than random time function.

Finally, a reimplementation in a language that is compilable to machine native format, such as C, is necessary in order to overcome the problems associated with Java virtual machine. This kind of reimplementation will also widens the scope of available hand held devices that can execute Haggle node.
Appendix A

Appendix

A.1 Sample Data

Table A.1 and Table A.2 show the time taken for different methods of the Haggle engine. The list comprise of only the ten methods with the highest processing time. The definitions for the columns are as follows:

- **Base Time**: The amount of time (in seconds) the method has taken to execute. Not including the execution time of any other methods called from this method.

- **Average base time**: The average base time required to execute this method once.

- **Cumulative base time**: The amount of time (in seconds) the method took to execute including the execution time of any other methods called from this method.

- **Calls**: The number of times the method was invoked

Table A.3 shows classes with the highest number of total objects allocated among the to at the point the maximum heap memory allocated was fully utilised.
### Table A.1: 10 methods with highest processing time for a Haggler node in isolation

<table>
<thead>
<tr>
<th>Method</th>
<th>Base Time (sec)</th>
<th>Average Base Time (sec)</th>
<th>Cumulative Time (sec)</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>SendDaemon.getSendMessages() java.util.List</td>
<td>192.825515</td>
<td>0.001304</td>
<td>192.825515</td>
<td>147881</td>
</tr>
<tr>
<td>SendDaemon.access$200(haggle.app.sendsocket.SendDaemon) java.util.List</td>
<td>30.565523</td>
<td>0.000164</td>
<td>232.391038</td>
<td>241487</td>
</tr>
<tr>
<td>SendDaemon.access$300() java.util.HashMap</td>
<td>6.181287</td>
<td>0.000078</td>
<td>6.181287</td>
<td>79732</td>
</tr>
<tr>
<td>DO.DO(int)</td>
<td>1.790331</td>
<td>0.000196</td>
<td>2.618142</td>
<td>9144</td>
</tr>
<tr>
<td>DefaultNameManager.getTopLevelNames(haggle.data.DO, java.util.Set, java.util.Set) void</td>
<td>3.13427</td>
<td>0.000867</td>
<td>16.25002</td>
<td>3616</td>
</tr>
<tr>
<td>DefaultNameManager.getName(int) haggle.data.Name</td>
<td>2.506011</td>
<td>0.000453</td>
<td>14.176974</td>
<td>5529</td>
</tr>
<tr>
<td>DefaultResourceManager.selectBestCandidateTask(haggle.connectivity.Connectivity java.util.Map)</td>
<td>2.328128</td>
<td>0.00175</td>
<td>12.446925</td>
<td>1330</td>
</tr>
<tr>
<td>DefaultResourceManager.doCostBenefit(int, haggle.resource.Cost, haggle.resource.Benefit) int</td>
<td>2.109733</td>
<td>0.000796</td>
<td>5.984602</td>
<td>2650</td>
</tr>
<tr>
<td>Haggle$LogCommitHandler.commit(java.util.LinkedList) void</td>
<td>1.863479</td>
<td>0.00617</td>
<td>7.236923</td>
<td>302</td>
</tr>
<tr>
<td>LogHandler$Entry.toString() java.lang.String</td>
<td>1.677968</td>
<td>0.00036</td>
<td>4.002353</td>
<td>4662</td>
</tr>
</tbody>
</table>
Table A.2: 10 methods with highest processing time for a Haggle node sending data

<table>
<thead>
<tr>
<th>Method</th>
<th>Base Time (sec)</th>
<th>Average Base Time (sec)</th>
<th>Cumulative Time (sec)</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>SendDaemon.getSendMessages() java.util.List</td>
<td>28,111.93</td>
<td>0.001304</td>
<td>28,111.93</td>
<td>21561415</td>
</tr>
<tr>
<td>SendDaemon.access$0(haggle.app.sendsocket.SendDaemon) java.util.List</td>
<td>5,305.12</td>
<td>0.000154</td>
<td>33,417.04</td>
<td>34389171</td>
</tr>
<tr>
<td>SendDaemon.access$1() java.util.HashMap</td>
<td>469.30423</td>
<td>0.000075</td>
<td>469.30425</td>
<td>6290029</td>
</tr>
<tr>
<td>DO.hashCode() int</td>
<td>216.50832</td>
<td>0.000061</td>
<td>216.508322</td>
<td>3560156</td>
</tr>
<tr>
<td>DO.DO(int)</td>
<td>190.20896</td>
<td>0.000121</td>
<td>284.244567</td>
<td>1578196</td>
</tr>
<tr>
<td>DefaultNameManager.getTopLevelNames(haggle.data.DO, java.util.Set, java.util.Set) void</td>
<td>492.51151</td>
<td>0.000541</td>
<td>1,858.20</td>
<td>909552</td>
</tr>
<tr>
<td>DefaultNameManager.getName(int) haggle.data.Name</td>
<td>316.56485</td>
<td>0.000276</td>
<td>1,236.09</td>
<td>1144915</td>
</tr>
<tr>
<td>DefaultResourceManager.selectBestCandidateTask(haggle.connectivity.Connectivity) java.util.Map</td>
<td>157.08946</td>
<td>0.001579</td>
<td>858.349494</td>
<td>99476</td>
</tr>
<tr>
<td>Haggle.getDataManager() haggle.manager.DataManager</td>
<td>183.4194</td>
<td>0.00006</td>
<td>183.419404</td>
<td>3046603</td>
</tr>
<tr>
<td>Haggle$LogCommitHandler.commit(java.util.LinkedList) void</td>
<td>143.69626</td>
<td>0.005918</td>
<td>438.159445</td>
<td>24282</td>
</tr>
</tbody>
</table>
Table A.3: Number of class objects Allocated and maximum heap memory utilisation

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Number of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 Nodes</td>
</tr>
<tr>
<td>haggle.data.DO</td>
<td>19191236</td>
</tr>
<tr>
<td>haggle.data.Attribute</td>
<td>37198</td>
</tr>
<tr>
<td>haggle.run.LogHandler$Entry</td>
<td>19944</td>
</tr>
<tr>
<td>haggle.data.Name</td>
<td>19693</td>
</tr>
<tr>
<td>haggle.data.DOBuilder$Pair</td>
<td>11576</td>
</tr>
<tr>
<td>haggle.manager.data.SimpleDataManager$AttributeTuple</td>
<td>11518</td>
</tr>
<tr>
<td>haggle.manager.data.SimpleDataManager$ObjectTuple</td>
<td>9449</td>
</tr>
<tr>
<td>haggle.data.DOBuilder</td>
<td>9439</td>
</tr>
<tr>
<td>haggle.resource.Cost</td>
<td>9072</td>
</tr>
</tbody>
</table>
Bibliography


