WARDROP ROUTING IN WIRELESS SENSOR NETWORKS

Robin Rishi (101212)

Project Guide –
Dr.Nitin

Submitted in partial fulfilment of the Degree of
Bachelor of Technology

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT

Table of Content

Page

Chapter – 1

Wireless Sensor Networks

1.1 Introduction ……………………………………………………………………1

1.2 Individual Wireless Sensor Node Architecture …..2

1.3 Wireless Sensor Networks Architecture………………………………………...3

1.3.1 Star Network (Single Point-to-Multipoint)……………………………..……..3

1.3.2 Mesh Network…………………………………………………………………4

1.3.3 Hybrid Star – Mesh Network……………………………………..…..........5

1.4 Challenges……………………………………………………………….….......6

1.6 Applications of Wireless Sensor Networks………………………….………....8

1.7 Future Developments 11

Chapter - 2

Literature Survey

2.1 Introduction ………………………………………………………….…....13

2.2 Routing Algorithm……………………………………………….………..…..17

2.3 Connection to wardrop equilibrium……………………… …………...….....18

2.4 Why wardrop routing can automatically lead to flow avoiding routes………....18

2.5 The STARA Algorithm……………………………………………………….20

2.6 The Challenges…………………………………………..…………………21

2.7 Controlling Paths and flow avoiding Routes…………………………………….22

2.8 Controlling paths with M-STARA………………………………...……….22

2.9 Eliminating and controlling with P-STARA…………………………………….24

FUTURE WORK……………………………………………………

APPENDIX 28

REFERENCES 65

Certificate

This is to certify that the work titled **— “WAR DROP ROUTING IN WIRELESS SENSOR NETWORKS**”, submitted by Robin Rishi, partial fulfillment for the award of degree of Bachelor of Technology in Computer Science Engineering to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Signature of Supervisor ……………………..

Name of Supervisor Dr. Nitin

Designation Associate Professor

Date ……………………..

Acknowledgement

As we conclude our project with the God's grace, we have many people to thank; for all the help, guidance and support they lent us, throughout the course of our endeavor.

First and foremost, we are sincerely thankful to Dr.Nitin, our Project Guide, who has always encouraged us to put in our best efforts and deliver a quality and professional output. His methodology of making the system strong from inside has taught us that output is not the END of project. We really thank him for his time & efforts.

We are deeply indebted to all those who provided reviews and suggestions for improving the materials and topics covered in our project, and we extend our apologies to anyone we may have failed to mention.

Robin Rishi

Enrollment no. 101212

CSE

Signature ……………………..

ABSTRACT

Routing protocols for multihop wireless networks have traditionally used shortest path routing to obtain paths to destinations and do not consider traffic load or delay as an explicit factor in the choice of routes. We focus on static mesh networks and formally establish that if the number of sources is not too large, then it is possible to construct a perfect flow-avoiding routing, which can
boost the throughput provided to each user over that of the shortest path routing by a factor of four when carrier sensing can be disabled or a factor of 3.2 otherwise. So motivated, we address the issue of designing a multipath, load adaptive routing protocol that is generally applicable even when there are more sources. We develop a protocol that adaptively equalizes the mean delay along all utilized routes from a source to destination and does not utilize any routes that have greater mean delay. This is the property satisfied by a system in Wardrop equilibrium. We also address the architectural challenges confronted in the software implementation of a
multipath, delay-feedback-based, probabilistic routing algorithm. Our routing protocol is 1) completely distributed, 2) automatically load balances flows, 3) uses multiple paths whenever beneficial, 4) guarantees loop-free paths at every time instant even while the algorithm is suntil converging, and 5) amenable to clean implementation. An ns-2 simulation study indicates that the protocol is able to automatically route flows to “avoid” each other, consistently out performing shortest path protocols in a variety of scenarios. The protocol has been implemented in user space with a small amount of forwarding mechanism in a modified Linux 2.4.20 kernel. Finally, we discuss a proof-of-concept measurement study of the implementation on a six node testbed

CHAPTER 1

WIRELESS SENSOR NETWORKS

1.1 INTRODUCTION

Sensors integrated into structures, machinery, and the environment, coupled with the efficient delivery of sensed information, could provide tremendous benefits to society. Potential benefits include: fewer catastrophic failures, conservation of natural resources, improved manufacturing productivity, improved emergency response, and enhanced homeland security. However, barriers to the widespread use of sensors in structures and machines remain. Bundles of lead wires and fiber optic “tails” are subject to breakage and connector failures. Long wire bundles represent a significant installation and long term maintenance cost, limiting the number of sensors that may be deployed, and therefore reducing the overall quality of the data reported. Wireless sensing networks can eliminate these costs, easing installation and eliminating connectors.

The ideal wireless sensor is networked and scaleable, consumes very little power, is smart and software programmable, capable of fast data acquisition, reliable and

accurate over the long term, costs little to purchase and install, and requires no real maintenance.

Selecting the optimum sensors and wireless communications link requires knowledge of the application and problem definition. Battery life, sensor update rates, and size are all major design considerations. Examples of low data rate sensors include temperature, humidity, and peak strain captured passively. Examples of high data rate sensors include strain, acceleration, and vibration.

Recent advances have resulted in the ability to integrate sensors, radio communications, and digital electronics into a single integrated circuit (IC) package. This capability is enabling networks of very low cost sensors that are able to communicate with each other using low power wireless data routing protocols. A wireless sensor network (WSN) generally consists of a base station (or "gateway") that can communicate with a number of wireless sensors via a radio link. Data is collected at the wireless sensor node, compressed, and transmitted to the gateway directly or, if required, uses other wireless sensor nodes to forward data to the gateway. The transmitted data is then presented to the system by the gateway connection. The purpose of this chapter is to provide a brief technical introduction to wireless sensor networks and present a few applications in which wireless sensor networks are enabling.

1.2 Individual Wireless Sensor Node Architecture

A functional block diagram of a versatile wireless sensing node is provided in Figure 1.1. A modular design approach provides a flexible and versatile platform to address the needs of a wide variety of applications. For example, depending on the sensors to be

deployed, the signal-conditioning block can be re-programmed or replaced. This allows for a wide variety of different sensors to be used with the wireless sensing node. Similarly, the radio link may be swapped out as required for a given applications' wireless range requirement and the need for bidirectional communications. The use of flash memory allows the remote nodes to acquire data on command from a base station, or by an event sensed by one or more inputs to the node. Furthermore, the embedded firmware can be upgraded through the wireless network in the field.

The microprocessor has a number of functions including:

1) Managing data collection from the sensors

2) Performing power management functions

3) Interfacing the sensor data to the physical radio layer

4) Managing the radio network protocol

A key feature of any wireless sensing node is to minimize the power consumed by the system. Generally, the radio subsystem requires the largest amount of power. Therefore, it is advantageous to send data over the radio network only when required. This sensor event-driven data collection model requires an algorithm to be loaded into the node to determine when to send data based on the sensed event. Additionally, it is important to minimize the power consumed by the sensor itself. Therefore, the hardware should be designed to allow the microprocessor to judiciously control power to the radio, sensor, and sensor signal conditioner.

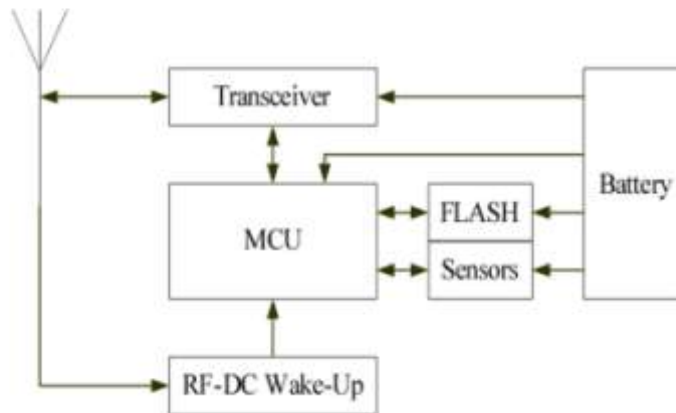

Figure 1.1: Wireless sensor node functional block diagram

1.3 Wireless Sensor Networks Architecture

There are a number of different topologies for radio communications networks. A brief discussion of the network topologies that apply to wireless sensor networks are outlined below.

1.3.1 Star Network (Single Point-to-Multipoint)

A star network (Figure 22.3.1) is a communications topology where a single base station can send and/or receive a message to a number of remote nodes. The remote nodes can only send or receive a message from the single base station, they are not permitted to send messages to each other. The advantage of this type of network for wireless sensor networks is in its simplicity and the ability to keep the remote node's power consumption to a minimum. It also allows for low latency communications between the remote node and the basestation. The disadvantage of such a network is

that the basestation must be within radio transmission range of all the individual nodes and is not as robust as other networks due to its dependency on a single node to manage the network.

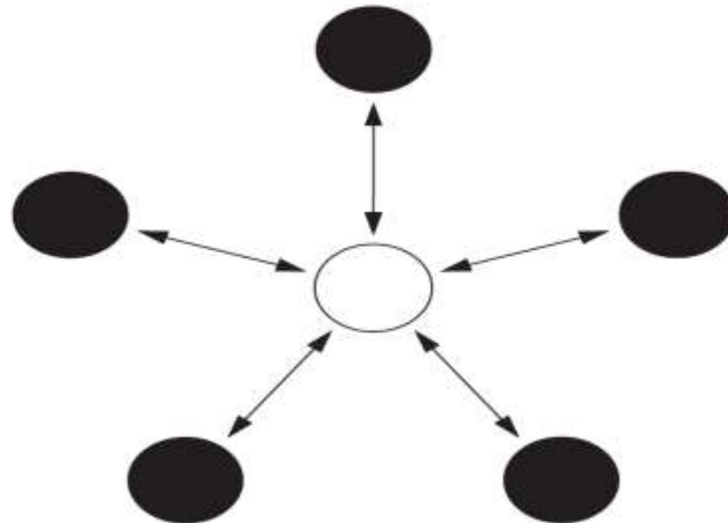

Figure 1.2: Star network topology

1.3.2 Mesh Network

A mesh network allows for any node in the network to transmit to any other node in the network that is within its radio transmission range. This allows for what is known as multihop communications; that is, if a node wants to send a message to another node that is out of radio communications range, it can use an intermediate node to forward the message to the desired node. This network topology has the advantage of redundancy and scalability. If an individual node fails, a remote node still can communicate to any other node in its range, which in turn, can forward the message to the desired location. In addition, the range of the network is not necessarily limited by the range in between single nodes, it can simply be extended by adding more nodes to

the system. The disadvantage of this type of network is in power consumption for the nodes that implement the multihop communications are generally higher than for the nodes that don't have this capability, often limiting the battery life. Additionally, as the number of communication hops to a destination increases, the time to deliver the message also increases, especially if low power operation of the nodes is a requirement.

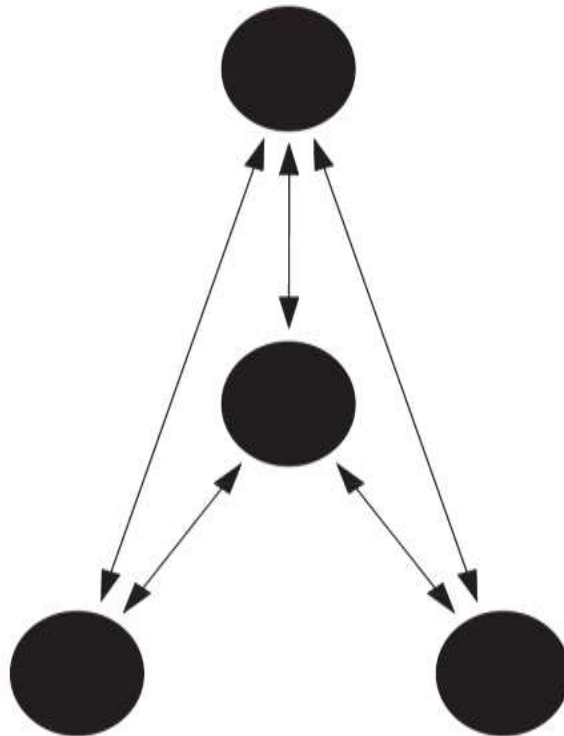

Figure 1.3: Mesh network topology

1.3.3 Hybrid Star – Mesh Network

A hybrid between the star and mesh network provides for a robust and versatile communications network, while maintaining the ability to keep the wireless sensor nodes power consumption to a minimum. In this network topology, the lowest power sensor nodes are not enabled with the ability to forward messages. This allows for minimal power consumption to be maintained. However, other nodes on the network

are enabled with multihop capability, allowing them to forward messages from the low power nodes to other nodes on the network. Generally, the nodes with the multihop capability are higher power, and if possible, are often plugged into the electrical mains line. This is the topology implemented by the up and coming mesh networking standard known as ZigBee.

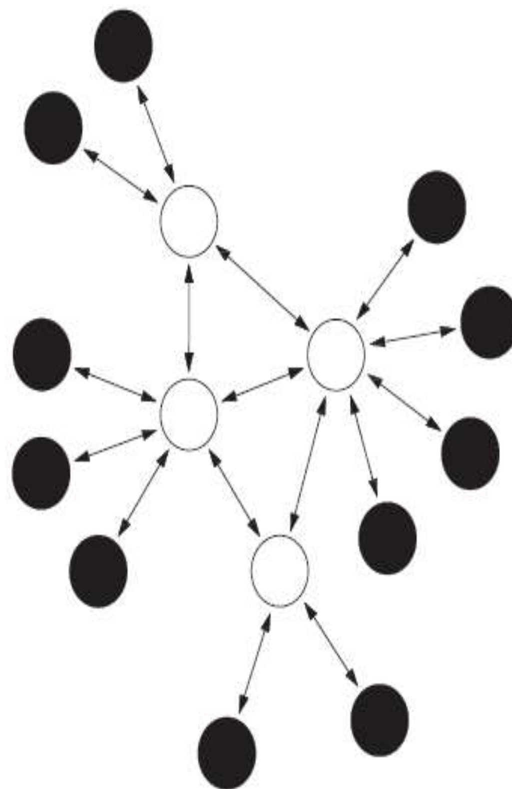

Figure 1.4: Hybrid star-mesh network topology

1.4 Challenges

In spite of the diverse applications, sensor networks pose a number of unique technical challenges due to the following factors:

>> **Ad hoc deployment:** Most sensor nodes are deployed in regions which have no infrastructure at all. A typical way of deployment in a forest would be tossing the sensor nodes from an aero plane. In such a situation, it is up to the nodes to identify its connectivity and distribution.

>>**Unattended operation:** In most cases, once deployed, sensor networks have no human intervention. Hence the nodes themselves are responsible for reconfiguration in case of any changes.

>>**Untethered:** The sensor nodes are not connected to any energy source. There is only a finite source of energy, which must be optimally used for processing and communication. An interesting fact is that communication dominates processing in energy consumption. Thus, in order to make optimal use of energy, communication should be minimized as much as possible.

>> **Dynamic changes:** It is required that a sensor network system be adaptable to changing connectivity (for e.g., due to addition of more nodes, failure of nodes etc.) as well as changing environmental stimuli. Thus, unlike traditional networks, where the focus is on maximizing channel throughput or minimizing node deployment, the major consideration in a sensor network is to extend the system lifetime as well as the system robustness.

1.6 Applications of Wireless Sensor Networks

Structural Health Monitoring – Smart Structures

Sensors embedded into machines and structures enable condition-based maintenance of these assets. Typically, structures or machines are inspected at regular time intervals, and components may be repaired or replaced based on their hours in service, rather than on their working conditions. This method is expensive if the components are in good working order, and in some cases, scheduled maintenance will not protect the asset if it

was damaged in between the inspection intervals. Wireless sensing will allow assets to be inspected when the sensors indicate that there may be a problem, reducing the cost of maintenance and preventing catastrophic failure in the event that damage is detected. Additionally, the use of wireless reduces the initial deployment costs, as the cost of installing long cable runs is often prohibitive.

In some cases, wireless sensing applications demand the elimination of not only lead wires, but the elimination of batteries as well, due to the inherent nature of the machine, structure, or materials under

test. These applications include sensors mounted on continuously rotating parts , within concrete and composite materials, and within medical implants.

Application Highlight – Civil Structure Monitoring

One of the most recent applications of today's smarter, energy-aware sensor networks is structural health monitoring of large civil structures, such as the Ben Franklin Bridge (Figure 1.7), which spans the Delaware River, linking Philadelphia and Camden, N.J. The bridge carries automobile, train and pedestrian traffic. Bridge officials wanted to monitor the strains on the structure as high-speed commuter trains crossed over the bridge.

Figure 1.7: Ben Franklin Bridge

A star network of ten strain sensors were deployed on the tracks of the commuter rail train. The wireless sensing nodes were packaged in environmentally sealed NEMA rated enclosures. The strain gauges were also suitably sealed from the environment and were spot welded to the surface of the bridge steel support structure. Transmission range of the sensors on this star network was approximately 100 meters.

The sensors operate in a low-power sampling mode where they check for presence of a train by sampling the strain sensors at a low sampling rate of approximately 6 Hz. When a train is present the strain increases on the rail, which is detected by the sensors. Once detected, the system starts sampling at a much higher sample rate. The strain waveform is logged into local Flash memory on the wireless sensor nodes. Periodically, the waveforms are downloaded from the wireless sensors to the base station. The base station has a cell phone attached to it which allows for the collected data to be transferred via the cell network to the engineers' office for data analysis. This low-power event-driven data collection method reduces the power required for continuous operation from 30 mA if the sensors were on all the time to less than 1 mA continuous. This enables a lithium battery to provide more than a year of continuous operation.

Resolution of the collected strain data was typically less then 1 micro strain. A typical waveform downloaded from the node is shown in Figure 1.8. Other performance specifications for these wireless strain sensing nodes have been provided in an earlier work .

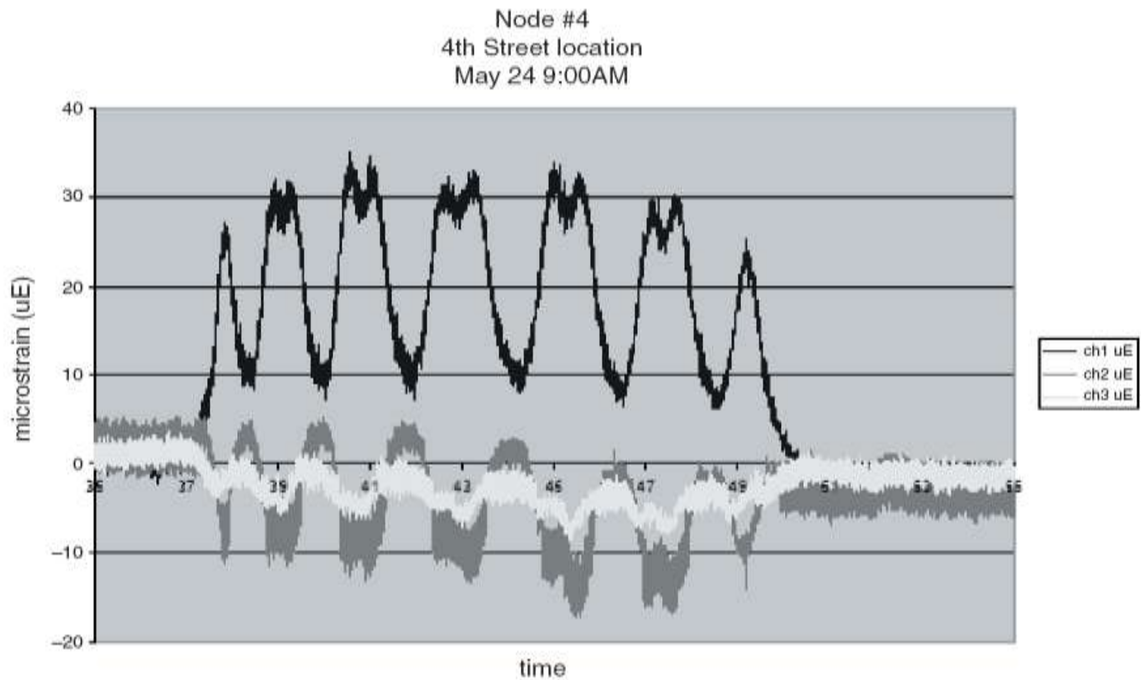

Figure 1.8: Bridge strain data

1.7 Future Developments

The most general and versatile deployments of wireless sensing networks demand that batteries be deployed. Future work is being performed on systems that exploit piezoelectric materials to harvest ambient strain energy for energy storage in capacitors and/or rechargeable batteries. By combining smart, energy saving electronics with advanced thin film battery chemistries that permit infinite recharge cycles, these systems could provide a long term, maintenance free, wireless monitoring solution.

CHAPTER 2

Wardrop Routing in Wireless Networks

Abstract—Routing protocols for multihop wireless networks have traditionally used shortest path routing to obtain paths to destinations and do not consider traffic load or delay as an explicit factor in the choice of routes. We focus on static mesh networks and
formally establish that if the number of sources is not too large, then it is possible to construct a perfect flow-avoiding routing, which can boost the throughput provided to each user over that of the shortest path routing by a factor of four when carrier sensing can be
disabled or a factor of 3.2 otherwise. So motivated, we address the issue of designing a multipath, load adaptive routing protocol that is generally applicable even when there are more sources. We develop a protocol that adaptively equalizes the mean delay along all
utilized routes from a source to destination and does not utilize any routes that have greater mean delay. This is the property satisfied by a system in Wardrop equilibrium. We also address the architectural challenges confronted in the software implementation of a
multipath, delay-feedback-based, probabilistic routing algorithm.
Our routing protocol is
1) completely distributed
2) automatically loadbalances flows,
3) uses multiple paths whenever beneficial,
4) guarantees loop-free paths at every time instant even while the algorithm is suntil converging,
5) amenable to clean implementation.

An ns-2 simulation study indicates that the protocol is able to automatically route flows to “avoid” each other, consistently out-performing shortest path protocols in a

variety of scenarios. The protocol has been implemented in user space with a small amount of forwarding mechanism in a modified Linux 2.4.20 kernel. Finally, we discuss a proof-of-concept measurement study of the implementation on a six node testbed.

Index Terms—Wireless networks, wardrop routing, delay-adaptive multipath routing, performance study, implementation.

INTRODUCTION

Routing research in wireless multihop networks has traditionally focused on shortest path routing protocols.A lot of effort has gone into designing protocols that route packets efficiently in mobile networks with minimal overhead, e.g., [1], [2], [3], [4], and [5].

This paper is addressed toward static or at best networks with slow mobility, for example, mesh networks or ad hoc networks in office environments where users do not move

around with their laptops. Our focus is on wireless networks, e.g., mesh networks, where the focus is on boosting the throughput-delay performance, and energy limitations are not a constraining factor. Looking ahead to the next generation of wireless routing protocols for such quasi-static networks, users may want their routing to be adaptive to the load on the network. For example, users may want to improve their throughput-delay performance

by intelligently routing flows in a manner to avoid interference from other paths as far as possible. Users may also want to utilize multiple paths so that they obtain more throughput. In addition, looking from a network standpoint, one may want the routing algorithms implemented at each node to combine together to balance load across the network. At the same time, one would like to retain some of the key features of current protocols,

including the completely distributed operation, loop-free paths, and ease of implementability.

In this paper, we present a completely distributed, load adaptive, multipath routing protocol for quasi-static wireless mesh networks. For every source-destination (SD) pair, the protocol adaptively equalizes mean delays along all utilized routes and avoids using any paths with greater or equal mean delay. This is the property satisfied by a system in Wardrop equilibrium. Such an equilibrium is potentially useful in practice for a variety of reasons:

1. Adaptive delay-based routing can automatically route around hotspots in interference-constrained wireless networks.

2. Equalizing the average delay along used paths can reduce resequencing delays for packets in receiver socket buffers.

3. TCP congestion control reacts adversely to reordered packets and thus misbehaves when TCP is used over multiple paths. Equalizing the average delay along used routes can reduce packet reordering and potentially improve TCP behavior when it is used on top of multipath routing.

The work in this paper builds on earlier work on design [6] and convergence [7] of delay-adaptive routing algorithms in wireless networks. The goal of this paper is to bridge the gap between the theory of delay-adaptive routing and its implementation and use in practice as a
routing protocol for 802.11-based wireless mesh networks. The issues addressed range from theoretical characterization and algorithmic properties to a detailed simulation study and architectural challenges in implementing multipath delay-adaptive routing protocols.

1.) We prove a new result that formalizes the potential performance improvement for interference avoiding multipath routing in wireless networks. We show that when the number of active SD pairs is appropriately small in comparison with the number of nodes in the network, and the SD pairs are randomly distributed, then it is possible to choose two paths per each SD pair such that no two paths from different SD pairs cross or interfere with each

other and that the two paths for the same SD pair meet only at their end points. The throughput benefit realizable by such path disjoint multipath routing depends on whether carrier sensing (with twice the transmission range) can or cannot be turned off. The throughput that can be furnished to each SD pair is four times what would be furnished
by minimum hop shortest path routing if carrier sensing can be disabled or 3.2 times otherwise.

2.) The algorithm used to construct the flow-avoiding routing only illustrates the feasibility of such an approach and is not amenable to distributed implementation in a wireless network. Further, in practice, we would like our routing protocol to be load adaptive even when there are a large number of flows in the network that necessarily cross each other. So motivated, we design a multipath routing protocol with the following properties:

a. The routing algorithm converges to a set of admissible routes for each SD pair with the
following properties: for each SD pair, the mean delays experienced along the multiple paths are all the same. The potential delay along any unutilized admissible path through the network is greater than or equal to the delay along each of the utilized paths. This is the property satisfied by a system in Wardrop equilibrium.

b. The admissible paths all have no more than twice the number of hops of the minimum hop path.

c. Even during the transient phase while the algorithm is in the process of converging, the
utilized paths are guaranteed to be loop free.

3.) We demonstrate, via a theoretical example, how our delay adaptive routing can help automatically achieve flow avoiding routing whenever possible. We conduct a detailed ns-2 simulation study of the algorithm to study the throughput delivered, the number of hops along the paths, the delays experienced, the rate at which the algorithm converges, and the overhead consumed by the routing algorithm. The simulation study indicates

that Wardrop routing is effective in routing flows to “avoid each other” and minimize interference.

4.) We propose a software architecture for implementing the multipath, delay-feedback-based, Wardrop equilibrating protocol. The architecture respects the IP stack layering. Even though it is adaptive to the end-to-end delay, it does not utilize transport layer mechanisms to implement network layer routing.

5.) We implement the above protocol in a Linux 2.4.20-6 kernel. The routing policy is implemented completely in user space, with a small amount of forwarding mechanism in the kernel itself.

6.) We conduct a proof-of-concept measurement study of the protocol on a six-node testbed.

The rest of this paper is organized as follows: In Section 2, we survey related work. In Section 3, we demonstrate, by theoretical proof and simulation, the regime in which shortest path routing can lead to a loss of throughputperformance in large wireless networks. In Section 4, we introduce the traffic-adaptive routing algorithm, the challenges to be faced in developing a practical protocol and our solutions resulting in the P-STARA protocol. In Section 5, we describe the protocols for distributed delay estimation and link delay measurement. In Section 6, we present the results of a detailed simulation study. In Section 7, we present the implementation architecture, and in Section 8, we carry out a measurement study of the protocol.

THE ROUTING ALGORITHM

The theoretical result in the previous section indicates that traffic-aware routing can provide considerable benefits. However, the scheme used to prove the result is centralized and should only be taken as proof of existence. Instead, we examine a more general adaptive approach. For every SD pair, we will attempt to drive routes toward an equilibrium, where the mean delay along all utilized paths is equalized, and all unutilized paths have greater or equal potential mean delay. In a communication network, such an equilibrium has attractive properties vis-a-vis multipath routing:

1. When packets have to be resequenced at the receiver and delivered in-order to the application, equalizing the average delay along utilized paths reduces receiver socket buffer space requirements and receiver socket buffer resequencing delays.
2. Equalizing the average delay along utilized paths mitigates TCP congestion misbehavior that results from TCP's adverse reaction to multiple paths and reordered packets.
3. Route adaptation using delay feedback allows rerouting of flows around traffic bottlenecks in wireless environments. This allows flows to automatically "avoid" each other and minimize interference.

2.3 Connection to the Wardrop Equilibrium

The above equilibrium can be interpreted as a routing solution for a global optimization problem that minimizes the sum of the integral of delays on all links in the network [24]. In this sense, it is similar to the minimum-delay optimization approach in [18] and [19]. There is another interpretation—as the property of a system in Wardrop equilibrium. Suppose that wireless connectivity is represented by a graph $G = (V, E)$,

and let $SD = \{s, d\}$, where $s, d \in V$, represent the set of SD pairs. Consider the problem of routing packets over such a network where the performance measure we are interested in minimizing is the expected delay. The setting is a noncooperative one in which each packet wishes to minimize the time taken to get from source to destination. The route chosen by each packet affects the latency experienced by other packets along its route, as well as in the vicinity of its route due to wireless interference. Since each packet has an infinitesimally small impact on the load of the network, the solution of this noncooperative game corresponds to the Nash equilibrium when the number of agents goes to infinity. For each $\{s, d\}$ pair in SD, this corresponds to a solution where all flow paths have equal latency, which is lower than the latency experienced on any unutilized path. In the absence of this property, it would be possible for some packet to reduce its latency by switching to the unutilized path [25]. This is the "Wardrop equilibrium," defined in [26]:

1. *Wardrop's first principle.* All utilized paths from a source to a destination have equal mean delays.
2. *Wardrop's second principle.* Any unutilized path from a source to a destination has greater potential mean delay than that along utilized paths.

2.4 Why Wardrop Routing Can Automatically Lead to Flow Avoiding Routes

Before describing our algorithm, we illustrate a simple example to show that routing toward a Wardrop equilibrium can automatically result in flow avoiding routes in wireless networks. Consider the example in Fig. 5, where there are two flows traversing a wireless network. The first flow f_1, from s_1 to d_1 has two available paths: path P_1 of length $l_1 = 4$, and path P_2 of length $l_2 = 2l_1 = 8$. The second flow f_2, from s_2 to d_2, is a one-hop flow that interferes with exactly one link of the path P_1. Suppose the input rates for the two flows are x_1 and $x_2 = wx_1$, respectively. Let us determine the Wardrop equilibrium for this system as a function of the parameter w, which captures the relative magnitude of the interference from the second flow on the first flow. In general, the link delay along a directed edge in a wireless network is a complex function of the topology, traffic on interfering edges, and the routing and scheduling algorithms used in the wireless network. As a first approximation, we assume that the delay along each directed edge is a convex nondecreasing function $f(x + y)$ of the

traffic x on that edge, and the cumulative traffic y on interfering edges. Since this example is merely illustrative, we will use the function $f(x) = x^n$, with $n = 3$.

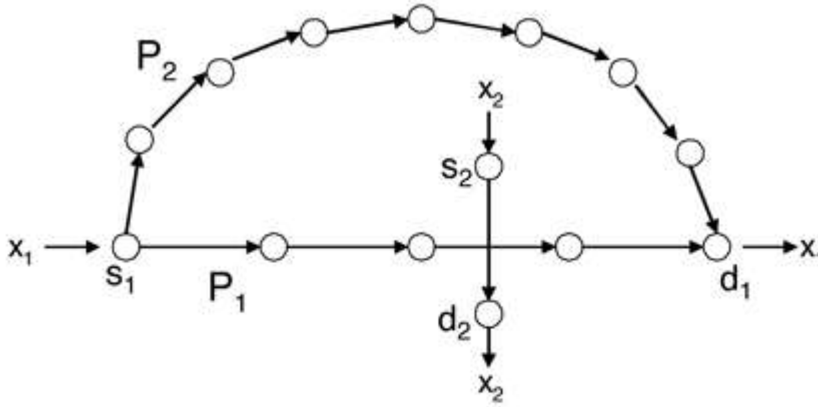

Fig. 5. Flow avoidance with Wardrop routing: a simple two flow example.

Suppose, in Wardrop equilibrium, that the optimal routing for flow f_1 directs q fraction of the traffic along path P_1 and $1-q$ fraction of the traffic along path P_2. The delay along path P_1 is given by $D(1,1) = (l_1 - 1)(qx_1)^n + (qx_1 + x_2)^n = x_1^3(3q^3 + (q+w)^3)$. The delay along path P_2 is given by $D(1,2) = (l_2)((1-q)x_1)^n = 8x_1^3(1-q)^3$. In Fig. 6, we have plotted the values of $\frac{D(1,1)}{x_1^3}$ and $\frac{D(1,2)}{x_1^3}$ as a function of the fraction of traffic q on the first path P_1 for different values of w. For $w < 2$, it is easy to see that the solution where all the traffic is on one of the two paths (i.e., $q = 0$ or $q = 1$) are not Wardrop equilibria, since the delay on the unutilized path in these cases is strictly lower than the delay on the utilized path. Thus, for a given $w < 2$, the Wardrop equilibrium is obtained by solving $D(1,1) = D(1,2)$ for q. On the other hand, when $w \geq 2$, the delay $D(1,1)$ on path P_1 is always greater or equal the delay $D(1,2)$ on path P_2, irrespective of what fraction of traffic q flows along P_1. This implies that for $w \geq 2$, the unique equilibrium consists of routing all traffic along path P_2, and completely avoiding the interference bottleneck along path P_1 caused by the presence of flow f_2. This equilibrium routing is plotted as a function of w in Fig. 7. This example shows the potential of traffic-adaptive routing to automatically lead to flow avoiding routing whenever possible in a wireless network.

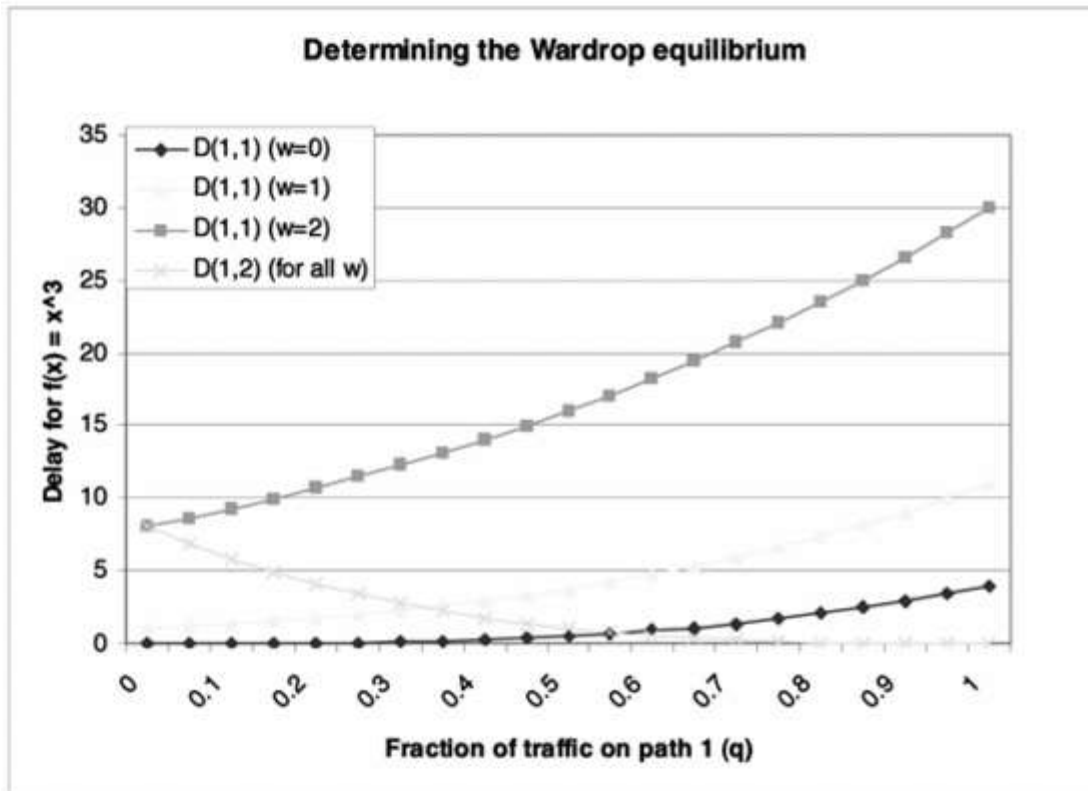

Fig. 6. Determining the Wardrop equilibrium: the equilibrium for a given w is attained at the value of q for which the delays $D_{1,1}$ and $D_{1,2}$ are equalized.

Previous | View All | Next

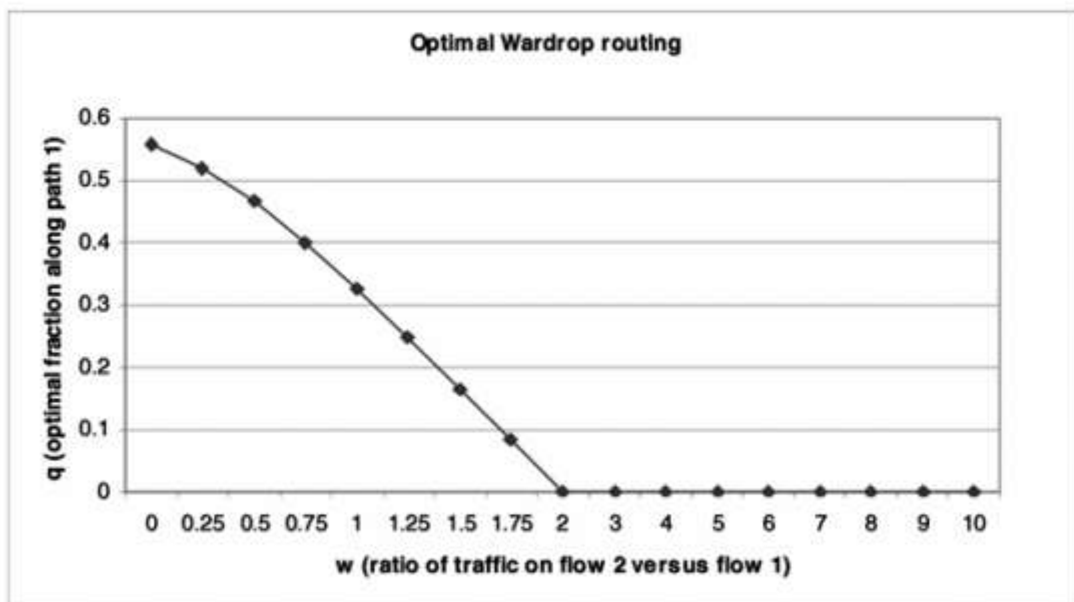

Fig. 7. The optimal routing as a function of w, the ratio of traffic betweens flows f_1 and f_2.

2.5 The STARA Algorithm

Given these attractive features of routing toward a Wardrop equilibrium, our focus in the rest of this paper will be on the design and implementation of a practical distributed algorithm that equalizes the mean delays along all utilized paths in the network and ensures that all unutilized paths have greater or equal mean delay. We start off by describing earlier work on the design [6] and convergence [7] of delay-adaptive routing algorithms in wireless networks. The problem addressed there was to construct an algorithm for reducing or increasing the flows along paths that would equilibrate to a Wardrop equilibrium. The basic idea is to estimate end-to-end delay along various paths and adaptively shift traffic from higher delay paths to lower delay paths, equilibrating only when all utilized paths have the same average delay. For each node i and destination d, let $N(i,d)$ denote the neighbours of i used to forward packets to destination d. Under the STARA algorithm, when node i receives a packet destined for d, it probabilistically sends it out to a neighbour in $N(i,d)$. Let p_{id}^{j} be the probability that it is forwarded to node $j \in N(i,d)$. The routing probabilities $[p_{id}^{j} : j \in N(i,d)]$ are adjusted based on delay feedback.

Let D_{id}^{j} denote the average delay experienced by packets going from node i to d via the immediate neighbour j. This information is obtained by delay feedback using ACKs from the destination. Also, denote by $\overline{D}_{id}$ the average delay experienced by all packets from node i to d, without regard to what next node they were forwarded to. STARA iteratively adjusts the routing probabilities $[p_{id}^{j} : j \in N(i,d)]$. It increases the probability p_{id}^{j} of sending a packet via node j if the delay D_{id}^{j} via j is less than the average delay $\overline{D}_{id}$ over all neighbours and decreases it otherwise.

The algorithm consists of two components: an iterative scheme for delay estimation and an iterative scheme for updating routing probabilities [6]:

1. *Delay estimation.* End-to-end ACKs are used to record $\hat{D}_{id}^{j}[n]$, the measured delay of the packet sent at time n. Exponential forgetting is used to estimate the

$$D_{id}^{j}[n] = \gamma D_{id}^{j}[n-1] + (1-\gamma)\hat{D}_{id}^{j}[n].$$

delay:

2. *Probability update scheme.* The routing probabilities are updated as follows:

$$p_{id}^{j}[n+1] = \left[p_{id}^{j}[n] + \alpha[n] q_{id}^{j}[n] \times \left(\overline{D}_{id}[n+1] - D_{id}^{j}[n+1]\right)\right]^{+},$$ where

$$\overline{D}_{id}[n+1] = \sum_{j=1}^{N(i,d)} D_{id}^{j}[n+1] q_{id}^{j}[n],$$

and

$$q_{id}^{j}[n] = (1-\epsilon) p_{id}^{j}[n] + \epsilon \times 1/N(i,d).$$

The above algorithm features two modifications from the scheme discussed above. The $[\cdot]^{+}$ is the projection map onto the simplex of probability vectors, and $\alpha[n]$ are appropriate step sizes. When a packet for destination d arrives at node i, the node routes the packet to one of its neighbours $j \in N(i,d)$ with probability q_{id}^{j} rather than p_{id}^{j}. The actual routing probabilities used are thus a convex combination of the p_{id}^{j} s and a uniform probability distribution on all neighbours. This ensures a positive probability of probing to obtain delay information on unutilized routes [7] and is a standard feature in adaptive control; see [27]. Note that the Wardrop equilibrium is now defined with respect to $q = \{q_{id}^{j}\}$, i.e., $q_{id}^{j} > 0$ only if $D_{id}^{j} = \sum_{k \in N(i,d)} q_{id}^{k} D_{id}^{k} = \min_{\{k \in N(i,d)\}} D_{id}^{k}$.

An important issue in using delay information is that no two clocks are synchronized. The above algorithm works even when clocks in the network are not synchronized [6]. The probability update scheme only uses the difference between delays $(\overline{D}_{id} - D_{id}^{j})$ for a SD pair. Thus, even if the clocks at i and d differ by an offset, by taking the difference $\overline{D}_{id} - D_{id}^{j}$, the offset is eliminated since it is present in both $\overline{D}_{id}$, as well as D_{id}^{j}.

2.6 The Challenges

Our work on rendering Wardrop routing practical will build on the above algorithm. We will preserve all the valuable features, such as immunity to clock offsets at the nodes. However, to obtain a practical protocol that in fact delivers improved performance, we need to make several modifications. We begin by identifying three problems that render difficult the use of the algorithms presented in here

1. The routes followed by packets are unrestricted. They can be arbitrarily long. This causes several problems:
    1. There are many bad routes that packets can go out on and delays experienced by such packets can be exceedingly long.
    2. Since the algorithm requires feedback on all these bad routes before it can adapt, its convergence can be very slow, rendering it impractical.
2. After the routing probabilities have converged to the correct estimates, the algorithm produces loop-free routes. However, packets can follow loopy paths while the algorithm is converging, which, in practice, is always the case.
3. The delay measurement in relies on acknowledgments to carry measurements back to sources and intermediate nodes on a per-packet basis. This poses problems in implementation:
    1. A scheme relying on transport layer ACKs to solve the network layer routing problem violates layering and does not extend to unreliable transport layers (e.g., UDP).
    2. Further, there is no guarantee that ACKs will follow the reversed path as the data packet. Thus, intermediate nodes will not be able to obtain delay information.

To address these problems and obtain a practical protocol amenable to deployment, we redesigned the algorithm in [7] to obtain two protocols M-STARA and P-STARA:

1. We propose new mechanisms to control the length of paths followed by packets, while the algorithm is converging. These mechanisms preserve the attractive properties of load adaptation and multipath utilization, while retaining the property of convergence toward the appropriately defined Wardrop equilibrium.
2. We propose a mechanism that guarantees that routes followed by packets are loop-free even while the algorithm is suntil converging. This mechanism is not

only compatible with the path length control mechanism above but also continues to preserve convergence toward the appropriately defined Wardrop equilibrium for the resultant load adaptive multipath routing protocol.

3. We propose a completely distributed delay measurement mechanism to replace the ACK-based scheme with the attendant problems discussed earlier. The new mechanism retains immunity to clock offsets. It consists of a light-weight link delay measurement protocol and a distance-vector like neighbourhood broadcast of average delay information.

2.7Controlling Paths and Eliminating Loops

Our first objective is to control paths so that the algorithm does not have to investigate all possible paths from source to destination, which would render the convergence very slow. The basic idea that we use is to modify the definition of the neighbourhood $N(i, d)$, the set of neighbours to which node i is allowed to forward packets destined for node d.

2.8 Controlling Paths with M-STARA

Let $S(i, d)$ denote the shortest path distance in hops from node i to destination d. Set $N^0(i, d) := \{j \in N(i, d) : S(j, d) \leq S(i, d)\}$. We call this algorithm, which only allows the nodes in $N^0(i, d)$ to be used for forwarding packets at i destined for d, *M-STARA*. (The reason for the name, denoting Multiplicative-STARA, is that with one additional modification, it can be used to strongly control path lengths. For brevity, we omit discussion on this, referring the reader to) The advantage of M-STARA is that it can be easily built on top of STARA by running a distance vector protocol to produce distances to all destinations. One other important property is that by merely suitably defining the notion of "neighbourhood," we can prove convergence to a suitably defined Wardrop equilibrium.

2.9 Eliminating Loops and Controlling Path Lengths: P-STARA (Parity Stara)

The above mechanism restricts the set of paths investigated by M-STARA and is easy to implement, but it does not provide any guarantees on path lengths. In fact, it does not eliminate routing loops.

Note that under any algorithm that converges to a Wardrop equilibrium with respect to the delay, after the routing probabilities have converged to the correct estimates asymptotically, the resulting probabilistic routes are guaranteed to be loop-free. However, while the adaptive algorithm is suntil converging, which in practice is *always*, packets can indeed follow loopy paths. Thus, we need to guarantee loop-freedom of the algorithm at every instant.

To solve these two problems of controlling the path lengths as well as eliminating routing loops, we propose an additional mechanism, resulting in P-STARA.

The basic idea is to introduce a packet state which alternates between "odd" and "even," as a packet moves from hop to hop. When the packet state is "odd," only those neighbours that strictly decrease the distance in hops to the destination are considered for forwarding. When the packet state is "even," the neighbourhood $N^0(i, d)$, defined above for M-STARA, is used. In this way, we will show that over every two hops the distance to the destination is decreased by at least one hop. This simultaneously eliminates all routing loops, as well as strictly upper bounds the path length to be no more than twice the shortest path length.

An equally important advantage, which we will expound on, is that this scheme admits a simple and completely distributed delay estimation algorithm matched to the definitions of the two neighbourhoods used. To understand the simplicity of the solution provided by P-STARA, one should note the difficulties of controlling route length as well as killing loops while preserving Wardrop equilibration, all without sacrificing simplicity of implementation. For example, if one just allows a budget of k over the shortest path length by initially setting an "excess budget" packet field of k and then progressively decrements it by one after each hop, then one can guarantee that the path length does not exceed the shortest path length by more than k hops. However, one would then have to use k separate routing tables in order to assure the Wardrop property since packets would be differently treated depending on the excess budget with which they arrive at a node. In comparison, the parity feature of P-STARA allows us to get away with just two tables, kills all loops, and yet allows us to control path lengths by a factor of two proportional to the shortest path length.

We now describe the P-STARA algorithm. Define $N^1(i,d) := \{j \in N(i,d) : S(j,d) = S(i,d) - 1\}$, recalling the definitions of $N^0(i,d)$ and $S(j,d)$ from above

1. We include a field $F(p)$ in each packet, which is decremented by one by every node along the path. The source initially sets $F(p)$ to an arbitrary value. In practice, a field like the IP TTL field already behaves in this manner and can be used as $F(p)$.
2. We define the state of the packet as $X(F(p)) = (1 + F(p)) \ mod \ 2$.
3. When the packet has state $X(F) = 0$, node i only consider neighbours in $N^0(i,d)$ as valid for routing. When the packet has state $X(F) = 1$, it considers neighbours in $N^1(i,d)$ as valid for forwarding.
4. Since $F(p)$ is decremented at each node, at successive nodes along its path, the packet's state $X(F(p))$ has different values (“0” or “1”).
5. Corresponding to the two values of $X(F)$, we maintain two separate probability vectors, $(p_{id}^j)_F$, and delay estimate vectors $(D_{id}^j)_F$, for $F = 0, 1$.
6. When a packet for destination d arrives at node i with field F, the node routes the packet to one of its neighbours $j \in N^{X(F)}(i,d)$ using the probabilities $(q_{id}^j)_F$. The probability update and delay estimation rules for P-STARA are given by (4), (5),and(6) below:

$$\left(p_{id}^j\right)_F[n+1] = \left[\left(p_{id}^j\right)_F[n] + \alpha[n] * \left(q_{id}^j\right)_F[n] * \left((\overline{D}_{id})_F[n+1] - \left(D_{id}^j\right)_F[n+1]\right)\right]^+, \quad (4)$$

$$(\overline{D}_{id})_F[n+1] = \sum_{j=1}^{N(i,d)} \left(\left(D_{id}^j\right)_F[n+1] * \left(q_{id}^j\right)_F[n]\right), \quad (5)$$

$$\left(q_{id}^j\right)_F[n] = (1-\epsilon) * \left(p_{id}^j\right)_F[n] + \epsilon * 1/N^{X(F)}(i,d). \quad (6)$$

We now prove that P-STARA has all the desired properties of controlling path lengths and eliminating all routing loops even during the transient phase, while providing the load adaptation of the multipath routing protocol so that it converges to a Wardrop

equilibrium.In fact, we will prove that it provides loop-free routes even when the distance estimates are not accurate or are suntil converging, provided only that the distance estimation scheme uses the destination originated sequence numbering technique of DSDV [10]. For brevity, we avoid an explanation of the sequence numbering technique used by DSDV, as well as the technical definitions of Cesaro convergence in [7], and provide only a brief outline of the corresponding proofs.

Theorem 2.

1. Suppose packet p with source s and destination d follows path P with length $L(P)$. Then, $L(P) \leq 2S(s, d)$.
2. P-STARA produces loop-free paths from every source s to every destination d, provided that the distance estimates $S(s, d)$ are accurate.
3. P-STARA produces loop-free paths from every source s to every destination d, even when the distance estimates $S(i, d)$ are not accurate or are suntil converging, provided only that the distance vector scheme uses sequence numbering to avoid loops.
4. The routing probabilities $q = \{q_{id}^{j}\}$ produced by P-STARA converge almost surely to the set of ϵ -Cesaro-Wardrop equilibria with respect to the restriction to the allowed routes.

Proof.

1. Since $F(p)$ is decremented by 1 at every step, $F(p)$ is alternately even and odd at every node along the path P. Thus, after every two hops along the path P, a packet is closer to the destination by at least one more hop (since we are forced to choose the shortest path neighbour at one of these two hops). Thus, $L(P) \leq 2S(s, d)$.
2. Let P be a path from s to d. P-STARA never forwards packets upstream, i.e., from i to j if $S(j, d) > S(i, d)$. Thus, if P contains a cycle $(i_1, i_2, \ldots i_n)$, then $S(i_1, d) = \ldots = S(i_n, d)$. However, if i_1, i_2, i_3 are three consecutive nodes along P, then $S(i_1, d) - S(i_3, d) \geq 1$. This implies that P contains no cycle of size n for any $n \geq 2$.

3. We can redefine the neighbourhood sets $N^1(i,d)$ and $N^0(i,d)$ (see [28] for details) so that the sequence numbers do not decrease along a path P to the destination. Thus, a loop can be formed only if the sequence numbers are the same. In this case, we can obtain a contradiction by using the fact that the distance estimates $S(i,d)$ decrease by at least one after every two successive hops along the path P.
4. The last property capitalizes on the fact that all we have changed is the definition of admissible neighbourhoods with respect to a destination. This preserves the convergence properties of the original scheme. Thus, the proof paralleling that in [7] can be used.

Future Scope

The future vision of WSNs is to embed numerous distributed devices to monitor and interact with physical world phenomena, and to exploit spatially and temporally dense sensing and actuation capabilities of those sensing devices. These nodes coordinate among themselves to create a network that performs higher-level tasks.

Although extensive efforts have been exerted so far on the routing problem in WSNs, there are still some challenges that confront effective solutions of the routing problem. First, there is a tight coupling between sensor nodes and the physical world. Sensors are embedded in unattended places or systems. This is different from traditional Internet, PDA, and mobility applications that interface primarily and directly 32 with human users. Second, sensors are characterized by a small foot print, and as such nodes present stringent energy constraints since they are equipped with small, finite, energy source. This is also different from traditional fixed but reusable resources. Third, communications is primary consumer of energy in this environment where sending a bit over 10 or 100 meters consumes as much energy as thousands-to-millions of operations (known as R4 signal energy drop-off) [36].

Although the performance of these protocols is promising in terms of energy efficiency, further research would be needed to address issues such as Quality of

Service (QoS) posed by video and imaging sensors and real-time applications. Energy-aware QoS routing in sensor networks will ensure guaranteed bandwidth (or delay) through the duration of connection as well as providing the use of most energy efficient path. Another interesting issue for routing protocols is the consideration of node mobility. Most of the current protocols assume that the sensor nodes and the BS are stationary. However, there might be situations such as battle environments where the BS and possibly the sensors need to be mobile. In such cases, the frequent update of the position of the command node and the sensor nodes and the propagation of that information through the network may excessively drain the energy of nodes. New routing algorithms are needed in order to handle the overhead of mobility and topology changes in such energy constrained environment. Future trends in routing techniques in WSNs focus on different directions, all share the common objective of prolonging the network lifetime. We summarize some of these directions and give some pertinent references as follows:

• Exploit redundancy: typically a large number of sensor nodes are implanted inside or beside the phenomenon. Since sensor nodes are prone to failure, fault tolerance techniques come in picture to keep the network operating and performing its tasks. Routing techniques that explicitly employ fault tolerance techniques in an efficient manner are still under investigation.

• Tiered architectures (mix of form/energy factors): Hierarchical routing is an old technique to enhance scalability and efficiency of the routing protocol. However, novel techniques to network clustering which maximize the network lifetime are also a hot area of research in WSNs.

• Exploit spatial diversity and density of sensor/actuator nodes: Nodes will span a network area that might be large enough to provide spatial communication between sensor nodes. Achieving energy efficient communication in this densely populated environment deserves further investigation. The dense deployment of sensor nodes should allow the network to adapt to unpredictable environment.

• Achieve desired global behavior with adaptive localized algorithms (i.e., do not rely on global inter- action or information). However, in a dynamic

environment, this is hard to model.

• Leverage data processing inside the network and exploit computation near data sources to reduce communication, i.e., perform in-network distributed processing.WSNs are organized around naming data, not nodes identities. Since we have a large collections of distributed elements, localized algo- rithms that achieve system-wide properties in terms of local processing of data before being sent to the destination are still needed. Nodes in the network will store named data and make it available for processing. There is a high need to create efficient processing points in the network, e.g., duplicate suppression, aggregation, correlation of data. How to efficiently and optimally find those points is still an open research issue.

• Time and location synchronization: energy-efficient techniques for associating time and spatial coor- dinates with data to support collaborative processing are also required [20].

• Localization: sensor nodes are randomly deployed into an unplanned infrastructure. The problem of estimating spatial-coordinates of the node is referred to as localization. Global Positioning System (GPS) cannot be used in WSNs as GPS can work only outdoors and cannot work in the presence of any obstruction. Moreover, GPS receivers are expensive and not suitable in the construction of small cheap sensor nodes. Hence, there is a need to develop other means of establishing a coordinate system without relying on an existing infrastructure. Most of the proposed localization techniques today, depend on recursive trilateration/multilateration techniques (e.g., [38]) which would not provide enough accuracy in WSNs.

• Self-configuration and reconfiguration is essential to lifetime of unattended systems in dynamic, and constrained energy environment. This is important for keeping the network up and running. As nodes die and leave the network, update and reconfiguration mechanisms should take place. A feature that is important in every routing protocol is to adapt to topology changes very quickly and to maintain the network functions.

• Secure Routing: Current routing protocols optimize for the limited capabilities of the

nodes and the application specific nature of the networks, but do not consider security. Although these protocols have not been designed with security as a goal, it is important to analyze their security properties. One aspect of sensor networks that complicates the design of a secure routing protocol is in-network aggregation. In WSNs, in-network processing makes end-to-end security mechanisms harder to deploy because intermediate nodes need direct access to the contents of the messages.

- Other possible future research for routing protocols includes the integration of sensor networks with wired networks (i.e. Internet). Most of the applications in security and environmental monitoring require the data collected from the sensor nodes to be transmitted to a server so that further analysis can be done. On the other hand, the requests from the user should be made to the BS through Inter- net. Since the routing requirements of each environment are different, further research is necessary for handling these kinds of situations.

CODE

I have use java as the programming language and I have the following classes:

- AStarHeuristic.class
- AStarPathFinder$1.class
- AStarPathFinder$Node.class
- AStarPathFinder$SortedList.class
- AStarPathFinder.class
- ClosestHeuristic.class
- ClosestSquaredHeuristic.class
- ExampleBreakpoint.class
- GameMap.class
- ManhattanHeuristtic.class
- Monitoring$1.class
- Monitoring$2.class
- Monitoring.class
- Mover.class
- Path$Step.class
- Path.class
- PathFinder.class
- PathTest$1.class
- PathTest$2.class
- PathTest$3.class
- PathTest$4.class
- PathTest.class
- TileBasedMap.class
- UnitMover.class

Now,Codes for the respective classes follows:

```
package wardrop;


public interface AStarHeuristic {
	public float getCost(TileBasedMap map, Mover mover, int x, int y, int tx, int
ty);
}
package wardrop;
import java.util.ArrayList;
import java.util.Collections;


public class AStarPathFinder implements PathFinder {

	private ArrayList closed = new ArrayList();

	private SortedList open = new SortedList();


	private TileBasedMap map;
	private int maxSearchDistance;

	private Node[][] nodes;
	private boolean allowDiagMovement;
	private AStarHeuristic heuristic;
      int mapdirection=0;

	public AStarPathFinder(TileBasedMap map, int maxSearchDistance, boolean
allowDiagMovement) {
		this(map, maxSearchDistance, allowDiagMovement, new
ClosestHeuristic());
	}
```

```
	public AStarPathFinder(TileBasedMap map, int maxSearchDistance,
							boolean		allowDiagMovement,
AStarHeuristic heuristic) {
		this.heuristic = heuristic;
		this.map = map;
		this.maxSearchDistance = maxSearchDistance;
		this.allowDiagMovement = allowDiagMovement;

		nodes = new Node[map.getWidthInTiles()][map.getHeightInTiles()];
		for (int x=0;x<map.getWidthInTiles();x++) {
			for (int y=0;y<map.getHeightInTiles();y++) {
				nodes[x][y] = new Node(x,y);
			}
		}
	}


	public Path findPath(Mover mover, int sx, int sy, int tx, int ty) {
		if (map.blocked(mover, tx, ty)) {
			return null;
		}
		nodes[sx][sy].cost = 0;
		nodes[sx][sy].depth = 0;
		closed.clear();
		open.clear();
		open.add(nodes[sx][sy]);

		nodes[tx][ty].parent = null;

		int maxDepth = 0;
		while ((maxDepth < maxSearchDistance) && (open.size() != 0)) {

			Node current = getFirstInOpen();
			if (current == nodes[tx][ty]) {
```

```
					break;
				}

				removeFromOpen(current);
				addToClosed(current);
				for (int x=-1;x<2;x++) {
					for (int y=-1;y<2;y++) {

						if ((x == 0) && (y == 0)) {
							continue;
						}
						if (!allowDiagMovement) {
							if ((x != 0) && (y != 0)) {
								continue;
							}
						}

						int xp = x + current.x;
						int yp = y + current.y;

						if (isValidLocation(mover,sx,sy,xp,yp)) {
							float nextStepCost = current.cost +
getMovementCost(mover, current.x, current.y, xp, yp);
							Node neighbour = nodes[xp][yp];
							map.pathFinderVisited(xp, yp);
							if (nextStepCost < neighbour.cost) {
								if (inOpenList(neighbour)) {

	removeFromOpen(neighbour);
								}
								if (inClosedList(neighbour)) {

	removeFromClosed(neighbour);
								}
```

```
                                        }

                                        if (!inOpenList(neighbour) && !(inClosedList(neighbour))) {
                                                neighbour.cost = nextStepCost;
                                                neighbour.heuristic = getHeuristicCost(mover, xp, yp, tx, ty);
                                                maxDepth = Math.max(maxDepth, neighbour.setParent(current));
                                                addToOpen(neighbour);
                                        }
                                }
                        }
                }
        }

        if (nodes[tx][ty].parent == null) {
                return null;
        }
        Path path = new Path();
        Node target = nodes[tx][ty];
        while (target != nodes[sx][sy]) {
                path.prependStep(target.x, target.y);
                target = target.parent;
        }
        path.prependStep(sx,sy);

        return path;
}

protected Node getFirstInOpen() {
        return (Node) open.first();
}
```

```
	protected void addToOpen(Node node) {
		open.add(node);
	}

	protected boolean inOpenList(Node node) {
		return open.contains(node);
	}

	protected void removeFromOpen(Node node) {
		open.remove(node);
	}


	protected void addToClosed(Node node) {
		closed.add(node);
	}


	protected boolean inClosedList(Node node) {
		return closed.contains(node);
	}


	protected void removeFromClosed(Node node) {
		closed.remove(node);
	}


	protected boolean isValidLocation(Mover mover, int sx, int sy, int x, int y) {
		boolean invalid = (x < 0) || (y < 0) || (x >= map.getWidthInTiles()) || (y
>= map.getHeightInTiles());

		if ((!invalid) && ((sx != x) || (sy != y))) {
			invalid = map.blocked(mover, x, y);
```

```
		}

		return !invalid;
	}



	public float getMovementCost(Mover mover, int sx, int sy, int tx, int ty) {
		return map.getCost(mover, sx, sy, tx, ty);
	}

	public float getHeuristicCost(Mover mover, int x, int y, int tx, int ty) {
		return heuristic.getCost(map, mover, x, y, tx, ty);
	}

	private class SortedList {

		private ArrayList list = new ArrayList();
		public Object first() {
			return list.get(0);
		}

		public void clear() {
			list.clear();
		}

		public void add(Object o) {
			list.add(o);
			Collections.sort(list);
		}

		public void remove(Object o) {
			list.remove(o);
		}
```

```
		public int size() {
			return list.size();
		}



		public boolean contains(Object o) {
			return list.contains(o);
		}
	}

	/**
	 * A single node in the search graph
	 */
	private class Node implements Comparable {

		private int x;

		private int y;

		private float cost;

		private Node parent;

		private float heuristic;

		private int depth;

		public Node(int x, int y) {
			this.x = x;
			this.y = y;
		}

		public int setParent(Node parent) {
			depth = parent.depth + 1;
```

```
			this.parent = parent;

			return depth;
		}

		public int compareTo(Object other) {
			Node o = (Node) other;

			float f = heuristic + cost;
			float of = o.heuristic + o.cost;

			if (f < of) {
				return -1;
			} else if (f > of) {
				return 1;
			} else {
				return 0;
			}
		}
	}
}
package wardrop;
import java.util.ArrayList;
import java.util.Collections;
public class AStarPathFinder implements PathFinder {

	private ArrayList closed = new ArrayList();

	private SortedList open = new SortedList();


	private TileBasedMap map;
	private int maxSearchDistance;
```

```
	private Node[][] nodes;
	private boolean allowDiagMovement;
	private AStarHeuristic heuristic;
    int mapdirection=0;

	public AStarPathFinder(TileBasedMap map, int maxSearchDistance, boolean allowDiagMovement) {
		this(map, maxSearchDistance, allowDiagMovement, new ClosestHeuristic());
	}

	public AStarPathFinder(TileBasedMap map, int maxSearchDistance,
							   boolean allowDiagMovement, AStarHeuristic heuristic) {
		this.heuristic = heuristic;
		this.map = map;
		this.maxSearchDistance = maxSearchDistance;
		this.allowDiagMovement = allowDiagMovement;

		nodes = new Node[map.getWidthInTiles()][map.getHeightInTiles()];
		for (int x=0;x<map.getWidthInTiles();x++) {
			for (int y=0;y<map.getHeightInTiles();y++) {
				nodes[x][y] = new Node(x,y);
			}
		}
	}


	public Path findPath(Mover mover, int sx, int sy, int tx, int ty) {
		if (map.blocked(mover, tx, ty)) {
			return null;
		}
		nodes[sx][sy].cost = 0;
		nodes[sx][sy].depth = 0;
```

```
closed.clear();
open.clear();
open.add(nodes[sx][sy]);

nodes[tx][ty].parent = null;

int maxDepth = 0;
while ((maxDepth < maxSearchDistance) && (open.size() != 0)) {

	Node current = getFirstInOpen();
	if (current == nodes[tx][ty]) {
		break;
	}

	removeFromOpen(current);
	addToClosed(current);
	for (int x=-1;x<2;x++) {
		for (int y=-1;y<2;y++) {

			if ((x == 0) && (y == 0)) {
				continue;
			}
			if (!allowDiagMovement) {
				if ((x != 0) && (y != 0)) {
					continue;
				}
			}

			int xp = x + current.x;
			int yp = y + current.y;

			if (isValidLocation(mover,sx,sy,xp,yp)) {
				float nextStepCost = current.cost +
getMovementCost(mover, current.x, current.y, xp, yp);
```

```
						Node neighbour = nodes[xp][yp];
						map.pathFinderVisited(xp, yp);
						if (nextStepCost < neighbour.cost) {
							if (inOpenList(neighbour)) {

	removeFromOpen(neighbour);
							}
							if (inClosedList(neighbour)) {

	removeFromClosed(neighbour);
							}
						}

						if (!inOpenList(neighbour) &&
!(inClosedList(neighbour))) {
							neighbour.cost = nextStepCost;
							neighbour.heuristic =
getHeuristicCost(mover, xp, yp, tx, ty);
							maxDepth = Math.max(maxDepth,
neighbour.setParent(current));
							addToOpen(neighbour);
						}
					}
				}
			}
		}

		if (nodes[tx][ty].parent == null) {
			return null;
		}
		Path path = new Path();
		Node target = nodes[tx][ty];
		while (target != nodes[sx][sy]) {
			path.prependStep(target.x, target.y);
```

```
			target = target.parent;
		}
		path.prependStep(sx,sy);

		return path;
	}

	protected Node getFirstInOpen() {
		return (Node) open.first();
	}

	protected void addToOpen(Node node) {
		open.add(node);
	}

	protected boolean inOpenList(Node node) {
		return open.contains(node);
	}

	protected void removeFromOpen(Node node) {
		open.remove(node);
	}



	protected void addToClosed(Node node) {
		closed.add(node);
	}



	protected boolean inClosedList(Node node) {
		return closed.contains(node);
	}
```

```
	protected void removeFromClosed(Node node) {
		closed.remove(node);
	}


	protected boolean isValidLocation(Mover mover, int sx, int sy, int x, int y) {
		boolean invalid = (x < 0) || (y < 0) || (x >= map.getWidthInTiles()) || (y
>= map.getHeightInTiles());

		if ((!invalid) && ((sx != x) || (sy != y))) {
			invalid = map.blocked(mover, x, y);
		}

		return !invalid;
	}


	public float getMovementCost(Mover mover, int sx, int sy, int tx, int ty) {
		return map.getCost(mover, sx, sy, tx, ty);
	}

	public float getHeuristicCost(Mover mover, int x, int y, int tx, int ty) {
		return heuristic.getCost(map, mover, x, y, tx, ty);
	}

	private class SortedList {

		private ArrayList list = new ArrayList();
		public Object first() {
			return list.get(0);
		}

		public void clear() {
			list.clear();
```

```
	}

	public void add(Object o) {
		list.add(o);
		Collections.sort(list);
	}

	public void remove(Object o) {
		list.remove(o);
	}

	public int size() {
		return list.size();
	}



	public boolean contains(Object o) {
		return list.contains(o);
	}
}

/**
 * A single node in the search graph
 */
private class Node implements Comparable {

	private int x;

	private int y;

	private float cost;

	private Node parent;
```

```
		private float heuristic;

		private int depth;

		public Node(int x, int y) {
			this.x = x;
			this.y = y;
		}

		public int setParent(Node parent) {
			depth = parent.depth + 1;
			this.parent = parent;

			return depth;
		}

		public int compareTo(Object other) {
			Node o = (Node) other;

			float f = heuristic + cost;
			float of = o.heuristic + o.cost;

			if (f < of) {
				return -1;
			} else if (f > of) {
				return 1;
			} else {
				return 0;
			}
		}
	}
}
package wardrop;
```

ClosestHeuristic.class:

```
public class ClosestHeuristic implements AStarHeuristic {

	public float getCost(TileBasedMap map, Mover mover, int x, int y, int tx, int ty)
{
		float dx = tx - x;
		float dy = ty - y;

		float result = (float) (Math.sqrt((dx*dx)+(dy*dy)));

		return result;
	}

}
```

ClosestSquaredHeuristic.class

```
package wardrop;

/**
 *
 * @author Admin
 */

public class ClosestSquaredHeuristic implements AStarHeuristic {

	public float getCost(TileBasedMap map, Mover mover, int x, int y, int tx, int ty)
{
		float dx = tx - x;
		float dy = ty - y;

		return ((dx*dx)+(dy*dy));
	}
```

```
}
```

ExampleBreakpoint.class

```
package wardrop;

/**
 *
 * @author Admin
 */
public class ExampleBreakpoint {

    private static int x = 5;

    public ExampleBreakpoint() {
    }

    public static void main(String app[]) {
        x++;
        System.out.print(x);
    }
}
```

GameMap.class

```
package wardrop;
import java.util.Locale;
import javax.swing.JOptionPane;
public class GameMap implements TileBasedMap {


    public static final int WIDTH = 30;

    public static final int HEIGHT = 30;

    public static final int GRASS = 0;
```

```
public static final int WATER = 1;

public static final int TREES = 2;

public static final int PLANE = 3;

public static final int BOAT = 4;

public static final int TANK = 5;

private int[][] terrain = new int[WIDTH][HEIGHT];

private int[][] units = new int[WIDTH][HEIGHT];

private boolean[][] visited = new boolean[WIDTH][HEIGHT];

/**
 * Create a new test map with some default configuration
 */
public GameMap() {
    // create some test data
   /* if (d == 1) {
       fillArea(0, 0, 5, 5, WATER);
       fillArea(0, 5, 3, 10, WATER);
       fillArea(0, 5, 3, 10, WATER);
       fillArea(0, 15, 7, 15, WATER);
       fillArea(7, 26, 22, 4, WATER);

       fillArea(5, 0, 25, 3, WATER);
       fillArea(27, 3, 3, 27, WATER);

       fillArea(17, 9, 10, 3, TREES);
       fillArea(20, 12, 5, 3, TREES);
```

```
    fillArea(8, 8, 7, 3, TREES);
    fillArea(10, 11, 3, 3, TREES);

    fillArea(15, 20, 3, 3, TREES);

    units[15][15] = TANK;
    units[2][7] = BOAT;
    units[20][25] = PLANE;
}

/*if (d == 2) {
    fillArea(0, 0, 5, 5, WATER);
    fillArea(0, 5, 3, 10, WATER);
    fillArea(0, 5, 3, 10, WATER);
    fillArea(0, 15, 7, 15, WATER);
    fillArea(7, 26, 22, 4, WATER);

    fillArea(5, 0, 25, 3, WATER);
    fillArea(27, 3, 3, 27, WATER);

    fillArea(17, 9, 10, 3, TREES);
    fillArea(20, 12, 5, 3, TREES);

    fillArea(8, 8, 7, 3, TREES);
    fillArea(10, 11, 3, 3, TREES);

    fillArea(15, 20, 3, 3, TREES);

    units[15][15] = TANK;
    units[2][7] = BOAT;
    units[20][25] = PLANE;
}
if (d == 3) {
```

```
        fillArea(0, 0, 5, 5, WATER);
        fillArea(0, 5, 3, 10, WATER);
        fillArea(0, 5, 3, 10, WATER);
        fillArea(0, 15, 7, 15, WATER);
        fillArea(7, 26, 22, 4, WATER);

        fillArea(5, 0, 25, 3, WATER);
        fillArea(27, 3, 3, 27, WATER);

        fillArea(17, 9, 10, 3, TREES);
        fillArea(20, 12, 5, 3, TREES);

        fillArea(8, 8, 7, 3, TREES);
        fillArea(10, 11, 3, 3, TREES);

        fillArea(15, 20, 3, 3, TREES);

        units[15][15] = TANK;
        units[2][7] = BOAT;
        units[20][25] = PLANE;

    }*/
}

/**
 * Fill an area with a given terrain type
 *
 * @param x The x coordinate to start filling at
 * @param y The y coordinate to start filling at
 * @param width The width of the area to fill
 * @param height The height of the area to fill
 * @param type The terrain type to fill with
 */
```

```
public void drawmap(int map)
{
        if (map==1) {
        fillArea(0, 0, 5, 5, WATER);
        fillArea(0, 5, 3, 10, WATER);
        fillArea(0, 5, 3, 10, WATER);
        fillArea(0, 15, 7, 15, WATER);
        fillArea(7, 26, 22, 4, WATER);

        fillArea(5, 0, 25, 3, WATER);
        fillArea(27, 3, 3, 27, WATER);

        fillArea(17, 9, 10, 3, TREES);
        fillArea(20, 12, 5, 3, TREES);

        fillArea(8, 8, 7, 3, TREES);
        fillArea(10, 11, 3, 3, TREES);

        fillArea(15, 20, 3, 3, TREES);

        units[15][15] = TANK;
        units[2][7] = BOAT;
        units[20][25] = PLANE;
    }
if (map==2) {
        fillArea(0, 0, 5, 5, WATER);
        fillArea(0, 5, 3, 10, WATER);
        fillArea(0, 5, 3, 10, WATER);


        fillArea(5, 9, 10, 3, TREES);
        fillArea(10, 12, 5, 3, TREES);

        fillArea(12, 9, 2, 3, TREES);
```

```
        fillArea(15, 10, 1, 3, TREES);

        fillArea(8, 8, 7, 3, TREES);
        fillArea(10, 11, 3, 3, TREES);

        fillArea(15, 20, 3, 3, TREES);

        units[15][15] = TANK;
        units[2][7] = BOAT;
        units[20][25] = PLANE;
    }

}

private void fillArea(int x, int y, int width, int height, int type) {
    for (int xp = x; xp < x + width; xp++) {
        for (int yp = y; yp < y + height; yp++) {
            terrain[xp][yp] = type;
            // Monitoring.jTextArea1.append(""+terrain[xp][yp]);
        }
    }
}



public void clearVisited() {
    for (int x = 0; x < getWidthInTiles(); x++) {
        for (int y = 0; y < getHeightInTiles(); y++) {
            visited[x][y] = false;
        }
    }
}

/**
 * @see TileBasedMap#visited(int, int)
```

```
     */
    public boolean visited(int x, int y) {
        return visited[x][y];
    }

    /**
     * Get the terrain at a given location
     *
     * @param x The x coordinate of the terrain tile to retrieve
     * @param y The y coordinate of the terrain tile to retrieve
     * @return The terrain tile at the given location
     */
    public int getTerrain(int x, int y) {
        return terrain[x][y];
    }

    /**
     * Get the unit at a given location
     *
     * @param x The x coordinate of the tile to check for a unit
     * @param y The y coordinate of the tile to check for a unit
     * @return The ID of the unit at the given location or 0 if there is no unit
     */
    public int getUnit(int x, int y) {
        return units[x][y];
    }

    /**
     * Set the unit at the given location
     *
     * @param x The x coordinate of the location where the unit should be set
     * @param y The y coordinate of the location where the unit should be set
     * @param unit The ID of the unit to be placed on the map, or 0 to clear the unit at
the
```

```
 * given location
 */
public void setUnit(int x, int y, int unit) {
   units[x][y] = unit;
}

/**
 * @see TileBasedMap#blocked(Mover, int, int)
 */
public boolean blocked(Mover mover, int x, int y) {
   // if theres a unit at the location, then it's blocked
   if (getUnit(x, y) != 0) {
      return true;
   }

   int unit = ((UnitMover) mover).getType();

   // planes can move anywhere
   if (unit == PLANE) {
      return false;
   }
   // tanks can only move across grass
   if (unit == TANK) {
      return terrain[x][y] != GRASS;
   }
   // boats can only move across water
   if (unit == BOAT) {
      return terrain[x][y] != WATER;
   }

   // unknown unit so everything blocks
   return true;
}
public float getCost(Mover mover, int sx, int sy, int tx, int ty) {
```

```
        return 1;
    }
    public int getHeightInTiles() {
        return WIDTH;
    }
    public int getWidthInTiles() {
        return HEIGHT;
    }

    /**
     * @see TileBasedMap#pathFinderVisited(int, int)
     */
    public void pathFinderVisited(int x, int y) {
        visited[x][y] = true;
    }
}
```

ManhattanHeuristic.class

```
public class ManhattanHeuristic implements AStarHeuristic {

        private int minimumCost;

        public ManhattanHeuristic(int minimumCost) {
                this.minimumCost = minimumCost;
        }



        public float getCost(TileBasedMap map, Mover mover, int x, int y, int tx,
                        int ty) {
                return minimumCost * (Math.abs(x-tx) + Math.abs(y-ty));
        }

}
```

Mover.class

```
package wardrop;

public interface Mover {

}
```

Path$Step.class

```
package wardrop;
import java.util.ArrayList;
public class Path {

	private ArrayList steps = new ArrayList();

	public Path() {

	}


	public int getLength() {
		return steps.size();
	}

	public Step getStep(int index) {
		return (Step) steps.get(index);
	}


	public int getX(int index) {
		return getStep(index).x;
	}
```

```
public int getY(int index) {
        return getStep(index).y;
}

public void appendStep(int x, int y) {
        steps.add(new Step(x,y));
}

public void prependStep(int x, int y) {
        steps.add(0, new Step(x, y));
}

public boolean contains(int x, int y) {
        return steps.contains(new Step(x,y));
}

public class Step {

        private int x;
        private int y;
        public Step(int x, int y) {
                this.x = x;
                this.y = y;
        }

        public int getX() {
                return x;
        }

        public int getY() {
                return y;
        }

        public int hashCode() {
```

```
                return x*y;
            }

            public boolean equals(Object other) {
                if (other instanceof Step) {
                    Step o = (Step) other;
                    return (o.x == x) && (o.y == y);
                }
                return false;
            }
    }
}
```

PathFinder.class

```
package wardrop;

public interface PathFinder {

    public Path findPath(Mover mover, int sx, int sy, int tx, int ty);
}
```

PathTest$1.class

```
package wardrop;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionListener;
import java.awt.event.WindowAdapter;
```

```
import java.awt.event.WindowEvent;
import java.awt.image.BufferedImage;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

import javax.imageio.ImageIO;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;


public class PathTest extends JPanel {

        private static GameMap map =new GameMap();

      public static JTextArea textdisplay=new JTextArea();
      public static JTextArea direction=new JTextArea();
        private PathFinder finder;
        private Path path;
      static PathTest test;
        private Image[] tiles = new Image[6];
        private Image buffer;
        private int selectedx = -1;
        private int selectedy = -1;
        private int lastFindX = -1;

        private int lastFindY = -1;
      private int mapselection=0;
      static JComboBox box=new JComboBox();
```

```
static  JFrame jf=new JFrame();

    public PathTest() {
        try {
        box.addItem("Load War field -1");
        box.addItem("Load War field -2");
        box.addItem("Load War field -3");
        box.addItem("Load War field -4");
        direction.append( "The Signal Covering area by node" );
        tiles[GameMap.TREES] = ImageIO.read(getResource("trees.png"));
            tiles[GameMap.GRASS]                                    =
ImageIO.read(getResource("grass.png"));
            tiles[GameMap.WATER]                                    =
ImageIO.read(getResource("water.png"));
            tiles[GameMap.TANK]                                     =
ImageIO.read(getResource("tank.png"));
            tiles[GameMap.PLANE]                                    =
ImageIO.read(getResource("plane.png"));
            tiles[GameMap.BOAT]                                     =
ImageIO.read(getResource("boat.png"));
        } catch (IOException e) {
            System.err.println("Failed to load resources: "+e.getMessage());
            System.exit(0);
        }
      //int        d=Integer.parseInt(JOptionPane.showInputDialog(null,"Selection
map"));

        finder = new AStarPathFinder(map, 500, true);
     /*design.setLayout(null);
      design.setSize(500, 500);
      design.setVisible(true);*/
      //design.add(g);

        addMouseListener(new MouseAdapter() {
```

```
                    public void mousePressed(MouseEvent e) {
                            handleMousePressed(e.getX(), e.getY());
                   int x=e.getX();
                   int y=e.getY();
                   if((x>50 && x< 536) && (y<556 && y>50 ))
                   textdisplay.append("The   node  movement  between  node  \n"+x+"
:\t"+ y +"\n" ) ;
                    }
              });
              addMouseMotionListener(new MouseMotionListener() {
                    public void mouseDragged(MouseEvent e) {
                    }

                    public void mouseMoved(MouseEvent e) {
                            handleMouseMoved(e.getX(), e.getY());
                   int x=e.getX();
                   int y=e.getY();
                   if((x>50 && x< 536) && (y<556 && y>50 ))
                    direction.append(  "X  position  \t  :"+x  +"\t"+"Y  position  \t  :"+x
+"\n" );
                    }
              });
          this.add(textdisplay);
       }

       private InputStream getResource(String ref) throws IOException {
              InputStream                         in                              =
Thread.currentThread().getContextClassLoader().getResourceAsStream(ref);
              if (in != null) {
                    return in;
              }
              return new FileInputStream(ref);
       }
```

```
	private void handleMouseMoved(int x, int y) {
		x -= 50;
		y -= 50;
		x /= 16;
		y /= 16;

		if ((x < 0) || (y < 0) || (x >= map.getWidthInTiles()) || (y >=
map.getHeightInTiles())) {
			return;
		}

		if (selectedx != -1) {
			if ((lastFindX != x) || (lastFindY != y)) {
				lastFindX = x;
				lastFindY = y;
				path = finder.findPath(new
UnitMover(map.getUnit(selectedx, selectedy)),
						selectedx,
selectedy, x, y);
				repaint(0);
			}
		}
	}

	private void handleMousePressed(int x, int y) {
		x -= 50;
		y -= 50;
		x /= 16;
		y /= 16;

		if ((x < 0) || (y < 0) || (x >= map.getWidthInTiles()) || (y >=
map.getHeightInTiles())) {
			return;
		}
```

```
		if (map.getUnit(x, y) != 0) {
			selectedx = x;
			selectedy = y;
			lastFindX = - 1;
		} else {
			if (selectedx != -1) {
				map.clearVisited();
				path = finder.findPath(new UnitMover(map.getUnit(selectedx, selectedy)),selectedx, selectedy, x, y);
			if (path != null) {
					path = null;
					int unit = map.getUnit(selectedx, selectedy);
					map.setUnit(selectedx, selectedy, 0);
					map.setUnit(x,y,unit);
					selectedx = x;
					selectedy = y;
					lastFindX = - 1;
				}
			}
		}

		repaint(0);
	}
  @Override
	public void paintComponent(Graphics graphics) {
	super.paintComponent(graphics);
		// create an offscreen buffer to render the map

		if (buffer == null) {
			buffer = new BufferedImage(600, 600, BufferedImage.TYPE_INT_ARGB);
		}
		Graphics g = buffer.getGraphics();
```

```
        g.clearRect(0,0,600,600);
        g.translate(50, 50);

        // cycle through the tiles in the map drawing the appropriate
        // image for the terrain and units where appropriate
        for (int x=0;x<map.getWidthInTiles();x++) {
            for (int y=0;y<map.getHeightInTiles();y++) {
                g.drawImage(tiles[map.getTerrain(x, y)],x*16,y*16,null);
                if (map.getUnit(x, y) != 0) {
                    g.drawImage(tiles[map.getUnit(x,
y)],x*16,y*16,null);
                } else {
                    if (path != null) {
                        if (path.contains(x, y)) {
                            g.setColor(Color.blue);
                            g.fillRect((x*16)+4, (y*16)+4,7,7);
                        }
                    }
                }
            }
        }
        // if a unit is selected then draw a box around it
        if (selectedx != -1) {
            g.setColor(Color.black);
            g.drawRect(selectedx*16, selectedy*16, 15, 15);
            g.drawRect((selectedx*16)-2, (selectedy*16)-2, 19, 19);
            g.setColor(Color.white);
            g.drawRect((selectedx*16)-1, (selectedy*16)-1, 17, 17);
        }
        graphics.drawImage(buffer, 0, 0, null);
    }

  private static void BoxItemStateChanged(java.awt.event.ItemEvent evt) {
```

```
        JOptionPane.showMessageDialog(null, "select");
        if (box.getSelectedIndex()==01) {
          jf.getContentPane().remove(test);//setBounds(0, 40, 600, 600);
          map.drawmap(1);
          jf.getContentPane().add(test).setBounds(0, 40, 600, 600);

        }
        if (box.getSelectedIndex()==02) {
          //jf.getContentPane().remove(test);//setBounds(0, 40, 600, 600);
          map.drawmap(2);
          jf.getContentPane().add(test).setBounds(0, 40, 600, 600);
        }
      }



        public static void main(String[] argv) {
                test = new PathTest();
          jf.setLayout(null);
          JScrollPane jp=new JScrollPane(textdisplay);
          JScrollPane jp1=new JScrollPane(direction);

          jf.getContentPane().add(test.box).setBounds(700, 500, 70, 30);


          jf.getContentPane().add(new JScrollPane(jp)).setBounds(650, 40, 200, 200);
          jf.getContentPane().setBackground(Color.BLACK);
          jf.getContentPane().add(new   JScrollPane(jp1)).setBounds(650,   250,   200,
200);
          jf.getContentPane().setBackground(Color.BLACK);

          box.addItemListener(new java.awt.event.ItemListener() {
                  public void itemStateChanged(java.awt.event.ItemEvent evt) {
                BoxItemStateChanged(evt);
               } });
```

```
            jf.setVisible(true);
            jf.setSize(950,650);
            jf.setLocation(10,30);
            jf.setTitle("Welcome to The Wireles-communication ");
            JLabel jLabel1=new JLabel();
            jLabel1.setFont(new java.awt.Font("Monotype Corsiva", 0, 35)); // NOI18N
            jLabel1.setText("Monitoring System");jLabel1.setBackground(Color.BLUE);
            jf.getContentPane().add(jLabel1).setBounds(350, 5, 250, 40);
            jf.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent e) {
                              System.exit(0);
                        }
                  });
            jf.setResizable(false);
        }

}
```

TileBasedMap.class

```
package wardrop;

public interface TileBasedMap {

   public int getWidthInTiles();

   public int getHeightInTiles();

   public void pathFinderVisited(int x, int y);

   public boolean blocked(Mover mover, int x, int y);

   public float getCost(Mover mover, int sx, int sy, int tx, int ty);
```

}

REFERENCES

[1]T. Clausen, P. Jacquet, C. Adjih, A. Laouiti, P. Minet, and
P. Muhlethaler, “Optimized Link State Routing Protocol (OLSR),”
IETF RFC 3626.

[2] D.B. Johnson and D.A. Maltz, “Dynamic Source Routing in
Ad Hoc Wireless Networks,” Mobile Computing, Kluwer Academic
Publishers, 1996.

[3] C.E. Perkins, E.M. Royer, and S. Das, “Ad Hoc on Demand

Distance Vector Routing," Proc. Second IEEE Workshop Mobile Computing Systems and Applications (WMCSA '99), 1999.

[4] V. Park and S. Corson, "A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks," Proc. IEEE INFOCOM, 1997.

[5] M.P. Zygmunt Haas and P. Samar, "The Zone Routing Protocol for Ad Hoc Networks," IETF Internet draft, work in progress, 1999.

[6] P. Gupta and P.R. Kumar, "A System and Traffic Independent Adaptive Routing Algorithm for Ad Hoc Networks," Proc. IEEE Conf. Decision and Control (CDC '97), 1997.