Computation Sharing Offered within a Mobile Ad Hoc Context-Aware Network

Fintan McGee

A dissertation submitted to the University of Dublin,
in partial fulfilment of the requirements for the degree of
Master of Science in Computer Science

2007

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: ____________________

Fintan McGee

14/09/2007

Acknowledgement

I would like to thank my supervisor, Siobhán Clarke, for her support and guidance throughout the project. I would like to also thank Andronikos Nedos, for his help and direction and helping to focus the project. I would also like to thank my NDS class , (especially Andrew for giving me use of his laptop for experiments).

I would also like to thank my family for their support throughout my masters, especially when the going got tough.

Fintan McGee

University of Dublin , Trinity College
September 2007

Computation Sharing offered within a Mobile Ad Hoc Context-Aware network

Fintan McGee

University of Dublin, Trinity College, 2007

Supervisor: Siobhán Clarke

As devices supporting mobile ad-hoc networking capabilities become more pervasive there are potentially benefits, such as an increase in application throughput, in one node within the network being able to harness the processing power and functional capabilities of other nodes within the network. MANETs (Mobile Ad Hoc Networks) are however generally heterogeneous in their technology capabilities. Processing power is not guaranteed to be uniform across the network, nor is a specific functionality guaranteed to be available at any given node at a specific moment in time. Each node has its own system context describing characteristics such as processor speed, memory and processing load, as well as what services are available at that node. This system context information reflects the ability of the node to process tasks distributed from other nodes in the MANET.

This dissertation describes the design, implementation and evaluation of middleware, which disseminates context information around a MANET and distributes processing tasks to individual nodes based on the context of nodes targeted to share in the computations. Nodes are initially targeted for sharing based on the services they advertise as being available and assigned tasks based on their system context information. An API has been developed which allows an application running on a MANET node to request that some computations, organised as a job consisting of discrete tasks, to be shared across a MANET. Distribution of tasks based on context information is weighted using values supplied by the application requesting the computation sharing.

The evaluation of the middleware's task distribution uses a simple test application which submits a job for sharing. A measurement is taken of the latency from when the job is distributed to when the results are returned to the original sharing node. A comparison is done between the distribution of tasks evenly across nodes and the distribution of tasks using context information. The evaluation reveals that performance benefits in terms of test application throughput are seen when using context information to aid distribution of tasks. However the evaluation also reveals that the choice of standard underlying UDP and TCP messaging protocols limits the ability of the middleware to distribute tasks across the full span of the ad-hoc network. In order to gain as large a performance benefit as possible an enhanced message delivery protocol needs to be developed.

Table of Contents

List of Figures

Chapter 1 Introduction

This section describes the motivation, objectives, scope and contribution of this dissertation.

Motivation

As wireless technology becomes more pervasive and the ability to create networks on an ad-hoc basis containing nodes which are mobile becomes more widespread, there is an opportunities to share the computation power of the nodes of a Mobile Ad-Hoc Network (MANET) in a manner similar to GRID computing. Rather than complete a computation locally on one node of the MANET, as the processing power of the node may be too small to process the data in a reasonable time or the functionality required for the processing may not be available on the local node, the computations could be distributed across the network to suitable nodes in the MANET.

However MANETs by their nature are made up of heterogonous nodes. Nodes in a MANET may have the limited processing power of a mobile Phone or PDA or they may be the latest high power laptop or desktop. If a computation is shared among nodes, and one of the nodes has considerably low processing power, it may slow down the overall computation completion. In order to avoid this situation, a potential problem node should be sent less data to process, in proportion to their capability.

Each node in a MANET can be considered to have a system context. This is information about the node such as processor frequency, number of processor cores, processor load and memory available. This information can be used to approximate the processing capability of a node. If this context information is used to determine how individual computation tasks should be distributed across the MANET, the distribution of tasks can be optimised to improve performance. A node with low capability will no longer significantly delay the completion of the overall computation.

Objectives

The objective of this dissertation is to demonstrate how system context information can be used to improve the performance of computation sharing in a MANET. To this end middleware will be developed with the following capabilities:

- Retrieval of node context information.
- Dissemination of the context information across the MANET.
- Dissemination of service information across the network. This service information describes the functionality that can be shared.
- Dissemination of computations sharing job data across the network using context information (which can be weighted).
- Dissemination of computations sharing job data across the network evenly, without context. (For comparison to when context is used).
- Execution of services using shared computation data.
- Interface to receive computation sharing jobs from requesting applications.
- Returning of completed computations jobs to the original node and application.

In order to allow applications to use the middleware application an API will be developed to allow a local application to request for a computation to shared across the network, as well as specify context weights in the case where context information is being used to distribute the information.

Contribution

The main contribution of this dissertation is statistical evidence, gathered by experimental evaluation, concerning the impact of using context information to determine the distribution of tasks when sharing computations on a MANET. There is a further contribution in the development of middleware to support the distribution of computation sharing tasks using user weighted context information and an analysis of issues faced in building such an implementation, such as the choice of underlying message delivery protocol. An API to enable an application running locally on a node to use the middleware is also being provided.

Scope

This scope of this dissertation covers development of a real-world implementation of computation sharing middleware for ad-hoc networks. The implementation will not run on a simulator but on computers acting as nodes in a real ad-hoc network. The purpose of this implementation is to show the benefits of the use of context information when sharing computations across an ad-hoc network in a real-world scenario, as well as to illustrate the issues encountered in such an implementation, that may not be encountered within a simulation. Of particular relevance is the performance of messaging protocols such as TCP and UDP within an ad-hoc network using 802.11 wireless.

Approach

The approach taken to this project was as follows:

- Study into the state of the art or relevant areas such as GRID computing, ad-hoc networking, peer-to-peer architectures, information distribution within an as-hoc network.

- Research into the technical components of the system. This including research on libraries to provide XML parsing, networking capabilities and retrieving system context. Research on a suitable implementation of an ad-hoc routing protocol.

- Design of the computation sharing middleware

- The implementation of the middleware

- Evaluation of the middleware and the benefit of computation sharing using a real world ad-hoc network

Structure

The structure of this dissertation is as follows:

- **Chapter 1** is an introduction to the dissertation, describing the motivation , objectives contribution, and scope of the project

- **Chapter 2** gives the background on relevant research areas to the project, to help the reader put the project in context and further understand the motivation of the project.

- **Chapter 3** develops on chapter 2 by describing the state of the art of the background areas mentioned. It also describes some related work to computation sharing across a MANET

- **Chapter 4** describes the design of the middleware, highlighting what decisions were made and what alternatives approaches were considered

- **Chapter 5** describes the implementation of the project. It describes in detail using flowchart some of the more complex algorithms used by the middleware, as well as the message flow.

- **Chapter 6** describes the evaluation of the project, describing in detail 2 experiments completed as part of the evaluation.

- **Chapter 7** discusses what conclusions can be draw from the experimental evaluation described in chapter 6.

- **Chapter 8** describes options for future work, following on from this project. It also describes features that were not implemented as part of the middleware due to time constraints, should be considered for implementation as part of any future work.

Chapter 2 Background

This chapter provides background information on areas of computer science which are related to the research topic. These areas were the starting point for all research, and a presented here to familiarise the reader with concepts upon which the project is based.

Grid Computing

GRID computing, using distributed systems, has for some time been utilised to solve large scale scientific problems, which would otherwise be too computationally time consuming to solve on single machines. The objective of GRID computing is to support the sharing of resources, such as bandwidth, processing capacity, storage, between individuals and organisations[1]. The GRID represents resources as services which are accessible across a network. The use of distributed systems to share processing power is not just limited to large scientific organisations and academic institutions with significant network resources. BOINC[2] is a utility which allows a home PC to be used as a node in a GRID of home PCs located across the planet. Home users can contribute their PC processing power to projects such as the search for extra terrestrial intelligence (SETI) or climate prediction. Within the BOINC network a centralised server distributes work packets to the subscribing home PC nodes and stores the results once the computations are finished.

Peer-to-peer systems

Peer-to-Peer systems are defined by Oram et al. [3] as *"a self organising system of equal, autonomous entities (peers)* [which] *aims for the shared usage of distributed resources in a networked environment avoiding central services."*

Peer-to-Peer computing dates back to the late 1960s with the establishment of the ARPANET. The ARPANET was a physical network designed to allow research facilities in the United States to share computing resources and documents. While ARPANET lacked some of the characteristics of modern peer-to-peer networks every host on the network was treated equally therefore it can be considered to be one of the first peer to peer networks[4].
In the late 1990s peer to systems gained widespread awareness (and notoriety) with the rise of Napster, an online peer-to-peer file sharing service. Many other peer-to-peer based file sharing services have since become popular; however there is more to peer-to-peer architectures than providing a mechanism for people to download music and video. Peer–to-peer systems can allow users to store document remotely in a secure manner [5] and also allow files to be distributed in such a manner that the overloading of any central server is avoided.

Mobile Ad-Hoc Networks (MANETs)

Due to the rising pervasiveness of wireless connectivity for devices, it is increasingly easy for devices to connect to each other without any physical networking infrastructure. For example employees in an office location without access the local ethernet network can connect their laptops together using wireless to allow them to share documents. Devices connected to each other in this manner are said to be connected on an Ad-hoc basis. Each node in the network acts as both a host (which receives inbound connections) and as a router (which routes messages to their target). Due to the fact that there is not centralised network infrastructure it, and the fact that many devices have battery power (or a self contained source of power), in many cases the nodes of an ad-hoc network are free to move around in the physical world. If this is the case the network is referred to as a Mobile Ad-hoc Network or MANET for short.

In wired physical networks the network topology is generally constant. Network nodes are usually available except in the case of failure, and each node normally has fixed relationship with its neighbours. If a node joins or leaves the network, it is in a controlled manner. Nodes will generally have a fixed IP address and location. DHCP does mean that a laptop joining an office network every morning may have a varying IP address, however the laptop is a host on the network and not a router so is not a Node in the networking sense and is generally not relied to be consistently available (unless the network is being administered in an incompetent manner).

In ad-hoc networks all of the assumptions about nodes been continuously available, reachable at a known IP via a known route and having a fixed relationship with other nodes can be completely disregarded. Within an ad-hoc network (particularly a Mobile Ad-Hoc Network) the network topology may be changing all of the time. This renders the routing protocols, normally used for fixed networks, useless. There are many routing protocols which have been put forward for use in MANETs. The most interesting of these is Ad-Hoc On Demand Distance Vector (AODV). This is an on demand algorithm that determines routes to destinations only when required. AODV has been chose as the underlying routing protocol used by MANET nodes for this project.

Real-world Applications of Ad hoc Networks

Ad-hoc network are very valuable in situations where there is little existing infrastructure. Possible situations where ad-hoc networks are applicable include

- In the aftermath of natural disasters, such as earthquakes, where existing infrastructure is damaged beyond use. Ad-hoc networks can be used by emergency workers communicates, as well as distribute information to workers the help rescue teams target the right areas to focus their rescue efforts on

- During Military operations military vehicles on a battlefield may not have any access to existing infrastructure. An ad-hoc network would allow units and vehicles to communicate and keep track of each other position. It would also be possible to distribute the analysis of reconnaissance data or the decryption of encrypted enemy communications across nodes in the network.

- A fleet of vehicles in the near future maybe be able to communicate with each other using ad-hoc networking.[6] Such communications can be used to warn of impending obstacles in the road, changing weather conditions. As research progresses on computer controlled vehicles, ad-hoc networks will be vital for inter-vehicular communications.

- In many countries in the developing world there is little telecommunications or networking infrastructure. The One Laptop Per Child Program[7] (OLPC) is an education project which aims to encourage learning in children growing up in developing countries by providing them access to a (relative) cheap, robust fully functional laptop. Due to the lack of networking infrastructure each laptop is equipped with wireless functionality that allows mesh networks to be set up between laptops in an ad-hoc fashion. The routing protocol use by the OLPC laptops is a variation of the AODV routing protocol. While the laptops themselves are not exceptionally powerful there is potential for a group of laptops to network and use their combined processing power to perform computationally intensive tasks more quickly.

Context Information

When humans communicate with each other very often we use information implicit from the situation to gain a more thorough understanding of meaning. Often the context of a piece of information has a large impact on how that information should be understood. The question “Who is making that statement?” can often have an impact on the understanding of what is being said. When we make decisions, having more information, or understanding the context of the situation, allows us to make better choices.
The same logic also applies to computer applications. Context information can help a computer to make decisions which result in better performance.

A real world example of context information being used by a computer system is the London taxi company “Zingo”. When a customer calls the taxi service on their mobile phone, the position of the mobile phone is retrieved and is compared to the GPS positions of London taxis. The user is then connected directly to an available taxi driver in a nearby taxi. The context information being used here is the location of the user, which can be retrieved from the mobile phone, and the location of the taxi, which is retrieved from the taxi's GPS device. The performance benefit is a user (the customer) being connected directly to a taxi drive who is nearby and available to collect the user quickly.

The definitions of context vary from application to application and situation to situation. Dey[8] defines context as *"Any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."*

From this definition of context if the applications purpose is to process user data in a more timely fashion, the system information is a valuable piece of context. If an application is to distribute data for processing across remote nodes in a network the system information of each node may be able to inform the application about how tasks should be distributed in a manner which improves performance for the end user.

Chapter 3 State of the Art

This chapter will give an overview in the current state of the art of relevant research areas such as ad-hoc routing, grid computing, grid computing in a mobile environment and data sharing in mobile devices. It elaborates on the background information given in the previous chapter, giving specific examples of the state of the art.

Routing in ad-hoc networks

Due to the topological differences between ad-hoc networks and more traditional networks, new protocols are required for the routing of data packets. Established wired network protocols such a Distance Vector Routing and Link State Routing depend on relatively static network sand cannot be used in ad-hoc networks.

Ad Hoc On Demand Distance Vector routing

Ad-hoc On-Demand Distance Vector (AODV)[9] is a routing protocol developed for ad-hoc networks and is quite popular amongst researchers of MANETs. AODV has been chosen as the underlying routing protocol to be used in the development of middleware for this dissertation.

As its name suggests AODV is a distance vector protocol[10]. This means the algorithm works by each node maintaining a 2 dimensional vector (i.e. a table) giving the best distance known to a remote node in a network as well as the address of the next node in that route. The protocol is also on-demand. This means that the protocol only determines the route to the destination node when the source network node wants to send data to that destination node. This is what allows the protocol to adapt to the changing topology of an ad-hoc network. In order when to spot that a route to a node is no longer viable due to a node dropping out of the network, the AODV protocol periodically sends special “HELLO” messages to each of its neighbours. If a neighbour node suddenly drops out of the network, it will fail to respond to the “HELLO” message and the source of the message will know that any routes destination nodes which were accessed via the now defunct neighbour node are unavailable. This results in the routing table being purged of all routes that used the neighbour.

```
# Time: 21:34:36.578 IP: 134.226.51.176, seqno: 1 entries/active: 3/3
Destination     Next hop        HC  St. Seqno Expire Flags Iface
192.168.1.101   192.168.1.101   1   VAL 1     1829         eth1
192.168.1.100   192.168.1.100   1   VAL 1     2021         eth1
134.226.51.199  134.226.51.199  1   VAL 1     1971         eth1
```

Figure 1 Sample output from an AODV_UU Routing Table

An implementation of AODV called ADOV –UU has been developed at Uppsala University. It is freely available and is capable of running on systems which use any recent version of the Linux operating system.

GRID computing

GRID computing is a popular research topic in research institutions across the world
The Globus project[11] is one of the first and a most visible projects in the area of GRID computing. It was started in 1996 and is supported by numerous universities, such as the university of Chicago and Wisconsin scientific institutions, such as NASA and members of industry such as Microsoft and IBM.
The project is concerned with developing technologies to support the building of large scale grid applications as well as building software tools, such as the “Globus toolkit™” , which supports the development of service oriented applications for distributed computing, as well as supporting the infrastructure to do so[11].

The BOINC platform (mentioned in chapter 1) also is indicative the start of the art in grid computing. The BOINC project boast over 1,000,000 users each contributing to computations, with approximately 250,000 currently actively participating on a daily basis[12]. Ireland alone has contributed over 1000 GigaFLOPS. The BOINC platform overall performance at 683.697 TeraFLOPS[12]. FLOPS are a measure of Floating point operations per second and are an indicator of computer performance. To put these measurement in perspective the IBM BlueGene/L supercomputer, said to be the world's fastest supercomputer boast a peak speed of 360 teraFLOPs[13].

The BOINC project is not a perfect example of GRID computing as it does rely on central servers to distribute tasks to client nodes, however it does clearly demonstrate how harnessing large numbers if individual computer systems can be of benefit. In the case of BOINC the benefit is in giving scientific projects access to a vast amount of processing power that they otherwise would not be able to use.

The Globus Toolkit™ and Globus project shows how massively distribute systems can be implemented using services. The BOINC platform clearly illustrates the performance benefits that can be gained from sharing computations across a number of nodes. These 2 concepts are important input to the design of any computation sharing middleware The middleware should be able to distribute data to several nodes and be able to remotely process that data using services running on each node.

Computation Sharing using a mobile agent

A novel approach to GRID computing within MANETs using a mobile agent is described by Wang et Al.[14] (2006). A mobile agent is defined as a "*self-contained and identifiable computer programs that can move within the network and act on behalf of the user or another entity*" by Pham et Al[15]. In the approach described, a mobile agent carries task data and the logic (essentially code) to perform the data around the network looking for nodes with resources to be able to process the data. The mobile agent also stores the resource requirements for the job, in order to be able to determine whether or not a node is capable for executing the job. If a node is processing a job, but suddenly cannot does not have to resources to do so, the agent can save the job and move onto another node. The agent is also capable of distributing the workload across several jobs.

When an agent is migrating between nodes, it can query the resource status of nodes in the network, to work out optimal paths. For more details on how the agent migration occurs, see[14].

The mobile agent approach is very interesting, however it does have some potential drawbacks. The bundling of execution code with data within the agent does provide some security concerns. It would be necessary for the mobile agent to execute within a completely segregated virtual machine in order to prevent malicious code executing on a node.   The bundling of code does have the advantage that the agent has no need for a service discovery mechanism, to search for services at a node that are capable of process the job data. However this also means that execution code must be specifically written for the agent. The agent cannot take advantage of pre-existing services which may exist at a node. Also, highly complex execution logic may require a lot of code, which in turn may result in an increase in the footprint of the agent as it migrates between nodes.

The agent also needs to migrate around the network in order to find nodes which have suitable resources. This could be a significant issue if the jobs carried by the agent were quite data intensive. An agent migrating around the network carrying a large volume of data with it will incur a significant overhead, in term of the amount of time it takes the agent to migrate from one node to the next.

If it is necessary of the mobile agent to return to the originating node (which in the vast majority of cases it will be) there maybe an issue if the network has partitioned resulting in the agent being in an MANET isolated from the originating node. Obviously to originating node can dispatch another agent, but if the original agent had completed 90% of its task this seems like an awfully large overhead.

The agent approach does have many merits, however all evaluation has been done using simulators. In a real-world wireless environment transmission overheads, as well as interference and lost packets could result in quite a negative impact on the ability of the agent to navigate around the network.

Agent based computation-sharing in an MANET does have potential, but I believe that it is limited to processing without a large data overhead, processing without complex execution logic and ad-hoc networks where interconnections between nodes are strong enough to avoid data having to be retransmitted.

Peer to Peer Applications on MANETs:

Within any network a set nodes running peer-to-peer applications can be considered to make up their own logical network. This is called an overlay network and the topology of this network may be completely unrelated to the underlying topology of the physical network. For example, if a peer to peer application is running across the internet, two nodes which may be physically close to each other may have several nodes between them in the overlay network. This is the case for peer to peer networking approaches such as Chord[16].

This logical/physical incongruence is not suitable for running P2P applications on Mobile Ad-hoc Networks, as the topology of nodes within the MANET also reflects their availability and the cost of sending data to those nodes.

Mobile Peer-to-Peer Protocol

Schollmeier et al[17] describe a Mobile Peer-to-Peer Protocol (MPP) that is designed to allow the operation of peer to peer services on ad-hoc networks by over coming the topology issue described above. The MPP protocol reuses existing network protocols where possible and spans from the network layer to the application layer, giving a P2P application direct access to the network layer. Within the network layer MPP uses a Enhanced Dynamic Source Routing (EDSR) protocol, which is an extension to the existing DSR protocol, but with enhanced functionality which allows P2P application search for specific nodes and content. The protocol can allow nodes to be searched for based on a hashed description of services, or a hash value of a file when searching for content.

I believe the strongest point of the MPP protocol is the integration of peer-to-peer functionality with the Network layer functionality. However the MPP protocol is not entirely suitable for computation sharing, as services are only located on demand so when a computation needs to be shared, a list of candidate node needs to be built up for distributing the tasks across. The purpose of computation sharing is generally to decrease overall processing time, so if each request has an extra over head of several discovery messages being created once the initial request for computation sharing arrives. MPP is also designed for general peer-to-peer applications such as file-sharing and remote file storage, so if provides more functionality and a heavier footprint than is necessary for computation sharing. Finally the only implementation of the MPP protocol is using the System Definition Language (SDL). There is no implementation available as an API for a standard operating system. To build an implementation of MPP for a real world network would require operating system programming skills and a timeframe beyond the bounds of this dissertation.

Information Dissemination in mobile ad-hoc networks

As part of any computation sharing middleware that distributes computation tasks to services on remotes nodes based on node system context, there needs to be functionality to provide service information and system context information about each node.
One approach to do this is described by Nedos et al. [18]. As part of an approach to semantic service discovery in MANETs, a gossip protocol is used to exchange randomised subset of concepts, which describe services available, between nodes.
Part of the reason for using this protocol is that there is no centralised service registry available within MANETs due to there ad-hoc nature. Any such registry would need to have a consistent well know location, which is not possible within MANET architectures.

Whilst there is a lot more complexity to the semantic service discovery described by Nedos et al, the use of the gossip protocol is of benefit to any computation sharing middleware which needs to have node system context and service information distributed across a network. Semantic service matching is outside the scope of this project , however randomly distributing node system context and service name information using the gossip protocol is a useful mechanism, which will avoid flooding a MANET with context updates.

Data Formats within A-hoc Networks

XML[19] (Extensible Mark-up Language) is a well established Internationally agreed upon standard for formatting data . It is used by Nedos et al [18]. It is also used by other MANET oriented middleware implementations such as XMIDDLE (Mascolo et al, 2001)[20]. XMIDDLE is a middleware application for MANETS which allows the replication and reconciliation of data across nodes in a mobile ad-hoc network.

XML is frequently chosen for data formats because of its hierarchical data structure, the widespread availability of parsers and is human readable form. These reasons also make it valid choice as a message format for the computation sharing middleware.

Chapter 4 Design

This chapter describes the design of the middleware, test application and API. The design decisions made for each component are described as well as potential alternatives.

Middleware overview

At a very high level, the purpose of the middleware is to receive a processing job from an application, divide that job into smaller job parts which are distributed to remote nodes within the MANET. Each of the remote nodes is also running the middleware, and the middleware application on each node processes the data contained in the job part and returns the result make to the middle ware on the originating node.
Once all remote nodes have returned data to the middleware on the originating node, the completed job is returned to the application.

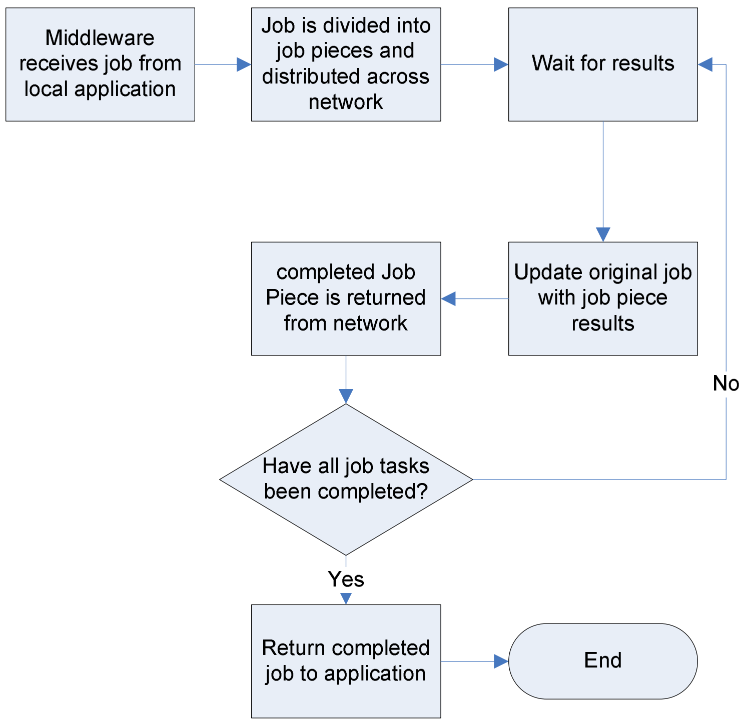

Middleware receives job from local application
Job is divided into job pieces and distributed across network
Wait for results
Update original job with job piece results
completed Job Piece is returned from network
Have all job tasks been completed?
No
Yes
Return completed job to application
End

Figure 2 Process diagram giving an overview of the middleware functionality

Figure 3 is a sequence diagram showing the sequence of messages in an example of middleware task distribution in an ad- hoc network consisting of four nodes. The local node is running both the application requesting the computation sharing and an instance of the middleware. Each of the remote nodes is also running the middleware. For simplicity it is assumed that all nodes are capable of processing the job data. When the middleware application running on the local nodes receives a request it divides the job into smaller job parts and distributes 3 of these to the other 3 nodes in the network. The fourth job part is processed on the local node itself

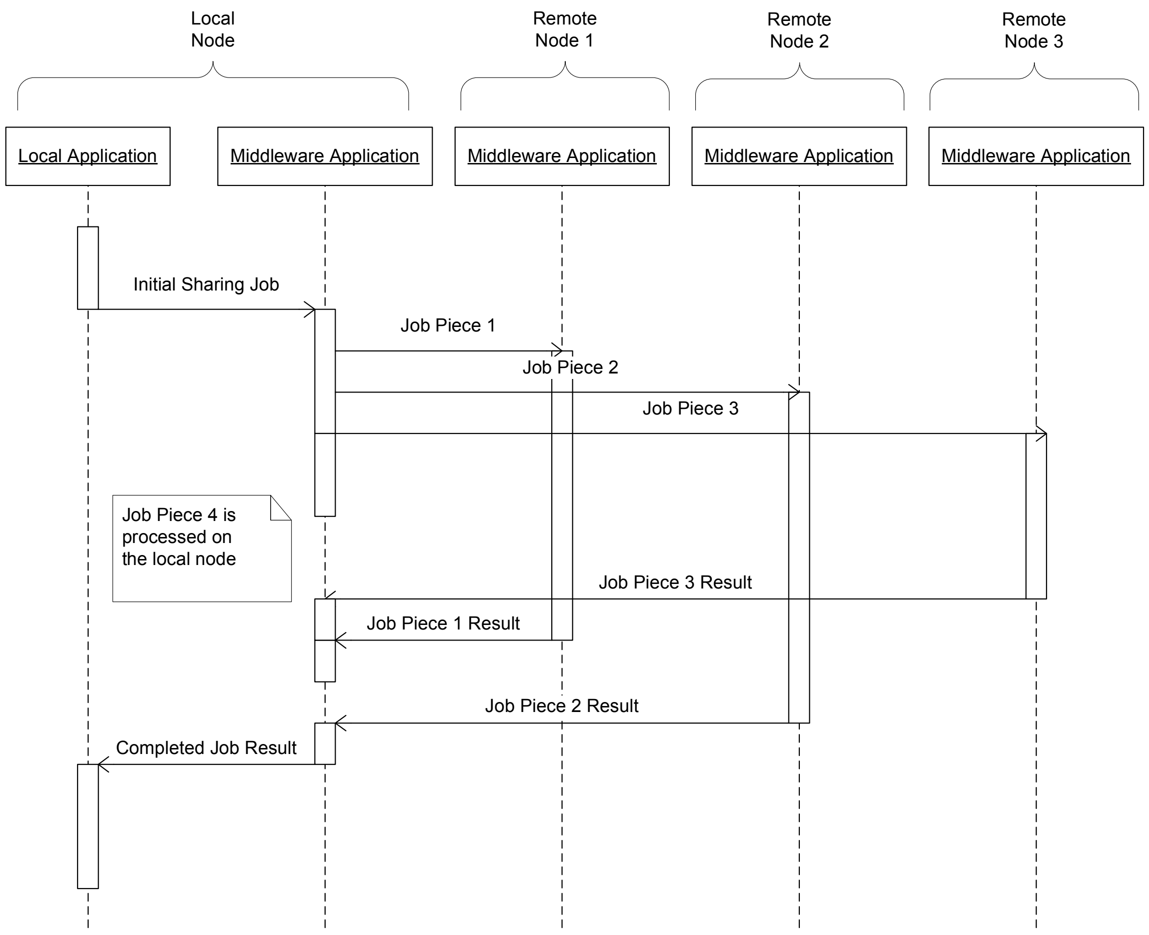


Figure 3 An example of job distribution in a 4 node MANET

Architecture

Programming language

The middleware, API and test application have all been implemented using C++, an object oriented language which is compiled natively on a target system. The reason for choosing C++ is that, due to the fact that code is compiled natively for a specific device, performance is significantly better than for a language that runs upon a virtual machine e.g. Java.

As a requirement for this project it is necessary extra system context information from the OS, there are cross-platform libraries available to perform this function. A language such as java was not deemed suitable as it required that each node run a virtual machine, which would consume system resources and that system context information would not be readily available. A negative aspect of C++ is that it is more complex than most other object oriented languages. It features fewer programmer friendly features such as array bounds checking and garbage collection, which can be found in many more recent languages such as java. Despite the increased technical challenge in implementation, it has been chosen for its performance and the availability of the necessary libraries.

Node Architecture

For the purposes of the project implementation each node in the network is running the Linux operating system, and hence the middleware application is built specifically for Linux. Linux is a freely available operating system which is widely supported and frequently used in networks, particularly in research. There are many different distributions of Linux. The distribution used for this project was Ubuntu v7.04.
Each node contains an 802.11 wireless card capable of communicating with other nodes in the network in an Ad-Hoc manner. The main alternative is the windows operating system, however Linux was deemed preferential due the amount of free open-source libraries available, as well as the fact that it is freely available without and charge. The libraries used for this project were

- The GNU Common C++ library[21] was used to provided multi-threading functionality, as well as socket functionality (basic TCP and UDP communications).

- The LibXML++ library[22] was used to provide XML message parsing and generation functionality

- The gLibTop library[23] was used to retrieve processor context information direct from the operating system kernel

Network Architecture –Peer to Peer

The high level architecture uses a peer to peer model. Due to the decentralised nature of MANETs any type of clients-server model would be impractical. Nodes can join or leave a MANET at any time. Use of a client server model would require on of the nodes in the network to be nominated (or elected by the other nodes) as server. If a set of nodes within an ad-hoc network are dependent on one node acting as server node, the loss of the server node would result in a significant overhead. A new server would need to be elected which would require a considerable amount of messages to be generated within the network. This also results in the network not being usable while an election takes place.

A peer-to-peer model also allows for nodes to communicate with each other in a more direct manner, without the necessity for messages to be routed through a central server. The negative side of using a peer to peer model is that all nodes need to have a view of the whole network, not just the main server. This leads to an increase in the volume and size of messages used to disseminate information about network size. However given the amount dynamic nature of MANETs, and the extra messages for a central server re-elect and node discovery a peer-to-peer model is the most suitable for the middleware.

Building the Network Model

In order to be able to decide what nodes to share computations with, the middleware application needs to know what nodes actual exist in the network. Also in order to be able to decide how to distribute tasks based on context information, the middleware needs to be aware of the system context at each node as well as the services available at each node. There are two possibilities for this.

The first option is to build up a network model in advance of receiving any sharing requests. This means nodes have to share their context and service information with each other on a regular basis. This means little time has to be wasted between a node receiving a computation sharing request and that request being distributed across the network. Unfortunately this also results in many extra update messages being sent around the network from each node.

The second option is to build up a network view only once a request is received. However this would mean a considerably delay when a node receives a request for computation sharing. A node will need to query several other nodes in the network to determine which nodes are candidates for sharing. This will take time especially if discovery messages have to travel to nodes several hops away from the originating node. Given that the purpose of computation sharing is to reduce overall processing time for a job, this overhead will impact any performance gain form the sharing of the computations.

In order to maximise the time benefit gained from sharing computations across the ad-hoc network, the middleware design is adapting the first approach, where a complete network model is built in advance of any computation sharing requests being received by a node

Messages between Nodes

Message Format

XML is being used as the data format for all messages. XML [19] (Extensible Mark-up Language) is a well established internationally agreed upon standard. It is text based and is widely used for integration between applications. There are also many C++ development libraries available to generate messages in an XML format. XML is a human readable text based format, however is possible to transmit non-text based data using XML. This can be done by converting the data to a text form guaranteed to avoid any XML control characters. This conversion process does increase the size of the message however, to send binary data would require 2 messages to be sent, one for the text and control information and one for the binary data. Encoding the binary data within the XML message saves on extra messages and simplifies the messaging design. Examples of the XML messages used to disseminate context updates can be seem in Appendix B.

Message Transmission Protocol

There are two possible protocols for the transmission of data between nodes. The first is the Unreliable Datagram Protocol (UDP).

Unreliable Datagram Protocol

UDP[24] is a very simple protocol which allows data to be sent without a connection. Each UDP packet consist on and 8-bit header, followed by a data payload. The UDP protocol does not have any flow-control, error control or retransmission if data is corrupted. As a result of the lack of flow control and the lack of a continuous connection a sender of a UDP packet cannot be sure that the target actually received the packer, unless it is specifically acknowledged. Another drawback of UDP is that it is packet oriented. The maximum packet size for a UDP message is generally 64 kilobytes. This means that is a message larger than 64 kilobytes is to be sent it must be divided up into smaller packets and reassembled at the destination into the original full message.

Transmission Control Protocol

The second protocol is the Transmission Control Protocol (TCP). TCP[25] is designed specifically to provide and reliable end to end data stream over an unreliable internet work. TCP provides functionality such as flow control and error control. As a result of TCP's end-to-end connectivity, once a connection is established, a message can be assumed to have reached the target successfully and without corruption. As TCP is steam based and manages flow control there is no restriction on how big a message (within reason) can be sent using TCP. The protocol will be responsible for breaking the data into individual packets and reassembling them at the destination. However, when it comes to transmitting data, the enhanced functionality of TCP means that it has a higher overhead than UDP. More work has to be done a the Operating System level to transmit a TCP packet than a UDP one

Choice of message protocol for Implementation

Initially for the implementation of the middleware TCP was chosen as the message transmission protocol for messages between nodes which required a response. While sending a TCP messages has a higher overhead than a UDP messages the end to end connectivity of TCP was a very attractive feature in the design of the middleware. However experimentation revealed that TCP does not perform well running on a wireless ad-hoc network using AODV for connectivity. TCP connections could only be made to nodes which were direct neighbours in the ad-hoc network. As a result of this the middleware was designed to be able to run using either TCP or UDP. When the middleware application is started, a command line parameter specifies which protocol is to be used. The addition of UDP also meant that extra functionality had to be added to the middleware. In effect aspects of the TCP protocol have to be rebuilt at the application layer. Messages received via UDP must be acknowledged and messages which were are larger than 64 kilobytes would be broken up into pieces which can be reassembled at the destination node.

Sharing Jobs

When an application wants to share computations, it submits a job consisting of several tasks to the middleware. As part of the job submissions the target application, a job ID, a timeout value and parameters for the target application must be specified. It is also possible to specify context weights as part of a job. This indicates how much preference a one piece of context information is given over another when the middleware evaluates how jobs are to be distributed. If no context weight information is specified the job is divided evenly amongst the nodes. The individual tasks belonging to a job are atomic. A task cannot be divided across nodes. A task has can have its own application parameter data, but must use the service specified by the parent job. Each task also stores the data that is to be processed.

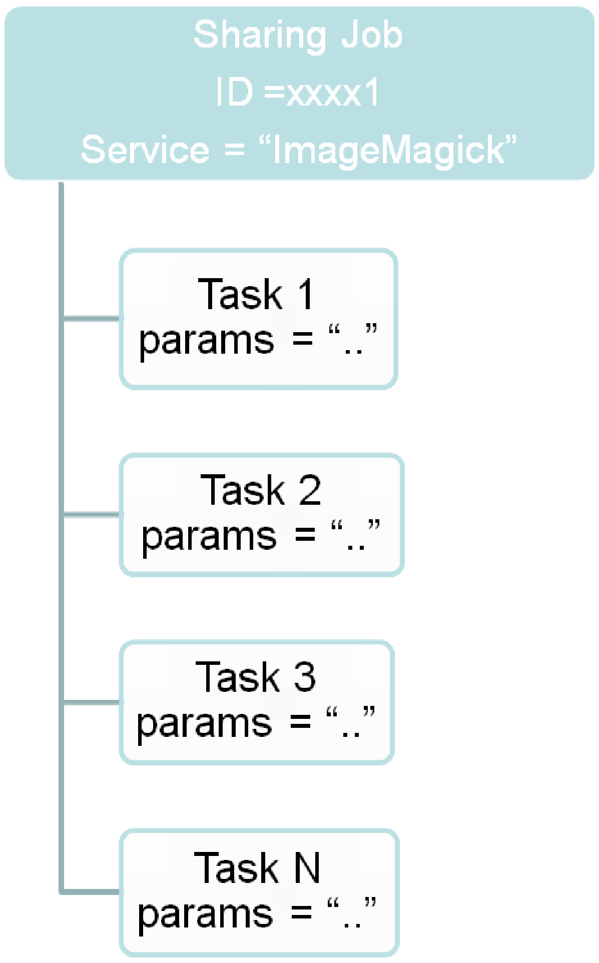


Figure 4 Sharing job Hierarchy

When a job is divided across nodes, it is the tasks of the job that get sent to different nodes, as part of new smaller job messages. Each of these new smaller job messages is nearly identical to the original job message, except they have fewer tasks and a unique "piece ID". Tasks are considered to be the lowest level of granularity when it comes to dividing process across nodes.

For example if a job with 5 tasks is to be shared evenly across 3 nodes, 2 of the nodes will receive 2 tasks while the final node will receive 1 task.

Services

There is no functionality within the middleware to process task data directly. Other applications are used to process the data. These applications are considered services as they offer functionality as a service to other nodes in the network. Normally services are remotely invoked using a messaging protocol such as SOAP over an agreed port. Due to time required to build such services and the short timescale of this project, a very simple service model has been used. All services supported by the middleware are actually command line applications which can be invoked with parameters. When a task is received to be processed on a node, the task data is saved to disk and a command line application is invoked to processes that data. The resulting output data of the command line application is read from the application's output file and stored as the task result data.

Middleware Components:

The middleware is made of up 5 main components

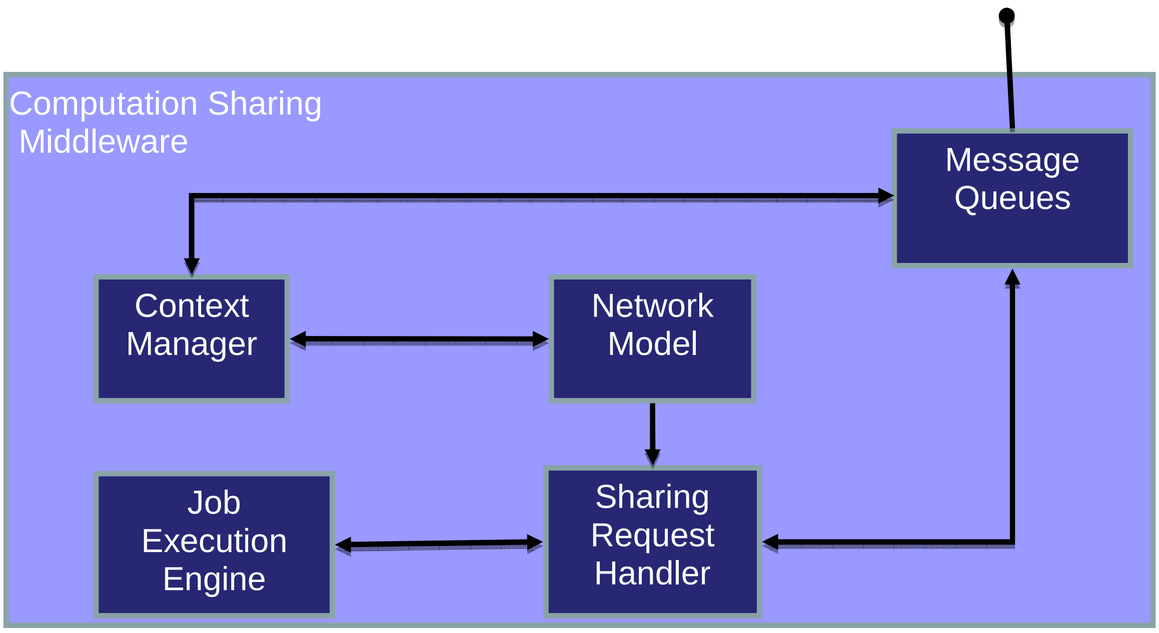


Figure 5 Middleware component overview

Context Manager

The context manager component looks after both the system context as well as the list of services which are available on the nodes. The context manager periodically retrieves and dispatches context information (both system context and service information) to other nodes in the network. A simple form of the gossip protocol is used to do this. When the context manager tries to send a system context update message, a target node is chosen at random and the context manager sends that node its own context and the context of every node in its network model. The same applies to service directory update messages.

System context Retriever

The context Manager retrieves the local system context. This includes processor and battery information. Processor context information is read using a system library. The battery information is retrieved by parsing the result of a command line call which outputs the battery status. The data retrieved is formatted into XML messages ready to be dispatched

Service Directory

The service directory stores a list of services available on the local node. This list is read in from a text file. It is up to the node administrator to specify what services are available on the node. A shared service directory class reads this file, and parses the data into an XML message ready to be dispatched around the network.

Context Update Messages

Periodically a nodes context manager will sent an update to other nodes in the network. These updates are made using a TCP or UDP, depending on the parameter the application was started with. It is expected that a target node is active and if it is not the node must be removed from he nodes network list. In order to verify if a node is active when using UDP the node must send a UDP response.

Computation Sharing Messages

Computation Sharing messages are also sent via TCP or UDP, depending on the application configuration. These messages can be very large in size as they contain computation sharing task data. It is also vital the messages are received successfully. When running in UDP mode computation sharing messages are broken into 64 Kilobyte packets. Each packet is acknowledged by the receiver before the next one is sent, as UDP does not guarantee delivery. When messages are sent using TCP there is no need to perform any such acknowledgement or divide the message into packets.

It is also necessary to be sure that the task data is not corrupted when received by the target node, so an MD5 checksum is included for each task. The MD5[26] digest algorithm is used produce a message digest value for the task data when it is being sent. When the receiving node receives the task data it re-computes the MD5 digest value. I this value does not match the one included in the XML for the task, the receiver knows that the data has been corrupted.

Network Model

Each Node running the computation sharing middleware stores a list of other Nodes in the MANET which are also running the computation sharing middleware. Against each entry in the node list is stored the IP address of a node, the name of the node as well as the system context information and a list of available services at that node. Periodically updates to the system context and service information for nodes within the network model are received via the context manager component.

Network Discovery

When the computation sharing service is started, one of the first pieces of code to execute is concerned with network discovery. A network discovery class reads data from the AODV routing table. This data includes a list of all the IP addresses of neighbour nodes in the network, as well as possibly the IP Addresses of nodes more than one hop away who have recently been communicated with. The AODV routing table data is output by the AODV module approximately every 30 seconds. (This frequency of the output is a configurable parameter, set when starting the AODV module). Due to the fact that new nodes may join the network at any time (or may start running the middleware application at any time) the network discovery mechanism periodically checks the AODV routing table data for nodes which are not part of the network model already and sends them a discovery message.

Discovery Messages

All discovery messages are UDP based whether the application is running in TCP mode or UDP mode. This is as a response in not necessarily expected and the messages are small in size. The reason a response is not necessarily expected is that it is not assumed that all nodes within the network will be running the middleware application. Each node also has a UDP server running, listening for discovery messages from other nodes. If a discovery message is received, the node which sent the message is added to the local nodes data model, provided it does not exist in the model already.

Job Execution Engine

This Component takes a job piece from the Sharing Request Handle and executes it on the local system. The middleware does itself does not contain any code or functionality to process the job itself.

The application name and parameters are read from the job and the job is executed using the command line. Once the job has been executed the original task data is replaced with result data, and each task in the job is marked as complete. The completed job piece is then sent back to the node it originated from.

Sharing Request Handler

This component handles computation job sharing messages. This includes sharing request and sharing result messages. A Sharing request can be an initial sharing request received from a local application, or sharing request from a remote node (which has received an initial sharing request from a local application). If a sharing request job is received from a remote node it is passed to the job execution engine.

If a sharing request message is an initial sharing request from the local node, it is divided into smaller job pieces which are sent out to other nodes for processing. The job's tasks can either be divided evenly between all nodes capable of executing the job, or it can be divided amongst the nodes using context information, resulting in some nodes getting more jobs than others. The local node can also receive some of the jobs tasks. If this is the case the job piece is passed directly to the job execution engine once all the other job pieces have been dispatched to remote nodes.

Once the job task data has been processed by the remote nodes, the resulting task data is returned to the originating node for the job.

This component also handles sharing results from other nodes, for requests which originated from the current node. When the a job piece containing the results of a shared computation is received, the results are stored by the sharing request handle until all job pieces have been returned and the original job, now containing result data, can be returned to the local application.

Job Timeouts

When a job is divided into pieces for remote processing, it is given a unique piece ID and a timeout timestamp. If the job piece has not been process and returned by the time this timestamp has been reached the job piece is considered to have timed out. When a job piece times out it is redistributed across the network for processing. To redistribute a job the sharing request handle simply shares it across the network like it would a job from a local node. When a job piece is redistributed it is broken again into smaller job pieces (if possible) and redistributed across

Figure 6 illustrates the messages sequence between nodes when a job times out. In the diagram, remote node 2 leaves the network while it is processing job piece 2. When job piece 2 times out at the local node it is redistributed as job pieces 5, 6 and 7. Job piece 7 is processed on the local node, while job pieces 5 and 6 are sent to the remaining remote nodes.

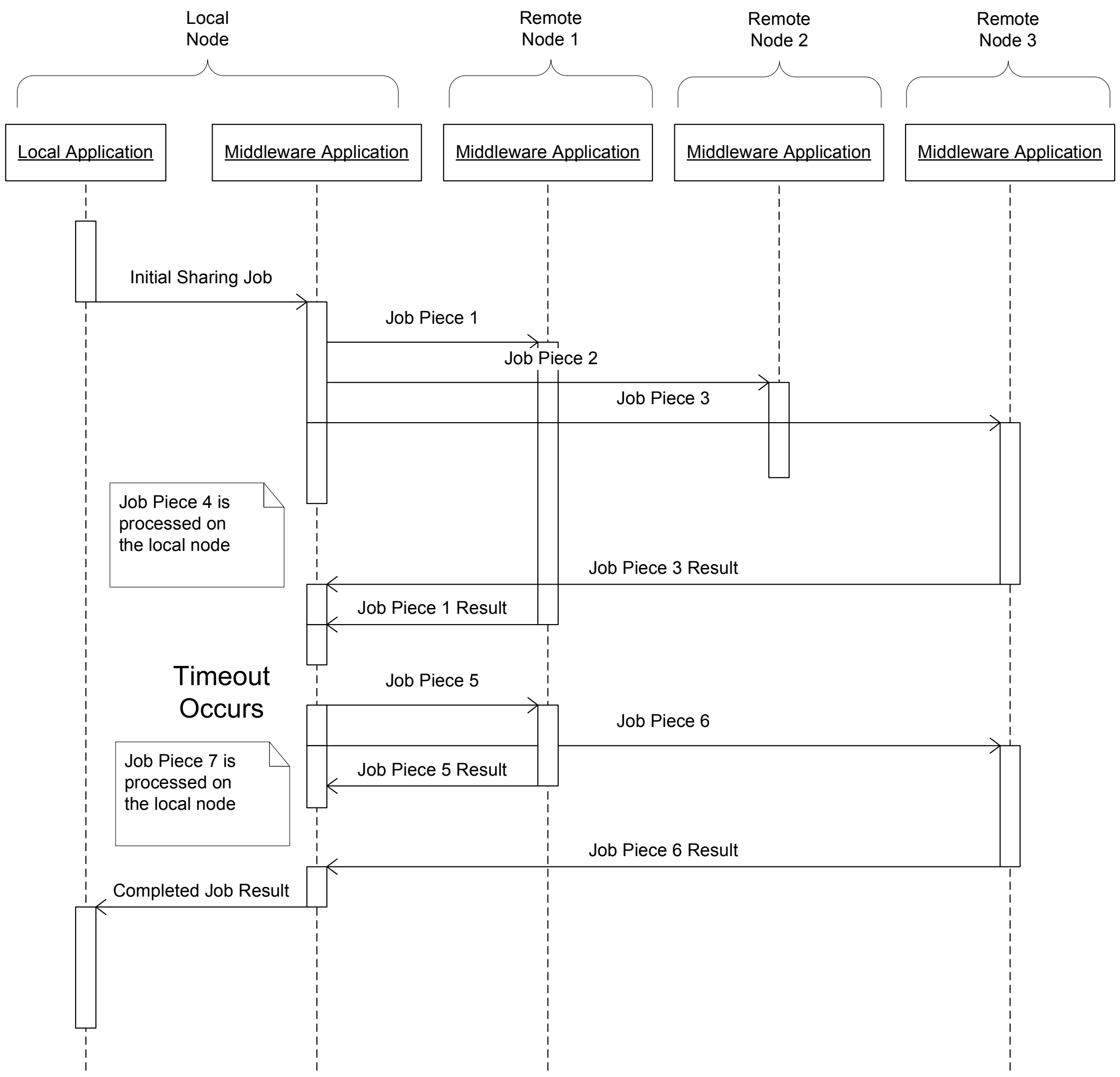


Figure 6 Sequence diagram showing message flow in case of a timeout

Message Queues

The messages queues are the point of entry and exit for all messages in the middle ware except discovery messages. Discovery messages are the responsibility of the network model. There are 2 messages queues within the application the inbound message queue (the message receiver) and the outbound message queue (the message dispatcher).

The Message Receiver

When the middleware application is started a server thread is launched which listens on a predefined port for inbound messages. This server thread listens for either UDP or TCP messages depending on which parameter value the middleware application was launched with. Once an inbound message has been received in its entirety, it is passed to the message receiver. The messages receiver contains a queue of messages which have been received. A thread processes the queue taking messages from the front of this queue and passing the message to the relevant component. The context manager receives all system context and service directory updates. The sharing request handle receives any messages concerning computation sharing. Once a message has been dispatched to the correct component it is removed from the queue.

The message Dispatcher

Whenever the Sharing Request handler or the Context Manager needs to send an outbound message, the message is added to the queue of messages in the message dispatcher. A message dispatching thread reads messages from the message queue and sends them to the target node (specified in the message's XML header) using either UDP or TCP messaging, depending on which parameter value the middleware application was launched with. Once a message has been sent successfully or has failed transmission 20 times it is removed from the outbound queue.

Local Application API Design:

In order to allow applications running locally on a node to created jobs and pass them to the middleware for computation sharing a small API has been designed. The API allows a local application to create a computation sharing object. All interaction between the local application and the middleware is done using one class called "computationSharer".
The following functionality is required in the computation sharer class to allow a local application to use the middleware.

- The ability to create job for sharing in a format that can be understood by the middleware

- The ability to create tasks for the job, as well as add task parameters and data.
- The ability do deliver the job as a computation sharing request to the middleware
- The ability to wait for and store the result of the computation sharing job.

Chapter 5 Implementation

This chapter will describe some aspects of the implementation including

- The algorithm for the distribution of tasks evenly
- The algorithm for the distribution of tasks using context information
- The gossip algorithm used for dissemination of context and service information

Task Distribution Functionality

When the sharing request handler component receives data sent from a local application a check is done to see whether any context weight information is included. Depending on whether or not context weight information has been specified, tasks are distributed evenly across nodes or distributed using the weighted context information. The original job received from the local application is divided into job pieces when it distributed across the remote nodes. When a job piece is returned it is used to update the original job with the result data. Once all tasks of the original job have been processed it is returned to the local calling application.

Distribution of tasks evenly across nodes

When a job is to be distributed evenly across node the sharing request handler first reads what service is required by the job and a call is made to the network node model to see which nodes support the service. When the network model finds a node supporting the service specified it send the node a short “ping” message to verify the node is still contactable. This “ping” consists of sending a short TCP or UDP message to the target node. If no response is received, it is assumed that the target node has left the network. The target node is not returned to the sharing request handler as a node suitable for sharing, and is removed from the network node model altogether.

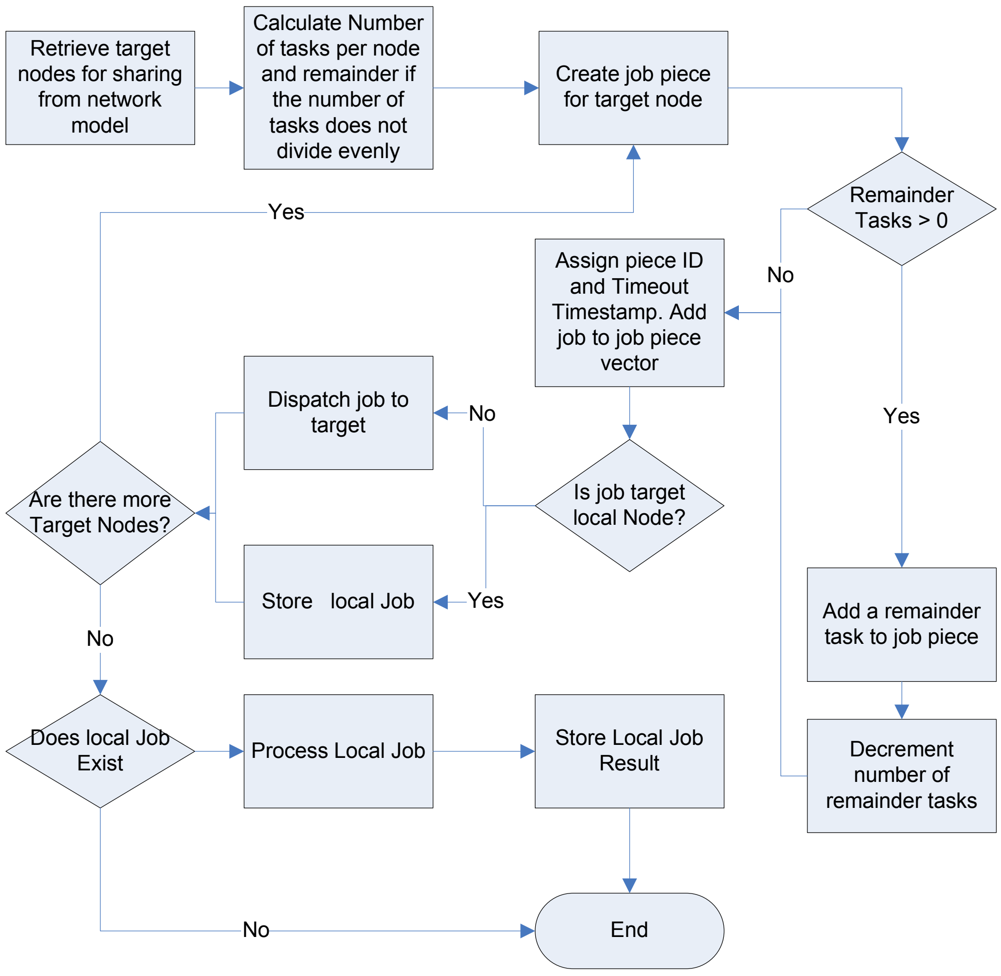

Retrieve target nodes for sharing from network model
Calculate Number of tasks per node and remainder if the number of tasks does not divide evenly
Create job piece for target node
Remainder Tasks > 0
No
Yes
Assign piece ID and Timeout Timestamp. Add job to job piece vector
Is job target local Node?
No
Yes
Dispatch job to target
Store local Job
Are there more Target Nodes?
Yes
No
Add a remainder task to job piece
Decrement number of remainder tasks
Does local Job Exist
Process Local Job
Store Local Job Result
No
End

Figure 7 Process flow for the distribution of tasks evenly across nodes

Once the list of target nodes is returned to the sharing request handler, the number of tasks to be distributed is integer divided by the number of nodes. The remainder of this division is also stored. A new job is created for each node and each node’s job is assigned the number of tasks resulting from the integer division. If there are any remainder tasks, one of them is also assigned to the node job and the total number of remainder tasks is decremented.

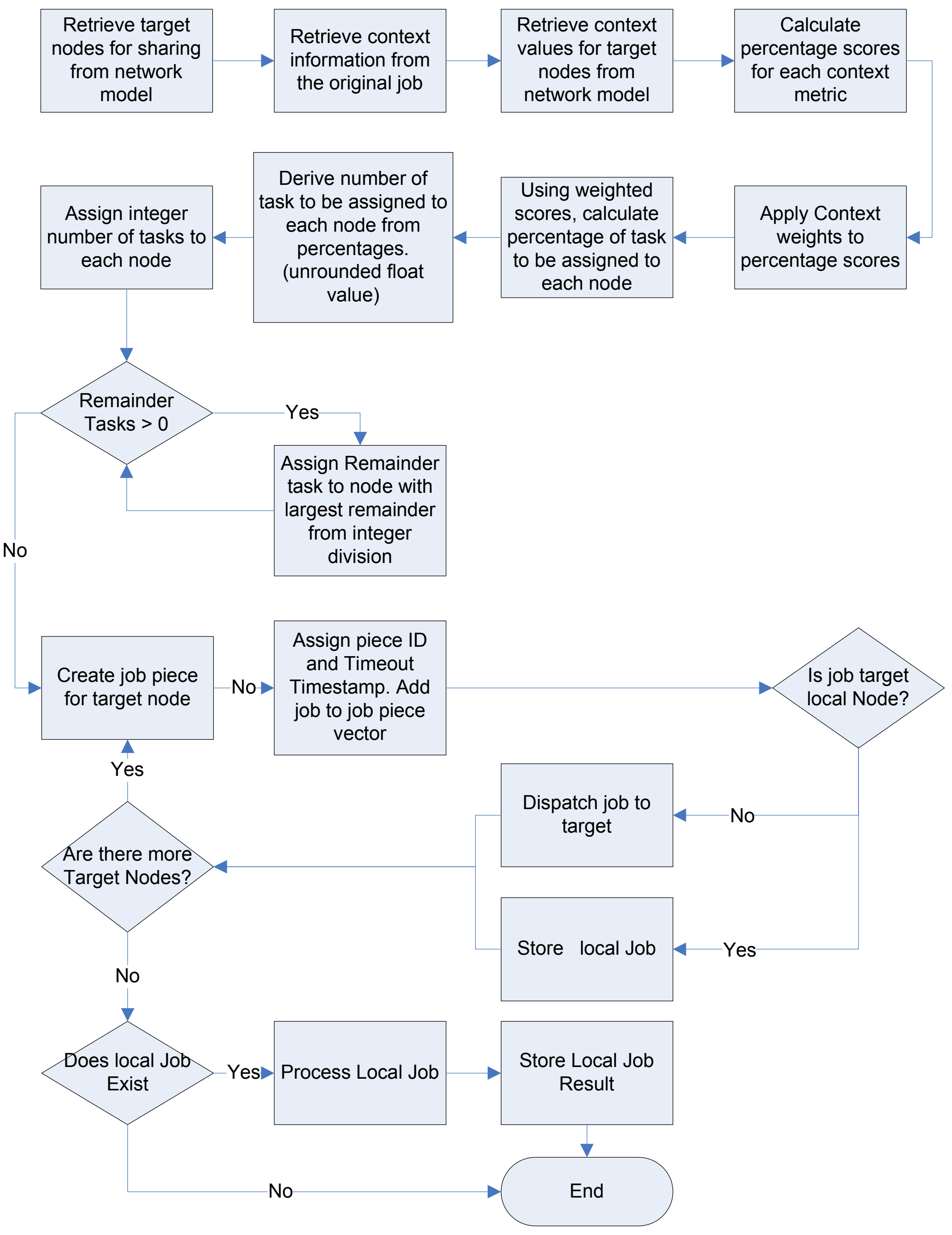
Each new job created for a node is given a unique piece ID to differentiate them from each other, and the original job received by the middleware. Each of the new jobs created to send the tasks to the sharing nodes can be considered to be a job piece, as they are all pieces of the job originally submitted to the middleware.

Each of the job pieces is given a timestamp, which is equal to the current time plus a timeout value for that job piece. If the job piece has not been processed and returned by the time this timestamp has been reached, it is considered to have timed out and must be redistributed for sharing across the network. Finally a copy of the job piece is saved in a vector, in order to track which job pieces have been completed or have timed out.

Once the job piece has been added to the job piece vector it is ready to be dispatched. The job piece is dispatched to its target node, using the outbound message queue, unless its target node is the local node. If the job pieces target node is the local node there is no point in sending it via UDP or TCP, as it can be executed within the sharing request handler. A job piece that is for the local node is only processed once all of the other jobs have been dispatched. Once the local job piece has been processed it is used to update the original job vector and its tasks are marked as complete.

Distribution of tasks across nodes using context

When a job is to be distributed using context information the first step for the sharing request handler is the same as for when no context is specified. The target list of nodes for sharing is retrieved from the network node model in the same manner as is done when jobs are to be distributed evenly. Once the list of target nodes has been retrieved, the context weight information is obtained from the job. This information tells the sharing request handler which pieces of context information is to be used. The values of each piece of context are retrieved from the network model.



Retrieve target nodes for sharing from network model
Retrieve context information from the original job
Retrieve context values for target nodes from network model
Calculate percentage scores for each context metric
Apply Context weights to percentage scores
Using weighted scores, calculate percentage of task to be assigned to each node
Derive number of task to be assigned to each node from percentages. (unrounded float value)
Assign integer number of tasks to each node
Remainder Tasks > 0
Yes
Assign Remainder task to node with largest remainder from integer division
No
Create job piece for target node
No
Assign piece ID and Timeout Timestamp. Add job to job piece vector
Is job target local Node?
Dispatch job to target
No
Store local Job
Yes
Yes
Are there more Target Nodes?
No
Does local Job Exist
Yes
Process Local Job
Store Local Job Result
No
End

Figure 8 Process flow for the distribution of tasks across nodes using context information

Next the sharing request handler needs calculate the relative strength of each value returned for each context metric. A percentage is assigned to each metric value for each node, where 100% indicates the best value across all target nodes. For system context metrics such as processor speed the maximum value will be considered the best. For example if there are two nodes, one with a processor speed of 2 GHz and one with a processor speed of 500 MHz, for the metric processor speed they will be assigned values of 100% and 25% respectively. For other metrics such as processor load averages a lower score is better. For example if there are 2 nodes, with a 1 minute load average of 1.5 and 3, they will be assigned scores of 50% and 100% respectively.

At this stage each node now has a percentage score for each system context metric. The next step is to apply the context weights. Each node's system context value score is multiplied by the context weight value specified for that node and the sum of these weighted scores is assigned to each node. The sum of all of the weighted scores is then calculated and each node is assigned a percentage based on the nodes overall weighted score relative to the sum of all weighted scores. The result is a percentage break down of what percentage of tasks should be assigned to each node. The percentage values are double values which may have many decimal places and the number of tasks is unlikely to divide evenly between these percentages

From this percentage values the number of tasks to be assigned to each node is calculated. This number of tasks to be assigned to each node will also be a double value. As tasks are discrete and nodes cannot be assigned a fraction of a task, each node is assigned an integer number of tasks using a "floor" operation. E.g. a node which has initially been assigned 5.8 tasks will be assigned 5 tasks. If after this integer assignment there any remainder tasks they will be assigned to nodes based on which nodes have the largest remainder from the integer division.

Once each node has an integer value indicating how many tasks should be assigned to that node, a new job piece is created for each node containing the specified number of tasks. As with when jobs are being distributed evenly across nodes each job piece is given a Piece Id and a timeout value. Also as before, if the target node for a computation is the local node the local node's job piece is not passed to the message dispatch queue. It is executed locally once all of the other jobs have been sent.

Processing of completed tasks

When a completed job piece is received into the inbound message queue it is passed to the sharing request handler. The sharing request handle finds the original job and updates its task with the result data and marks the tasks as complete. The copy of the completed job piece is also removed from the job piece vector, to avoid a timeout occurring erroneously.

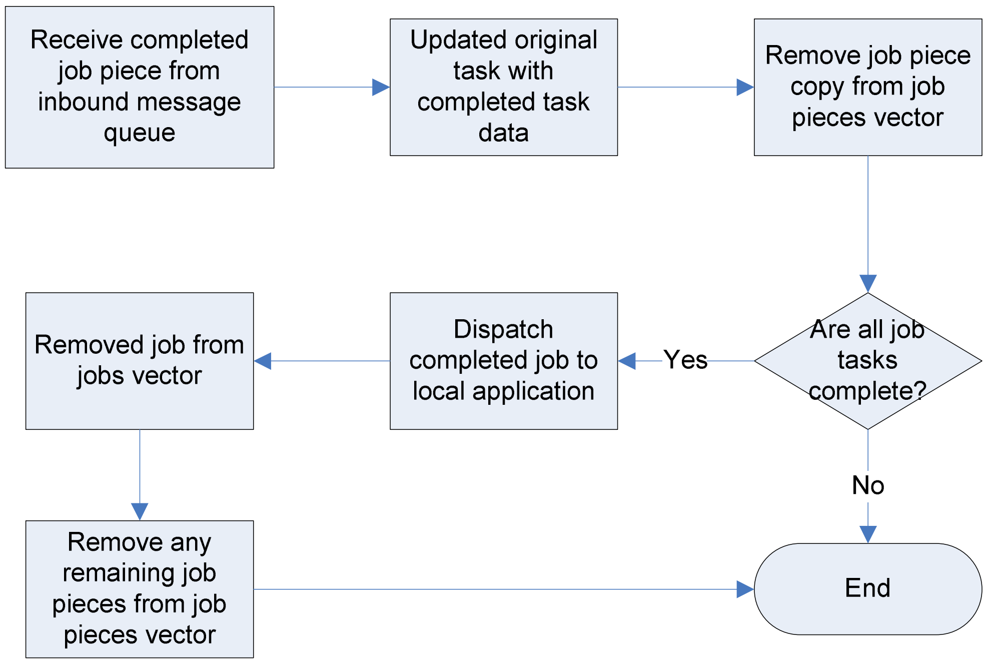

Receive completed job piece from inbound message queue
Updated original task with completed task data
Remove job piece copy from job pieces vector
Are all job tasks complete?
Yes
No
Dispatch completed job to local application
Removed job from jobs vector
Remove any remaining job pieces from job pieces vector
End

Figure 9 Process flow for the handling of completed job pieces

If all tasks of the original job have been marked complete at this stage it is dispatched back to the local application and deleted from the jobs vector. It is possible that even at this stage, when the job is complete, there may be job pieces still in the job piece array. This may occur if job pieces timed out and were redistributed, and subsequently the completed timed out job piece was received. To prevent this any job pieces with a job ID matching the completed job are deleted form the job pieces array.

Job Timeout Functionality

The purpose of storing a copy of job pieces which have been distributed is to allow the sharing request handler to be aware of when a job has timed out. As stated previously whenever a job piece is dispatched it is stamped with a timeout timestamp. This timeout timestamp is derived from the original job submitted from the local application. When a job is created by the local application it is assigned a timeout value. This value is the time, in seconds, in which a single task can be consider to have timed out if no result is returned. The timeout timestamp for a job piece is calculated as the current time plus the timeout value times the number of tasks within the job.

In order to monitor if job pieces have timed out a thread is run within the sharing request handler. This thread iterates through the job pieces vector comparing the timeout timestamp of each job piece to the current time. If the current time is greater than the timestamp value the job piece is redistributed. The job piece is distributed around the networking the same manner as the original job received from the local application, this means that the single job piece which has timed out is shared across several nodes. If the original job contained context weight information the timed out job piece will be redistributed using that information. Once a job piece has been redistributed it is deleted from the job pieces array to prevent another timeout occurring. The job pieces array is made thread safe by using a mutex as there is a risk of concurrency issues due to it being accessed by multiple threads.

Dissemination of System Context and Service information using a Gossip Protocol

Context information is disseminated using a Gossip Protocol. Rather than flood the network with updates periodically each node only send information to one other node. A very simple version of the protocol has been implemented. Within the context manager component a thread runs. Periodically (every 20 seconds) a context update and a service update are sending to a node selected at random from the network node list. When a system or service update is to be sent the context manager component refreshes the local data. In the case of the service directory, the data is read from the file system. In the case of system context, the operating system is queried for the current system context.

The service directory information, as well as the battery and processor context information, is parsed into XML data structures, ready to be included into context update messages. When a system context update (or service directory) update message is sent, it is not just the information from the local node that is distributed. The context manager queries the network node model for system context and service information retrieved from other nodes. This information is included with the local node information when any update messages are sent. Each system context or service directory update contains a timestamp which is also stored against the node in the network model. This allows a node to determine if the values for a remote node’s context in its network model are more recent than those being received in an inbound message. If this is the case, the update for can be discarded for that particular node within the network model.

API implementation

The API has been created to allow local applications to interface with the middleware using just one simple object. It consists of 2 components, a computation sharer component and a response server component.

The computation share component is used to create the job, add task data and specify job and task parameters, such as timeout, context weights and service parameters. The computations sharer also dispatches the job to the middleware application. When the computation sharer instance is created, it in turn creates a response server which listens for a response from the middleware.

When the computation sharer has sent the data, the local application can wait on a semaphore within the computation server object. When the response server receives the result data from the middleware application, it posts the semaphore allowing the original application to continue. Figure 10 shows the process that a local application ahs to follow to use the API. Appendix C contains some sample code illustrating how a local application can use the API.

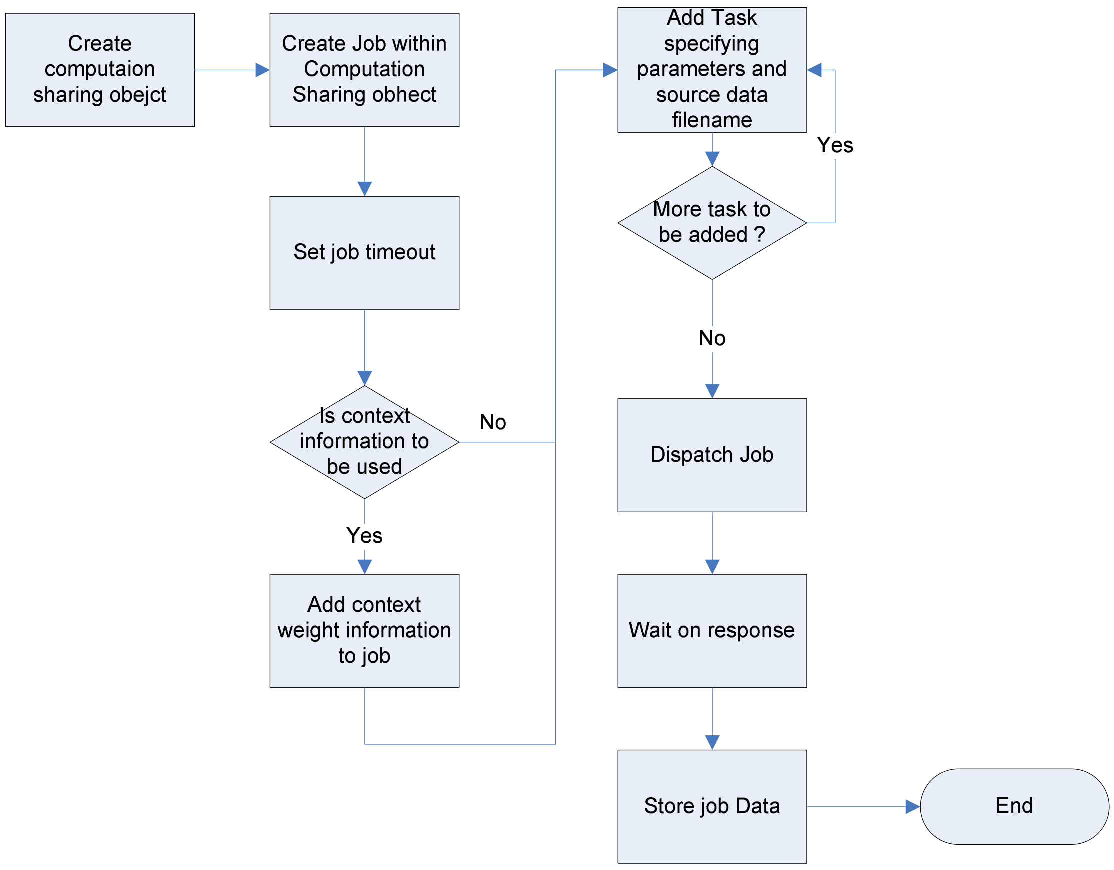

Create computaion sharing obejct
Create Job within Computation Sharing obhect
Set job timeout
Is context information to be used
No
Yes
Add context weight information to job
Add Task specifying parameters and source data filename
More task to be added ?
Yes
No
Dispatch Job
Wait on response
Store job Data
End

Figure 10 Process followed by local applications to send a job to the middleware using the API

Chapter 6 Evaluation

The middleware is evaluated in 3 ways. The first is in the benefit in throughput of distributing a job across the network using context information. The second is a comparison of using TCP for message delivery versus UDP. The third is the success of the gossip protocol in disseminating information around the network.

Evaluation Setup

A test application was developed which distributes jobs around the network. These jobs consisted of 11 image processing tasks. The image processing tasks to be executed are identical, meaning both the source image and processing parameters for each task are the same. The reasoning behind this is to avoid a case where 1 node in the network completes its assigned tasks more quickly, not due to its system contexts values such as processing speed, but due to the fact that smaller, less computationally intensive tasks were received. Each source image is 572 kilobytes in size and so represents a significant size of data to distribute around the network.

For a complete job to be processed on the originating test node with no other applications running, it takes 3 minutes and 3 seconds, equivalent to 16.4 seconds per task.

The evaluation program is run using a network consisting of 5 nodes, each running the computation sharing software. Every node, including the originating node from which the evaluation task comes, runs the image processing service. This means that each node is capable of executing job tasks.

Each run of the test application distributes 10 jobs around the network. 5 without context information, so the jobs are spread evenly across the network, and 5 using context information, so that the jobs were distributed to each node based on weighted context information. The program has 5 iterations. Each iteration consists of 1 job being sent with context weight information, and 1 job being sent without. After iteration is complete the application sleeps for a short while before moving onto the next iteration. Prior to starting the computation middleware AODV is stared on each node using the shell script attached in appendix A (with a unique IP address for each node).

Each of the five nodes in the network runs Ubuntu Linux and is equipped with an 802.11 wireless card. Each node is also running AODV-UU, an AODV implementation developed at Uppsala University, Sweden.
The node technical specifications are as follows:

- 1 x 3.1 GHz Desktop 1GB Ram
- 3 x 1.3 MHz Laptop
- 1 x 500 MHz Laptop

The nodes used are varying in speed and memory to accurately reflect the heterogeneous nature of ad hoc networks in real world situations.

Experiment 1: TCP Messaging in an AD-Hoc net work where all nodes are neighbours.

For the first experiment all nodes in the network were neighbours, meaning that each node could contact any other node directly.

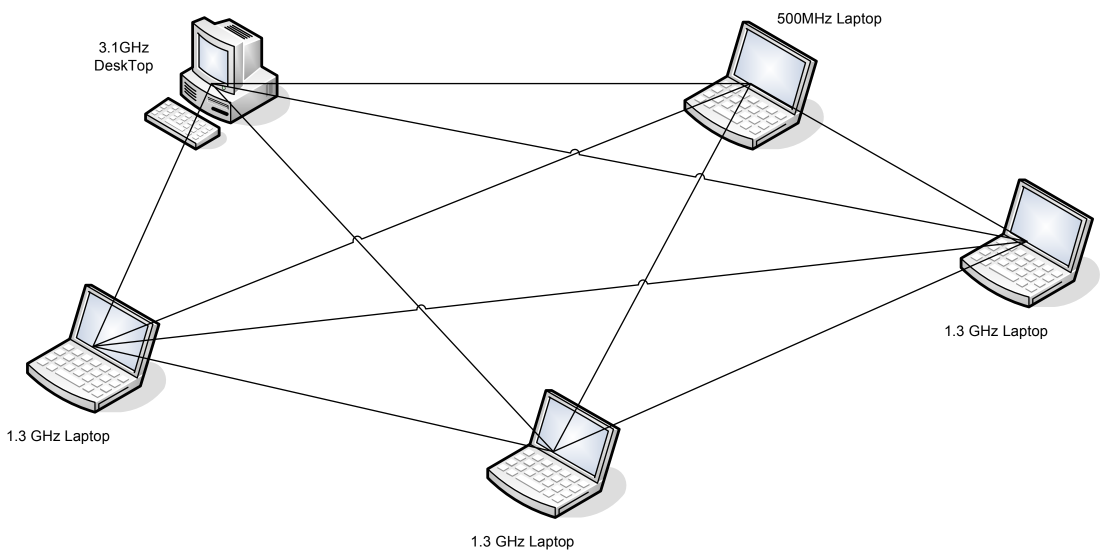


Figure 11 Network topology for experiment 1

TCP messaging was used. The following context distribution weights were used:

- ☐ CPU Frequency : 50
- ☐ Load Average 1 minute : 3
- ☐ Number of processes :3

The processor speed is more heavily weighted as of all the context information available, processor speed will have the biggest impact on the tasks being processed.
The following table illustrates the time in millisecond from when the computation sharing middleware received the job specified until the result data is returned to the calling application. A high time out value of 180 seconds was used to ensure each job piece had time to complete on its target node.

Job ID	**Context Free (ms)**	**Context Aware (ms)**	**Difference (ms)**	**Performance gain (%)**
1	253378	121253	132125	52.1454112
2	242787	136054	106733	43.96157949
3	236827	133696	103131	43.54697733
4	239370	123470	115900	48.41876593
5	247682	142968	104714	42.27759789

Figure 12 Experiment 1 output results

The averaged result over the 5 iteration:

Context Free (ms)	**Context Aware (ms)**	**Difference (ms)**	**Performance gain (%)**
244008.8	131488.2	112520.6	46.1133369

Figure 13 Experiment 1 output results averaged

From these result it can be seen that in the case of context free distribution the middleware performs even worse than if the job was execution locally on the originating node. This is due to the heterogeneous nature of the nodes, and results from the fact the node with the least processing power will receive the same amount of tasks as the node with the most processing power.

Where context information is used, the nodes with the least power are given fewer task resulting in a shorter processing time overall.

Sample distribution information taken from an iteration of this experiment is displayed in the table below:

Node Specification	**Context Aware Tasks Assigned**	**Context Free Tasks Assigned**
1.3 GHz Laptop	2	3
1.3 GHz Laptop	2	2
3.1 GHz Desktop	4	2
500 MHz laptop	1	2
1.3 GHz Laptop	2	2

Figure 14 Experiment 1 task distribution sample

Impact of reduction of job timeout value

Analysis of the logs output by the middleware revealed that even when using the context aware distribution of tasks the 500 MHz laptop node still proved to be a bottleneck, requiring over twice the amount of time to process a single task than any other node. In an effort to improve throughput the timeout value for a task was reduced to 60 seconds. This improved the overall through put as can be seen in the table in figure x.x

Job ID	Context Free (ms)	Context Aware (ms)	Difference (ms)	Performance gain (%)
1	176046	114373	61673	35.0323211
2	172872	109224	63648	36.8179925
3	173353	99473	73880	42.61824139
4	180407	95649	84758	46.98154728
5	162558	89727	72831	44.80308567

Figure 15 Experiment 1 output results with reduced timeout value

Context Free (ms)	Context Aware (ms)	Difference (ms)	Performance gain (%)
173047.2	101689.2	71358	41.25063759

Figure 16 Experiment 1 averaged results with reduced timeout

As can be seen from figures 15 and 16, a reduction in the timeout value for tasks did improve the processing time for both context free and context aware distribution of tasks. With the reduced timeout value context free processing result in the completion of tasks in a time that is marginally better than processing all of the tasks on the originating node. In the case of the context aware distribution, improvement was also seen. The performance gain on average is slightly worse than with the longer timeout, however it is still significant.

Experiment 2: UDP messaging in an Ad-Hoc network where all nodes are not neighbours

For the second experiment the network was set up so that one of the 1.3 GHz laptops was 2 hops away from the originating node. All messaging used the UDP protocol. This is as early experimentation showed the TCP messages were incapable of travelling more than I hop in distance across the MANET. The same context weight information is used as in experiment 1.

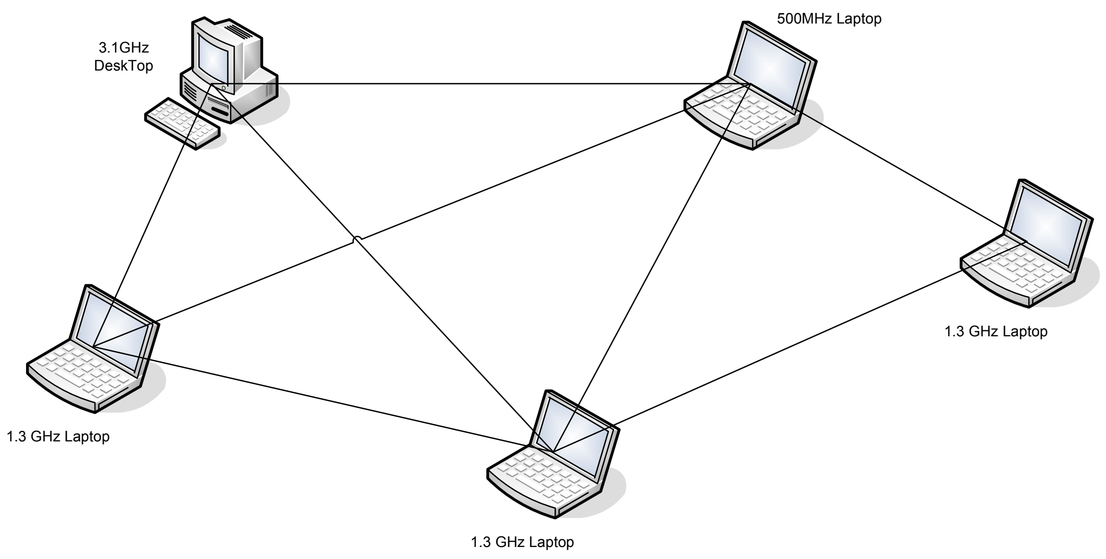


Figure 17 Network topology for experiment 2

Only one result, for distribution without context was received:

Job ID	Context Free (ms)	Context Aware (ms)	Difference (ms)	Performance gain (%)
1	1126791	N/A	N/A	N/A

Figure 18 Experiment 2 partial result

1126791 milliseconds is equivalent to over 18 minutes. It can be clearly seen that this is significantly worse than running the job on just 1 node.
Analysis of the message dispatch logs and debug output revealed that in the first 20 minutes of operation the computation sharing middleware, running across all nodes, sent a total of 298 messages. The corresponding value for the middleware running using TCP connections was 563 messages. Counter- intuitively it appears UDP was delivering messages more slowly, despite the fact that UDP messages have a lower overhead. The problem was traced to fact that many UDP messages were being lost in wireless transmission. Numerous UDP messages were timing out, when after a timeout period the sender did not receive an acknowledgement. The message would then be resent. Normally the message would be successfully delivered after several attempts. However in the worst case scenario, if after 20 delivery failures the message has not been acknowledge it is discarded, as delivery is assumed to be impossible.

This frequent timeout affects computation sharing request messages the most. Most context and service directory update messages are generally less than 64 kilobytes in size and fit in a single UDP packet. However a computation sharing message containing 2 tasks is over 1.5 megabytes in size and requires 25 packets to be sent. Given the unreliability of the delivery of UDP packets it is likely that one outbound packet out of 25 may fail to be delivered 20 times and result in the message as a whole not being delivered.The frequent timeouts also result in a backlog in the outbound message queue. This means that when a message is dispatched to the message queue from the sharing request handler there is a delay before it is actually dispatched. This delay means that a job piece may time out and be redistributed even before it has been dispatched from the outbound message queue.

The above 2 problems result in all job pieces which have been dispatched from the sharing request handler timing out and being distributed. The only tasks which are executed are the ones which are to be processed on the local node, and as a result never have to be dispatched. Each time a job piece is redistributed, so of its tasks are distributed to the local node. After enough job pieces time out, every task will have been processed on the local node and the results will be returned to the local application. For experiment 2, this process of repeatedly distributing task took approximately 18 minutes. Due to the huge disparity between the amount of time to process the tasks directly and using the computation sharing middleware, the experiment was stopped. Using context information in the distribution of tasks would have resulted in no significant benefit.

In order to verify that the UDP timeouts were as a result of transmission over wireless, the experiment was rerun, using just 3 nodes and an ethernet router to connect the nodes. No UDP packets timed-out and the test application ran until completion. It can be reasonably deduced the packet loss was due to the wireless transfer medium.

Gossip Protocol

In order to measure to distribution of context information for each experiment the network discovery component output the current view of the network model every 30 seconds, indicating which nodes existed in the model , as well as if service and system context information was available for each node.

Node Specification	TCP full visibility	UDP full visibility
3.1 GHz Desktop	0:03:00	N/A
1.3 GHz Laptop	0:02:30	0:30:30
1.3 GHz Laptop	0:02:30	N/A
500 MHz laptop	0:03:10	N/A
1.3 GHz Laptop	0:02:30	0:09:00

Figure 19 Gossip Performance, using TCP and UDP protocols

Figure 19 shows the time for each node to reach full visibility of all the other nodes in the network for both the UDP and TCP runs of the experiment. A node has full visibility of another node if it has both system context and service list data for that node. When the experiment was completed using TCP, it can me seen that it took approximately 3 minutes for all nodes to have visibility of the network. There is a lot of randomness in this time, as it depends on how many nodes were already active and if the local node was selected as an update target by a node which already full visibility.

In the case of the UDP messages, for many nodes full visibility was never achieved, as the two nodes that did achieve full visibility, did not maintain it. The reason for this is the repeated time outs of UDP messages. Eventually after a message is not acknowledge by a remote node, the local node will delete the remote node from its network model as it reasonable to assume that if a node is not contactable it is no longer part of the network. Examination of the middleware logs showed that UDP messages were even timing out and causing nodes which were physically quite close to be delete from a neighbours network model. During the TCP run of the experiment, no nodes were ever deleted from the network model until they were shut down.

Chapter 7 Conclusions

The experimental evaluations of this project give a very useful insight in how to best implement computation sharing in and ad-hoc network using context information. Despite the fact that the sharing of computations over multiple hops within an ad-hoc network was not achieved, useful data was retrieved from each experiment.

The Benefits of Computation Sharing using Context Information

The results from experiment 1 clearly show that there are benefits to be obtained by distributing computations around an ad-hoc network. Not only does the use of system context information improve on the performance of distributing tasks across nodes event, but in the case of the heterogeneous test network and the jobs used for the experiment, the use of system context information was required in order to gain a significant benefit from the sharing of processing tasks.
It can reasoned , that given the heterogeneous nature of mobile ad-hoc networks in general, in order to benefit as much as possible from the distribution of context information around the network, some knowledge of the context of other nodes in the network is required. If an ad-hoc network does not take system context information into account the performance gain of processing data across the network can be weakened by the introduction of a node with low processing power. This could be seen as a security risk, as it provides a vector of attack for reducing the computation power available within an ad-hoc network, by maliciously introducing a node with low processing power into the network.

Choice of network messaging protocol

Two different protocols were used for sending messages across the network, UDP and TCP. Each has there own advantages and disadvantages, however when it comes to computation sharing TCP, in the author's opinion, is the best option. However for TCP to be of use for sharing tasks with nodes which are greater than one hop away the protocol needs to be enhanced or extended to allow this functionality.

Unreliable Datagram Protocol

UDP is the most straightforward of all protocols. It is light weight as it does not rely on a fixed two way connection being set up, so it allows messages to be sent more quickly. UDP data packets traverse an ad-hoc network (using the AODV protocol) more easily. TCP connections generally cannot be made across nodes more than 1 hop away. However due to the simple implementation of the protocol, if data is desired to be sent in a reliable manner, as is required by computation sharing middleware, then the use of UDP requires much TCP functionality to be implemented at the application layer. Re-implementing message packet acknowledgement at the application layer is much more expensive than using similar functionality implemented at the network layer.

The primary networking medium used for ad-hoc networks is 802.11 wirelesses. This itself is an unreliable medium, prone to interference, and incapable of detecting collisions between messages on the network. When UDP datagrams are used over this medium, a large number of packets are lost, requiring the sender to wait for an acknowledgement and timeout before resending. This can add a huge overhead in message transmission, as was demonstrated by experiment 2.
Given the performance of UDP in experiment 2, where the computations failed to be distributed across the network in any sort of timely manner and resulting in all remote tasks timing out and being processed on the local node, UDP cannot be recommended as a transport protocol for a computation sharing middleware implementation.

Transmission Control Protocol

The use of TCP carries with it an extra over head of setting up a continuous connection with a target host, however given the issue with UDP described above this is a performance impact that is worth absorbing.
The biggest disadvantage of TCP is that it is not capable of transmitting data to nodes grater than 1 hop away within the network. Obviously network topology will have a direct impact on task distribution when using TCP. If, as was the case with experiment 1, the originating node for the computation sharing has numerous direct neighbours, it will clearly perform well as was evidenced by the experimental results. It is worth reiterating here that when UDP was used computations were not even shared successfully with neighbouring nodes. So despite the inability of TCP to transfer data over a distance of greater than 1 hop it can still be considered to be superior to UDP for computation sharing purposes. One possible mechanism to over come this would to enable nodes to forward TCP messages onto other nodes. This technique would render the use of an underlying ad-hoc protocol such as AODV unnecessary.

Choice of ad-hoc networking Protocol

AODV was chosen as the underlying ad-hoc protocol due to the widespread availability of an implementation, and its popularity for use in research. However, AODV has limitations in that TCP messages cannot be sent of a distance of grater than 1 hop. It also has an overhead in the amount of messages it sends as part of its network discovery.
Due to these limitations and the messaging overhead I do not believe that AODV is suitable as a protocol, to be used by middleware for computation sharing.
I believe that an integration of ad-hoc networking functionality and peer-to-peer computation sharing functionality is required, to benefit fully from computation sharing in an ad-hoc environment. An approach such as this (for example using the aforementioned MPP protocol) would greatly reduce message overhead and improve the ability of connecting to nodes greater than 1 hop away. Another option worth investigating is using an approach similar to indirect TCP[27] and split an N hop TCP connection into N individual TCP connections and forward the message along each TCP connection until it reaches the target.

Use of the Gossip protocol

The gossip protocol was chosen as a means of disseminating data around the network. From the results of the TCP evaluation experiment it can be seen that this was a successful mechanism for ensuring that all nodes get a full view of the network. When running under UDP, a full network view was not achieved by the majority of nodes, however this is due to issues with UDP rather than the use of the gossip protocol. As the implementation and evaluation were focused more on the distribution of computation sharing tasks, there is limited output data concerning the gossip protocol. For a full evaluation of the benefit of using the protocol compared to other approaches it would be necessary to do a comparison between the protocol and other alternatives such as flooding the network. There are also potential optimisations such as instead of using the gossip protocol in a random fashion, using an implementation where updates are sent to the least recently updated node in the network model. This is left as future work and is beyond the scope of this dissertation.

Chapter 8 Future Work

The scope of potential research in Mobile Ad-Hoc Networks is massive. The same can be said for peer to peer applications and the use of context awareness within applications. Within the timeframe available I have only been able to scratch the surface of these areas, however I hope to have provided a starting point for much further research into computation sharing in MANETs, using context information. This section details future areas of research that have been suggested by my experience of working on this dissertation, which I have not had time to explore fully.

Choice of System Context

There are many different pieces of context information supplied by the middleware application which have not been used as part of the evaluation. For example system memory and battery charge are pieces of system context which respectively could be very relevant for a specific type of memory intensive task, or for use in nodes where battery life is absolutely critical. Further research is warranted to discover which types of task are better suited for different context weight values.

Use of context information beyond system context

In the current implementation of the middleware context information is taken only about the node processor and battery. It would also be possible to use network information such as node distance (in hops) and network link state information (should that information be available).

Another item that could be eligible for consideration as a piece of context is the task itself. The task size or complexity could be used as information to help determine how to best distribute a set of tasks. This could be context conformation derived purely from the task size, by the middleware application, or a rating defined by the calling application.

Extending the Functionality of the Middleware:

Currently the middleware provides the basic functionality required to enable computation sharing between neighbouring nodes in an ad-hoc network, however there were many other pieces of functionality which were considered as nice to have features, but could not be implemented due to time constraints.

These are:

- Message Compression
    - When sending messages no from of compression is used. Given that the amount of data associated with a task may result in a significant over head, data compression could be used to speed up message delivery times. Of course there is the question of compressing the data at source and uncompressing it at the destination and the impact that this would have on system through put. An analysis would need to be done on whether message compression would benefit overall throughput.

- Message Encryption and Authentication
    - Currently all messages are sent as plaintext and are open to interception. There is no authentication of message senders, so it may be possible for a malicious node to compromise the quality of shared computations. Also there is no redundancy built into job distribution to allow the results of different nodes to be compared, in an effort to detect malicious nodes. Such a technique is used by the BOINC platform to detect the presence of malicious nodes

- Forwarding of jobs from nodes unable to complete their tasks

    - Currently if a node running the middleware suddenly runs out of battery power the node will shut down and if any jobs were currently being processed on the at node will be lost. The job will time-out at the originating node and be redistributed. Given that the middleware application is aware of the battery power level as part of the system context information associated with the node, a more elegant solution would be for the remote node doing the processing to forward the partially completed job piece to another node, once the battery power dips below a predefined threshold.

- Storing of Job results if the originating node is unavailable
    - It is possible that while jobs are being processed on remote nodes, the jobs originating node may leave the network temporarily and be unavailable when the remote node tries to return the task result data. Given the dynamic nature off MANETS, it is entirely possible for a Node to leave the network and then rejoin it moments later. In order to prevent computations from being redone, it would be useful for remote nodes to store computation results for a fixed period so that if the originating node returns to the network, the result data can be returned.

- Support of fully fledged services.
    - Currently within the middleware, services are actually applications which are invoked using a command line call from within the middleware. A full implementation using services which are accessible using a protocol such as SOAP would greatly add to the usability of the middleware.

- Support minimum context values.
    - As was seen in experiment 1 even if context information is used a node with limited processing power can still reduce the throughput of the middleware. While this can be partially remedied with a carefully selected timeout value, another option would be to allow the local application calling the middleware to specify a minimum threshold for a context value, so that if any potential sharing target nodes have a value lower than this , they will be ignored.

Integration of Middleware and Ad-hoc routing

The middleware application runs on top of an ad-hoc routing protocol, by reading the routing table output to a file by the AODV protocol. Much tighter integration of the middleware and the routing protocol would raise the possibility of combining the AODV discovery messages with middleware discovery messages. This would reduce the messaging overhead of the middleware. Given that, when the middleware was running using UDP, many UDP messages were dropped in a wireless environment, that were not dropped in a wired environment, reducing the number of overall message may increase the percentage of UDP messages successfully delivered, and would also result in a lower drain on the power sources of nodes within the ad-hoc network.

Given that the middleware did not distribute tasks successfully in a 5 node network when using UDP, one option would be two implement a store and forward TCP protocol, similar to indirect TCP, a proposal by Bakne and Badrinath [27].
My suggested approach for future work designing and implementing a TCP based computation sharing MANET middleware application is as follows:

- The network model would also act as a routing table for all nodes. Due to the limits of TCP, nodes can only connect to there direct neighbours, however using the existing gossip protocol nodes would still receive context and service updates from nodes which are not their direct neighbours.

- When a node receives an update for a non neighbour node it should also contain information about the path the data took to get there including hop count and the ID of the node which forwarded the packet (i.e. the next node in the path to the target node). The network work node model should only store the next node in the shortest path to the target node.

- When a message is to be sent to a non-neighbour node, the local node looks up the target node in the network model and forwards the message to the next node in the path to the target node. This node forwards the data to the next node in the path, until the target node is reached. Due to the overhead of forwarding TCP messages across nodes, the hop count to a target node would need to be used as in input into what nodes tasks will be distributed to. For example to distribute tasks which are 10 hops away would be pointless as the overhead of message delivery would be too significant.

Appendix A AODV Start- up Shell script

```
#!/bin/bash
echo "Hello $USER"
echo "Thjis script starts AODV running on this machine in AD_HOC MODE"

echo "sudo /etc/dbus-1/event.d/25NetworkManager stop"

sudo /etc/dbus-1/event.d/25NetworkManager stop

echo "sudo modprobe ndiswrapper"
sudo modprobe ndiswrapper


echo "sudo ifconfig eth0 down"
sudo ifconfig eth0 down


echo "sudo iwconfig eth1 essid aodvnet mode Ad-Hoc"
sudo iwconfig eth1 essid aodvnet mode Ad-Hoc

echo "sudo ifconfig eth1 134.226.51.176"
sudo ifconfig eth1 134.226.51.176
echo "sleep 2"
sleep 2

echo "sudo iwconfig eth1 essid aodvnet mode Ad-Hoc"
sudo iwconfig eth1 essid aodvnet mode Ad-Hoc

echo "sleep 2"
sleep 2
echo "All done - displaying status"

ifconfig
iwconfig

echo "modprobe koadv"
sudo modprobe kaodv

sleep 2

echo "sudo aodvd -i eth1 -r 30"
sudo aodvd -i eth1 -r 30
```

Appendix B XML Messages Examples

Context Update XML Message

```xml
<?xml version="1.0"?>
<!DOCTYPE contextsharing_xml_doc SYSTEM "contextsharing_xml_doc.dtd">
<computationsharing:computationsharing
xmlns:computationsharing="http://foo" type="context_update"
target="192.168.1.103" port="24679" source="192.168.1.100" Time="Mon
Sep 10 21:25:05 2007&#10;" epochTime="-250035923">
  <context:context xmlns:context="http://foo" epochTime="1189455905069"
nodeIP="192.168.1.100" nodename="192.168.1.100"
tcpresponseport="24679">
    <battery:BatteryContext xmlns:battery="http://foo">
      <status>No Battery</status>
      <charge>100</charge>
      <acpower>on-line</acpower>
    </battery:BatteryContext>
    <Processor:ProcessorContext xmlns:Processor="http://foo">
      <CPU>
        <cputotal>210160</cputotal>
        <user>39390</user>
        <nice>288</nice>
        <sys>16758</sys>
        <idle>141871</idle>
        <tickfrequency>100</tickfrequency>
        <scaledfrequency>3000</scaledfrequency>
        <nominalfrequency>3000</nominalfrequency>
        <name>Intel(R) Pentium(R) D CPU 3.00GHz </name>
        <cores>2</cores>
      </CPU>
      <memory>
        <memtotal>1002</memtotal>
        <memused>896</memused>
        <memFree>106</memFree>
        <memShared>0</memShared>
        <memBuffered>90</memBuffered>
        <memCached>353</memCached>
        <memUser>452</memUser>
        <memLocked>0</memLocked>
      </memory>
      <load>
        <processes>135</processes>
        <nonsystemprocesses>53</nonsystemprocesses>
        <nonidleprocesses>0</nonidleprocesses>
        <nonidlenonsystemprocesses>0</nonidlenonsystemprocesses>
        <loadaverage1>3.47</loadaverage1>
        <loadaverage5>2.98</loadaverage5>
        <loadaverage15>1.96</loadaverage15>
      </load>
    </Processor:ProcessorContext>
  </context:context>
</computationsharing:computationsharing>
```

Service Directory Update XML Message

```xml
<?xml version="1.0"?>
<!DOCTYPE contextsharing_xml_doc SYSTEM "contextsharing_xml_doc.dtd">
<computationsharing:computationsharing
xmlns:computationsharing="http://foo" type="service_update"
target="192.168.1.103" port="24679" source="192.168.1.100" Time="Mon
Sep 10 21:25:05 2007&#10;" epochTime="-250035922">
  <service_directory:service_directory
xmlns:service_directory="http://foo" epochTime="1189455905069"
nodeIP="192.168.1.100" nodename="192.168.1.100"
tcpresponseport="24679">
    <service:service xmlns:service="http://foo">
      <type>application</type>
      <name>imagemagick</name>
    </service:service>
    <service:service xmlns:service="http://foo">
      <type>application</type>
      <name>a2mp3</name>
    </service:service>
    <service:service xmlns:service="http://foo">
      <type>application</type>
      <name>mp32ogg</name>
    </service:service>
  </service_directory:service_directory>
</computationsharing:computationsharing>
```

Appendix C Local Application API Utilisation Sample Code

```
// Create the computation sharer object
ComputationSharer CS(globalSocketType);

//Create the sharing Job Naming the job and the service to be used to process the job
CS.createJob(JobName,"imagemagick");

// Specify Context weights if this information is to be used
CS.setContextWeight(CPU_NOMINAL_FREQUENCY,10);
CS.setContextWeight(LOAD_AVERAGE_5_MINUTE,1);
CS.setContextWeight(TOTAL_PROCESSES,5);

// Set Job Timeout in seconds
CS.setJobTimeout(60);

//Add Tasks to Job specify input data filename and task parameters
CS.addTask("Task1", "./Picture001.jpg", "-charcoal 5 -thumbnail 256");
CS.addTask("Task2", "./Picture002.jpg", "-charcoal 5 -thumbnail 256");

//Dispatch the job to the middleware application
CS.dispatchJob();

//Use computation sharer semaphore to wait on the response
CS.waitforResponse();

// save the result data from the job
// filenames will be the input names with a “-out” suffix e.g. "./Picture001-out.jpg",
CS.writeTaskDataToFile()
```

Bibliography

1. Mauthe, A. and O. Heckmann, *Distributed Computing – GRID Computing* in *Peer-to-peer Systems and Applications*, R. Steinmetz and K. Wehrle, Editors. 2005, Springer. p. 193-206.
2. Anderson, D.P. *BOINC: A System for Public-Resource Computing and Storage*. in *5th IEEE/ACM International Workshop on Grid Computing*. 2004.
3. Oram, A., *Peer-to-Peer: Harnessing the power of disruptive technologies*. 2001: O'Reilly & Associates inc.
4. Eberspächer, J. and R. Schollmeier, *Past and Future*, in *Peer-to-peer Systems and Applications*, R. Steinmetz and K. Wehrle, Editors. 2005, Springer. p. 17-23.
5. Clarke, I., et al., *Protecting Free Expression Online With Freenet.* IEEE Internet Computing 2002. **6**(1): p. 40-49.
6. Festag, A., et al. *Fleetnet: Bringing Car-To-Car Communication Into The Real World*. in *11th World Congress on ITS*. 2004. Nagoya, Japan.
7. *One Laptop Per Child Foundation*. [http://www.laptop.org]
8. Dey, A.K., *Understanding and using context.* Personal and Ubiquitous Computing Journal, 2001. **5**(1): p. 4-7.
9. Perkins, C.E. and E.M. Royer. *Ad-hoc On-Demand Distance Vector Routing*. in *IEEE Workshop on Mobile Computing Systems and Applications*. 1999: IEEE.
10. Tannenbaum, A., *Distance Vector Routing*, in *Computer Networks*. 2003, Prentic Hall. p. 357-358.
11. Foster, I. and C. Kesselman, *Globus: A Metacomputing Infrastructure Toolkit.* International Journal of SUpercomputer Applications, 1997. **11**(2): p. 115-128.
12. *BOINC Statistics Webpage*. [http://boinc.netsoft-online.com/e107_plugins/boinc/bp.php?project=19]
13. *Top 500 Super Computer List – June 2007.* [http://www.top500.org/lists/2007/06]
14. Wang, Z., Q. Chen, and C. Gao. *Implementing Grid Computing over Mobile Ad-Hoc Networks based on Mobile Agent*. in *International Conference on Grid and Cooperative Computing*. 2006.

15. Pham, V.A. and A. Karmouch, *Mobile Software Agents: An Overview*, in *IEEE Communications Magazine*. 1998. p. 26-37.

16. Stoica, I., et al. *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*. in *ACM SIGCOMM Conference*. 2001. San Diego, California.

17. Schollmeier, R., I. Gruber, and F. Niethammer. *Protocol for peer-to-peer networking in mobile environments*. in *12th International Conference on Computer Communications and Networks*. 2003: IEEE.

18. Nedos, A., K. Singh, and S. Clarke, *Mobile Ad Hoc Services: Semantic Service Discovery in Mobile Ad Hoc Networks* in *Service-Oriented Computing – ICSOC 2006*. 2006, Springer.

19. *World Wide Web Consortium Extensible Mark-up Language (XML) homepage*. [http://www.w3.org/XML/]

20. Mascolo, C., et al., *XMIDDLE: A Data-Sharing Middleware for Mobile Computing*. International Journal on Wireless Personal Communications, 2002. **21**(1): p. 77-103.

21. *The GNU Common C++ Library*. [http://www.gnu.org/software/commoncpp/]

22. *The Lib XML++ library* [http://libxmlplusplus.sourceforge.net/docs/manual/html/index.html.]

23. *The gLibTop Library*. [http://library.gnome.org/devel/libgtop/stable/]

24. Tannenbaum, A., *The Internet Transport Protocols: UDP*, in *Computer Networks*. 2003, Prentice Hall. p. 524-532.

25. Tannenbaum, A., *The Internet Transport Protocols : TCP*, in *Computer Networs*. 2003, Prentice Hall. p. 532-556.

26. RIVEST, R.L., *RFC 1320 The MD5 Message Digest Algorithm*. 1992.

27. Bakre, A. and B.R. Badrinath. *I-TCP: Indirect TCP for Mobile Hosts*. in *15th International Conference on Distributed Computer Systems*. 1995.