MobNet: Mobile Network Simulation API

Avinash Bhadresha
Department of Computer Science
University College London
London
United Kingdom,
a.bhadresha@cs.ucl.ac.uk

May 12, 2005

Abstract

abstract goes here... ...

1 Introduction

The area of Mobile Computing is at present one of the most rapid growth area's in Computer Science. The miniaturization of components and increased technology has allowed computer devices to be used in scenarios never before envisaged. Mobile phones are becoming increasingly powerful and with 3rd and 4th generation networks their ability to communicate will be on par with today's fixed networks. This opens up a whole host of new models of interactions between humans and computers in their everyday lives.[1] It is not only mobile phones which are providing the thrust in mobile computing. The rapid proliferation of WiFi in homes and enterprises is enabling an always connected population of computer users. WiFi hotspots around major attractions allow mobile users access to Internet, email and work place applications on the move. This will be greatly improved with idea's such as Community networks and WiMax.

Another exciting area is Ad hoc networks in which there is no fixed infrastructure to aid communications. Each node in an ad hoc network is limited in its transmission range and can only directly communicate to those nodes within it's transmission range. With the use of multi-hop routing a node can communicate to another node outside of its transmission range. This opens up opportunities to share and exchange resources. This is already common place in mobile phones where images are shared via Bluetooth. In the last few months there has been a spate of new releases of mobile gaming

[1] http://www.theregister.co.uk/2005/01/28/mobile_phones_q4_04/

devices. Each of which feature WiFi or Bluetooth connectiviy in order to increase gameplay by allowing multiple gamers to connect and play together in an ad hoc network.

Mobile computing is a vast area and the discussion above gives only a taster of the applications available and technologies involved. What is clear though is that this depth complicates the design procedure for mobile applications. When making networked applications for mobile devices such as PDAs and mobile phones it can be very difficult to test the applications. First you need multiple devices, then you have to configure each one, finally you need to get people to use them and move around while you are testing. This makes testing a complicated and expensive process which is generally done at the end of the development phase. At present discrete event simulators are used to model mobile network applications in order to gauge how they may work in the real world. However this only allows a model of the application to be simulated and not the actual application itself.

We present a Mobile Network Simulator, the MobNet API, with which you can simulate multiple networked devices on one computer allowing the developer to quickly test out his/her Java applications. The MobNet API also transparently simulates the movement of these devices and only allows them to communicate with each other if they are in range therefore giving a computerised simulation scenario mimicking the real world. The MobNet API provides developers with a platform to quickly test out creations in order to assess how they adapt to different mobility scenarios. This allows for rapid application development allowing testing for fault tolerance and failure handling. The benefits to developers is reduced development time through the simulation of different scenarios rather than having to use real devices. Cost of development is also reduced as a result of not requiring multiple mobile devices to test the application. Developers can now attack risks associated with mobility such as packet loss, out of range from servers, etc early on in the development phase when the cost of re-development is cheapest.

Our implementation covers the most frequently used parts of the java.net API and we hope to fully replicate it in the future. The design has been carefully composed in order to easily extend the implementation for use with different mobility models. A unique feature we have achieved is the Visualiser which is also easily extensible.

The rest of this section discusses the issues to be considered during the development of mobile applications and current tools used for the evaluation and simulation of such applications. Section 2 details the design goals we tried to adhere to in order to compose the architecture followed by Section 3 which then gives an abstract overview of the final design. Section 4 provides detailed information on the implementation of the MobNet API for Java. Section 5 briefly introduces an example application which is then used to evaluate the API in Section 6. Finally in Section 7 we conclude the report

giving future directions for our research.

1.1 Development Issues

When developing applications for mobile devices utilising networks we there are a number of issues which have to be considered as a result of the mobile nature of these devices:[5][2]

- Due to the devices physical location its configuration may change unpredictably as the device is connected to or disconnected from the network, or even as the device moves between network connection points.
- Mobile users may elect to work online or offline at a given time depending on their connection.
- Wireless networks are susceptible to external interference and attenuation, potentially affecting reliability and decreasing the effective bandwidth.
- Battery-powered devices, by their very nature, can operate for only a limited time without recharging or replacing batteries.

Mobile developers therefore need to take into account these factors and design applications which operate transparently to the underlying network and are fault tolerant. They also need to take into account power optimization through efficient use of I/O. Designing such applications is a lot more complicated than traditional applications for fixed systems. Developers therefore need tools to aid them in the mobile application development process and a key benefit would be the ability to test prototypes early on in the development cycle. Another reason for developers to take testing seriously is that mobile phone developers are network are now restricting applications which can be used on their phones and networks to those which have been officially verified by a third party such as Symbian or Java. In the case of Java applications there is a Java Verification service in which Sun tests mobile applications in order to make sure they are resilient and robust enough for consumer mobile phones. Information on the criteria can be found at (www.javaverified.com) and the criteria for networking is given in the Appendix.

When trying to test applications developed for Mobile scenarios we run into a number of problems. We need multiple devices on which to test the software and this results in having to spend time in configuring each device to run the application. By definition mobile applications move around. If we try and factor in this movement by physically changing the positions

[2] `http://www.devx.com/Intel/Article/20799`

of these devices we have two problems. The first is how to actually move these devices? Do we expect a person to take each device and move around with it, or do we rely on some sort of automation. Secondly the range of movement to be tested is linked to the technology used for networked communication. For example testing a Bluetooth configuration is feasible in a laboratory where the range of Bluetooth is at most 10 metres and so movement to the boundaries of this restriction is possible. However when trying to test multiple WiFi devices we would need an area covering hundreds of square meters which is not feasible. As we add more and more devices to test their interactions the whole thing becomes a deployment nightmare. Mobile devices in most cases have a limited supply for power which in the case of Sensor networks is un-replenishable. Having to re-charge devices during experiments and monitor battery usage is extremely tedious and can hinder the productivity of testing. The result is that testing mobile ad hoc applications during development is very difficult and expensive to do at present. We now present a few of the current solutions in the area.

1.2 Current Solutions

Development of mobile wireless applications has been evaluated mainly by simulation and emulation. Ns-2[3] is a discrete event simulator targeted at networking research. Ns provides support for simulation of TCP, routing, and multicast protocols over wired and wireless (local and satellite) networks. One new simulation environment is called SWANS and is based on the JiST virtual machine platform. JiST[2] is a discrete event simulation engine that runs over a standard Java virtual machine. It is a new approach in building discrete event simulators. JiST converts an existing virtual machine into a simulation platform by embedding simulation time semantics at the byte-code level. Regular Java code is written to make JiST simulations and are compiled and executed in the standard way. SWANS is built atop the JiST platform. SWANS is organized as independent software components that can be composed to form complete wireless network or sensor network configurations. Its capabilities are similar to ns2 but is able to simulate much larger networks. Discrete event simulators such as ns-2 and JiST have been used to simulate a wireless scenarios but are mainly used to simulate routing protocols. These are not sufficient to fully test an application and only tests a simplified model of an application.

Emulators can be used to provide static test conditions when using wireless mobile devices. JEmu[4] is once such emulator which uss a central control module to emulate thee wireless layer of a protocol stack. The emulator program accepts a radio layer message from the mobile hosts and determines whether or not to forward the message depending on the location

[3] `http://www.isi.edu/nsnam/ns/`

information of the mobile host. EMWIN is a network emulator in which the target mobile wireless network is precisely mapped to an laboratory network consisting of several emulator nodes. Each emulator node is able to emulate multiple mobile hosts and can be configured such that predefined network conditions and traffic dynamics can be generated in an automatic manner. The result is a wired network which emulates the characteristics of the mobile wireless network. EMWIN is limited by the fact that a node can only emulate as many nodes as it has interface cards which decrease the ease at which multiple nodes can be emulated and increases the overall cost complexity of the emulation.

The above methods are quite cumbersome and need a lot of user configuration in order to set up the simulation/emulation scenario's. A lightweight framework for the development, development and evaluation of applications and algorithms for wireless mobile ad hoc networks has been developed called FRANC. FRANC contains a Virtual Network layer that can emulate a multihop environment by filtering and discarding received packets. This layer creates a multihop topology even if the nodes involved are within physical communications distance therefore allowing easy evaluation of applications in the multi-hop topology. Also each layer in the protocol stack can be configured to emulate various characteristics in order to provide a consistent test bed. However, FRANC is designed to be used on physical nodes where each node is actually running the application. This again enforces a level complexity into evaluating applications as we need multiple physical nodes.

In summary, the tools available for application developers to test and evaluate their programmes are quite complex and expensive to deploy. Most of the tools concentrate on the lower layers of the communications stack and are heavyweight in their use. In the following sections we introduce MobNet a lightweight API simulating the Java.net API to aid MANET application development and evaluation.

2 Design Goals

When designing the MobNet Simulation API there were a number of design goals which had to be adhered to. Below we discuss some of the goals and why they were chosen.

2.1 Which Platform?

There are many mobile software development platforms in the market at present however the dominant ones are BREW, J2ME, .NET Compact Framework.[4]

BREW(Binary Runtime Environment for Wireless) is an application execution environment released by Qualcomm. BREW is a vendor neutral application development and execution environment capable of running on any network and any handset. It sits right above the hardware and can run with many different device operating systems such as Palm OS. It is an up and coming platform and has gained enterprise support with partnerships with Oracle and IBM, the latter providing a J2ME JVM running on top of BREW.

The Java 2 Micro Edition is the most popular platform for developing applications for Mobile Devices. J2ME is a subset of the Java 2 Standard Edition and is used for computationally low power devices such as Mobile phones and PDA's.

Microsoft have launched the .NET Compact Framework which like J2ME is a scaled down version of the full .NET Framework. While J2ME programmes can run on a virtual machine on top of a mobile devices native platform the .NET Compact Framework requires a smartphone running on the Compact Framework. The advantage of the .Net Framework is that programmers can use the same set of tools and programming models used to develop applications for desktop PC's. This now opens the gate for a whole new range of developers used to developing using Microsoft's development tools to develop applications for mobile phones without having to re-train.

We ruled out BREW due to its relative immaturity in the market compared to .NET CF and J2ME. J2ME was then chosen as the target platform for which to build the Simulation API because of its impressive cross platform support and prospective wide reach. The final decision was to create a Java tool which would allow any Java application to be simulated. As J2ME is a subset of Java this should allow all combinations of J2ME to be supported in our simulation API.

[4] `http://weblogs.asp.net/nleghari/articles/44057.aspx`

2.2 Transparent or Opaque API?

When designing the Simulation API we had to think long and hard about how the API would fit into the application being developed. Our initial aim was to produce an simple and easy to use simulation API which did not need too much integration work by the developer, therefore making it as transparent as possible. This is because a tool such as this would only be used if it did not impose too much of an overhead into the development process. The ideal solution would be to allow an application developer to develop without having to make any changes in order to use the simulator. In practice having this goal imposes a lot of inflexibility on how we implement such a system therefore we reached a compromise in which only minimal amounts of code has to be changed (described later).

2.3 Pluggable Mobility Models

It is essential that the Simulation API supports different mobility models in its use. This will allow applications developers to test their software in various different mobile scenarios. There are various types of mobility models as classified by Camp, Belong and Davies [3] [5]:

- Random Walk Mobility Model: A simple mobility model based on random directions and speeds.
- Random Waypoint Mobility Model: A model that includes pause times between changes in destination and speed.
- Random Direction Mobility Model: A model that forces Mobile nodes to travel to the edge of the simulation area before changing direction and speed.
- A Boundless Simulation Area Mobility Model: A model that converts a 2D rectangular simulation area into a torus-shaped simulation area.
- Gauss-Markov Mobility Model: A model that uses one tuning parameter to vary the degree of randomness in the mobility pattern.
- A Probabilistic Version of the Random Walk Mobility Model: A model that utilizes a set of probabilities to determine the next position of an Mobile Node
- City Section Mobility Model: A simulation area that represents streets within a city.

Our simulation API will contain a Mobility Model interface which will allow different models to be used with the simulator.

[5] `http://toilers.mines.edu/papers/pdf/Models.pdf`

2.4 Visualiser

Application engineers can often benefit from viewing in 2D form how the nodes they are simulating are moving about in order to infer move information about the behaviour of their programmes. The inclusion of a visualiser is therefore a feature which cannot ignored and will be one of the highlights of the API.

3 Abstract Approach

3.1 High Level Overview

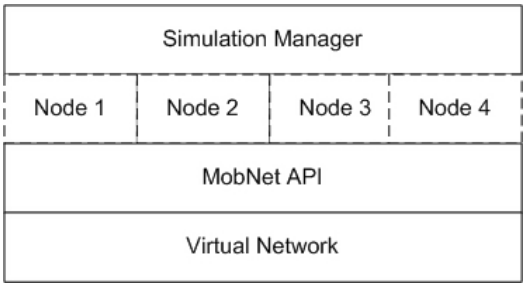


Figure 1: High Level overview of system components.

The MobNet simulation environment is built of 3 components:

- MobNet Java API which mirrors the java.net.* API and provides the application programmer with the same API so the application programmers need not have to change any method calls or arguments in order to run their applications on the simulator. This enables applications to be quickly ported for use in the simulator are only an import statement needs to be change to any class which uses java.net.
- A Virtual Network component which simulates the physical layer and deals with exchanging packets between nodes involved in the simulation.
- Simulator Manager, this controls the simulation and allows the developer to set parameters such as mobility models and which applications are to be simulated. The manager also contains the visualiser component.

Figure 1 shows how the 3 components integrate together with the applications being simulated. Each application, here labelled as a node, runs under the Simulation Manager. The Simulation Manager is responsible for launching and configuring the applications and starting the underlying simulation environment. Once the applications are running they then have access to the Virtual Network via the MobNet API. This API shadows the java.net API therefore applications programmers need make no changes to their networking code. The MobNet API and Virtual Network combine to allow multiple applications to transparently appear as independent hosts on one computer.

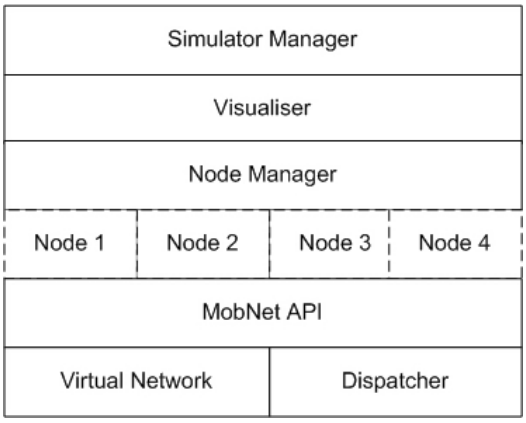


Figure 2: 2nd Level overview of system components.

3.2 2nd Level Overview

We will now introduce more detail into the explanation of the system. Figure 2 shows 3 more components the Visualiser, Node Manager and Dispatcher.

The Node Manager is responsible for keeping track of details and properties each node possesses. These include the colour a node appears in the Visualiser, the current co-ordinates in the simulation field and the name of the application Java class to be executed as the application for this node.

The Visualiser directly takes the co-ordinates of each Node from the Node Manager and draws and on-screen 2D representation of the current simulation field. Each node being simulated is automatically given a colour by the node manager in order to help the application developer trace different types of applications to their position in the simulation field. This feature for example could be used to quickly find where a server type application is compared to the rest of the other nodes, helping to provide feedback on performance.

Each node deposits packets into the Virtual Network through the MobNet API. The Dispatcher then services these packets in a FIFO manner and sends them to their destination node. The Dispatcher is an integral part of the Simulation and runs as a thread servicing the Virtual Network packet queue. We will now explain in more detail the relationship between Nodes and the Virtual Network.

4 The MobNet API for Java - Implementation

- Implemented and tested using Java 1.4.205
- Current Number of Lines of Code: 5600

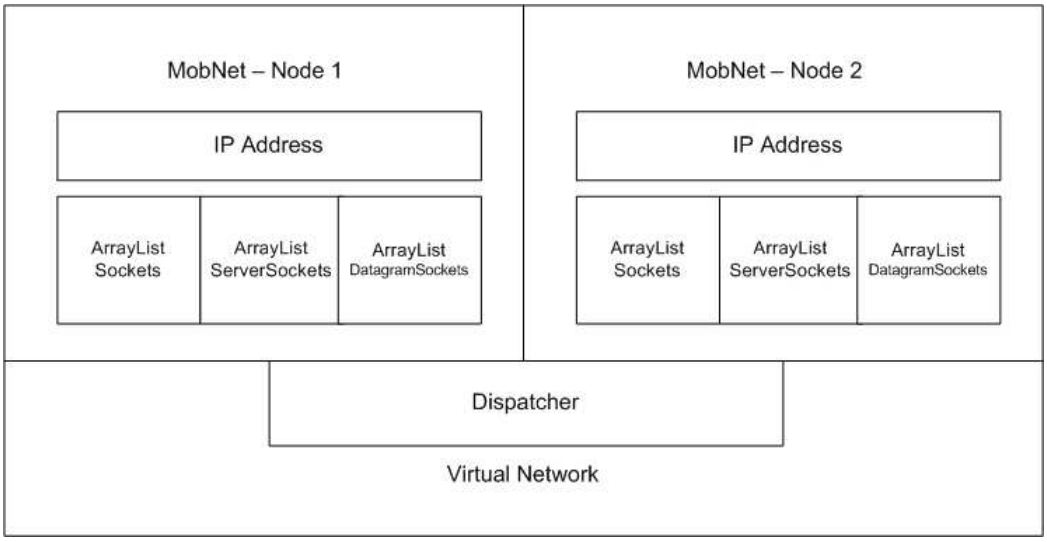


Figure 3: Two Nodes connected by the Virtual Network

Each node in the simulation contains a MobNet object. This encapsulates the Virtual Network and provides a handler to all ServerSockets, Sockets and DatagramSockets created by a node. Sockets therefore have access to the Virtual Network and packets are sent to the Virtual Network to be dispatched to the destination. The Dispatcher gains access to destination Sockets via the Simulation Manager. The Simulation Manager has a handle to the MobNet objects of each Node. The Dispatcher finds the destination IP address of a packet and then looks up the MobNet object for the required IP Address(node). It then checks to see if the packet is a Datagram or a TCP/IP type packet before depositing the packet directly into the destination Socket via the ArrayList containing all Sockets.

These are the basic principles by which the application nodes have a simulated network in which they can send and receive data and is the core of the system. Next we will discuss how application nodes achieve separation from each other, gain IP Addresses and communicate with the common core.

4.1 The MobNet ThreadGroup

One of the hardest issues we had to resolve was how to let each node execute as a separate entity yet transparently allow it access to the Virtual Network.

How should a node gain access to the the Virtual Network transparently from the application developer. Obviously each node would have to share the same Virtual Network and so an initial solution is to have a static class which encapsulates the Virtual Network. This solves the immediate problem of nodes having access to a network. However a further problem arises. In a real life scenario each node can be pre-configured with an IP Address. This is used by networking layer of the Operating System on which the application is running to append each outgoing packet with the source IP Address. We therefore need each application node in the simulator to have an IP Address which it can use exclusively. One answer is to set it as a variable in each Java Class which uses networking features however this detracts from one of our design aims which is to make the use of the simulator as transparent as possible for the developer. By making the developer explicitly include variables in each class we are adding development overhead which we were not keen on doing.

The solution was to implement a new type of ThreadGroup which encapsulates a MobNet, under which the application node runs in a thread normally. When the Simulation Manager launches an application node class it creates a new thread, the Application Container. The thread is then paired with a MobNet Thread Group and will run under it. This then acts as a container for the application and provides a way for an application to access it's MobNet. When a Socket is created by an application node the MobNet API transparently traverses the thread group structure in which the thread is running until it finds the one containing the MobNet. In our implementation this is done by the MobNetGrabber class. Once the Socket has a reference to the MobNet it then can add itself into the ArrayList of Sockets and also acquire its IP Address, solving the previous problem. Figure 3 shows how the programmer has the flexibility of implementing networking calls in whichever thread structure he wants to. This is because the top Level thread will always be the one created by the Simulation Manager and as such it will always contain the handle to the nodes MobNet.

In order a give a clear picture on how each part of the MobNet API works, while giving a transparent view to the application developer, we now give an example run through on how the ServerSocket and Socket parts of the API work together with the Virtual Network.

4.2 MobNet.ServerSocket

Program listing 1, shows how a ServerSocket is created by an application developer with our API. As you can see it is exactly the same as the java.net package. When an instance of ServerSocket is created the thread hierarchy in which the call is made is traversed until the MobNet ThreadGroup is found. Once the MobNet is found a reference to the ServerSocket is added into the ArrayList of ServerSocket's. The ServerSocket contains a buffer

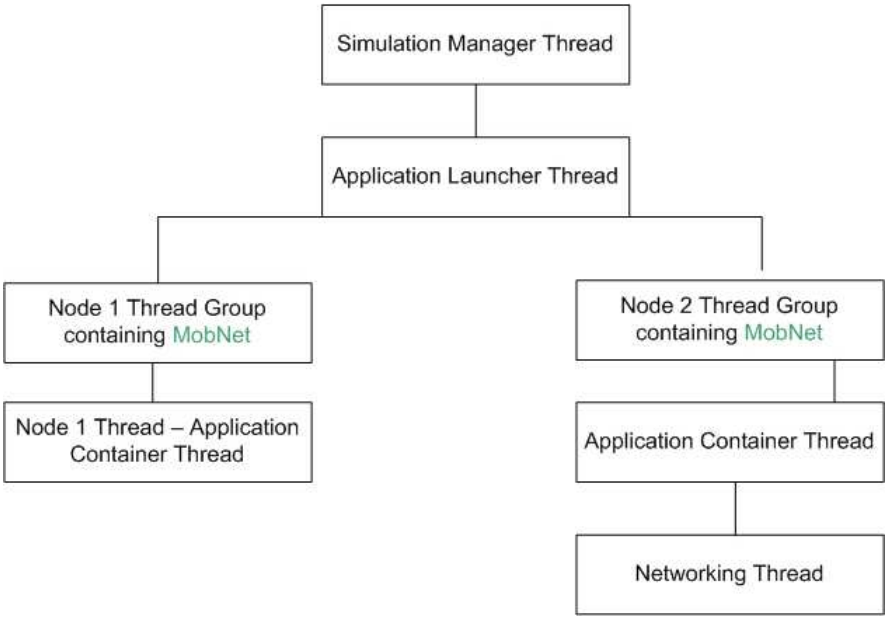


Figure 4: An example thread structure containing two applications.

Program 1 Example code for setting up a Server Socket

```
ServerSocket dateServer = new ServerSocket(3000);
System.out.println("Waiting for connections.");
Socket client = dateServer.accept();
print("Accepted a connection");
```

which accepts New Connection packets, called PIPs, from other hosts. New Connection packets are created when a host wishes to connect to a Server-Socket, show pictorially in Figure 5. The host sends it's own IP Address and port number to the desired ServerSocket. The New Connections buffer therefore acts as a list of hosts wishing to connect to Server. If there are no hosts wishing to connect, the use of accept() will block. The ServerSocket thread waits and gets notified when a New Connection packet arrives.

The reason we use New Connection (PIP) packets is that a client Socket needs to somehow notify the ServerSocket that it wishes to connect and supply its details (IP, port number). If we had used a direct method call this would introduce a bottleneck as all calls would be synchronous. Instead we have an asynchronous queue, the New Connections buffer, which the ServerSocket then services.

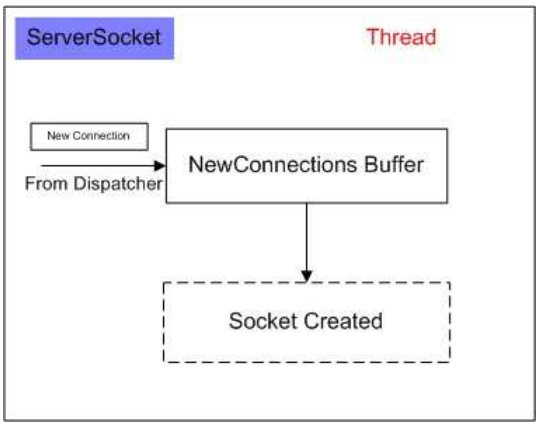


Figure 5: Pictorial view of a ServerSocket.

4.3 MobNet.Socket

Compared to ServerSockets, Sockets are more complicated in their construction. A client has to synchronise with a server and only once both parties have set up the connection can data transfer begin. This enforced the use of synchronisation primitives for threads and the solution is now described.

Initially when a Socket is created a boolean isReady is set to false. This boolean represents the successful set up of a Socket to a Server and vice versa. The Socket then calls the Manager and requests to be connected to a ServerSocket as a specified IP Address and port number. The Manager now finds the MobNet for the desired destination IP Address and finds the correct ServerSocket. A PIP is then input into the ServerSocket's New Connections buffer. A PIP is a class which encapsulates the port number and IP Address of the client which is wanting to connect to the Server. The PIP also connects a reference to the client's Socket object, which has just initiated this set up process. Now the ServerSocket starts processing the new connection request and while this is happening the socket, using a class called MobNetGrabber (described in Section 4.1), gains reference to the nodes MobNet. A Buffer is then created in order to store the Socket's incoming packets and then the Socket is added into the ArrayList of Socket's in the MobNet. On the server side, the ServerSocket recieves a PIP and starts to create the reciprocal Socket to the client. There is a difference in the creation of a Socket on the server side. A Socket created by a server does not search for a ServerSocket at the required destination, it just creates the Socket and waits to be used. Once the server has set up the Socket to the client the two can now synchronise. The server has a PIP which contains a reference to the clients Socket. The boolean isReady is now set to true, and the clients Socket can now begin communications. Figure?? gives a brief

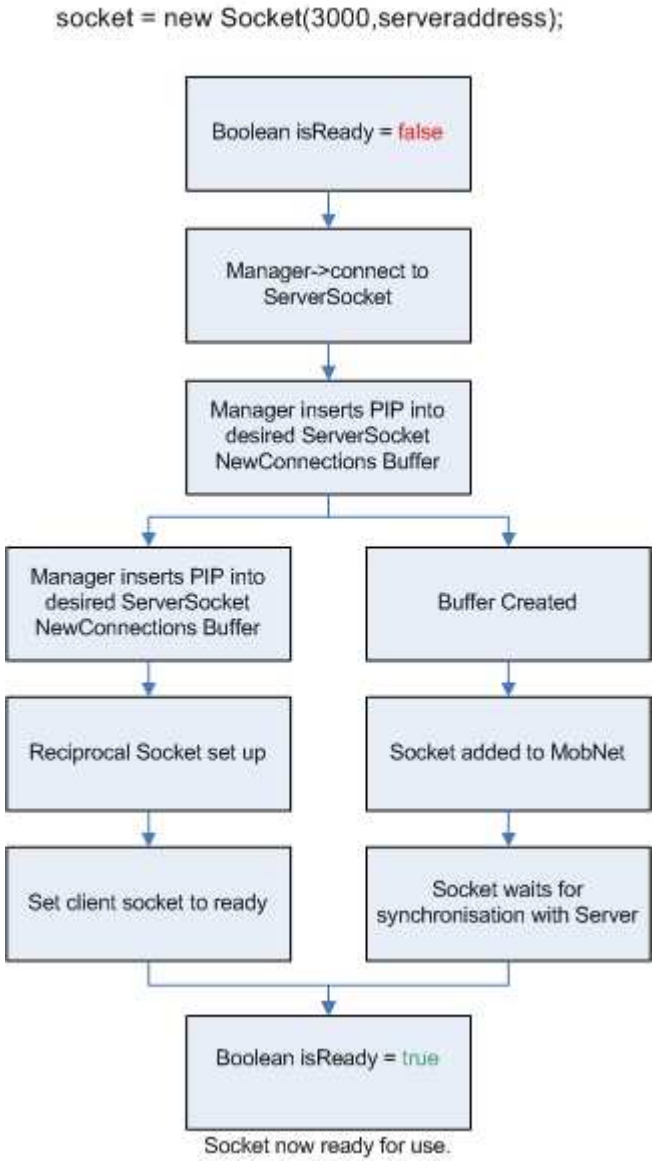


Figure 6: Brief overview of Socket set-up

overview of the process described above.

4.4 MobNet.OutputStream

A Socket contains an implementation of an OutputStream and InputStream in order to let the developer read and write to the Socket. The OutputStream effectively wraps data to be sent in a Packet addressed to the destination of the Socket. The Packet is then sent to the Virtual Network via the MobNet. The class MobNet has a method which allows nodes to deposit Packets into the Virtual Network. MobNet.OutputStream mirrors the java.io version in that it has three ways of writing bytes to the output, 1 byte at a time, a byte array at a time or a part of a byte array at a time.

4.5 MobNet.InputStream

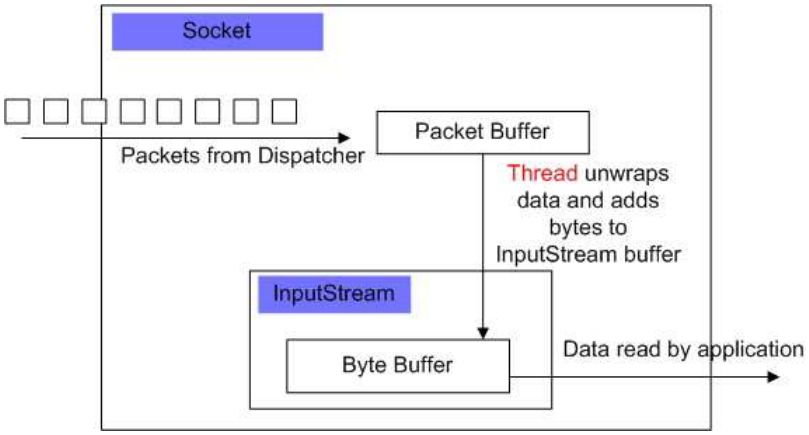


Figure 7: Overview of how packets reach an InputStream

The Dispatcher sends Packets to a buffer in the desired Socket, via the Socket lookup mechanisms described in Section 6. Inside the Socket a thread monitors the buffer and unwraps the data in any Packets received and passes the data into the byte buffer of an InputStream, shown in Figure7.

4.6 Mobnet.DatagramSocket

The DatagramSocket was a lot easier to implement than the TCP/IP based Socket. This is because DatagramSocket's do not rely on synchronisation with the reciever and so implementation is relatively straight forward. When a new DatagramSocket is created a buffer is created to store incoming packets, it is bound to a port and a reference to it is kept in the corresponding MobNet to which it belongs to. DatagramPackets are then created by the

client and sent via the DatagramSocket using the mechanism of depositing them into the Virtual Network. We have created our own implementation of DatagramPackets mainly to overcome the fact that as yet our API only supports String IP addresses. When being sent to the Virtual Network DatagramPackets are encapsulated in our API's Packet representation for compatibility with the Dispatcher (Section 4.7. Incoming packets are received using the recieve() command which blocks if no DatagramPackets are available, with a timeout made possible through the use of the SocketTimer (Section **??**.

4.7 The Dispatcher

The Dispatcher is part of the core of the simulation environment. It is responsible for the distribution of packets to their destinations and is therefore used very heavily. It also has the task of filtering packets which are sent between reachable and unreachable hosts. A very simple and efficient design is therefore used to implement this. The Virtual Network is in effect a buffer containing packets. The Dispatcher takes a Packet from the Virtual Network, looks up the source and destination IP Address. Via the Node Manager it checks to see if the two nodes are in range. If they are in range it finds the desired MobNet for the destination IP Address via the Simulation Manager. The required socket for the port number specified in the Packet is then found and the Packet is the deposited into the Socket's Packet buffer. Packets involving nodes which are not in range are dropped.

4.8 The Node Manager

Along side the Dispatcher, the Node Manager is one of the most fundamental parts of the simulation environment. The Node Manager provides a common interface to configure, store and change characteristics of each node. The Node Manager stores a representation of each node in a Class called VisualNode. The NodeManager contains the method to create VisualNode entities which is used by the Simulation Manager and will be described later. It is implemented using a Singleton Pattern in order to provide each component with a consistent centralized access method and to ensure no duality in representations.

VisualNodes contains the following information:

- IP Address - A String representation of this nodes IP Address.
- X - The current X co-ordinate position of the node in the simulation field in integer form.
- Y - The current Y co-ordinate position of the node in the simulation field in integer form

- Colour - The colour this node is represented by in the Visualiser.
- ToX - The X co-ordinate to which this node is heading
- ToY - The Y co-ordinate to which this node is heading.
- StepX - The incremental increase in X co-ordinate for one frame of animation in the Visualiser.
- StepY - The incremental increase in Y co-ordinate for one frame of the animation in the Visualiser.

Internally the co-ordinates are kept as doubles, but when co-ordinates are requested by other components they can only access the integer representations because the added accuracy is not needed by these other components.

The Node Manager contains methods which are used to modify the position of Nodes, used during simulation set up, as well as a method to enquire if two nodes are in range (used by the Dispatcher). There are also 2 methods which are used to modify the positions of nodes during a simulation. A move nodes method is used to move the position to which each node is heading to which is used in conjunction with method called nodestep which moves the they amount they should move in a frame of the Visualiser.

4.9 The Visualiser

One of the most defining features of this API is the Visualiser component. The component allows a developer to quickly see the state of the simulation field and can help analyse application performance. Built on top of the Node Manager, the Visualiser provides the heart beat of the simulation. The Visualiser initiates the movement of nodes via the Node Manager and then displays a 2D representation on screen resulting in the movement being fully synchronised with the Visualiser.

All nodes are held static until the simulation starts. When the simulation is started a thread runs through and moves the nodes in the Node Manager and the on-screen representations. At present the frame rate is fixed to 6.25 frames per second. Therefore this also represents the granularity of the movements each node makes in the Node Manager. We had to choose a compromise on the frame rate and overall efficiency. The greater the frame rate the greater the overhead in moving the nodes and re-adjusting the co-ordinates in the Node Manager. However the lower the frame rate the less realistic the simulation of the movement of nodes. By trial and error analysis it was decided that 6.25 frames per second represented an ideal point between granularity of node movement, smoothness of animation and program efficiency. In future versions we hope to enable the user to be able to fine tune this parameter.

As stated previously the Visualiser runs on top of the Node Manager and interacts with it via a very simple interface. The NodeManger has two methods movenodes and nodestep, described in Section 4.8. The Visualiser calls movenodes every 50 frames allowing the nodes to change position. On an inter-frame basis the Visualiser calls nodestep in order to increment the positions of each node by 1 frame.

The NodeManager and VisualNodes export interfaces in order to make the Visualiser an easy to replace Component. The Visualiser itself extends from JPanel allowing the Visualiser to be upgraded/extended with great ease. In order to change the Visualiser all that needs to be done is to extend a JPanel and then work with the defined interfaces for the NodeManager and VisualNodes. The NodeManager is used to move the co-ordinates of the nodes, while the VisualNode's are used to extract the exact co-ordinates in a time frame in order to render the nodes to screen.

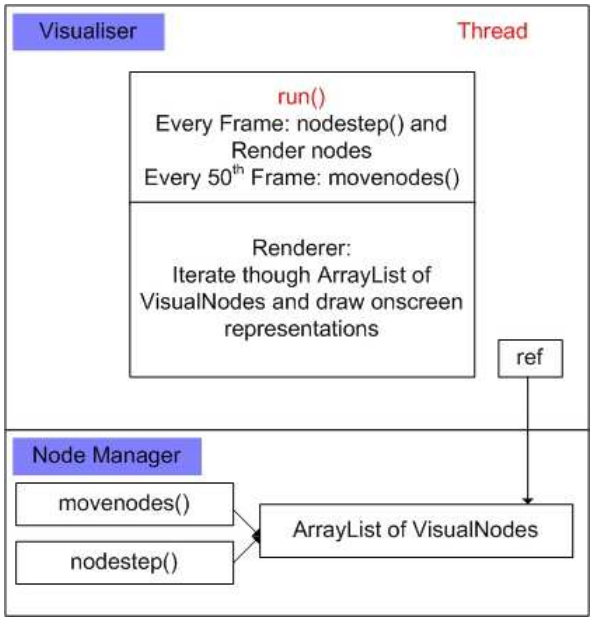
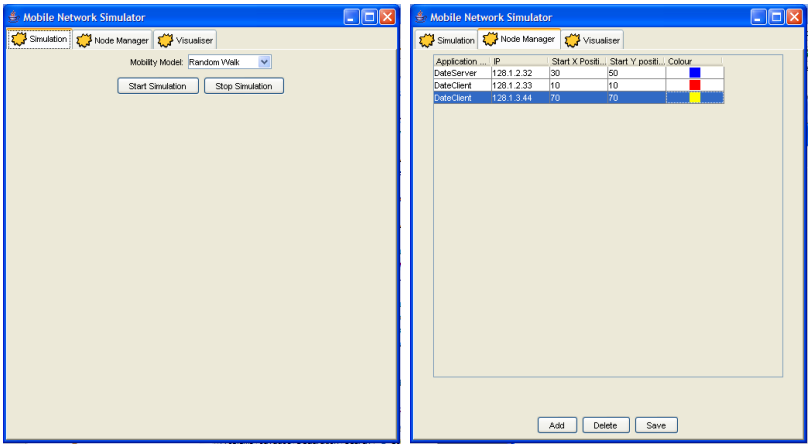


Figure 8: Overview of Visualiser interaction with NodeManager

4.10 The Simulator Manager

The Simulator Manager is implemented in a Singleton Class called SingletonManager. This provides a centralised repository for nodes to store their MobNet's and centralised access for the Dispatcher to get access to MobNet's. MobNet's are stored in an ArrayList and can be retrieved by IP address. The manager also has the task of initialising and starting the Dispatcher.

4.11 The Simulator GUI



(a) Simulator Set-up (b) Node Manager

(c) Visualiser

Figure 9: The Simulator GUI

A GUI is included in order to configure and boot-strap the simulation. The first screen, Figure 9(a), is fire's off basic start-up mechanism for the simulation. The user select's the mobility model required for the simulation and then hits Start Simulation. This will cause all the applications registered in the Node Manager screen, Figure 9(b), to be launched.

The Node Manager is used to configure the nodes required in the simulation. It allows the user to fix the start position, IP Address and application Class to be simulated for each node. These details are stored and edited via a class called NodeTableModel. A seperate data model is required by the JTable component in order to render the table's contents. The NodeTable-

Model flushes all changes to a Node's details to the NodeManager component to ensure both sets of data are entirely consistent. The Node Manager pane automatically assigns colours to a node via a separate class called a ColorManager. The Visualiser requires colors to be of the type java.lang.color, however the Node Manager JTable can only display java.awt.ImageIcon objects. Therefore each colour is paired with a corresponding image and stored as a ColorIcon. The ColorManager sequentially cycles through and outputs a different colour from a set of 13 when a Color request is made by the Node Manager. The colour is then assigned to a node.

Once all information is input into the Node Manager the simulator is ready to go. When Start Simulation is pressed the each entry in the NodeTableModel is iterated in order to launch the applications. A MobNet for each application node with the corresponding IP Address is also created. The simulation environment is now ready.

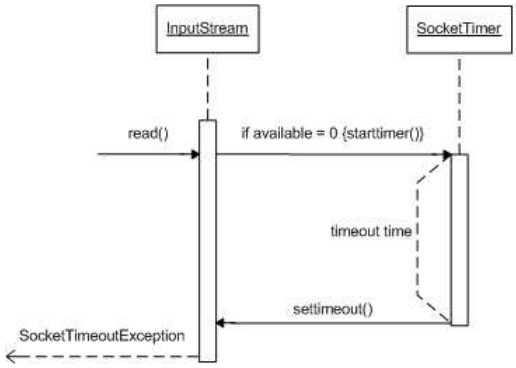
The Visualiser, shown in Figure 9(c), is explained in Section 4.9.

4.12 The SocketTimer

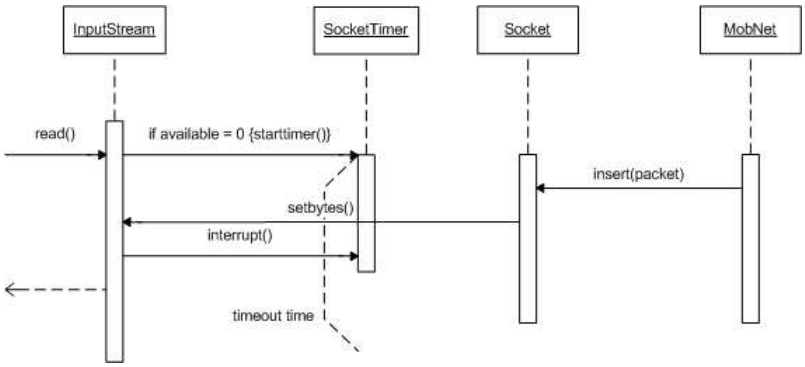
One of the most trickiest things to implement was a timer which would throw and exception after a given amount of time for use in Socket timeouts. Class InputStream needs to be able to throw a SocketTimeoutException after a configurable amount of time. Given this basic requirement you would think that it would be simple to implement. Java provides Timer and TimerTask classes to encapsulate something you want to occur after a given time period. TimerTask is like the Thread class and includes a run() method in which you put the code you want executed. However the run() method cannot throw an exception and propagate it up to the Timer making this mechanism unsuitable for our purpose. The solution was to create a new class called SocketTimer. A SocketTimer is a Thread which sleeps for the given timeout period. On awakening it sets a flag in the InputStream making the InputStream throw a SocketTimeoutException. However if the Socket receives data during the time that the Timer is sleeping, the Timer is interrupted and aborted. This is summarised in the sequence diagrams in Figure 10

4.13 Handling Graphical User Interfaces

Unfortunately when trying to test out graphical applications we found that our API could not support applications in which Sockets are created as a response to GUI Events. The reason for this is because of the way Events are handled in Java and how our MobNet Thread Group mechanism works. Our MobNet Thread Group traverses the current thread hierarchy to find the MobNet Thread Group so that it can then gain access to the MobNet object for the application that is calling the method in the MobNet API.



(a) Socket with timeout occuring



(b) Socket with timeout avoided

Figure 10: SocketTimer sequence diagrams

However, in Java all GUI events are handled by the Event Manager which runs as a separate thread in the Java Virtual Machine. This means that the encapsulation we previously relied on to separate calls to the API is lost therefore we can no longer use the Thread Group mechanism to find out which application made the call.

5 Example Application

Contained in the appendix is code for a sample Date Server application. This is included to give a taste of how easy it is to convert a normal network application for use in our simlulation. The only change to be made is changing the import statements from java.net.* to mobnet.*.

6 Evaluation

Our system relies heavily on the continuous interaction of components and multi-threading. Figure 11 shows how thread creation scales as more clients are added to the system. By inspection we can see that 9 threads are created per Client being simulated. Threads are lightweight and do not impose too much of a performance overhead in creation, however it is important that they are managed properly in order to avoid starvation and bottlenecks being created by suffocation. The MobNet API is thread safe and in testing, deadlock has not been achieved so far. Even so at a minimum of 9 extra thread per client created this is on the high end of the Thread scale. However we believe that improvements in desktop processors will allow this to be a reasonable overhead and will not impose to big a performance penalty.

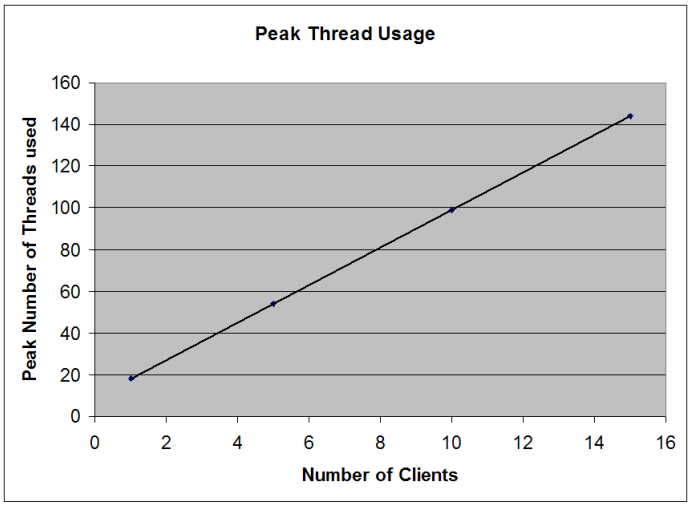


Figure 11: Thread usage scaling per client

Intel's Hyper-Threading Technology enables multi-threaded software applications to execute thread in parallel. Hyper-threading allows processor level threading to be utilised which offers more efficient use of processor resources. It also provides greater parallelism and improved performance.[1][6]

Dual-Core Technology is the next big development in processor architecture. A dual-core processor is a single physical package that contains two microprocessors. The microprocessors share the same packaging and the same bus interface into the chipset and memory. However, they operate as distinct central processing units (CPUs), with the exception that they may share the higher level cache. Therefore, dual-core processors are the next logical step for Hyper-Threading Technology. Again this should allow

[6] `http://www.intel.com/technology/hyperthread/`

multiple multi-threaded applications to run smoothly together.[1][7]

The Visualiser is one of the most impressive features of our work, but it also imposes a performance penalty. The penalty comes from the use of animation and rendering techniques but we hope to streamline the animation process in the future to reduce the performance overhead it imposes. Part of the reason there is a performance overhead is because we have created a dynamically re-sizeable Visualiser which can automatically alter all font and shape metrics. Therefore calculations take places when rendering each frame which could be reduced if we resorted to a fixed size for the Visualiser. A surprise was how well memory requirements scale for the Visualiser.

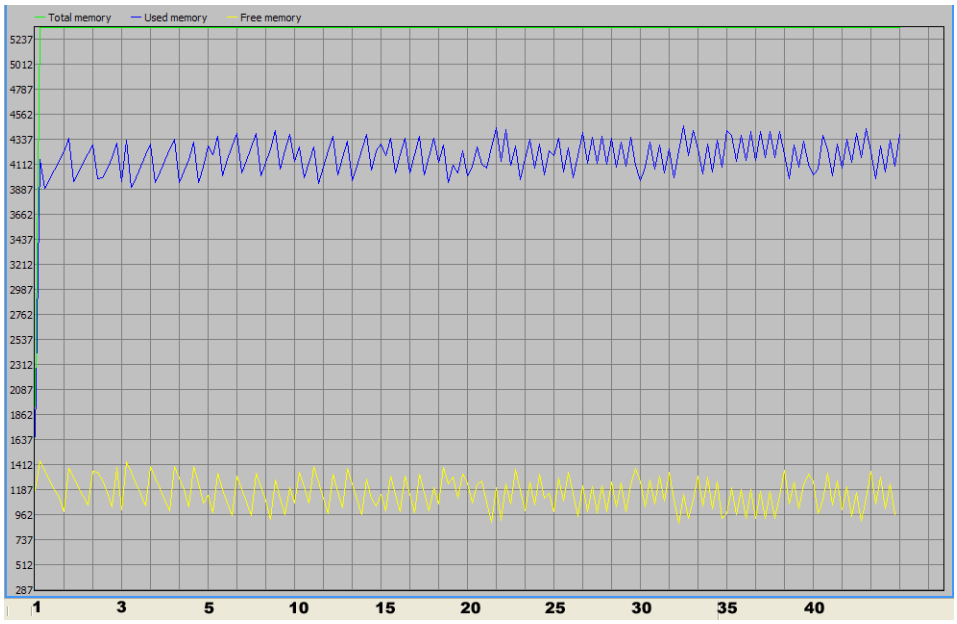


Figure 12: Java VM Memory usage with increasing number of visualised nodes

Figure 12 shows how the Java VM handles the increasing load of nodes with some ease. As the number of nodes reaches 45 the increase in memory usage is very small and almost negligible.

We have analysed the CPU share each package uses in the java implementation. The DateServer and DateClient applications were used to simulate a number of nodes and at the end of the simulation the CPU load data was gathered. This allows us to gain a sense of which components are used most and points us in direction in which to aim efficiency improvements. The Visualiser was not used in these tests in order to avoid the foreseen skew that it would impose on the results.

[7] `http://cache-www.intel.com/cd/00/00/20/57/205705_205705.pdf`

We found that as more clients are simulated, the gui package has a lesser share of CPU load. This is because without the Visualiser the gui package is only used to set up and start the simulation therefore as you would expect it becomes it is less prominent when more clients are simulated.

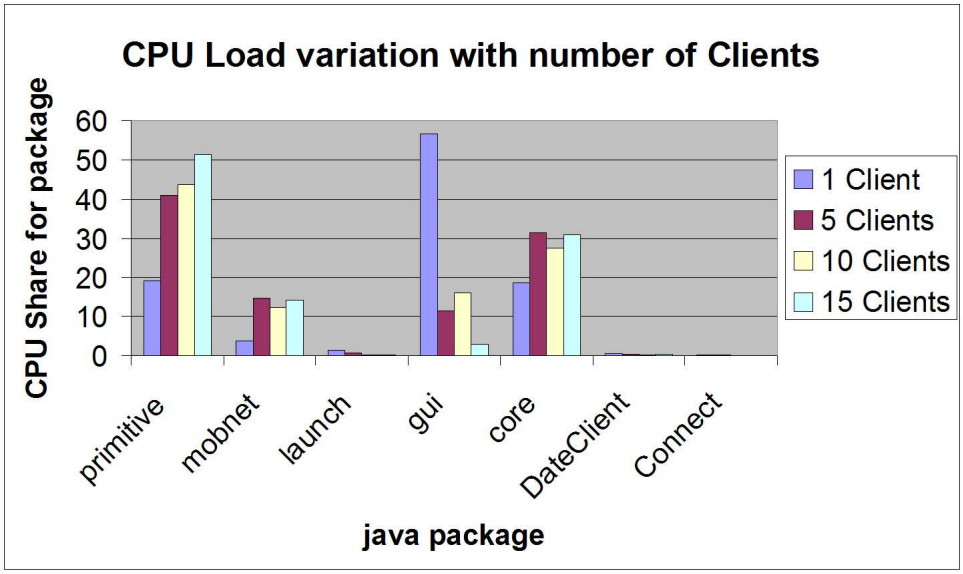


Figure 13: CPU Share per package

We found that the use of MobNet and core packages were linked and generally stabilised as the number of nodes being simulated increased. The exact share these two packages have on CPU load is depends on the type of applications being simulated.

The primitive package as expected increased in CPU share as the number of clients increased. This is because the primitive package contains types which are heavily used, the Packet and Buffer classes. As the number of nodes increases these two classes become the most prominent to be used as they are the basic building blocks of the simulation API. Therefore we should concentrate efforts aim to streamline the API by increasing the efficiency in the creation and use of such objects

7 Conclusions

This paper has described the MobNet Simulation API, a tool to aid development of mobile Java applications. The API features a GUI tool to set up the simulation environment, a Visualiser tool and an API that shadows java.net. The API at present is not complete and does not fully replicate the java.net API (due to time constraints) therefore we would hope that in the future we could fully replicate the java.net API. At present our API only accepts IP addresses and however it should be possible in the future to integrate a DNS component to convert host names to IP addresses.

The simulation environment only provides basic simulation primitives and we would like to include finer grained simulation environments which could include different technologies such as different ranges, for example for Bluetooth and 802.11. This would then allow developers to try out different connectivity profiles and see how tolerant their applications are to different network conditions. We could include multi-hop routing and then different routing protocols into the simulator. At present their is uniform movement of all nodes in the simulation environment. It is possible to make nodes move at different paces and with different mobility models.

Discussed in Section 1.1 was the fact that the battery life of mobile devices is an issue in the development of applications. Therefore it would beneficial for developers if our applications had some kind of battery life emulation which could be used to optimise applications.

The possibilities of extensions are endless but the complexity rises with each level of added detail. However, future research should be conducted in order to fully exploit the use of multiple processors /cores to make the simulation environment more efficient while being detailed.

At present the bottleneck on performance is the Visualiser component which uses a lot of CPU time rendering. A more efficient implementation could be developed which would make use of Swing Timers and a fixed size of rendering, as discussed in Section 6.

While the GUI allows users to set the various properties of the simulation, at present, it does not allow users to save their preferences and the applications involved in the scenario. This feature would allow scenario's to be saved, possibly to XML, saving the developer the time associated with setting up the simulation.

We have achieved almost full transparency for the application developer in order to use his/her application on the simulation platform. In order to achieve full transparency we could either implement a class re-writer which would overwrite the java.net.* import statement with the mobnet.* import. This is still rather crude and would require the source code. The ideal solution and something which could be worked on in the future would be to dynamically substitute calls for the java.net API for calls to our MobNet API.

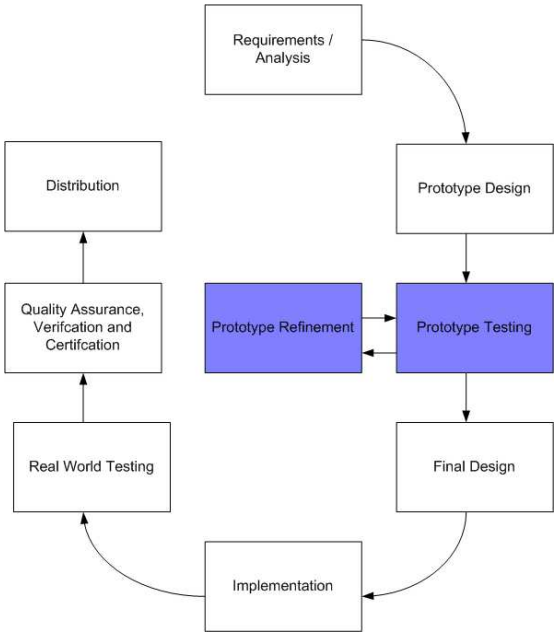


Figure 14: The Mobile Development Life-Cycle, blue highlight shows the phases where the MobNet API is beneficial

A feature which we would like to add is similar to the Statistics layer in FRANC (give ref to paper). It would provide feedback to the developer and a trace of the the actual simulation. A log of node movements, packets being sent, packet losses, packets recieved, etc could be used in order aid the developers comprehension of the simulation.

Being able to handle any GUI's is a goal which at present seems infeasible because of the design of the Java Virtual Machine. One option is to redesign it, however this would affect the level of transparency we are trying to achieve with our API.

While Java is the most popular mobile platform, I believe that Symbian OS and Microsoft's .NET Compact Framework will become increasingly important in the arena and so a port to these platforms would be highly recommended.

The work we have presented here is a first for the field and should provide an interesting base from which to direct future research. A key driver for the future development of this or any other similar simulation tool is the benefits that such tools can provide to developers. Our tool offers developers increased flexibility and help in the evaluation prototypes which saves time and cost. Figure 14 shows where the MobNet API tool fits into the mobile application development cycle. While development cost and time can be reduced through such tools, research should and will continue in this area.

References

[1] Jeff Andrews. Preparing for hyper-threading technology and dual-core technology. Technical report, Intel, 2004.

[2] Rimon Barr, Zygmunt J. Haas, and Robbert van Renesse. Jist: An efficient approach to simulation using virtual machines. *SOFTWAREPRACTICE AND EXPERIENCE*, 2004.

[3] Tracy Camp and Jeff Boleng andVanessa Davies. A survey of mobility models for ad hocnetwork research. Technical report, Dept. of Math.and Computer Sciences Colorado School of Mines, Golden, CO, 2002.

[4] Juan Flynn, Hitesh Tewari, and Donal OMahony. Jemu: A real. time emulation system for mobile ad. hoc networks. *n/a*, 2002.

[5] Intel Research. Intelmobileapplication architecture guide. Technical report, Intel, 2004.

A Example Code

Program 2 A Date Server

```
import java.io.*; import mobnet.*; import java.util.*;

public class DateServer extends Thread {

   private ServerSocket dateServer;

   public static void main(String argv[]) throws Exception {
     new DateServer();
   }

   public DateServer() throws Exception {
    ThreadGroup tg = Thread.currentThread().getThreadGroup();
    String name=tg.getName();
    tg.list();
    print("found group with name="+name);
     dateServer = new ServerSocket(3000);
     System.out.println("Server listening on port 3000.");
     this.start();
   }
   private void print(String s)
   {
    System.out.println("DateServer: "+s);
   }
   public void run() {
     while(true) {
       try {
        System.out.println("Waiting for connections.");
        Socket client = dateServer.accept();
        print("Accepted a connection");
        Connect c = new Connect(client);
       } catch(Exception e) {}
     }
   }
}
```

Program 3 Connect Class

```
import mobnet.*; import java.io.*; import java.util.*;

class Connect extends Thread {
   private Socket client = null;
   private ObjectInputStream ois = null;
   private ObjectOutputStream oos = null;

   public Connect() {}

   public Connect(Socket clientSocket) {
     client = clientSocket;
     try {
      ois = new ObjectInputStream(client.getInputStream());
      oos = new ObjectOutputStream(client.getOutputStream());
     } catch(Exception e1) {
         try {
            client.close();
         }catch(Exception e) {
           System.out.println(e.getMessage());
         }
         return;
     }
     this.start();
   }

   public void run() {
      try {
         oos.writeObject(new Date());
         oos.flush();
         // close streams and connections
         ois.close();
         oos.close();
         client.close();
      } catch(Exception e) {}
   }
}
```

Program 4 The DataClient Class

```
public class DateClient {
   public static void main(String argv[]) {
      String serveraddress;
      ObjectOutputStream oos = null;
      ObjectInputStream ois = null;
      Socket socket = null;
      Date date = null;
      try {
        // open a socket connection
        // find server IP

            serveraddress = "1";
        socket = new MobSocket(3000,serveraddress);
        // open I/O streams for objects
        oos = new ObjectOutputStream(socket.getOutputStream());
        ois = new ObjectInputStream(socket.getInputStream());
        // read an object from the server
        date = (Date) ois.readObject();
        System.out.print("The date is: " + date);
        oos.close();
        ois.close();
      } catch(Exception e) {
        System.out.println(e.getMessage());
      }
   }
}
```

B Java Verified Criteria

The following is an excerpt from the Java Verified Program application testing program, specifically the document: Unified Test Criteria for Java Technology-based Applications for Mobile Devices.

4.6 Network

Test Identifier	NT2 **(R)**
Summary	If the application is network enabled, appropriate error messages must be displayed when the application attempts to send / receive data when data services are not available. The application should respond gracefully to the loss of network connectivity.

Test Identifier	NT3 **(R)**
Summary	The application should be able to handle delays. For instance when making a connection it may need to wait for permission from the user

Test Identifier	NT4 **(R)**
Summary	The application must be able to handle situations where connection is not allowed.

Test Identifier	NT5 **(R)**
Summary	The application must be able to close the connection which it's using after the session is over.

Figure 15: Criteria for Network part of Application