A LINK-STATE BASED ON-DEMAND ROUTING PROTOCOL SUPPORTING REAL-TIME TRAFFIC FOR WIRELESS MOBILE AD HOC NETWORKS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Gökçe Görbil

August, 2007

ABSTRACT

A LINK-STATE BASED ON-DEMAND ROUTING PROTOCOL SUPPORTING REAL-TIME TRAFFIC FOR WIRELESS MOBILE AD HOC NETWORKS

Gökçe Görbil
M.S. in Computer Engineering
Supervisor: Assist. Prof. Dr. İbrahim Körpeoğlu
August, 2007

Wireless ad hoc networks have gained a lot of popularity since their introduction and as many wireless network interface cards provide support for ad hoc networking, such networks have also seen real-life deployment for non-specialized purposes. Wireless mobile ad hoc networks (MANETs) are currently the most common type of ad hoc networks, and such networks are especially esteemed for their mobility support and ease of deployment due to their ad hoc nature. As most common network applications, such as the Web, FTP, email, and instant messaging, are data-centric and do not operate under strict time constraints, MANETs have been deployed to enable such non-real-time applications in the past. However, with the increasing use of real-time applications over ad hoc networks, such as teleconferencing, VoIP, and security and tracking applications where timeliness is of importance, real-time traffic support in multi-hop wireless mobile ad hoc networks has become an issue.

We propose an event-driven, link-state based, on-demand routing protocol to enable real-time traffic support in such multi-hop wireless mobile ad hoc networks. Our protocol, which is named *Elessar*, is based on link-state topology dissemination, but instead of the more common periodic link-state messaging scheme, we employ event-driven link-state messages in Elessar, where topology changes are the events of interest. Through such an approach, we aim to lower the overhead of our protocol, especially for low-mobility cases, which is currently the most commonly encountered case with ad hoc networks deployed with machines directly interacting with humans, such as PDAs and laptops. Due to its link-state nature, our protocol is able to support non-real-time traffic without any further action. In order to support real-time traffic, however, we employ a direct cost dissemination mechanism, which only operates on-demand when there are one or more real-time flows in the network. We aim to provide soft quality-of-service (QoS) guarantees to real-time flows through intelligent path selection, without any resource reservation. We also aim to provide such QoS guarantees throughout the lifetime of a real-time flow, even in the face of node failures and mobility, by dynamic path adaptation during the lifetime of the flow. Elessar is able to support real-time and non-real-time traffic concurrently, as well as various different types of concurrent real-time traffic, such as delay- and loss-sensitive traffic. Our protocol, therefore, does not aim to support a single type of real-time traffic, but rather a plethora of different types of real-time traffic. Elessar is completely distributed, dynamic and adaptive, and does not require the underlying MAC protocol to be QoS-aware.

We analyse our design choices and the performance of our protocol through realistic simulation experiments conducted on the OMNeT++ discrete event simulation platform, using the INET framework. We have used the IEEE 802.11b MAC protocol during our simulations and have employed the random waypoint mobility model to simulate mobility. Our experimental results show that Elessar is able to efficiently provide real-time traffic support for different types of traffic flows, even in the face of mobility. Our protocol operates best for small-to-medium-sized networks where mobility rates are low-to-medium. Once the mobility rate exceeds a certain threshold, intelligent path selection cannot cope satisfactorily with the high dynamism of the environment and the overhead of Elessar exceeds acceptable levels due to its event-driven link-state nature.

Keywords: Wireless ad hoc networks, routing protocol, real-time traffic support, quality-of-service (QoS).

ÖZET

KABLOSUZ MOBİL TASARSIZ AĞLARDA GERÇEK ZAMANLI TRAFİK DESTEĞİ VEREN BAĞ DURUMU TABANLI İSTEĞE DAYALI YOL ATAMA PROTOKOLÜ

Gökçe Görbil
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Assist. Prof. Dr. İbrahim Körpeoğlu
Ağustos, 2007

Geliştirildiklerinden ve kullanıma sunulduklarından beri, kablosuz tasarsız ağlar bayağı rağbet görmektedir ve bir çok kablosuz ağ arayüz kartı tasarsız ağlara destek sağladığı için, bu tip ağ yapıları gerçek hayatta özel amaçlara yönelik olmayan bir çok kullanım alanı bulmuştur. Şu anda gerçek hayattaki en yaygın tasarsız ağ tipi "kablosuz mobil tasarsız ağlar (KMTA)"dır ve bu ağlar, özellikle hareketlilik destekleri ve tasarsız doğalarından kaynaklanan konuşlandırma kolaylıklarından dolayı büyük itibar görmektedir. Web, dosya aktarımı, e-posta ve hızlı mesajlaşma gibi yaygın ağ uygulamaları veri odaklı olduğundan ve sıkı zaman kısıtlamaları altında çalışmadığından, KTMA'lar geçmişte bu tip gerçek zamanlı olmayan uygulamaları olanaklı kılacak şekilde kullanılmıştır. Fakat, telekonferans, IP üzerinden ses aktarımı, güvenlik ve takip uygulamaları gibi vakitliliğin önemli olduğu uygulamaların kullanımı yaygınlaştıkça, çoklu sekmeli KTMA'larda gerçek zamanlı trafik desteği önemli bir konu olarak ortaya çıkmaktadır.

Çoklu sekmeli KTMA'larda gerçek zamanlı trafik desteği veren, olaya ve isteğe dayalı, bağ durumu tabanlı bir yol atama protokolü öneriyoruz. *Elessar* adını verdiğimiz protokol, bağ durumlu topoloji dağıtımına dayanmaktadır, ama daha yaygın olan periyodik bağ durumu mesajlaşması yerine, Elessar'da, ilgilendiğimiz olayların topoloji değişimleri olduğu olay tabanlı bağ durumu mesajları kullanıyoruz. Böyle bir yaklaşımla protokolümüzün getirdiği ek yükü azaltmayı hedefliyoruz. Protokol ek yükünü özellikle diz üstü ve avuç içi bilgisayarları gibi insanlarla doğrudan etkileşim içerisinde bulunan araçlardan oluşan, düşük seviyeli hareketlilik barındıran tasarsız ağlarda azaltmayı amaçlıyoruz. Bağ durumu tabanlı doğasından dolayı, protokolümüz gerçek zamanlı olmayan trafiği herhangi bir ek işlem gerektirmeden desteklemektedir. Gerçek zamanlı trafiği desteklemek için ise sadece ağda bir veya birden fazla gerçek zamanlı trafik akışı olan durumlarda, talebe bağlı olarak çalışan bir doğrudan tutar dağıtım mekanizması öneriyor ve kullanıyoruz. Kaynak rezervasyonu yapmadan, akıllı yol seçimleri sayesinde gerçek zamanlı trafik akışlarına gevşek hizmet kalitesi güvencesi veriyoruz. Hizmet kalitesi güvencelerini, ağ düğümlerinde oluşabilecek aksaklıklara rağmen ve hareketlilik durumlarında bile, dinamik yol ayarlamaları sayesinde trafik akışının ömrü boyunca verebiliyoruz. Elessar, gerçek zamanlı olan ve olmayan trafik akışlarına eş zamanlı destek verebildiği gibi, gecikme duyarlı ve kayıp duyarlı trafik gibi birden fazla gerçek zamanlı trafik akış tipini de eş zamanlı olarak destekleyebilmektedir. Yani protokolümüz sadece tek tip gerçek zamanlı trafik akışına destek vermek yerine bir çok trafik tipini desteklemektedir. Elessar tamamen dinamik, kendinden ayarlamalı ve dağıtımlı olup, aşağıda yer alan katmanların hizmet kalitesi sağlandığının farkında olmasını gerektirmemektedir.

OMNeT++ kesikli olay simülasyon platformu ve bu platform için geliştirilmiş INET iskelet yapısı üzerinde gerçekleştirdiğimiz gerçekçi simülasyon deneyleriyle, aldığımız tasarım kararlarını ve protokolümüzün performansını değerlendirdik. Bu deneylerimizde IEEE 802.11b ortam erişim protokolünü ve hareketliliği simüle edebilmek için rastlantısal yol noktası hareketlilik modelini kullandık. Deney sonuçlarımız göstermektedir ki Elessar, hareketlilik durumlarında bile değişik gerçek zamanlı trafik tiplerini etkili bir biçimde desteklemektedir. Deneylerimiz sonucunda görülüyor ki önerdiğimiz protokol en iyi performansını küçük ve orta ölçekli ağlarda, düşük ve orta hareketlilik seviyeleri için göstermektedir. Hareketlilik seviyesi belirli bir eşiği aştıktan sonra akıllı yol seçimleri ortamdaki yüksek dinamizmle tatminkar bir şekilde baş edememekte ve Elessar'ın getirdiği ek yük kabul edilebilir seviyeleri geçmektedir.

Anahtar sözcükler: Kablosuz tasarsız ağlar, yol atama protokolü, gerçek zamanlı trafik desteği, hizmet kalitesi.

To my parents, Hale and Yavuz Görbil . . .

Acknowledgement

First of all, I would like to express my gratitude to my supervisor, Assist. Prof. Dr. İbrahim Körpeoğlu, for his guidance and support throughout this thesis. I have learned a lot from him and he has been very helpful to me, both professionally and socially. His friendly and pleasant personality has rendered this study more enjoyable.

I would also like to thank Assist. Prof. Dr. Defne Aktaş and Assist. Prof. Dr. Tolga Çapın for taking the time to read and evaluate my thesis.

I am grateful to my parents, Hale and Yavuz Görbil, for their understanding, support, and love throughout this work. They have always believed in me and encouraged me to achieve higher standards and have never withheld from me their care and love. This work would not have been possible without you. I am also thankful for my sister, Müge, for her witty comments and her joyful spirit. She has given me mirth and has been very understanding and supportive.

I would like to thank my labmates, Alper Rifat Uluçınar, Büşra Çelikkaya, Eyuphan Bulut, and especially Berk Berker, for their constructive comments and friendship during this study. It has been a pleasure sharing the same workplace with you.

I hereby thank my friends, Ata Türk and Aylin Tokuç, for their companionship and emotional support.

I am grateful to TÜBİTAK for their financial support during this research. This presented work has been supported under grant TÜBİTAK CARREER 104E28.

And last, but not least, I would like to thank my special friend Meltem Çelebi. I am grateful for all the moments we shared together.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

1.1 Motivation

Wireless networks had been gaining popularity for quite some time now, and wireless LANs employing IEEE 802.11 technology seem to be ubiquitous these days. As the widespread acceptance and deployment of such one-hop wireless networks increase, so do the popularity of wireless multi-hop networks, however at a lower rate of penetration.

There are many wireless multi-hop network types, including mobile ad hoc networks (MANETs) [87], sensor networks [8, 9], and mesh networks [10, 11], to name a few. Among such multi-hop networks, MANETs are the ones that are closest to the end user in terms of deployment and interaction. MANETs are also technically viable these days, meaning they can easily be deployed in real-life by people who are just end users and not researchers. Although several other wireless multi-hop networks may promise wider areas of open research and greater benefits to mankind once they are put to real-life use, MANETs promise actual real-life deployment in the present under conventional roles and in the very near future under various roles in which they are not currently used.

A wireless mobile ad hoc network (MANET) is a self-configuring network that does not require any existing infrastructure to be deployed. Nodes in the network connect to each other and form a network in an ad hoc manner, hence the name ad hoc networks. MANETs are multi-hop networks where each node in the network may act as a router to forward packets on behalf of other nodes. As the name suggests, nodes in a MANET are generally mobile, meaning they can move about during their lifetime in the network. Another source of dynamism for MANETs is the joining and leaving of nodes to and from the network during the lifetime of the network. One final factor which makes MANETs highly dynamic is the changing conditions of the wireless medium due to ongoing flows in the network and transient/permanent interferences. Nodes in a MANET are generally devices that have sufficient energy and processing capabilities. Examples for such nodes are PDAs, laptops, and hand-held cellular devices such as cell phones.

One current application area of MANETs is their use for real-time applications [32, 59, 62]. Such networks may be used in several different roles when deployed for the support of real-time traffic. They may be used to support real-time audio/video applications, such as VoIP, teleconferencing, etc. They may also be used as easily deployable personal security networks for real-time intrusion detection or monitoring systems. When used as an emergency network deployed in times of disaster, military conflict, and/or emergency medical situations, the network may be required to support various degrees of different QoS parameters in order to support delay-sensitive and/or loss-sensitive applications, where timeliness and reliability are of high importance.

We focused our research on the real-time traffic support issue under wireless multi-hop MANETs, trying to improve real-time traffic support in such networks. When undertaking this research, we had in mind whether our proposed solution will actually be feasible in real-life deployment and to what effect we will be able to improve real-time traffic flows in MANETs.

1.2 Our Contributions

Our solution to the problem of real-time traffic support on MANETs focused on the network layer, and we hereby propose a routing protocol named *Elessar*[1] that supports real-time traffic in wireless multi-hop MANETs. Our routing protocol is a link-state based protocol [33, 63], however instead of the more conventional use of periodic link-state updates, we make use of event-driven updates, where topology changes constitute events of interest. We are able to support both real-time and non-real-time (normal) traffic concurrently in the network, where support for real-time traffic is based on an on-demand mechanism [7, 12, 88, 89, 111] which is only initiated when one or more nodes want to send real-time data. This mechanism which enables real-time traffic support is also deactivated when all real-time flows in the network have ended. Therefore our routing protocol operates reactively for real-time traffic, whereas we support normal traffic in a proactive manner. Such normal traffic support is achieved through the event-driven link-state mechanism.

We achieve real-time traffic support through intelligent path selection at the source node. Our protocol does not employ any resource reservations and therefore provides only soft quality-of-service (QoS) guarantees to real-time flows at the moment. However, our protocol may easily incorporate a reservation mechanism that reserves resources along least-cost paths found by the route discovery mechanism of our protocol. The protocol is able to support various different types of real-time traffic concurrently, such as loss-sensitive and delay-sensitive traffic. It should also be noted that our protocol is able to provide almost any type of soft QoS, with the only requirement being that link costs available to Elessar are meaningful representatives of the QoS type requested by a real-time flow. We currently assume that there is an underlying mechanism providing link costs to Elessar periodically, where such link costs may be measurements of link delay, loss rate, available bandwidth, etc.

[1]The name Elessar is derived from the pronounciation of LSR, which stands for link-state routing in short.

We employ source routing in our protocol [7, 12, 88, 89, 111]. The overhead due to embedded routes in packet headers is justifiable since our protocol is targeted towards small-to-medium sized wireless MANETs, having a diameter between 5 and 10. Our protocol supports dynamism resulting from node mobility and dynamic node joins and leaves and it is completely distributed, with no need for centralized components. Elessar is self-adapting to the current conditions in the network and provides QoS throughout the lifetime of a real-time flow, even in the face of node mobility. It requires little functionality from the underlying layers, and may enable several optimizations if relevant functions are available. Elessar currently supports single paths only, but it may easily be adapted to support multipaths as multipath capability is inherent in the design of our protocol.

We strived to design and implement our protocol as realistically as possible. We have implemented Elessar in OMNeT++ [98, 96, 97], a discrete-event simulation environment. In our implementation, for feasibility and ease of deployment concerns in real-life, we chose currently the most common MAC protocol, the IEEE 802.11b MAC protocol [25, 24, 43, 64, 103], as our underlying link layer protocol, instead of more sophisticated MAC protocols [37, 60, 106, 107, 108, 109] which have been developed specifically to support real-time traffic in MANETs but have failed to achieve widespread use. Such concerns were also a factor in our choice of wireless network interface cards, where we opted for currently available technologies instead of more promising, but currently unused technologies.

1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 presents related work on the subject and Chapter 3 presents a simple link capacity calculator that we have designed and implemented for use in our protocol simulations. We present our proposed protocol in Chapter 4 along with various implementation details provided in Chapter 5. We present experimental results obtained from our simulation testbed in Chapter 6 and we conclude and provide directions for future work in Chapter 7.

Chapter 2

Related Work

There is extensive research on wireless ad hoc networks. In this chapter, we mainly focus on two research fields in ad hoc networks: routing and QoS support.

2.1 Routing Protocols for Wireless Ad Hoc Networks

Many different routing protocols have been proposed for use in wireless ad hoc networks. Such protocols are categorized according to different criteria, including when and how they provide routes, whether they operate on flat or hierarchical topologies, whether they provide single or multiple paths, and what type of routing information is exhanged [7, 12, 22, 88, 89].

According to their underlying information mechanisms, routing protocols may be classified as *link-state* [33, 63, 90] and *distance-vector* [33, 63] protocols. In *link-state* routing, routing information is exchanged in the form of link-state packets, where each packet includes link information about the originating node's one-hop neighbors. Such link-state messages may be created periodically or when a link state change occurs. When a link-state message is created, it will be flooded into the entire network. Through this mechanism, every node in the network can construct and maintain a view of the global topology and locally compute routes to all other nodes. The potential problem with link-state routing is high routing overhead due to the globally broadcasted link-state messages.

In *distance-vector* routing, every node maintains a distance-vector table for each destination in the network. Each node exchanges this table with its neighbors periodically. Upon reception of such a distance-vector, a node computes new routes and updates its distance-vector. The complete route from a source to a destination is formed in a distributed manner by the combination of information provided in the distance-vectors of nodes along the path from the source to the destination. The problems with distance-vector routing are slow convergence and possibility of routing loops.

Depending on when the route is computed, routing protocols may be classified as *proactive* and *on-demand (reactive)* routing protocols. *Proactive* protocols compute routes a priori, so when a node wants to send a packet, there is no route acquisition latency as the route has already been computed. The disadvantage is that nodes need to store partial or full topology information and such information needs to be kept up-to-date, creating higher routing overhead. With *on-demand* protocols, on the other hand, a route to a destination is computed only when such a route is needed, creating a latency between the time a packet needs to be sent and the time when such a route is found. The advantage of on-demand protocols is lower routing overhead.

In light of such information, our protocol is a link-state protocol that is a hybrid of reactive and proactive routing. We use proactive routes for all non-real-time data flows and also for real-time flows until a better path is found to support such real-time traffic. We find this better path in order to support real-time traffic in a reactive manner. We aim to lower the route acquisition latency to practically zero with this hybrid scheme, while still keeping routing overhead to a minimum.

Based on when routing information is exchanged between nodes, routing protocols may be classified as *periodical* and *event-driven* update protocols. With *periodical* protocols, routing information is exchanged in a periodic manner, whereas with *event-driven* protocols, such updates are done only when certain events of interest occur. Periodic updates simplify routing protocols and maintain network stability, but the choice of suitable periods is an issue. Event-driven updates incur low overhead when event rate is low, but have the problem of high overhead when event rate is high. Elessar is an event-driven protocol where neighborhood changes of nodes are events of interest, and our event rate is affected by the mobility rate of nodes in the network.

With *flat-structured* routing protocols, every node in the network is at same level and has the same routing functionality. With *hierarchical* protocols, nodes in the network are dynamically organized into partitions called clusters and nodes may have different routing responsibilites based on their role in the cluster. *Flat routing* is efficient and simple for small networks but may cause problems due to initial route acquisition latency for very large networks. *Hierarchical routing* is suitable for large networks but has the additional overhead and complexity of the maintenance of clusters. Since we are dealing with small-to-medium sized ad hoc networks, using a flat routing structure is efficient and suited for our purposes.

Routing protocols may be classified as *source routing* and *hop-by-hop* routing protocols depending on how they route a packet. *Source routing* protocols place the entire route in the header of the packet to be routed at the source node, and intermediate nodes forward the packet according to the route in the header. This has the advantage that intermediate nodes do not need to maintain up-to-date routing information. A disadvantage of source routing is the overhead caused by the embedded routes in the packet header. With *hop-by-hop* routing, each node along the path from the source to the destination decide individually to which node to send the packet. A problem with hop-by-hop routing is that each intermediate node needs to maintain up-to-date routing information. We employ source routing in Elessar to ease the burden of packet processing and forwarding at intermediate nodes and we investigate the overhead caused by embedded routes in packet headers in simulation experiments.

One final criteria by which routing protocols may be classified is the use of *single* or *multiple paths* between source and destination during the lifetime of the flow. Our protocol currently uses single paths between sources and destinations, but it is capable of supporting concurrent multipath communication with little additional effort.

There has been many proposals for routing protocols in wireless ad hoc networks, including DSDV [82], GSR [28], FSR [52], CGSR [30], WRP [78], AODV [81, 83], DSR [56, 57], TORA [80], DST [86], ABR [94], SSA [39], ZRP [45], ZHLS [55], CEDAR [93], OLSR [31], STAR [44], and HSR [52], to name a few. We now take a closer look at some of the widely-accepted protocols related with our work.

Destination Sequenced Distance-Vector Routing (DSDV) protocol [82] is one of the earliest protocols proposed for wireless ad hoc networks. It is proactive in nature, and as its name implies, it employs distance-vector routing based on the classical distributed Bellman-Ford algorithm [19, 20, 36, 58]. The authors of DSDV have set out to design a routing method for ad hoc networks which preserves the simplicity of RIP [34, 48, 73], yet at the same time avoids the looping problem. To combat the routing loops problem, DSDV uses sequence numbers on each route table entry so that nodes can distinguish stale routes from the new ones. Since DSDV is distance-vector based, it requires periodic advertisement and global dissemination of connectivity information and that each node in the network maintain a table of next-hops for each possible destination (each node) in the network.

The *Temporally-Ordered Routing Algorithm (TORA)* [80] is based on the concept of "link reversals", where the protocol operates in a temporally-ordered sequence of computations, with each computation consisting of a sequence of directed link reversals. TORA was designed to be employed in large and dense mobile networks. TORA does not aim to provide optimal paths (i.e. shortest paths) and it does not maintain a route between all (source, destination) pairs. The authors have aimed to design a protocol where reaction to topological changes is minimized. The protocol creates a directed acyclic graph (DAG) for each destination and maintains this DAG using link reversals. The disadvantages of TORA are that it does not provide shortest paths between sources and destinations and it requires temporal ordering of events, which requires either synchronized clocks, which is hard to achieve for large networks or a relative time ordering mechanism, which is subject to errors and introduces situations where the protocol is unable to find a route even when one exists.

Zone Routing Protocol (ZRP) [45] is a flat-structured hybrid protocol developed for MANETs. The protocol uses the concept of "zones" in a flat-structured network, where a routing zone is established around each node according to some number named the *zone radius*. Each node knows the topology of the network within its routing zone only and receives updates only regarding topological changes within its zone. This is achieved through a proactive protocol running within the zone of each node. ZRP may make use of any proactive protocol to establish topology information at each node. Nodes find routes to other nodes outside their zone through a reactive mechanism, using route discovery, which is basically a flooding of the route request to all zones in the network. The route formed through such route discovery is not from node-to-node, but rather from zone-to-zone, which makes it more stable in the face of topological changes. The behavior of ZRP may be adjusted through the setting of the *zone radius*. For a zone radius of 0, ZRP operates in a pure reactive manner, and for a zone radius greater than or equal to the diameter of the network, ZRP is a pure proactive protocol.

The *Ad hoc On-Demand Distance Vector (AODV)* routing protocol [81, 83] is a popular routing protocol proposed for wireless ad hoc networks. AODV is based on a distance-vector algorithm but instead of relying heavily upon global periodic advertisements, it strives to operate on-demand. AODV is able to provide loop-free routes even while repairing broken links through the use of destination sequence numbers as employed in DSDV. AODV is loosely coupled with DSDV since it was designed with the intention of improving upon the performance characteristics of DSDV. AODV operates on-demand, meaning that nodes that do not lie on active paths do not maintain any routing information and do not take part in periodic routing table exchanges. Also, a node does not need to discover and maintain a route to a destination until it needs to send some data, unless, of course, either node is on some other active path. AODV uses a global broadcast route discovery mechanism and hop-by-hop routing. As with all on-demand protocols, AODV suffers from initial route acquisition latency.

DSR, namely *Dynamic Source Routing* [56, 57] is another popular routing protocol for MANETs. It is an on-demand protocol that uses source routing. DSR is composed of two mechanism: route discovery and route maintenance. Route discovery is achieved through global broadcast and each route request packet travelling in the network records in its header over which nodes it has past. When the destination receives such a packet, it sends a route reply to the source node containing the accumulated route in the route request header. When the source node receives such a reply, it has obtained a route from itself to the destination. Route maintenance is activated when an active route is broken. An explicit informing scheme is used as the node that detects the breakage sends a route error message back to the source. When the source node receives such a message, it may initiate another route discovery to find another path. DSR employs various additional features for performance purposes, such as caching overheard routes, early replies to route requests, route request hop limits, etc. DSR has seen widespread implementation due to its low overhead and simplicity. As with all on-demand protocols, DSR suffers from the initial route acquisition latency as well as route repairing latency when a route is broken. DSR also may suffer from higher overhead due to source routing for large networks.

Source Tree Adaptive Routing (STAR) [44] is a proactive routing protocol based on a link-state mechanism. A node in STAR sends to its neighbors its source routing tree in either incremental or atomic updates. Source trees are specified by stating the link parameters of each link belonging to the paths used to reach every destination. Therefore, a node disseminates link-state updates to its neighbors for only those links along paths used to reach destinations. A node broadcasts a link-state message to all its neighbors when its source tree changes. Each node computes its source tree based on information on its adjacent links and the source trees reported by its neighbors. STAR aims to lower the routing overhead associated with link-state protocols by allowing paths taken to destinations to deviate from the optimal (i.e. it uses non-shortest paths). The authors of STAR have shown through simulation experiments that STAR incurs lower overhead than DSR, which is one of the protocols with the lowest routing overhead proposed for wireless ad hoc networks.

Optimized Link-State Routing (OLSR) [31] is another routing protocol for MANETs that uses a link-state mechanism. It is proactive in nature and employs periodic message exchanges to maintain topology information at each node. OLSR aims to optimize the pure link-state mechanism by lowering the size of link-state messages and reducing the number of retransmissions in the network in order to achieve global broadcast. In order to achieve this, instead of using normal flooding for global broadcast, OLSR uses a technique called multipoint relays [54, 65, 53]. The idea of multipoint relays is to minimize the flooding of broadcast packets in the network by reducing duplicate transmissions in the same region. Each node in the network selects a set of its neighbors, and only these neighbors retransmit its broadcast packets. Such neighbors are called the multipoint relays of the node. The authors mention that OLSR is especially suited for large and dense networks with many active flows since the multipoint relays optimization and the link-state mechanism show their true values in this context. OLSR uses hop-by-hop routing and provides shortest paths.

All of the protocols discussed so far, with the possible exception of TORA, use a single path between source and destination. However, some of the protocols discussed above may easily be extended to support multipaths, and we would like to mention some of these here. *Split Multipath Routing (SMR)* [67] is an on-demand routing protocol that builds maximally disjoint paths. *AOMDV* [74] is a multipath extension to AODV which finds multiple loop-free link-disjoint paths. *AODVM* [105] is another multipath extension to AODV which finds multiple node-disjoint paths. *MSR* [99] is a multipath extension to DSR, where traffic is distributed among multiple paths according to the measurement of the round-trip-time of every path. MSR aims to achieve load balancing among multiple paths by this approach. *MP-DSR* [68] is another multipath extension to DSR, which is distinguished from MSR by the fact that MP-DSR is QoS-aware, attempting to provide end-to-end reliability as the QoS metric. We discuss MP-DSR in more detail in the next section.

2.2 QoS Support in Wireless Ad Hoc Networks

QoS support is a popular topic in wireless ad hoc networks and many different approaches have been proposed in the literature on the topic. Some works aim to provide a QoS framework that enables the network to support QoS traffic. Such frameworks may be considered in two different classes depending on whether the framework is complete or not. Complete frameworks aim to provide a full framework, from the application layer down to the link layer, where each layer is QoS-aware and cooperate with surrounding layers for efficient QoS support. Incomplete frameworks concentrate on one or more specific layers, such as the interaction between the transport layer and the network layer. We place frameworks that provide necessary tools and facilities for QoS support in this category.

Other works on QoS support focus on a single aspect of the problem, just focusing on a single layer, such as the transport or the network layer. Our approach falls into this category as we mainly focus on the network layer, striving to provide a Qos-aware routing protocol.

In this section, we first take a brief look at some of the proposed QoS frameworks and then provide information on routing protocols aiming to support QoS in wireless ad hoc networks.

The *DiffServ* [21] model is an architecture that specifies a simple, scalable and coarse-grained mechanism for classifying and managing network traffic and providing QoS guarantees on modern IP networks. DiffServ classifies traffic based on their requirements and orders these classes according to their priorities. DiffServ does not differentiate between individual flows but rather between classes, treating each packet based on its class and not its flow, so it's a class-based, coarse-grained approach for QoS support. DiffServ only provides a framework for traffic classification and differentiated treatment, but does not impose any rules on parameter selection for traffic classification or on how different classes of traffic should be treated by the network. DiffServ was not specifically designed for wireless ad hoc networks but it has seen fair use in MANETs due to its lightweight nature.

FQMM [102] is the first QoS model designed specifically for MANETs. It combines the high quality QoS of IntServ [23] and the service differentiation of DiffServ. In FQMM, traffic of the highest priority is given per-flow provisioning while other classes are given per-class provisioning. The problem with FQMM is that the model used for per-flow provisioning, IntServ, is not suitable for the highly dynamic nature of MANETs, so a better model is needed for flow-based traffic differentiation.

INSIGNIA [66] is an IP-based QoS framework for MANETs, designed to be lightweight and highly responsive to changes in network topology, node connectivity, and end-to-end conditions. Its in-band signaling mechanism, soft-state resource management, and fast reservation make it more suitable than IntServ for use in MANETs. Since INSIGNIA provides per-flow granularity, it may suffer from scalability problems when the number of flows in the network is large.

HQMM [47] is a hybrid QoS model proposed for MANETs that combines the per-flow granularity of INSIGNIA and the per-class granularity of DiffServ in order to provide a responsive and scalable QoS model. Just as in FQMM, HQMM provides per-flow provisioning for traffic with the highest priority while providing per-class provisioning for other traffic. Instead of employing IntServ for per-flow provisioning, HQMM employs INSIGNIA, while both FQMM and HQMM use DiffServ for per-class traffic provisioning.

Of course, there are many more works on QoS support in wireless ad hoc networks, and interested readers may wish to take a look at references [1, 2, 3, 17, 26, 41, 72, 75, 85, 91, 101, 110].

We have mentioned *MP-DSR* in the previous section. MP-DSR [68] is a QoS-aware multipath extension to DSR, aiming to provide end-to-end reliability as the QoS metric. The authors define end-to-end reliability as the probability of sending data successfully within a time window. End-to-end reliability is calculated from the reliabilities of the paths used for sending data and the reliability of a path is calculated from the availabilities of its links. *AODVM* [105] was another multipath protocol mentioned previously. AODVM is a multipath extension to AODV and it intends to provide end-to-end reliability as its QoS metric, just as MP-DSR does. However, AODVM is proposed for heterogeneous ad hoc networks, where some nodes are more reliable than others. AODVM tries to find a reliable path from a source to a destination where a reliable path is either one that consists entirely of reliable nodes or one that consists of reliable nodes connected by multiple node-disjoint paths.

[29] proposes an on-demand, link-state, multipath QoS routing protocol for MANETs. The QoS requirement this protocol tries to satisfy is bandwidth, and it aims to achieve this goal through the use multiple paths between source and destination. The protocol runs over a CDMA-over-TDMA channel [71], reactively collecting link-state information from source to destination in order to construct a partial topology at the destination. The destination then chooses multiple paths from the source to itself which collectively satisfy the bandwidth requirement and informs the source node of these paths. Link-state information from the source to the destination is collected through globally broadcasted route request packets. Upon reception of multiple route request packets that have accumulated link-state information on them, the destination has a partial topology from the source to itself, which forms a flow network. On this topology, the destination finds multiple paths, collectively satisfying the total bandwidth requirement and replies back to the source with multiple route replies following the reverse of the chosen paths. Each route reply confirms and reserves the bandwidth on the way back to the source.

For more QoS-aware routing protocols for wireless ad hoc networks, please see [4, 5, 6, 14, 15, 16, 18, 27, 40, 42, 46, 49, 50, 51, 61, 69, 70, 71, 77, 79, 84, 92, 104].

Chapter 3

Available Link Capacity Calculation

In this chapter, we present an algorithm to calculate residual available link capacities (bandwidths) of wireless links in a wireless network with ongoing data flows. Due to the broadcast effect of the wireless medium, the calculation of residual capacities is a non-trivial task and also depends on the MAC protocol used. Not only the capacities of links that the data flows are flowing over, but also capacities of nearby links will be reduced. We solve this problem under the assumptions of a single shared wireless channel and a perfect MAC protocol at the link layer.

3.1 Introduction

Wireless networks is a particularly popular and rich area for research due to its various application scenarios and benefits for the general public. Many researchers are working on different types of wireless networks, including wireless mesh networks, mobile ad hoc networks and sensor networks, to name a few. Each of these network types have fundamentally different requirements and application scenarios. However, they also share many common properties. Any researcher who has developed a scheme or protocol for use in any layer of these networks will invariably want to test the correctness and efficiency of his/her scheme. One useful tool to achieve this is network simulators. Simulators allow a researcher full control over the various parameters while testing the protocol, enable re-runs of the protocol with the same parameters, provide automated testing schemes, etc. They allow for a thorough testing of the protocol and shorten the testing and debugging phase before probable actual deployment of the protocol. Simulators also provide a means to observe the efficiency of the protocols under various conditions.

As powerful as simulators may be, simulation tools may lack certain gadgets that a researcher requires. Such gadgets may be very small and simple, but essential to the research. In other cases, a simulation package may include such tools and gadgets, but to spend the necessarily long time to learn such complex simulation tools in order to achieve preliminary results may be unjustified.

Considering such issues, we set out to provide an algorithm and a small software tool that enable the calculation of available link capacities (bandwidths) on wireless links in the face of active nodes. To be precise, when there are ongoing data flows across the wireless network, available link capacities (bandwidths) will naturally decrease along the path(s) used by the flows. However, due to the peculiar characteristics of the wireless medium, such as the broadcast property, not present in wired media and the properties of the medium access scheme used, determination of which wireless links in the network are affected by the current flows and to what extent are the effects may not be as trivial as it seems at an initial glance.

In this chapter, we provide a centralized algorithm for the calculation of available link capacities in a (generic) wireless network with current data flows. Our aim is not to achieve such calculation in a real network, but rather to find the available link capacities in a simulation setting. Therefore, using a centralized algorithm to achieve our goals may be easily justified in this case. We have also implemented this algorithm in two different settings, once as a software tool written in C++, using the MS Visual Studio .NET 2003 software package, intended for use in computers with Windows XP operating systems; and once as part of our simulation in OMNeT++, running on Linux operating systems. Please refer to Section 5.1 for a more detailed discussion on OMNeT++.

This chapter is organized as follows. In the next section, we provide our assumptions on the network model and the wireless channel properties and the rules we have derived based on these assumptions. In section 3, we present our algorithm. We conclude the report and present ideas for future development in section 4.

3.2 Assumptions and Rules

We have made several assumptions on the wireless channel properties. We present them below.

- There is a single, wideband channel shared by all nodes in the network.
- Half-duplex channels are assumed. Therefore, a node cannot both transmit and receive at the same time.
- A perfect medium access control (MAC) protocol is assumed. Therefore, the hidden and exposed terminal problems are effectively solved.
- The MAC protocol incurs no extra overhead and allocates and uses the wireless channel efficiently and fairly. Therefore, the whole bandwidth may be used for data traffic.

The hidden and exposed terminal problems and our assumptions on how the MAC protocol handles these problems is given below. In Figure 3.1, a very simple network topology is presented. While *node 2* is transmitting to *node 3*, *node 4* is a hidden node to *node 2*. Therefore, *node 4* should not transmit for this duration. However, *node 4* should be able to receive from *node 5*. Similarly, while *node 2* is transmitting to *node 3*, *node 1* is an exposed node. It should be able to transmit to *node 0*, although it will not be able to receive anything except from the transmission of *node 2* due to the broadcast nature of the transmission from *node 2* to *node 3*.

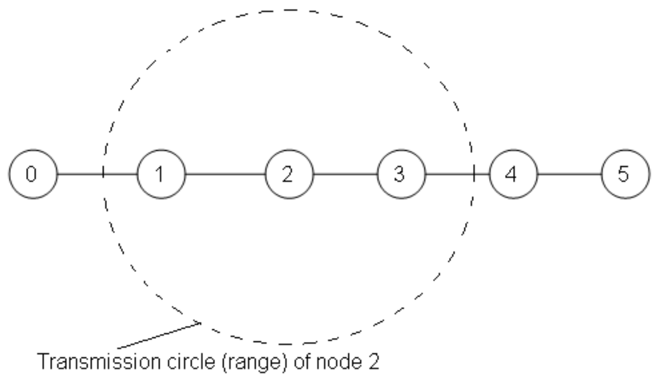

Figure 3.1: The hidden and exposed terminal problems.

The following rules have been identified based on the previous assumptions:

1. One-hop neighbors of the sender are not able to receive from other nodes. Note that they may receive and use the current transmission of the sender, even though the transmission may not be destined for themselves.

2. One-hop neighbors of the sender (except the receiver) must be able to send to other nodes (except to the sender).

3. One-hop neighbors of the receiver (except the sender) are not able to send to other nodes.

4. One-hop neighbors of the receiver (except the sender) must be able to receive transmissions from other nodes.

5. A transmitting node cannot receive at the same time.

6. A receiving node cannot transmit at the same time.

3.3 The Algorithm

Based on the rules stated in the previous section, the actions performed in order to calculate the link capacities are as follows:

1. Decrease the capacities of all outgoing edges of the sender.
2. Decrease the capacities of all incoming edges of the sender.
3. Decrease the capacities of all previously unprocessed incoming edges of one-hop neighbors of the sender.
4. Decrease the capacities of all previously unprocessed incoming edges of the receiver. Note that this action is already accomplished (i.e. all the incoming edges of the receiver have already been processed) by action 3 above. Therefore, we ignore this action in our algorithm specification.
5. Decrease the capacities of all previously unprocessed outgoing edges of the receiver.
6. Decrease the capacities of all previously unprocessed outgoing edges of one-hop neighbors of the receiver.

These actions are performed for each hop of each flow present in the network. Note that the sender and receiver nodes change for each hop of a flow.

We represent the wireless network as a simple directed graph $G = (V, E)$. Each wireless node corresponds to a node in the graph, and if node i can transmit to node j, then edge $(i, j) \in E$. Note that our implementation works on both directed and undirected graph representations, as we may transform an undirected graph to an equivalent directed graph in a very straightforward way. Most wireless networks will probably have bidirectional links due to identical transceivers, meaning that if node i can reach node j, then node j may reach node i (i.e. $(i, j) \in E \Leftrightarrow (j, i) \in E$). However, the characteristics of such links may be quite different from each other. For example, the noise of link (i, j) may be higher/lower than the noise of link (j, i). Therefore, even though the links are bidirectional, they may be asymmetric.

Even though we said that most wireless networks are bidirectional, there are certain applications of wireless networks where this property does not hold. In such cases, the graphs representing the networks may have unidirectional links, where the property $(i, j) \in E \Leftrightarrow (j, i) \in E$ no longer holds.

Our implementation is able to work on all types of networks, meaning that it supports both bidirectional and unidirectional links, both symmetric and asymmetric links, and both directed and undirected graph representations. In the rest of this report, we will focus on the more general case of directed graphs with unidirectional, asymmetric links.

Each node in the graph has a unique integer ID > 0. We represent the neighbors of each node in the adjacency list representation. The adjacency list of a node $i \in V$ is represented as $Adj[i]$. The current capacities of all edges are assumed to be kept in a table. We represent the capacity of edge (i, j) as $cap(i, j)$. Each flow in the network is represented as a path in the graph with its associated flow value (the amount of bandwidth used by the flow). We keep all current flows in G in a list, called $flows$. Each flow's flow value is kept in a table, where the flow value for flow $f \in flows$ is denoted by $val(f)$. The path of a flow f is denoted by $P(f)$. Figure 3.2 presents two flows on an example topology. The first flow's source is node 2 and its destination is node 6. The flow follows the path $2 \rightarrow 3 \rightarrow 4 \rightarrow 6$. Denoting this flow as f_1, $P(f_1) = \{(2,3),(3,4),(4,6)\}$. The second flow f_2 follows the path $1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6$, so $P(f_2) = \{(1,3),(3,2),(2,4),(4,5),(5,6)\}$.

Given a set of flows as $flows$, with paths and values for each flow, on a simple directed graph $G = (V, E)$, we wish to calculate the residual link capacities available. The **calculateCapacities** algorithm presents a way to achieve this goal. The algorithm uses a set structure S to keep track of processed edges so that an edge is not processed multiple times unnecessarily. We assume that the capacities of links are initialized to their correct values at network initialization.

Algorithm 1 calculateCapacities

```
1: $S \leftarrow \emptyset$

2: for each flow $f \in flows$ do
3:     for each edge $(i, j) \in P(f)$ do
4:         $S \leftarrow \emptyset$

5:         for each node $k \in Adj[i]$ do                          ▷ perform action 1
6:             $cap(i, k) \leftarrow cap(i, k) - val(f)$
7:             $S \leftarrow S \cup \{(i, k)\}$
8:         end for

9:         for each node $v \in V$ do                               ▷ perform action 2
10:            if $i \in Adj[v]$ then
11:                $cap(v, i) \leftarrow cap(v, i) - val(f)$
12:                $S \leftarrow S \cup \{(v, i)\}$
13:            end if
14:        end for

15:        for each node $k \in Adj[i]$ do                          ▷ perform action 3
16:            for each node $v \in V$ do
17:                if $k \in Adj[v]$ and $(v, k) \notin S$ then
18:                    $cap(v, k) \leftarrow cap(v, k) - val(f)$
19:                    $S \leftarrow S \cup \{(v, k)\}$
20:                end if
21:            end for
22:        end for

23:        for each node $k \in Adj[j]$ do                          ▷ perform action 5
24:            if $(j, k) \notin S$ then
25:                $cap(j, k) \leftarrow cap(j, k) - val(f)$
26:                $S \leftarrow S \cup \{(j, k)\}$
27:            end if
28:        end for

29:        for each node $k \in Adj[j]$ do                          ▷ perform action 6
30:            for each node $l \in Adj[k]$ do
31:                if $(k, l) \notin S$ then
32:                    $cap(k, l) \leftarrow cap(k, l) - val(f)$
33:                    $S \leftarrow S \cup \{(k, l)\}$
34:                end if
35:            end for
36:        end for
37:    end for
38: end for
```

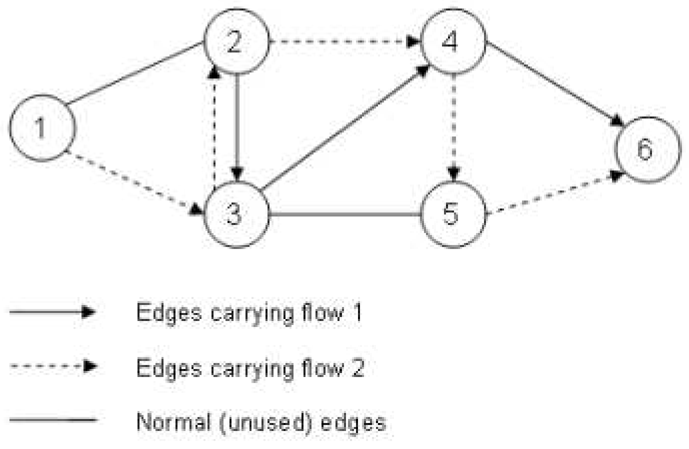

Figure 3.2: Example topology with two flows.

3.4 Future Work on the Link Capacity Calculator

We have presented an algorithm for calculation of available link capacities (bandwidths) of wireless links in a wireless network with ongoing data flows between pairs of nodes in the network. We have assumed a single wireless channel shared by all nodes and a perfect MAC protocol with no overhead. The proposed algorithm is aimed at helping researchers assign meaningful, real-life-like link capacities to edges for simulations and at aiding them find out about available link capacities without detailed simulations. By the proposed method, researchers would be able to gain essential knowledge of the network at an early stage in simulation, prior to developing their protocols to full extent and engaging in detailed simulations.

In a perfect world, our assumption of a perfect MAC protocol would be easily justified. However, we are not living in a perfect world, so our assumption is questionable. Even so, the presented algorithm is able to provide a general idea of the states of wireless links in a wireless network with ongoing data flows. Among ideas for future work, the inclusion of more MAC protocols with real-life characteristics holds the highest priority. We are currently focusing on MAC protocols that work on a single shared channel. A second major step in the development of the presented tool may be the consideration of MAC protocols that work on multiple wireless channels. Our tool is currently console-based; addition of a user-friendly graphical interface would probably be a less technically challenging, albeit a most welcome contribution for users of the tool.

Chapter 4

Our Proposed Protocol

4.1 Introduction

Elessar is an on-demand link-state routing protocol designed specifically for multi-hop wireless mobile ad hoc networks. Elessar supports both normal traffic and real-time traffic requiring QoS in the network. Elessar does not require any existing network infrastructure or administration and allows a completely self-organizing and self-configuring network. All wireless nodes in the network may not be in direct communication range of each other, so a packet from a source node to a destination node may go over multiple hops. In order to enable such communication, each node in the network acts as a router and relays packets on behalf of other nodes. The network topology changes due to node mobility and new nodes joining the network and nodes leaving the network. The topology changes may also be due to changes in the conditions of the wireless medium, i.e. due to interference. As the topology and such conditions change, Elessar dynamically adapts to these changes.

Elessar supports both normal and real-time traffic in the network. For normal traffic, the protocol operates in *normal mode.* For real-time traffic requiring QoS guarantees, Elessar operates in *QoS mode*, performing additional activities on top of the normal mode to enable efficient and optimal routing of real-time traffic, trying to fulfill the required QoS guarantees. It should be noted that Elessar does not employ resource reservation and it provides only soft QoS guarantees to real-time traffic flows.

Local path finding/selection and source routing is employed in Elessar. Each node knows the current network topology due to the link-state messages, and a node wishing to send a packet to another node may run a local path finding algorithm to find the path the packet must follow. This path is embedded into the packet header, and intermediate nodes forward the packet according to this path in the header. Since a local path finding algorithm along with source routing is used to find the paths, packet routing is trivially loop-free. Through the use of source routing, intermediate nodes do not need to maintain routing information for flows that are passing over them.

Elessar consists of the following mechanisms:

- Neighborhood beaconing
- Topology dissemination
- Directed cost dissemination
- Route discovery
- Route maintenance

The interactions between these mechanisms may be found in Figure 4.1. In 4.3, we explain each mechanism in detail.

4.2 Assumptions

We assume that the diameter of an ad hoc network will often be small (e.g. between 5 and 10). Packets may be lost or corrupted during transmission in the ad hoc network. We assume that a node receiving a corrupted packet can

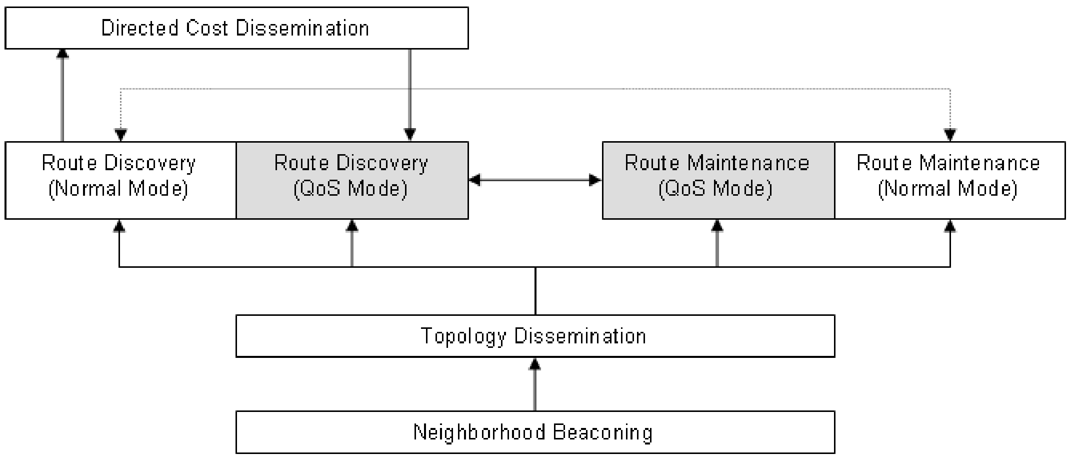
Directed Cost Dissemination
Route Discovery (Normal Mode)
Route Discovery (QoS Mode)
Route Maintenance (QoS Mode)
Route Maintenance (Normal Mode)
Topology Dissemination
Neighborhood Beaconing

Figure 4.1: Protocol mechanisms and interactions.

detect the error. Nodes within the ad hoc network may move at any time without notice; they may even move continuously. However, we assume that the speed of mobility is not so high as to render smart routing impossible. The discussion here focuses on operation of Elessar on *bidirectional* links in the wireless network, leaving the discussion of operation on unidirectional links to later sections in this chapter. We assume that nodes may enable *promiscuous* receive mode on their wireless network interface card (NIC), causing the hardware to deliver every received packet to the network protocol stack without filtering packets based on their link-layer delivery addresses. Elessar does not require this facility, but it may enable some optimizations if this option is available.

We assume that there are one or more mechanisms measuring link parameters and providing these as link costs to Elessar. More specifically, we assume that there are one or more mechanisms providing each node information about its links. We do not currently concern ourselves with the exact schemes used in the measurement of these link parameters. We only require that these mechanisms provide their measurements periodically and that each of these measurements represents the current condition of the links. Elessar will make use of these link parameters in route selection in order to support real-time traffic. By acquiring such parameters periodically, Elessar will be able to react to changes in network conditions dynamically and adapt itself to these changing conditions by providing different, more desirable routes as conditions change. Parameters of interest include loss probabilities, delays, and available bandwidths of links. We have mentioned that Elessar is able to support more than one type of real-time traffic. In order to support any type of real-time traffic effectively, link costs available to Elessar must be meaningful for the supported type of traffic. For example, in order to support delay-sensitive traffic, Elessar must have knowledge of link delays. Another example may be loss-sensitive traffic. For supporting such traffic, link loss probabilities must be available to the protocol.

Each node in the network keeps a local history about its link parameters. In this history, a node keeps the following fields for each link parameter and for each of its links: a *last update time* field which keeps the local reception time of the last update of the parameter, a *current value* field which keeps the latest measurement of the parameter, and an *average value* field which keeps the EWMA[1] of all the measurements received so far for this parameter.

4.3 Protocol Specification

In this section, we provide details on the mechanisms of the protocol. Elessar consists of five mechanisms that altogether allow efficient best-effort and real-time traffic routing.

Neighborhood Beaconing This is the mechanism by which nodes learn of their neighbors. This mechanism is active all the time and it employs periodic beacons between neighbors. This mechanism operates the same under both normal and QoS operation modes.

Topology Dissemination This is the mechanism by which nodes learn of the current topology of the whole network. Topology information is exchanged in the form of network-wide link-state messages, triggered whenever a topology change occurs. The topology dissemination mechanism is continuously

[1]EWMA: Exponentially Weighted Moving Average

active. This mechanism performs the same operations under normal and QoS modes.

Route Discovery Whenever a node S wishes to send a packet to a node D, the route discovery mechanism is employed in order to find a source route from S to D. Since each node has complete knowledge of the network topology, S runs a local path finding algorithm in order to find a path to D. In case of route caching, route discovery is performed only when S does not already have a valid route to D in its cache. This mechanism operates differently under normal and QoS operation modes.

Route Maintenance Route maintenance is the mechanism by which sender node S is able to learn of a route failure during a session to destination node D. Route maintenance is only employed when S is actively transmitting to D. This mechanism is also responsible for actions to be performed in order to recover from such route failures. This mechanism currently operates the same under normal and QoS operation modes, but different schemes may be used that differentiate the operation of the mechanism under different operation modes.

Directed Cost Dissemination Directed cost dissemination is the mechanism which enables a sender node S to use intelligent paths for real-time routing to destination node D. Through directed cost dissemination, S receives messages including current link cost information from nodes in the whole network or a subnetwork, and due to this newly acquired knowledge of link costs, S is able to find optimal paths under the QoS parameter(s) required by the real-time traffic flow. Directed cost dissemination is activated only when S is actively sending a real-time traffic flow to D. This mechanism is only part of the QoS operation mode.

The first four mechanisms are present in both normal and QoS operation modes of Elessar. The fifth mechanism, namely directed cost dissemination, is only part of the QoS operation mode. With the help of directed cost dissemination, Elessar is able to give real-time traffic support, providing soft QoS guarantees through intelligent path selection. Elessar is also able to dynamically adapt to changes in the network topology and link conditions, providing soft QoS guarantees throughout the lifetime of the real-time session by dynamic path finding/selection.

4.3.1 Neighborhood Beaconing

Neighborhood beaconing is the mechanism by which nodes learn of their immediate (one-hop) neighbors. This mechanism is active all the time and operates the same in both normal and QoS operation modes. This mechanism employs periodic beacons between neighbors for neighbor discovery/update. Each node in the network periodically broadcasts a message locally. Through such periodic message exchange, nodes learn of the condition of their neighbors.

Each node has a beacon timer. When the beacon timer of a node expires, the node creates a beacon message with the following fields:

- *Node ID* This is a number/string uniquely representing the creator of the beacon message.
- *Sequence number* This is a number used to identify the beacon message. It monotonically increases each time this node sends a beacon message. This sequence number is local to the node.
- *Beacon type* This field represents the type of the beacon message. The beacon message may be of four types: *join, leave, normal, reply.* A new node wishing to join the network uses the "join" beacon type. A node wishing to leave the network uses the "leave" beacon type. A node local-broadcasting a beacon uses the "normal" beacon type. When a node needs to send a reply in response to a beacon, it uses the "reply" beacon type. We will discuss the specific uses of beacon types below.

Each node has a neighbor list that includes its currently known neighbors. Each neighbor in the list is represented by its unique node ID. Each entry in the list also includes a timestamp field that tells when the entry expires. A node periodically removes expired entries from its neighbor list. When a node receives a beacon message from another node, it knows that the sender is its neighbor and that it is currently alive. If the sender was not previously in its neighbor list, it adds this node to its list and sets its expiration time accordingly. If the sender is already in the list, it updates the expiration time for the entry corresponding to that node. The pseudocode for the neighborhood beaconing mechanism is provided in the implementation details section.

If nodes are able to enable *promiscuous* mode on their wireless NIC, Elessar may use various optimizations in the neighborhood beacon exchange, reducing the number of beacon messages. Since the wireless medium is a broadcast medium, a unicast message is actually locally broadcasted. Therefore, when a node is sending any packet, other nodes may overhear this message. Any node overhearing such a packet knows that the sender of the packet is its neighbor and that it is alive, so it may update its neighbor list accordingly. So, whenever a node sends any type of message, it may restart its beacon timer since the currently sent message lets its neighbors know that it is alive, performing the essential job of a beacon message. Figure 4.2 illustrates this *passive beaconing* that may be enabled if promiscuous mode is available. In this figure, node 1 is unicasting a packet to node 2, but due to the broadcast medium property, its neighbors node 0 and node 3 overhear this transmission and all of node 1's neighbors, including node 2, determine that node 1 is their neighbor and that is alive. The modifications to the neighborhood beaconing algorithm when promiscuous mode is enabled is provided in the implementation details section.

The neighborhood beaconing mechanism also handles node joins and departures to/from the network. A new node wishing to join the network locally broadcasts a beacon of type "join". Any node receiving such a message replies with a beacon of type "reply". The reply message to a "join" beacon may include additional information about the network that the new node needs for correct operation. Through this message exchange, the new node learns of its neighbors and the nodes in its vicinity learn of the addition of this new node to the network. The reply beacon is utilized here in order to facilitate quick and correct

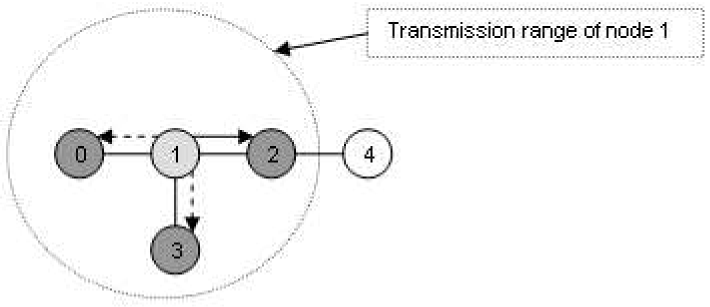

Figure 4.2: Passive beaconing.

integration of the new node into the network. Since its neighbors immediately send a reply beacon, the new node learns of its neighbors without having to wait for their individual normal beacons. In addition, the reply beacon allows the new node to learn of various network parameters that are needed for its correct operation as part of this network.

A node wishing to leave the network first locally broadcasts a beacon of type “leave”. After sending out this beacon, the node may leave the network. Any node receiving the “leave” beacon just removes the node from its neighbor list. Note that nodes receiving a “leave” beacon do not reply to the beacon. This situation is an example of a *graceful* departure from the network, where the leaving node informs its neighbors of its departure before-hand. A node may also leave the network ungracefully, perhaps due to node failure. Notice that a leaving node may leave the network gracefully for some nodes and ungracefully for others, as its “leave” beacon may not be received correctly by some nodes due to collisions and/or errors. An ungraceful leave is accomplished by the expiration times of entries in the neighbor list of nodes. Any entry which expires is removed from the neighbor list.

We would like to note that if the underlying MAC layer or any other mechanism provides a similar neighborhood information service, then the neighborhood beacon mechanism described here may be deactivated. It should also be noted that there are various improvements or variations that may be incorporated into the neighborhood beaconing mechanism, most notably in the periodic messaging scheme. In the most basic scenario, each node has the same period and this period is constant (hardcoded) in the protocol. The periods may be the same for all nodes but the period may be set by the network administrator or according to some protocol parameter, so that for each different network, the period is different. There are scenarios where the periods are the same for all nodes but these periods change dynamically during the lifetime of a network. Periods may be different for different nodes, where the periods are chosen according to the local conditions in parts of a network. Finally, periods may be randomized, where the period choice window may be the same for all nodes or different for each node. We briefly mention the possibilities here and leave the discussion and evaluation of such detailed issues as future work.

4.3.2 Topology Dissemination

Topology dissemination is the mechanism by which each and every node in the network learns of the whole network topology, and is able to update its view of the topology as it changes. This mechanism is essentially an event-driven link-state mechanism, with the events being topological changes due to node mobility, node/link failures, and node joins/departures to/from the network. The neighborhood beaconing mechanism is responsible for neighborhood discovery. Through neighborhood beaconing, nodes maintain their neighbor lists. Each time a change occurs in this neighbor list (addition of a new entry and/or removal of an old entry) due to neighborhood beaconing, a new topological change has occurred in the neighborhood of the owner of the neighbor list. Such changes constitute the events of interest, and the topology dissemination mechanism is triggered by these events. The node that detects such an event initiates a network-wide global broadcast to disseminate this information to all nodes in the network. Through this broadcast in the form of link-state messages, every node in the network is able to learn of the change in the current topology and update its view accordingly.

Each node in the network keeps its own global view of the network, meaning that each node knows the whole network topology. Of course, due to transmission errors, collisions, and the dynamic nature of the network, every node will not have the same and correct view all the time during the lifetime of the network. Our aim is to minimize the number of nodes with incorrect/incomplete views and the total amount of time the discrepancy exists in their views of the network. We wish to accomplish this goal with a basic network-wide broadcast mechanism, reacting to changes in the topology as soon as possible and making sure that these changes are propagated in the network as fast as possible. However, while trying to accomplish this goal, we also pay attention to the inherent problems with a broadcast mechanism, namely waste of resources (e.g. bandwidth, energy). We will briefly discuss in this section how Elessar tries to tackle this problem.

We have said that each node has its own view of the network. The network is represented as a directed graph at each node, and the main responsibility of the topology dissemination mechanism is to keep this representation as accurate as possible. The accuracy of this representation is quite important as the route discovery and maintenance processes depend heavily on a node's view of the network. These processes eventually will accomplish the task of routing in the network and any inefficiency in the routing process will degrade the performance of the protocol.

The details of the topology dissemination mechanism along with various considerations are provided below.

4.3.2.1 Link-State Message Content

Topology information will be exchanged in the form of link-state messages that will be disseminated throughout the whole network by the broadcast mechanism. There are various possibilities for the content of these messages, which are listed below:

- Whole network information vs. neighborhood information
- Topology information only vs. topology information and link costs
- Multiple costs vs. single cost

- Incremental messages vs. whole messages

These possibilities are described in more detail in the following discussion.

4.3.2.1.1 Whole network information vs. Neighborhood information: One option is a node originating a link-state message to include its whole view of the (updated) network in the message. This option is obviously very inefficient, especially for large networks. Inclusion of the whole network topology in link-state messages will increase packet length greatly, and when we consider that these messages are globally broadcasted, it can be seen that the overhead of these messages will be quite high with this option. This option may be desirable for small networks, where the overhead may be acceptable and the additional information provided in link-state messages may serve as a more robust topology information dissemination mechanism. As another option, a node originating a link-state message may include only (the IDs of) its neighbors in the message. With this option, local topology changes will be effectively disseminated in the whole network. The overhead of this option is quite lower than the previous one, as now an originating node includes only local information in a message. With this option, packet length of a link-state message is independent of the network size, but rather dependent on the degree[2] of a node. It should be noted that this is the option employed by most link-state protocols. It is also the option employed by Elessar.

4.3.2.1.2 Inclusion of link costs in link-state messages: We have mentioned in the assumptions section that each node in the network is periodically receiving information about its links from a measurement mechanism and that it is keeping a local history of these parameters. In the event-triggered link-state messages, nodes may include the averaged values of these link costs. The inclusion of link costs come at the price of more control overhead due to the increased packet lengths of link-state messages. When we include such link costs, we may enable a node to perform intelligent route selection even in normal operation

[2]Degree of a node is defined to be the number of its neighbors in a network.

mode. If a node knows the link costs of all the links in the topology, instead of finding the minimum-hop route, it may find the least-cost route. We should note that the found route is selected according to the condition of the network known to the node at the time of route selection, and since link-state messages are not periodic, but rather event-driven where the events are topology changes, a node does not receive continuous up-to-date information about all link costs in the network. Therefore, the network view of a node at the time of route selection may not be up-to-date in terms of link costs, so the advantages of inclusion of such link costs in link-state messages may be debatable. Furthermore, since a node will not receive continuous information about all link costs, it may not be able to adjust the used routes dynamically to changing link conditions. Considering that the job of the normal mode of Elessar is basically to transfer a packet from a source node to a destination node efficiently, we may argue that in this operation mode, we do not require knowledge of all link costs. Taking into account such issues, we have decided not to include link costs in link-state messages.

4.3.2.1.3 Multiple costs vs. Single cost: If a node includes its link costs in link-state messages, another issue that arises is which of these costs should be included in the messages. Recall that a node is receiving periodic information about its links. The received information may be about a single link parameter, such as link delay, or it may be about multiple parameters, such as delay and bandwidth. Considering that a node may be receiving information about more than one link parameter, a question that comes to mind is which of these parameters should be included in link-state messages, if link costs are included. There are various answers to this question. A node may include all link parameters as costs or it may include a subset of these parameters as costs. If a single link cost is desired, then an aggregate of multiple parameters may be used to represent the condition of the links. In this case, an additional question is how to calculate the aggregate cost. We have discussed the benefit of the inclusion of link costs in link-state messages in the above paragraph. Based on that discussion, the disadvantages of the inclusion of multiple costs seem to outweigh its advantages. Therefore, we believe that if a node has information about multiple link parameters and if link costs are to be included in link-state messages, then instead of the inclusion of multiple costs, a single cost should be included in these messages. This single cost may be one of the multiple costs that is best suited for optimal path selection in normal mode, or it may be an aggregate cost representative of the multiple costs.

4.3.2.1.4 Incremental messages vs. Whole messages: We have mentioned that a link-state message may include the whole topology information as seen by a node or the neighborhood information for that node only. No matter what the essential content of a link-state message is, we have the option of only sending information regarding changes that have occurred since the last link-state message sent by that node or sending all of the information about the current state of the node. We refer to the first option as an *incremental* message and the second option as a *whole* message. Figure 4.3 tries to illustrate the idea more clearly.

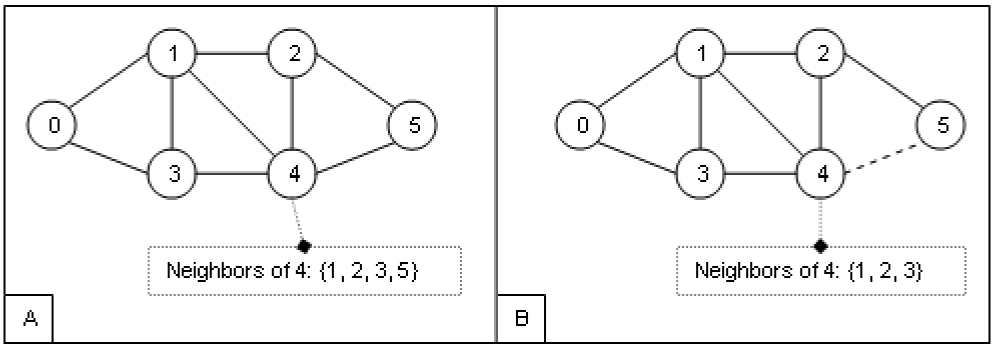

Figure 4.3: Topology dissemination: neighborhood change at node 4.

In Figure 4.3, part A, node 4 has four neighbors: $\{1, 2, 3, 5\}$. Assuming that the wireless link between nodes 4 and 5 has failed (due to interference or the mobility of node 5) at part B of the figure, the new neighbors of node 4 are $\{1, 2, 3\}$ only. Since node 4 has detected a topology change due to this link failure, it will send out a link-state message triggered by this change. Assuming that only neighborhood information is included in link-state messages, in the *whole* message case, node 4 will send out a message including its neighborhood information, i.e. it will send a message including the information that $\{1, 2, 3\}$ are neighbors of node 4. In the *incremental* message case, node 4 only needs to send information about the change, i.e. it will send a message including only the information that node 5 is no longer a neighbor of node 4. The data portion of the former message may look like $\{1, 2, 3\}$ and the latter message may be something like $(remove, 5)$. Notice that the incremental message is smaller than the whole message, therefore, we may lower the link-state message overhead by using incremental messages. To increase robustness of the topology dissemination mechanism with incremental messages, every k-th link-state message of a node may be a whole message, with k being a protocol parameter controlling the desired level of robustness. Notice that the whole message scheme corresponds to the case where $k = 1$.

The incremental message scheme especially shows its true advantage when a normal (whole) link-state message is significantly larger than an incremental message. This is true for the case where we send the whole topology information in a link-state message and also for the case where the average node degree in the network is high. Since the incremental scheme where each k-th message is a whole message is a general case of the whole message scheme, we implement this scheme in the current version of Elessar.

4.3.2.1.5 Generic Content: Aside from the various options discussed above, a link-state message needs to include fields for unique and correct identification of a link-state message by intermediate nodes. In Elessar, a link-state message includes the following fields:

- *Originating node ID:* This field identifies the creator of the link-state message. The creator of the message is essentially the node which has detected a change in its neighborhood.
- *Sequence number:* This is a number local to the creator of a message that increases monotonically each time a link-state message is created by the node. Together with the originating node ID, the sequence number uniquely identifies a link-state message.

4.3.2.2 Basic Topology Dissemination

In this section, we describe the basic operation of the topology dissemination mechanism. Topology information is disseminated through the whole network in the form of link-state messages. A node creates and broadcasts a link-state message only when it detects a change in its local neighborhood. Other nodes are responsible for replicating and forwarding such a message so that it reaches all the nodes in the network. It is important to note here that link-state messages are not periodic, as in most link-state protocols, but rather event-driven in Elessar. In a static network, no link-state messages will be generated after the network has converged. The overhead due to link-state messages increases as the dynamics of the network increase, most notably due to node mobility.

When a node detects a neighborhood change through the neighborhood beaconing mechanism, it creates a new link-state message carrying information about this change. Various options for the content of a link-state message were discussed above. The originating node creates the message and locally broadcasts it. Any node that receives a link-state message first checks to see if it has seen the message before. Notice that in order to accomplish this task, each node needs to maintain a table that keeps track of the most recent link-state messages seen by the node. Such a table does not need to maintain all link-state messages received, but it may rather keep track of the last n messages received by the node, where n is a protocol parameter.

If an intermediate node has received the link-state message before, it silently discards it. Otherwise, it first records the change included in the link-state message in its local view of the network, adds the message to its last seen link-state messages table, and then locally broadcasts the message, without changing the originating node ID or sequence number. Note that although a node may receive a link-state message multiple times, it will only broadcast it once, for the first time it receives the message. This basic topology dissemination mechanism is depicted in Figure 4.4. The pseudocode for the topology dissemination mechanism is provided in the design and implementation details section.

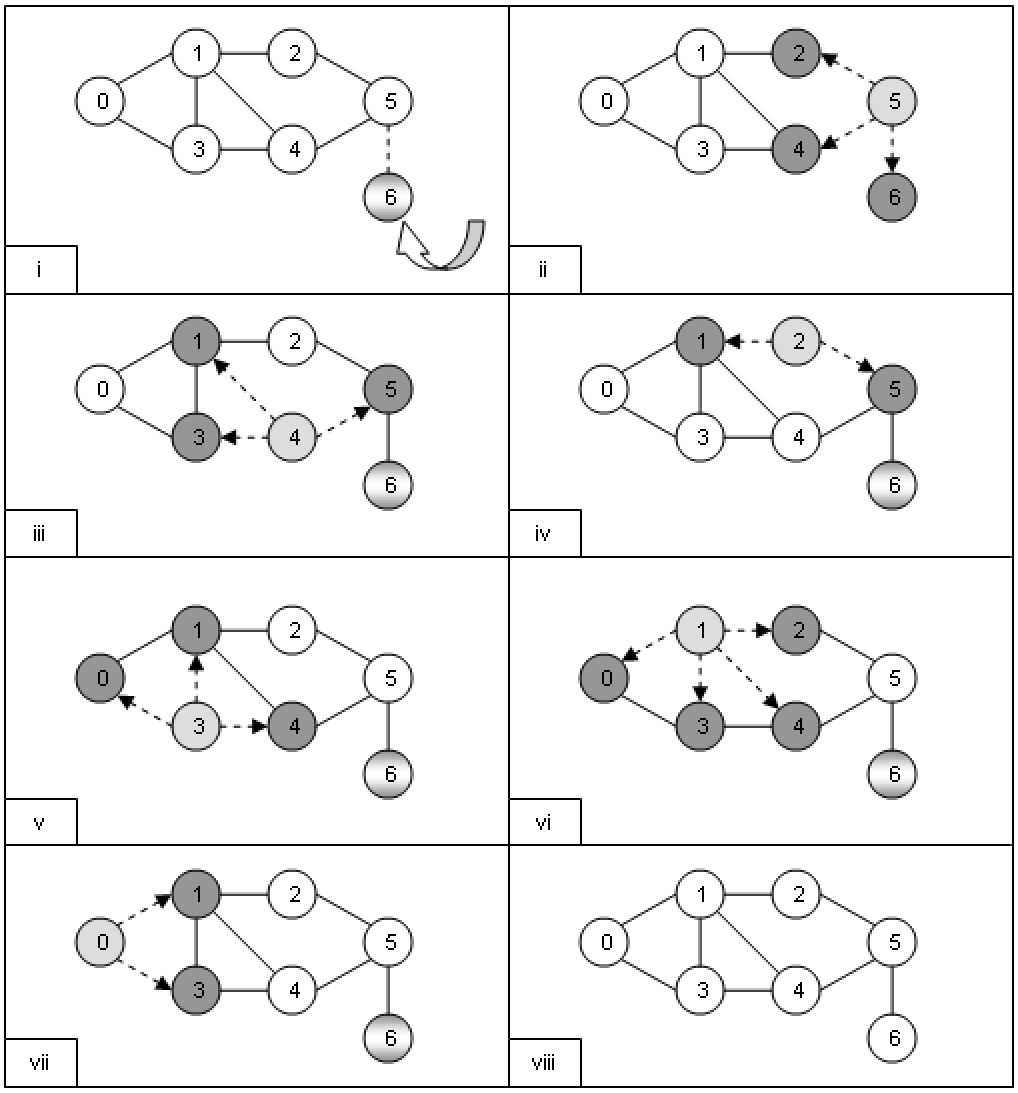

Figure 4.4: Topology dissemination.

In Figure 4.4, part (i), a new node joins the network. In this example, node 6 joins the network and becomes a neighbor of node 5. Node 5 will detect this neighborhood change through the neighborhood beaconing mechanism, and it will create and broadcast a link-state message for this change. The first nodes to receive the message are node 5's immediate neighbors, nodes 2, 4, 6, seen in Figure 4.4, part (ii). These nodes will each record the change in their local views and locally broadcast the message themselves. Since node 6 is the joining node, it does not forward the message. Assuming that the sending order is 4 and 2 (node 4 is the first among nodes 2 and 4 to send out the message), the topology dissemination mechanism goes through the following steps:

1. Node 4 broadcasts the message. Its neighbors 1, 3, and 5 receive the message. Node 5 silently discards the message since it has seen the message before (it is the originator node). Nodes 1 and 3 will forward the message. Let's assume that the forwarding order is 3, 1. These actions are shown in Figure 4.4, part (iii).

2. Node 2 broadcasts the message. Its neighbors 1 and 5 receive the message. Both nodes silently drop the message as they have seen it before. Note that this is the second time node 1 sees the message and it is the third time for node 5 (counting the original message). This step is depicted in Figure 4.4, part (iv).

3. Node 3 broadcasts the message locally and its neighbors 0, 1, 4 receive the message. Nodes 1 and 4 silently drop the message since they have received it before. Node 0 will forward the message. Figure 4.4, part (v) shows this step.

4. Node 1 locally broadcasts the message; nodes 0, 2, 3, 4 receive this transmission. All nodes silently discard the packet. Figure 4.4, part (vi) shows this step.

5. In Figure 4.4, part (vii), node 0 sends the message to its neighbors as a local broadcast. Nodes 1 and 3 receive the message and they both discard it.

6. At the current step, there are no nodes that need to forward the link-state message, so the dissemination comes to an end. At this step, all nodes in the network have received the link-state message and updated their view of the network. The final view of the network is shown in Figure 4.4, part (viii).

In the above discussion, we have assumed that all packets are correctly received by the nodes. Namely, packets do not suffer from corruptions or collisions. In the face of corruptions or collisions, every node in the network may not receive information about the change, leading to different local views of the network at different nodes. However, since each node potentially receives the same link-state message multiple times due to the broadcast nature, the chances that every node will receive at least one copy correctly are increased. Although reception of multiple copies indicates a waste of resources, these copies facilitate robustness of the dissemination mechanism. Note here that the broadcasting scheme currently used is basic flooding. Other broadcasting schemes that will provide the same functionality with more efficiency may be used instead of the flooding scheme. One such scheme may be multi-point relays [54, 65]. We leave the in-depth analysis of such schemes as future work and focus on the flooding scheme at the moment. We will discuss in the following sections how we can achieve more efficient dissemination when we use flooding.

When a change in the topology occurs, several nodes may be affected by this change simultaneously. An example case is provided in Figure 4.5. Although neighborhood beaconing is periodic, because the periods of nodes are not synchronized, all of the affected nodes will not learn of this change at the same time. The first node that learns of the change through the neighborhood beaconing mechanism is (most probably) going to be the one that will initiate the (first) topology dissemination mechanism.

In Figure 4.5, we see a topology change due to node mobility. Here, node 6 moves out of the range of node 5 and into the range of node 4. Due to node 6's mobility, three nodes' neighborhood's change; these nodes are node 4, 5, and 6. The previous neighbors of node 4 were $\{1, 2, 3, 5\}$, of node 5 were $\{2, 4, 6\}$, and

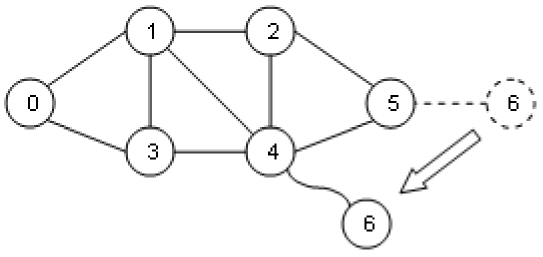

Figure 4.5: A topology change cetected by multiple nodes.

of node 6 was $\{5\}$. The new neighbors of node 4 are $\{1, 2, 3, 5, 6\}$, of node 5 are $\{2, 4\}$, and of node 6 is $\{4\}$. Since three nodes are affected by this movement, all three will be able to detect this change through the neighborhood beaconing mechanism. However, depending on the timing of link-state message dissemination and neighborhood beaconing, we may have several different scenarios. In order to simplify and clarify the discussion, we ignore all events related with node 6 (except its movement) at the moment. The possible scenarios that may occur are listed below:

- *Case 1: One node learns of its neighborhood change after it forwards the other node's link-state message.* Node 5 detects the change through neighborhood beaconing, but node 4 has not detected the change yet. Node 5 creates and sends a link-state message regarding the change and all its neighbors receive this message, including node 4. Node 4 learns of the topology change from this link-state message, but it currently does not know that node 6 is its new neighbor. Node 4 continues the link-state dissemination mechanism by sending the link-state message to its neighbors by local broadcast. Later, node 4 will detect the change in its own neighborhood (the arrival of node 6), and it will start another link-state dissemination for this change.

- *Case 2: One node receives the other node's link-state message after it has already sent out its own link-state message.* Node 4 and 5 both detect the change roughly at the same time. Both nodes send out a link-state message, so two link-state messages are created for the change, as in case 1.

- *Case 3: One node receives the other's link-state message before it has sent out its own link-state message.* Node 4 and 5 both detect the change roughly at the same time. However, before node 4 sends out a link-state message, it receives the link-state message originated by node 5. Now, there are two options. Node 4 may add the information regarding its neighborhood change to this received message and send out only one (aggregated) link-state message. The second option is the straightforward one, where node 4 does not do any data aggregation and sends out two link-state messages, creating a new link-state message of its own and also forwarding node 5's link-state message. Overhead due to link-state messages when we use aggregated messages will be lower than the straightforward case, improving efficiency of the protocol. However, aggregated messages complicate link-state message processing. We will first implement the straightforward case, leaving the option of aggregated messages as a protocol enhancement to be developed later.

4.3.2.3 Topology Dissemination in (Very) High Mobility

When the dynamics of the network is high due to (very) high node mobility, the number of link-state messages created will be high. In such a case, the event-driven approach may result in higher overhead than the periodic approach. To achieve better efficiency, when the link-state message creation rate exceeds a certain threshold, the nodes with high link-state creation rates may switch to periodic link-state dissemination. When the rate falls down again, those nodes would revert back to event-driven link-state dissemination. Another approach may be to let nodes with high link-state message creation rates to aggregate several of their own link-state messages into a single link-state message and send this out, in order to reduce the link-state message creation rate. To be more precise, a node with high rate would not sent out every link-state message it creates immediately, but rather would accumulate some of these messages into a single, possibly larger message, and send out this aggregated message. This way, only the m-th message would be sent out, but this message would also include all of the information of messages $\{1, 2, \ldots, m-1\}$, where m may be adjusted according to the link-state message creation rate.

There are several issues that need to be considered in this high node mobility case. If the periodic approach is employed, then the choice of the period of link-state messages becomes an issue. If the aggregation approach is used, then the choice of m is an important protocol parameter. In both approaches, the parameter in question may be adjusted according to the rate of link-state message creation at the nodes. However, more research and experiments are needed in order to achieve such a task. Another issue common to both approaches is how to set the threshold value for link-state message creation rate. The threshold value should represent the situation where the event-driven approach fails to be more efficient than the periodic approach, or the situation where the loss of responsiveness to topology changes caused by the aggregation approach is acceptable.

4.3.3 Directed Cost Dissemination

The mechanism that enables Elessar to support real-time traffic is directed cost dissemination. This mechanism is only part of the QoS mode of Elessar. Directed cost dissemination is activated only when there are one or more QoS-requiring data streams in the network. The basic idea is as follows. When a node S wants to send real-time traffic to another node D, it immediately starts sending the data over a path that may be non-optimal, therefore decreasing the delay in route acquisition. At the same time, S initiates the directed cost dissemination mechanism. This mechanism informs some or all of the nodes in the network that there is real-time traffic across the network and that these nodes should send their link costs directly to S. When S receives these costs, it has full topology and full/partial link cost information regarding the network. With this higher level of knowledge about the network, S may now select one or more optimal paths to D that satisfy the requirements of the real-time traffic as best as possible.

There are several questions that need answering to achieve better understanding of this mechanism. These questions include the following:

- How are nodes in the network informed of a real-time traffic flow?
- Which nodes should be informed of the real-time traffic flow?
- After nodes are informed, how exactly do they send their costs to S?
- Do informed nodes send their costs only once or multiple times (perhaps periodically)?

We will investigate these questions in the following sections.

4.3.3.1 Informing Nodes

Consider the case when a node S wants to send a long stream of real-time data to another node D. The adjective "long" here is important, as if the real-time data is too small, then the overhead of directed cost dissemination for QoS-route finding may not be justified. It should be noted that as the directed cost dissemination is taking place, the real-time data is being sent from S to D over one or more, possibly non-optimal paths, and if most or all of the data is sent by the time node S learns enough about the link costs to switch to more optimal paths, then the effort to enable such a decision is wasted. We will determine the amount of real-time data that will require the initiation of the directed cost dissemination mechanism through experiments.

So, let's consider the case where node S wants to send some real-time data to node D, and the directed cost dissemination mechanism will be activated for this real-time data stream. We are now faced with the questions of which intermediate nodes will send their known costs to S and how these nodes should be informed. The answer to the first question will be a result of the solution to the second problem, as the mechanism by which intermediate nodes are informed will also determine which nodes are informed and therefore which nodes send their costs to S. We currently propose two approaches for the informing of nodes and provide possible enhancements to these approaches.

4.3.3.1.1 Using a Sign Bit on Data Packets The first approach uses a special sign bit included in the data packets that are sent from S to D. We have mentioned that S starts sending the real-time data stream to D immediately, possibly over one or more non-optimal paths. Let's focus on the case where S uses a single path initially. Each data packet includes a sign bit telling whether that packet is a real-time packet or not. Each time an intermediate node is forwarding a packet, it checks its sign bit and if it sees that this is a real-time packet, it is informed of a real-time flow. It should be noted here that even though a data packet is a real-time data packet, its sign bit may not be set to indicate "real-time" due to the above discussion on the length of a real-time data stream. For a short real-time data stream for which the directed cost dissemination mechanism will not be initiated, the data packets have their sign bits set to "normal". For a long real-time data stream for which the directed cost dissemination mechanism will be initiated, all packets have their sign bits set to "real-time".

4.3.3.1.2 Using a Special Packet The second approach uses a special packet to inform all nodes along its path about a real-time data stream. When S wants to send a long real-time data stream to D, it first sends a special packet along the possibly non-optimal path(s) that it initially uses to send the packets to D. Focusing on the single path case, S sends the special packet along this initial path, and all nodes along this path are informed of the real-time flow. It is much better if the sending of the special packet is reliable, so for the delivery of this packet, either a hop-by-hop or an end-to-end acknowledgement mechanism, along with timeout and retransmission mechanisms would be desirable. We adopt this approach in order to inform nodes in our protocol implementation.

Using either the first or the second approach, the nodes along the initial single path from S to D are informed of the real-time flow. When an intermediate node is thus informed, it sends its link costs directly to S and it also informs its neighbors of the real-time flow and its source by a special information packet (that is locally broadcasted unreliably), so that its neighbors may send their costs directly to S. Each informed neighbor, in turn, sends its costs directly to S and informs its own neighbors, and the information mechanism continues like this, in a flooding manner. To limit the extent of the flood, a hop-limit is used on information packets. This hop-limit is decremented by one when a node locally broadcasts an information packet, and when it reaches zero, the flooding stops. The initial hop-limit value may be set according to a built-in protocol mechanism that calculates a reasonable hop-limit depending on the diameter of the current topology, or it may be determined by the source node of the real-time data stream, specified in the data packets from S to D, or in the special packet sent from node S.

If *promiscuous mode* may be enabled on wireless nodes, then the information of neighbors does not require a special information packet. Whenever a node overhears the transmission of one of its neighbor's link costs to S, it learns of the real-time flow and its source, so the overhearing node may send its link costs directly to S. Nodes overhearing the transmission of this node, in turn, may do the same, essentially forming a flooding mechanism. The hop-limit imposed above on information packets may here be imposed on the link cost messages, achieving the same purpose.

Similar to the situation seen in the topology dissemination mechanism discussed above, nodes would have to keep track of recently seen real-time flows, so that they do not take part in the flooding mechanism more than once. Keeping such a track may be accomplished by simply keeping a "last-recently-seen-real-time-flows" table at each node, discarding an entry from the table whenever it becomes out-of-date.

The information mechanism of the nodes is explained in Figures 4.6 and 4.7. In Figure 4.6, a sample topology along with a single real-time flow from node S to node D is presented. The nodes on the initial data path from S to D are colored gray. Figure 4.7 shows which nodes are informed of this real-time flow through the information mechanism. How these nodes are informed is irrelevant here. In part A of the figure, a hop-limit of 1 is imposed, and all non-black nodes (all nodes labeled "node on data path" and "node sending cost") are informed of the real-time flow and are sending their costs to S. In part B of the figure, a hop-limit of 2 is imposed, and again, all non-black nodes are informed and sending their

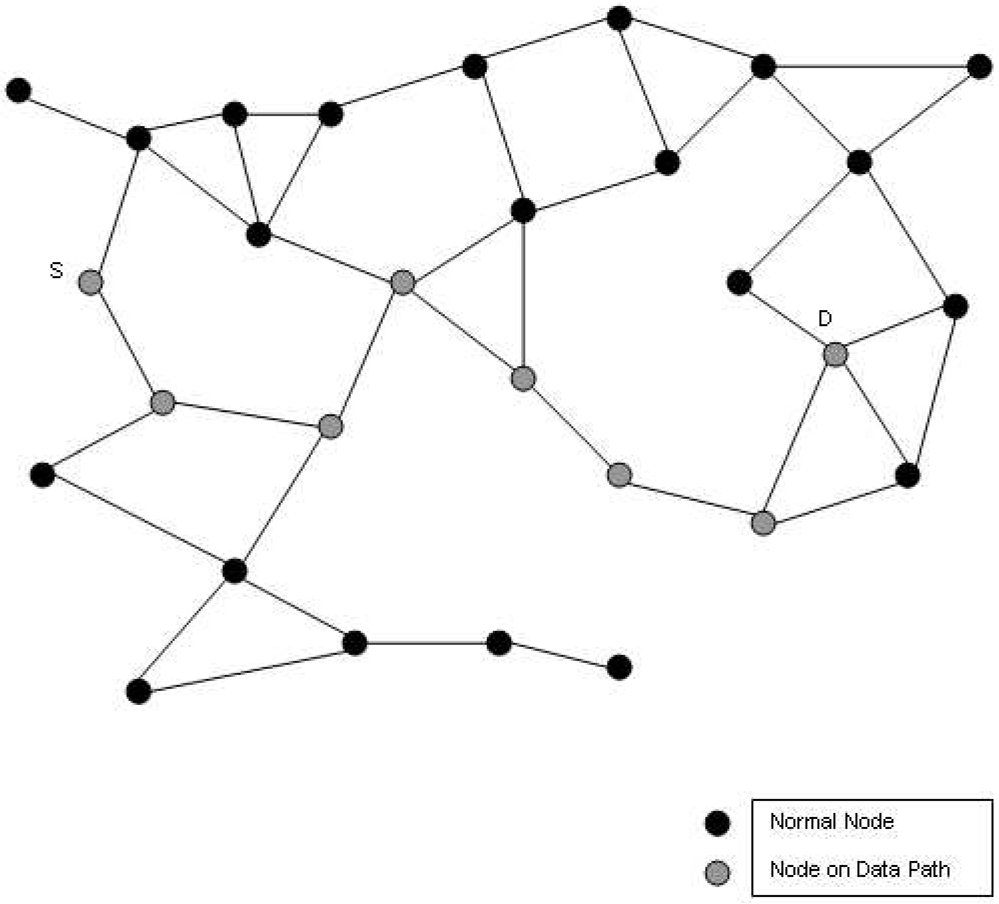

Figure 4.6: A sample topology with a real-time flow.

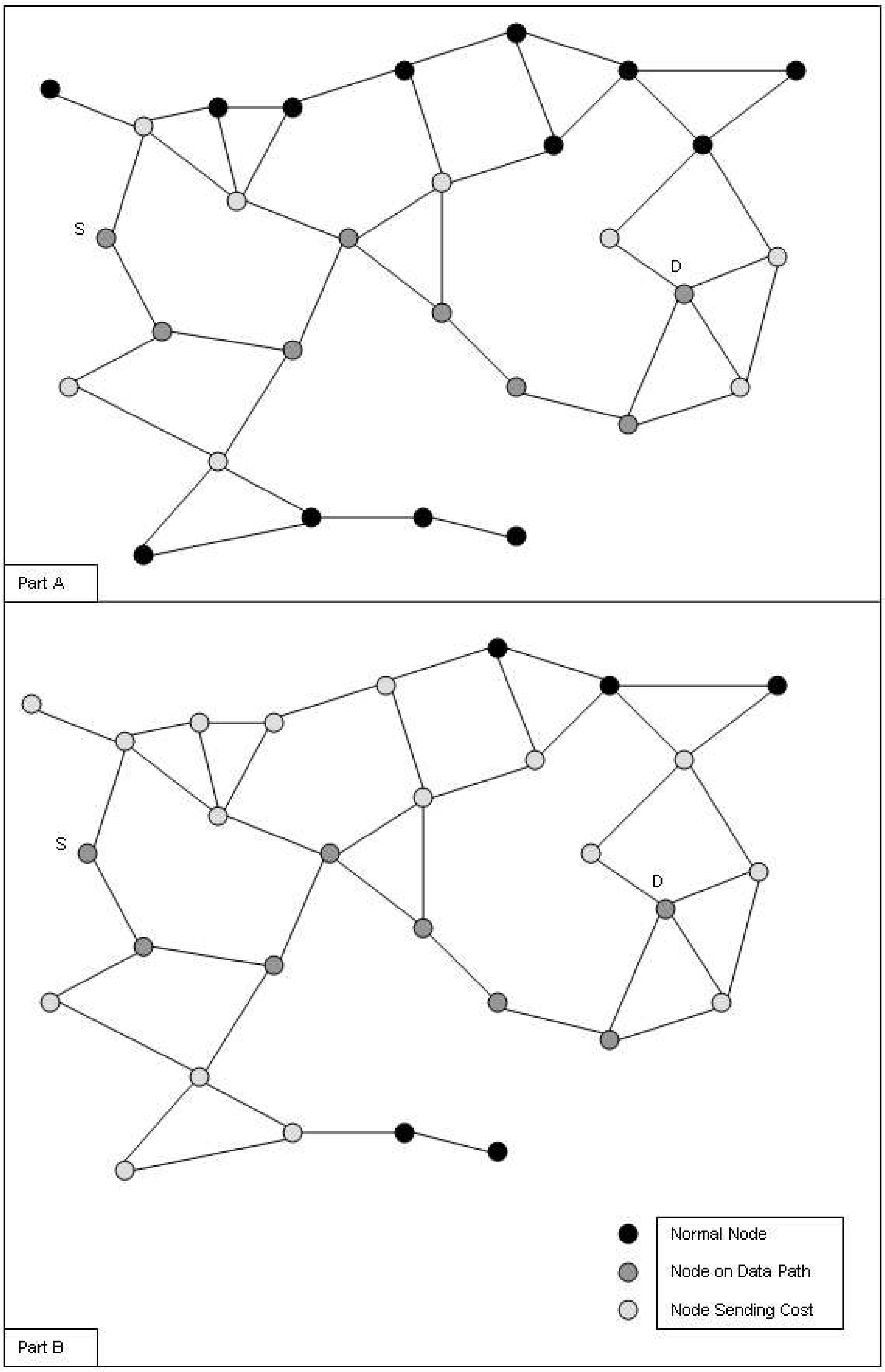

Figure 4.7: Information of nodes about a real-time flow.

costs to S.

4.3.3.1.3 Enhancements to Both Approaches A problem of the first approach using a sign bit is that if the real-time stream is very long, then the wasted bandwidth due to these single bits may be significant. We need to inform the nodes along the data path only once, so the first solution that comes to mind is to include this sign bit on only the first packet of the real-time flow. However, packets may be subject to corruptions and losses in the network, and if the first packet is lost, then the nodes are never informed. Therefore, we may be tempted to send the first packet of a real-time flow reliably, as discussed for reliable delivery in the special packet approach. However, with the introduction of the reliable delivery of the first packet of a real-time flow, the two approaches are almost the same, and in this situation, the use of the second approach may be more desirable as it introduces less complexity at the intermediate nodes during packet processing. Another solution for the wasted bandwidth problem may be to include the sign bit in the first b percent of the packets of the real-time flow, and send these packets as usual (i.e. unreliably). The parameter b may be calculated by the protocol according to the actual length of the flow and the loss rates of the channels. With this approach, even though some of the packets including the sign bit may be lost or corrupted, there are more packets with the same information that will be intact.

The problem with including the sign bit in only some of the packets is that it complicates packet processing at the intermediate nodes. In light of this, we are inclined towards the use of the second approach for the information of nodes. However, if there are session establishment procedures (signalling) done by a higher network layer at the start of each real-time flow, then the sign bit idea may be used with these procedures. If these procedures are accomplished over reliable channels, then using sign bits on signalling packets becomes much more desirable.

4.3.3.1.4 Use of Additional Paths for the Information of Nodes Regardless of whether the first or the second approach is used and whether node S initially uses a single path or multiple paths, another enhancement that may be applied during the information of nodes is the use of *additional paths.* In addition to informing the nodes over the initial data path(s) that S uses to send to D, one or more additional paths from S may be used in order to inform other nodes. These paths do not have to be from S to D, but may be from S to some other node, in order to facilitate selective gathering of link costs. The use of such additional paths is more suited to the second approach, where we use a special packet to inform nodes. The use of additional paths also makes more sense if S initially uses a single path. Generally, the use of two paths should be sufficient for the sizes of networks under consideration, and since the single data path used by S will be one of these two paths, only one additional path should be sufficient. Of course, a question that immediately arises is how this additional path is selected. We leave the detailed discussion of the use of additional paths for the information of nodes as future work, providing in Figure 4.8 a sample case that illustrates the idea more clearly.

In Figure 4.8, part A, there is a real-time flow from node S to node D, initially going over the nodes labeled "node on data path". The nodes on this data path are informed of the real-time flow. There is also a second path, nodes of which are labeled "node on 2nd path", used to inform additional nodes. The hop-limits for packets on both of these paths are set to be 1, and the nodes which are informed of the real-time flow are depicted in part B of the figure. All non-black nodes in part B of the figure are informed of the real-time flow and are sending their costs to S.

4.3.3.2 Cost Dissemination

Once a node is informed of a real-time flow and and its source, the node must send its link costs to the source node. Obviously, there are many ways in which such an intermediate node can send its costs to the source. The approach used in Elessar is to let an intermediate node send its costs to the source node *directly*,

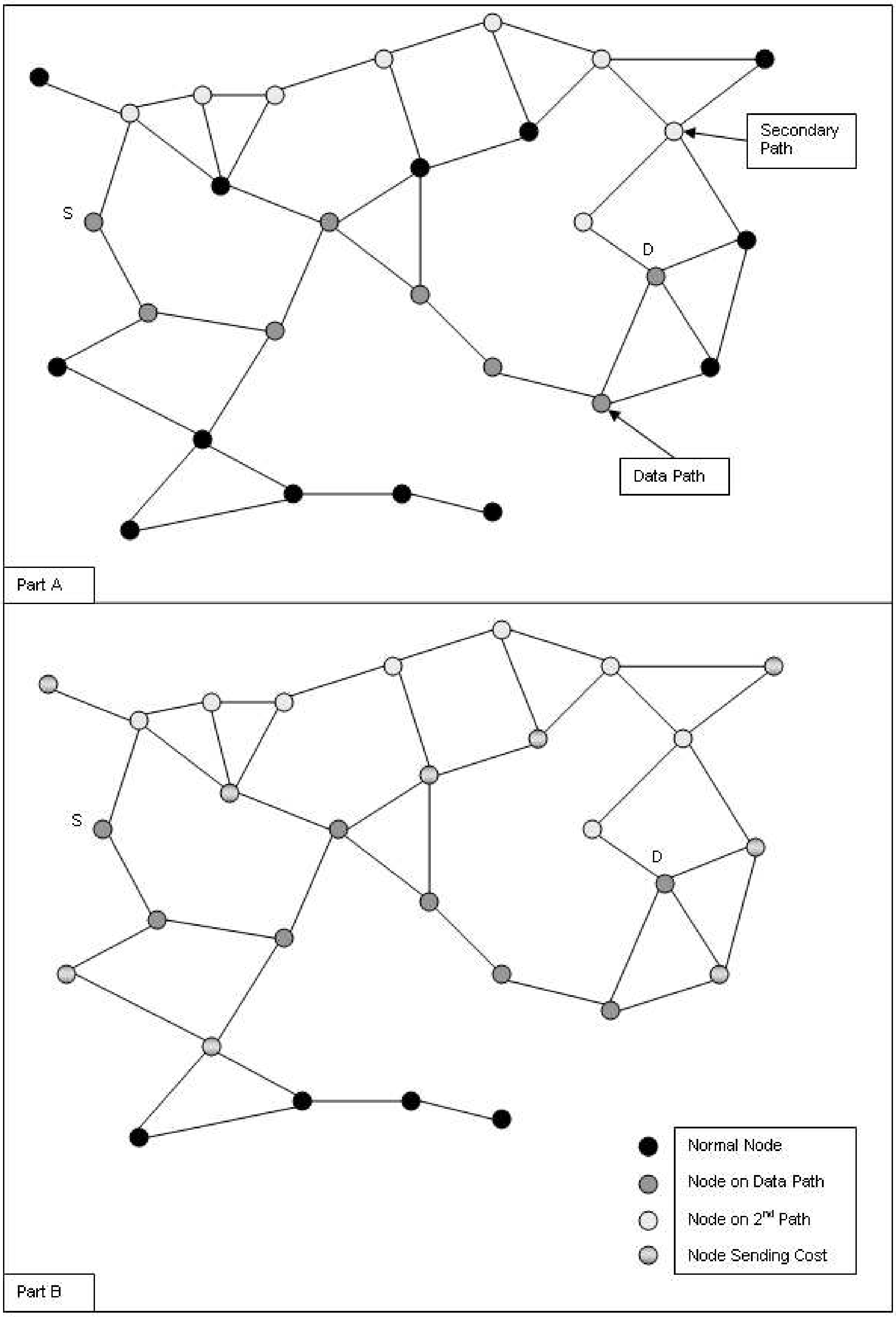

Figure 4.8: Information of nodes about a real-time flow: use of additional paths.

hence the name directed cost dissemination. Since each node in the network is receiving link-state messages as changes occur and updating its view of the network accordingly, an intermediate node may run a local path finding algorithm on its own view of the network in order to find a path from itself to the source node, and it may use this direct path to send its costs. It should be noted that the path from node to source may differ from the reverse of the route of the information message that let the node know of the real-time flow in the first place. In our opinion, such differences will facilitate load-balancing and prevent certain areas of the network to be congested during cost dissemination.

4.3.3.3 Dynamics of the Cost Dissemination Mechanism

One important aspect of directed cost dissemination is the frequency with which nodes send their costs to the source of a flow. Once a node is informed of a real-time flow, it needs to send its link costs to the source directly, at least once. The important question is whether such a node will send its costs more than once and if so, under which conditions it will do so. There are four options that we consider:

- An intermediate node sends its link costs exactly once.
- An intermediate node sends its link costs periodically, for a certain r number of times.
- An intermediate node sends its link costs periodically, for the whole duration of the real-time flow.
- An intermediate node sends its link costs once at the beginning, and possibly multiple times based on a threshold-value.

We will discuss these options in detail below.

4.3.3.3.1 Exactly Once This option is quite straightforward. An intermediate node which is informed of a real-time flow will send its link costs exactly once to the source node. Note that since such costs are sent unreliably, the source node may not get the costs of all informed nodes. In such a case, we do not use retransmissions to send the costs. The advantage of the *exactly once* option is its simplicity and low overhead. The disadvantage is that the source node will need continuous information regarding the network topology and link costs in order to be able to react to changes dynamically, and sending the costs only once does not achieve this requirement.

4.3.3.3.2 Periodic, with Bound With this option, an intermediate node informed of a real-time flow will send its link costs to the source of the flow *periodically*, however *with a bound* on the number of times it sends its costs. Two important issues with this option are the appropriate setting of the bound and the period of the cost messages. It may be reasonable to set the bound on the number of messages according to the length of the real-time flow. However, if the length of the flow is not known before-hand, then using a fixed number is the most suitable choice. Regarding the period, if it is too small, certain parts of the network may be congested due to the sending of many messages in a small time window, and the amount of information gathered may not be as useful since we would be getting samples from a small time window using a fixed number of cost messages with a small period. On the other hand, if the period is too large, the responsiveness of the protocol to changes (in the link costs) will be lost.

4.3.3.3.3 Periodic, without Bound This option is similar to the option above, but without the restriction on the number of cost messages. In this option, an intermediate node will send its link costs to the source of a real-time flow periodically, for the whole duration of the real-time flow. The issue with the setting of the period discussed above is also present here. One additional thing that needs to be done with this option is the information of all cost-sending intermediate nodes about the ending of the real-time flow. All such nodes need to informed of the end of the real-time flow, so that they will stop sending their costs to the source. This information step may be achieved using the same mechanism that was employed in order to inform the nodes of the start of a real-time flow.

However, this information step may pose a problem due to node mobility. Nodes were informed of the start of a real-time flow through the use of one or more specific paths, and these paths may have changed due to node mobility at the end of the real-time flow. In order to inform all nodes that were sending their costs of the end of the flow, the source node, which is getting cost messages from all cost sending nodes, and therefore knows which nodes to inform of the end of the flow, may send informing messages to all such nodes directly. Another option is to globally broadcast a message in order to reach all cost sending nodes. Of course, if the duration of the real-time flow is known at the start of the flow, cost sending nodes may be informed of this duration, and each individual node may decide for itself when to stop sending its costs based on this given duration. With an appropriate period, the *periodic, without bound* option allows the most responsiveness to changes (in the link costs) as it will inform the source during the whole duration of the real-time flow. However, it is also the option with possibly the highest overhead.

4.3.3.3.4 Threshold-Based The final option is the threshold-based one. This option is as follows: Intermediate nodes informed of a real-time flow will send their costs to the source node once, when they are informed. As the real-time flow continues, the source will get receive reports back from the destination, enabling the source to have an idea about the condition of the path(s) from itself to the destination. Once the source sees that the condition of the path(s) drops below a certain threshold point, it initiates the information step of the cost dissemination mechanism again. This way, (some of the) intermediate nodes will be informed again and they will send their link costs to the source, enabling the source to find possibly better path(s). As always, the intermediate nodes send their costs to the source using direct paths. The destination node will also send the receive reports back to the source directly. Instead of sending a receive report for each received packet, the destination may send a receive report periodically or for every k-th received packet. A similar discussion may be found in the "topology dissemination in (very) high mobility" section.

We would like to note here that the topology is subject to changes during directed cost dissemination, and the path(s) used by the source node in order to send the real-time flow may vary during the lifetime of the flow. In the face of such alterations, the flow will potentially go over nodes that were not on the initial path(s) and such (new) nodes also send their costs directly to the source once they determine that a real-time flow has started to pass over them. Therefore, all nodes over which data packets belonging to the real-time stream pass during its whole lifetime send their link costs to the source at least once, and possibly multiple times depending on the scheme adopted.

4.3.4 Route Discovery and Packet Forwarding

Route discovery is the mechanism that enables a node in the network to send a packet to another node. Due to the link-state nature of Elessar, each node has its own complete view of the network. Link-state messages are only produced when a topology change occurs. Therefore, if there are no collisions and packet losses, each node knows the exact current topology. However, a node does not have complete information on the link costs. With the current information at each node, a node wishing to send a packet to another node may simply run a local path finding algorithm to find one or more routes from itself to the destination node.

The route discovery mechanism consists of two different operation modes: *normal mode* and *QoS mode*. We will see below how a route is discovered in each operation mode. However, before discussing operation of the mechanism in each mode, we will first discuss the use of a route cache at each node and source routing.

4.3.4.1 Route Cache

Each node in the network maintains a route cache. The routes stored in the route cache of a node are routes found by the node itself through the use of a local path finding algorithm. The route cache is used to store currently/recently used routes by the node. Using the route cache, the sender node does not need to run the local path finding algorithm for each packet (belonging to the same stream) that it wishes to send to a receiver node. It just needs to find a path using a local algorithm for the first packet of a stream, and later packets belonging to that stream use the stored path in the route cache.

The entries in the route cache are valid as long as the topology considered when they were discovered remains the same. When the topology changes, all routes in the route cache are invalidated. Therefore, when a source node receives a link-state message, it will invalidate and erase all entries in the route cache. Another case when a route cache entry becomes invalidated is the reception of an error message through the route maintenance mechanism from some downstream node, if such error messages are employed. We will discuss the route maintenance mechanism in detail later, but let's just say here that when a downstream node is unable to forward a packet, it may send an error message back to the source node. When a source node receives such a message, it invalidates and erases the broken route from the route cache. We also clear the route cache periodically. The reason for this periodical cleansing is explained in Section 4.3.4.4.

The source node looks at the route cache for every outgoing packet. If it finds a route to the destination of the packet in the route cache, it uses that route[3]; otherwise, it runs the local path finding algorithm to find a route to the destination. If it finds a route, it records it in the route cache; if no route is found, this means that there is no path from the source node to the destination node, at least for the moment.

4.3.4.2 Source Routing and Packet Forwarding

In Elessar, we employ source routing in order to route a packet from a source to a destination. The source node runs a local path finding algorithm on its view of the topology, and embeds the found path in the header of each packet from

[3]In normal operation mode, a packet destined to node D may use a previously found route from the source to D. However, in QoS mode, a packet with QoS requirement Q, destined to node D, may only use a path to D, satisfying Q.

itself to the destination node. The source node then forwards the packet to its next hop along the path. Each intermediate node receiving a data packet checks its header and finds its node ID in the route field of the header. If it is the last node in the path (the destination node), it processes the packet and passes it up to the higher layers. If it is an intermediate node in the path, it sends the packet to its next hop in the route, without changing the packet header. If the node cannot find its ID in the route, then it silently drops the packet. Note that this final case should not happen in principle. Source routing is depicted in Figure 4.9. In the figure, node 1 creates and sends a packet destined to node 7 in the network, and the route of the packet is $\{1, 3, 4, 2, 7\}$. The originator of the packet creates the packet, embeds the route into its header, finds the next hop along the path (node 3) and forwards the packet to this node. An intermediate node, node 4 in this example, finds the next hop coming after itself in the packet's route (node 2), and forwards the packet to this node. When the destination node receives the packet, it processes it and passes it up to the higher layers. Node 7 is the destination node in the example, and even if the packet does not include an explicit *destination* field in its header, node 7 may determine that it is the packet's destination as it is the last node in the packet's route.

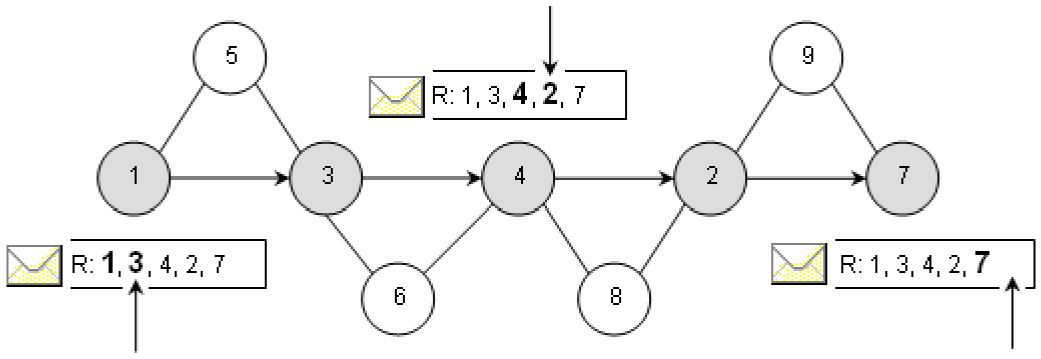

Figure 4.9: Source routing and packet forwarding.

In Elessar, since each node has complete knowledge of the topology due to the topology dissemination mechanism, we could have used hop-by-hop routing instead of source routing. However, source routing allows packet routing to be loop-free, even when the network is subject to rapid topology changes, and puts minimal amount of burden on the intermediate nodes during routing. Due to its simplicity and efficiency, we use source routing as our routing method. The only drawback of source routing is the additional overhead due to the inclusion of the path in each packet header. However, we have assumed that Elessar will operate on small-to-medium sized networks, with a network diameter of 10 to 15. Considering operation on such networks, the overhead due to source routing is low and therefore acceptable.

4.3.4.3 Route Discovery in Normal Mode

In normal operation mode, a node wishing to send a packet to another node in the network first checks to see if it has an up-to-date route registered for that destination in its route cache. If the sender finds a route in the route cache, that route is used. Otherwise, a local path finding algorithm is run on the local network topology and the found path is registered in the route cache. If no path is found, this means that there is no path from the sender to the receiver, at least for the moment. In such a case, the network layer may respond with a "destination unreachable" error message to the transport/application layer.

We currently use Dijkstra's shortest path algorithm [36, 38] as the local path finding algorithm. Setting the cost of each edge to 1, a sender node running this algorithm will find a minimum-hop path from itself to a desired destination in the network[4]. Note that since we have global topology knowledge, we may easily find multiple paths from a source to a destination in the network through the use of a local algorithm that finds multipaths between two nodes. We leave the discussion of multipath routing as future work.

Once a path is found to the destination, either from the route cache or through the use of the local path finding algorithm, the discovered path is embedded into the route field of the outgoing packet and sent towards the destination using source routing.

[4]Actually, using Dijkstra's algorithm, a sender node finds minimum-hop paths from itself to all possible destinations in the network.

4.3.4.4 Route Discovery in QoS Mode

When a source node S wants to send a real-time stream to a destination D, it will operate in QoS mode. If the length of the stream is known beforehand, S first checks to see if the length of the stream requires initiation of the directed cost dissemination mechanism[5]. If the answer is negative, then the packets are sent to D as in normal operation mode. However, if the stream requires directed cost dissemination, then a different, but similar, approach is taken.

When a real-time stream requiring the initiation of the directed cost dissemination mechanism needs to be sent from S to D, S first initiates the directed cost dissemination mechanism. For a detailed discussion on how directed cost dissemination may be initiated, please refer to the section on directed cost dissemination. After S initiates the directed cost dissemination mechanism, it immediately starts sending the real-time data to D. Since S currently knows only the topology, but not the link costs, it will only be able to send the data over an initial path that may be non-optimal. S uses a minimum-hop route from itself to D as the initial path; this path may easily be discovered as in route discovery in normal mode. S will start sending the real-time data over this initial path, thereby reducing the initial delay in data transmission to effectively zero. Since S has started the directed cost dissemination mechanism, it will receive link cost messages from several nodes as it continues transmission along the initial path. For each cost message received, S checks to see if it now has enough knowledge to find a more suitable path from itself to D. Once a more suitable path is found, S switches to that more optimal path. A "more suitable" path is determined according to the QoS requirements of the real-time stream and the type of link costs received. We leave detailed discussion of selection of more suitable paths to Section 5.3.3. It should be noted here that, just as in normal operation mode, multiple paths from S to D may be employed in QoS mode, and the discovery of such multipaths is relevantly easy since a local multipath finding algorithm will be able to discover such paths.

[5]Please see the section on directed cost dissemination for a discussion on this matter.

According to the exact operation of the directed cost dissemination mechanism, S may be able to react to changes in the network and choose different paths as time progresses. Please see the section on directed cost dissemination for a detailed discussion on the matter.

As a source node is receiving link costs from nodes in the network, it will record these costs on its own graph representation of the network. Of course, once the source node stops receiving such cost messages from the nodes, the costs recorded on the graph will become out-of-date. In order to facilitate the removal of such stale edge costs from the graph, each edge cost is removed from the graph periodically.

We have mentioned above that as the source node receives each cost from nodes, it checks to see if it now has enough information in order to find a more suitable path. However, running a local path finding algorithm each time a cost message is received in order to see if a more suitable path is found may be costly in terms of processing time, so instead, a different approach is taken in the actual implementation. In order to facilitate the discovery of better paths during the lifetime of a flow, we clear the route cache periodically. Remembering that the source node first checks the route cache to see if it has already found a path to a destination, clearing all routes in the route cache force the source node to run local path finding algorithms for each destination it wants to send a packet to, achieving the desired effect of determining if a more suitable path can now be found. The period of route cache cleaning must be selected with care, as a long period will cripple the responsiveness of the protocol to changes in link costs while a short period will incur a high overhead in processing time.

4.3.5 Route Maintenance

Route maintenance is activated when an intermediate node is unable to forward a packet using source routing towards its destination on behalf of the source node. Due to the dynamic nature of the network, link conditions and node neighborhoods will change during the lifetime of a stream, and such changes may lead to route breakages. When such a breakage occurs, it is the responsibility of the route maintenance mechanism to inform the source node of the route failure and to work around the problem, if possible, until the source node takes appropriate action. The route maintenance mechanism has normal and QoS operation mode components, although in essence, it operates the same in both operation modes.

4.3.5.1 Route Failure Detection

Let us assume that a source node S is actively transmitting to a destination node D along some path $P(S, D)$. Let a link $(x, y) \in P(S, D)$ from some node x to some node y along this path fail due to link failure or node mobility during this transmission (see Figure 4.10). The question is how x can detect the failure of link (x, y). Elessar does not employ acknowledgements (ACKs) for transmitted packets between hops, so x cannot tell whether its transmission to y was successful or not through an ACK-based mechanism. However, there are various other ways with which x may learn of the fate of its transmissions to y. If the underlying MAC protocol uses link-level ACKs, as most common IEEE 802.11x protocol families do, then the MAC protocol may provide this information to Elessar.

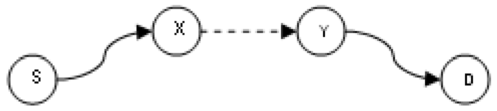

Figure 4.10: A path from S to D including link (x, y)

Another way for failure detection may be the use of *passive* ACKs. In this method, x listens for the transmission from y to the next hop of the packet it has sent to y. If it overhears the transmission of the packet from y, then it knows that y has received the packet correctly. Otherwise, the packet has failed to reach y. Of course, there are several issues with this method. For example, y may have other packets in its outgoing queue, so it may take a while for y to transmit the packet it has received from x. x should wait an appropriate amount of time so that it does not mistakenly think that a route failure has occurred. How to set this "appropriate waiting time" is an issue that would need to be resolved.

Another issue with the passive ACK method is that it cannot detect link failures at the last hop along the path since the destination node will not transmit the packet. And finally, nodes must be able to enable *promiscuous mode* on their wireless network interface cards in order to employ the passive ACK scheme.

Elessar is currently able to detect the failure of a link through its neighborhood beaconing mechanism. In this way, x does not learn of the failure of the link (x, y) through the failure of its data packets in reaching y, but rather through the failure of its neighbor beacon messages in reaching y.

4.3.5.2 Route Maintenance in Normal Mode

In whichever way x may learn of the link failure, it has detected a change in the topology and therefore it will need to broadcast a link-state message for this change. Since this link-state message is globally broadcasted, it will reach S, and since S clears its route cache with each new link-state message, it will recalculate the path from itself to D according to this new topology. Therefore, it would seem that we do not need any additional mechanisms in order to inform S of the route failure, which is correct. However, by sending some additional messages, we may convey some more information to S and possibly to the other up-stream nodes. We also achieve earlier information of S with explicit route failure messages, which may be important as S has an active stream flowing over the failed link. We have two different approaches to route maintenance depending on whether route failure messages are used or not.

4.3.5.2.1 Route Maintenance, Normal Mode, No Route Failure Messages This scheme does not use explicit route failure messages and forms the basis for the route maintenance scheme employing route failure messages. Returning back to the above discussion on the failure of link (x, y), when x learns of the failure, it will initiate the topology dissemination mechanism, but we do not concern ourselves with the details of that at the moment. For a detailed discussion on this, please refer to the section on topology dissemination. More importantly, x will start the route maintenance mechanism when it learns of the failure. In this particular scheme, x will try to work around the problem and see if it can find an alternate route from itself to D. x achieves this using the route discovery mechanism (in normal mode) as discussed in the previous section. For each data packet it receives, x will first try to send the packet according to the original path in the packet header[6]. If x finds out that it cannot reach the next-hop, it will look into its route cache to see if an alternate path exists[7]. If no path exists in the route cache, it will run the local path finding algorithm to discover a path, if one exists. If an alternate path is found, it is recorded in the route cache, the (broken) path embedded in the packet header is replaced with the alternate path, and the packet is forwarded along to its next-hop in this alternate path. If x is unable to find an alternate path to the packet's destination, then the packet is dropped. This is all node x does in this scheme.

4.3.5.2.2 Route Maintenance, Normal Mode, Using Route Failure Messages This scheme is an extension of the scheme that does not employ route failure messages. All the actions performed by x in the above scheme are also exactly done here. Furthermore, if x is unable to find an alternate route, it sends a route failure message back to S, following the *reverse path* from S to x in order to inform the upstream nodes from x of the situation[8]. By informing the upstream nodes, we prevent unnecessary transmissions to x from these nodes, as x has no way to relay these packets to D. The route failure message of x includes the information that x has no path to D. An upstream node receiving this route failure message stops sending packets towards x. In order to decrease the number of dropped packets, each upstream node tries to find an alternate path from itself to D and tries to send the current packets it has over this new route, if one exists. However, these upstream nodes do not produce route failure

[6]The action of first trying to follow the path in the packet header is logical and necessary. This way, when x receives packets following an alternate path to D, not using link (x, y), it will be able to send the packets correctly using their own routes.

[7]x has knowledge of the conditions of its immediate neighbors, so if it sees that the next-hop along the path is no longer its neighbor, it knows that it cannot send the packet to that node.

[8]x may send the route failure message in the reverse direction since we have assumed that links are bidirectional. If this assumption does not hold, then in every case, a route failure message is sent *directly* to S.

messages of their own. When S receives the route failure message, it will try to find a new path through the route discovery mechanism[9]. Note that in order to enable an upstream node to stop sending to x the packets belonging to the stream following the broken path, a table should be kept in the upstream node, which keeps the information that packets belonging to the stream should not be forwarded to node x.

Figure 4.11 illustrates the ideas presented above more clearly. In part (i), source node 1 is sending a data stream to destination node 8, using the route $\{1, 4, 5, 8\}$. In part (ii), node 8 moves away from all its neighboring nodes, and none of its neighbors can now reach it. Node 5 detects the breakage of the route, and tries to find an alternate path to node 8, without success. When route failure messages are used, node 5 creates and sends a route failure message back to node 1, following the *reverse* of the original data path in order to inform upstream nodes as well. In this case, upstream node 4, as well as source node 1, are informed of the failure of the route. In part (iii) of the figure, node 8 again moves, but this time node 3 is still connected node 8 while the other neighbors are disconnected. Node 5 detects the breakage in the route, and tries to find an alternate path, with success. Since node 5 has been able to find an alternate path, it reroutes the packets along this route, following nodes 3 and 8, so the route at node 5 effectively becomes $\{5, 3, 8\}$. Note that in this case, source node 1 will learn of the failure of link $(5, 8)$ through the topology dissemination mechanism since no route failure message is produced.

4.3.5.3 Route Maintenance in QoS Mode

The operation of the route maintenance mechanism in QoS mode is almost the same as its operation in normal mode. We again have the option of using explicit route failure messages or not. The scheme where route failure messages are not used is exactly like the one presented in Section 4.3.5.2.1. There is a slight difference when explicit route failure messages are employed. With this option,

[9]Note that this time, we are not guaranteed that S will be able to find at least one path to S, since the path $P(x, D)$ no longer exists. There may be no path from S to D in the network. However, if there is at least one path to D, we are guaranteed that S will find it.

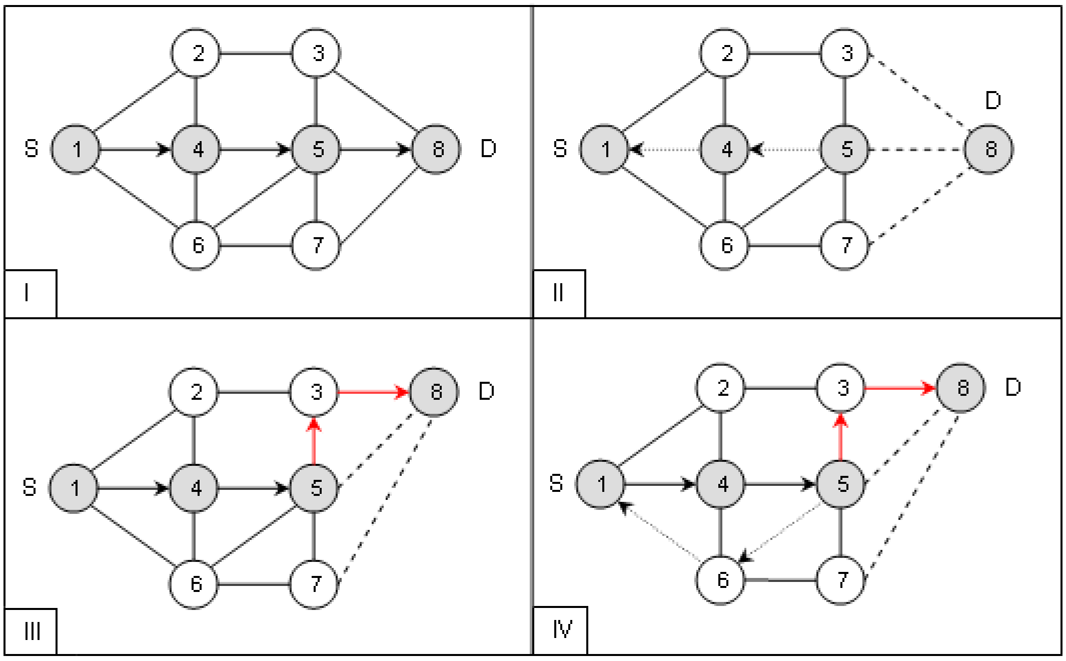

Figure 4.11: Example case for route maintenance.

all the actions performed by x in Section 4.3.5.2.1 are performed. Furthermore, if x has been able to find an alternate route, it sends a route failure message back to S directly. The path x uses to send this message to S is arbitrary, found by the route discovery mechanism. The route failure message also includes the alternate path x has found. This idea is demonstrated in Figure 4.11, part (iv). As stated previously, link $(5, 8)$ has broken due to node mobility and node 5 has been able to find an alternate path to reach node 8. If route failure messages are employed, then node 5 creates such a message, and sends it to node 1 directly, following an arbitrary path chosen by the route discovery mechanism at node 5. In the example, this route is $\{5, 6, 1\}$. Upon receiving this message, S takes necessary action. S will first try to find another route from itself to D, with the knowledge that link (x, y) no longer exists. Since the route failure message includes the path from x to D, S may use the previous path from itself to x along with the path from x to D readily[10]. Since the packets from S to D are real-time packets, when x finds out an alternate path to D, this path will not be optimal as x does not

[10]For the mathematically inclined, this may be represented by the following. We name the initial path from S to D $P(S, D)$. $P(S, D) = P(S, x) + (x, y) + P(y, D)$. Letting the new path found by x be $P(x, D)$, S may use the path $P(S, x) + P(x, D)$ in order to reach D.

have enough information on link costs. Therefore, we allow S to discover a path to D using the route discovery mechanism, since S may be able to find a better (least-cost) path through this mechanism. Note that we are never guaranteed that S will be able to find a QoS-path to D, even when a normal path from itself to D exists, since none of the possible paths from S to D may satisfy the QoS requirements. However, if there is at least one QoS path from S to D in the network, we are guaranteed that S will find it. The actions taken in the case where x is unable to find an alternate path to D are exactly the same as the ones presented in Section 4.3.5.2.2.

4.4 Summary

In this chapter, we have presented an overview of Elessar, an on-demand, link-state based routing protocol supporting real-time traffic in wireless mobile ad hoc networks. The aim of this chapter was to provide a general idea of how the protocol works and to discover various dimensions of the design space. Several important details of the protocol have been omitted here; these will be discussed in the next chapter on design and implementation details. Other issues, design choices, and parameter settings may only be justified through simulation experiments, which are presented in Chapter 6.

Chapter 5

Design and Implementation Details

We have implemented our proposed protocol on a discrete-event simulation tool named OMNeT++. Details on OMNeT++ are provided in Section 5.1. We present the information on main modules and messages used in the simulation in Section 5.2. Pseudocodes and algorithms for the main components of the protocol are provided in Section 5.3.

5.1 Our Simulation Tool: OMNeT++

We have implemented our protocol in OMNeT++ [98, 96, 97]. OMNeT++ is an object-oriented, modular, discrete event network simulator. It is capable of supporting any type of discrete event simulation, but was specifically designed with the simulation of telecommunication networks in mind. It is especially suited for traffic modeling of telecommunication networks, protocol modeling, modeling queuing networks, modeling multiprocessors and other distributed hardware systems.

An OMNeT++ model consists of hierarchically nested modules. The depth of module nesting is not limited, which allows the user to reflect the logical structure of the actual system in the model structure. Modules communicate through message passing, where messages can contain arbitrarily complex data structures. Modules can send messages either directly to their destination or along a predefined path, through gates and connections. Modules can have their own parameters, which can be used to customize module behavior and to parameterize the model's topology. Modules at the lowest level of the module hierarchy encapsulate behavior. These modules are termed *simple modules*, and they are programmed in C++ using the simulation library.

OMNeT++ simulations can feature varying user interfaces for different purposes: debugging, demonstration and batch execution. Advanced user interfaces make the inside of the model visible to the user, allow control over simulation execution and to intervene by changing variables/objects inside the model. This is very useful in the development/debugging phase of the simulation project. User interfaces also facilitate demonstration of how a model works.

OMNeT++ is capable of deployment in various platforms, including Linux and Linux-like systems, Cygwin under Windows, and MS Visual Studio under Windows.

OMNeT++ is free for academic and non-profit use.

5.1.1 Modules

The structure of the actual system simulated is captured through the definition of modules. In OMNeT++, there are two kinds of modules: *compound*, and *simple*. Simple modules are modules at the lowest level of the module hierarchy, and they encapsulate actual behavior of the system. Such simple modules are implemented in C++, using the simulation library. Compound modules are modules consisting of one or more simple and/or other compound modules. There is no limit in the nesting of compound modules, which enable OMNeT++ to capture a realistic description of the simulated system.

An OMNeT++ model consists of hierarchically nested modules, which communicate with each other through messages. Each module is an instance of a module type and each module may be parameterized to allow easy configuration of the simulation model during run-time. The top level module in each simulation model is the system module. The system module contains submodules, which can also contain submodules themselves (see Figure 5.1). The depth of module nesting is not limited; this allows the user to reflect the logical structure of the actual system in the model structure. Model structure is described in OMNeT++'s NED language [95].

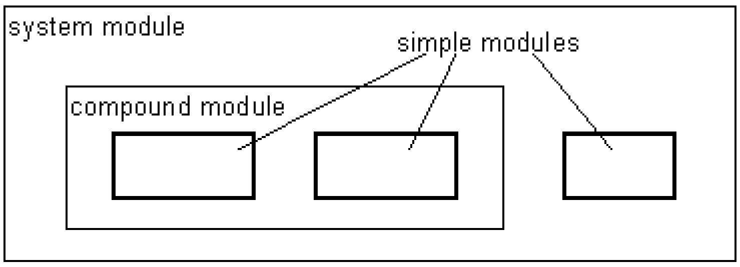

Figure 5.1: Modules in OMNeT++.

5.1.2 Components of a Simulation Model

An OMNeT++ model consists of the following components:

- **Topology and Module Descriptions**: These files, written in the NED topology description language provided by OMNeT++, provide descriptions for the network topology and the modules of the system. Connections between modules and module parameters, along with module definitions are specified in these files which have the *.ned* suffix.
- **Message Definitions**: Modules communicate with each other through messages. Files with the suffix *.msg* are message descriptions, which are automatically converted into corresponding C++ classes. Different types of messages may be defined, and each message type may contain various data fields, including complex and user-defined data types.

- **Simple Module Sources**: Simple module sources are provided in C++ files, with the extensions of *.cc* and *.h*. Note that all features provided by C++ may be used during the implementation of simple modules, including complex data structures, C++ library functions, object-oriented programming concepts such as polymorphism, etc, as well as data structures and functions provided by the simulation library.

5.1.3 The INET Framework for OMNeT++

The INET framework is an open-source communication networks simulation package, written for the OMNEST/OMNeT++ simulation system. The INET framework contains implementations of several Internet protocols, including UDP, TCP, IPv4, IPv6, ARP, ICMPv4, ICMPv6, OSPFv2, RIP, IEEE 802.11, Ethernet, PPP and MPLS with LDP and RSVP-TE signaling and several application models, such as TCP/UDP client/server models, various TCP/UDP/IP traffic generators, TCP/UDP applications, and various facilities such as different host definitions, scenario managers, static and dynamic routing support, several queue implementations, network configurator, routing and interface tables, and notification boards.

The INET framework also supports wireless and mobile simulations. These features of the INET framework have been derived from another framework written for OMNeT++, named the Mobility Framework.

5.1.4 The Mobility Framework (MF) for OMNeT++

This framework is intended to support wireless and mobile simulations within OMNeT++. The core framework implements support for node mobility, dynamic connection management and a wireless channel model. Additionally the core framework provides basic modules that can be derived in order to implement own modules. The framework can be used for simulating fixed and mobile wireless networks, distributed (ad-hoc) and centralized networks, sensor networks, multichannel wireless networks, and many other simulations that need mobility support and/or a wireless interface.

The MF provides modules for the following groups:

- **Application Layer Modules**: These modules include a base class for the application layer, a burst traffic application, polling and client applications to test centralized scenarios, and a test class for the application layer.
- **Network Layer Modules**: MF includes a base class for network layer modules and a simple flooding protocol in this group.
- **Network Interfaces**: There are three subgroups in this category: radios, MAC layer modules, and physical layer modules. MAC layer modules include implementations of a CSMA-based MAC layer, IEEE 802.11b MAC, and pure Aloha. Radio modules include implementation of a single channel radio model. Physical layer modules include decider modules which are used in order to decide whether a frame was received correctly, and SNR evaluation modules, which are used for signal attenuation, path loss and interference calculations.
- **Mobility Modules**: This category includes implementation of various mobility models, as well as base classes to enable the programmer to develop his/her own mobility models.

5.1.5 Modules Directly Used In Our Implementation

In the implementation of our proposed protocol, we have made extensive use of the simulation library of OMNeT++. We have also employed various modules provided by the INET framework, parts of which are inherited from the Mobility Framework. We have used two main components of the INET framework during the implementation of our protocol: *the IEEE 802.11b MAC implementation*, along with needed support modules such as a single channel radio module, SNR evaluators and deciders, and mobility support of INET, primarily using *the* *random waypoint mobility model.* Brief descriptions of employed modules provided by INET, as well as explanations of original modules may be found in the following section.

5.2 Modules and Messages Used in Our Protocol

In this section, we provide details on our simulation model, defined through modules in OMNeT++. Message types used by Elessar are also presented here.

5.2.1 Modules

The simulation model consists of various module definitions in OMNeT++. These modules may be *compound* or *simple* modules. Simple modules encapsulate the behavior of the system and are implemented in C++ using the simulation class library. Details on the inner workings and algorithms of simple modules are presented in Section 5.3. Compound modules are modules made up of other compound and/or simple modules.

All module definitions are written using the NED topology description language provided by OMNeT++. In this section, we provide the definitions of our modules, as well as brief descriptions of modules used from the INET framework. Our model consists of the following modules:

1. **AdhocNetwork** An original compound module defining the actual wireless mobile ad hoc network.
2. **FlowController** Original simple module used to create data flows in the network.
3. **MACTable** An original simple module used to disseminate the MAC addresses of nodes.

4. **CentralNode** An original simple module used to collect performance information on the protocol.
5. **ChannelControl** A simple module employed from INET, used to obtain information on the locations and connectivity of wireless nodes.
6. **MyMobileHost** An original compound module defining a wireless mobile ad hoc host in the network.
7. **TrafficGen** This is an original simple module used in order to create data packets. This module basically acts as the application layer that requests from the network layer transmission of various packets to other nodes.
8. **MyRouter** An original simple module that represents the network layer of a mobile host.
9. **NotificationBoard** A simple module employed from INET, used in order to be notified when a node moves.
10. **BasicMobility** A simple base module employed from INET, used to represent the mobility model of a mobile node.
11. **RandomWPMobility** A simple module employed from INET, used to instantiate the mobility model of a wireless node to the random waypoint mobility model.
12. **Ieee80211NicAdhoc** A compound module employed from INET, representing an IEEE 802.11b network interface card operating in ad hoc mode.
13. **Ieee80211Radio** A simple module employed from INET, representing the radio model used in Ieee80211NicAdhoc.
14. **Ieee80211Mac** A simple module employed from INET, representing the MAC layer used in Ieee80211NicAdhoc.
15. **Ieee80211MgmtAdhoc** A simple module employed from INET, representing the management module of Ieee80211NicAdhoc.

We provide details on these modules below.

5.2.1.1 Module AdhocNetwork

The *AdhocNetwork* module is a compound module representing the actual wireless mobile ad hoc network in the simulation. This is the top module in the simulation (aside from the *system module* discussed in Section 5.1.1).

This module is composed of the modules *ChannelControl*, an array of *MyMobileHost*s, *MACTable*, *FlowController*, and *CentralNode*. It has three parameters: *sizeX* and *sizeY*, two integers representing the size of the width and height of the playground in meters, respectively, and *nodeCount*, an integer representing the number of nodes in the network. A graphical representation of a network consisting of 10 nodes in an area of 250×250 m^2 is presented in Figure 5.2.

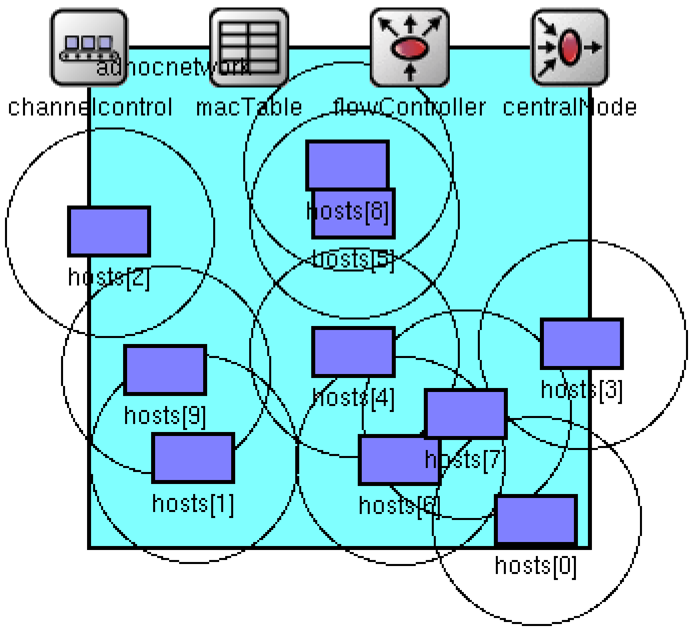

Figure 5.2: Wireless ad hoc network of 10 nodes in an area of (250 x 250).

5.2.1.2 Module FlowController

The *FlowController* is a simple module used to create and control data flows in the network. This is a central module, with one instance in each network. It directly communicates with the *TrafficGen* submodules of necessary hosts, giving them instructions on packet creation for data flows. We currently create data flows at random, choosing random source and destination node pairs. The number of data flows, the data rates, the flow types (i.e. normal vs. real-time), the QoS requirements of real-time flows, requests of cost messages for real-time flows are all configurable.

This module has one parameter, *hostCount*, representing the number of nodes in the network.

5.2.1.3 Module MACTable

This is a simple module, with a single instance in each network. It is used to provide a centralized look-up table in order to let nodes learn of their MAC addresses. Each network interface card is automatically assigned a random MAC address by the simulator, and the *MACTable* module provides a central table where each MAC address is stored, in order to be retrieved later by each individual node. It has one parameter, *size*, representing the number of nodes in the network and therefore the size of the MAC table in the module. Modules of type *MyMobileHost* communicate with the MACTable module through direct method calls.

5.2.1.4 Module CentralNode

CentralNode is a simple module, having a single instance in each network. It is used to provide a centralized location for data acquisition, collection, and recording during each simulation run. Modules of type *MyMobileHost* communicate through direct method calls with this module.

5.2.1.5 Module ChannelControl

This is a simple module, with a single instance in each network. This module gets informed about the location and movement of nodes, and determines which nodes are within communication or interference distance. This information is then used by the radio interfaces of nodes at transmissions. This module is provided by the INET framework.

5.2.1.6 Module MyMobileHost

This is a compound module, consisting of the modules *TrafficGen*, *MyRouter*, *BasicMobility*, *NotificationBoard* and *Ieee80211NicAdhoc*. This module represents a wireless mobile host in the network, having an IEEE 802.11b network interface card.

It has two parameters: $mobilityType$, which is a string used to instantiate the actual mobility model of the host, and $hostId$, an integer value representing the unique node identification number assigned to this node. Note that $hostId > 0$ for all hosts in the network. Each host is given their unique ID during the network setup phase in the description of the *AdhocNetwork* module. The mobility type of each host is also assigned during network setup, through the configuration file of the simulation. A graphical representation of this module may be found in Figure 5.3.

5.2.1.7 Module TrafficGen

TrafficGen is a simple module developed by us, with each mobile host having their own instance of this module. It is connected to the *MyRouter* module in each host and also receives instructions regarding the creation of data packets and flows from the *FlowController* module directly. It acts as the application layer of a mobile host, creating data packets and requesting the network layer to send them to their destinations.

This module has a single parameter, $myId$, representing the unique node ID of the host the module belongs to. This parameter is set in the definition of module *MyMobileHost*.

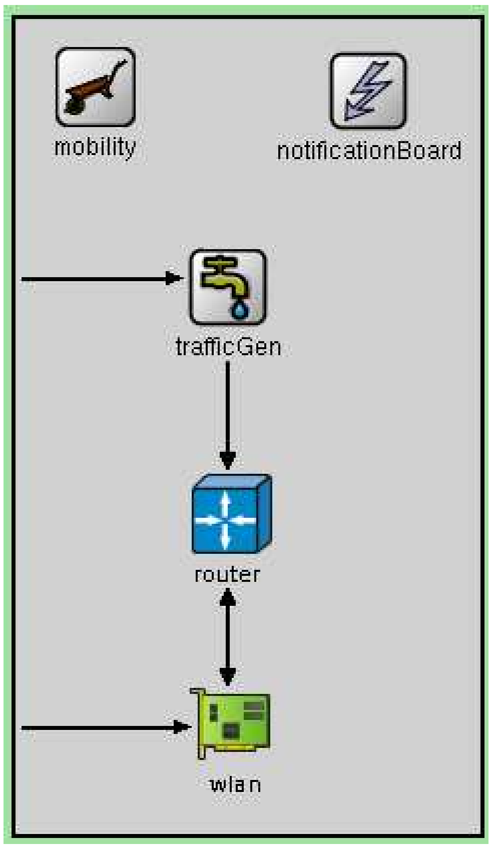

Figure 5.3: Internal representation of a wireless mobile host.

5.2.1.8 Module MyRouter

This module is practically the heart and soul of our simulation, as it captures the algorithms and actions proposed in our routing protocol. This module represents the network layer of a mobile host, performing route discovery and packet forwarding, as well as providing the implementation of supporting mechanisms for the routing protocol, such as link cost measurement. Each wireless host has an instance of this module. It is connected to the *TrafficGen* module, performing routing actions on behalf of the data packets sent from the *TrafficGen* module, and to the *Ieee80211NicAdhoc* module, which acts as the MAC layer of the host. *MyRouter* requests the *Ieee80211NicAdhoc* module to do the actual transmission of the data packets according to the information provided in the packet headers and the *Ieee80211NicAdhoc* module delivers received data frames up to module *MyRouter* for processing, mimicking the expected actions between the network and link layers.

MyRouter is a simple module with two parameters: *nodeId*, representing the unique ID of this node and *wholeK*. Using the incremental link-state message scheme, every *wholeK*-th link-state message is a whole message. Parameter *nodeId* is set in the module definition for *MyMobileHost*, while the *wholeK* parameter is set in the simulation configuration file.

5.2.1.9 Module NotificationBoard

This is a simple module employed from INET and each mobile host has an instance of this module. Modules can notify each other about "events", such as routing table changes, interface status changes (up/down), interface configuration changes, wireless handovers, changes in the state of the wireless channel, mobile node position changes, etc, using the *NotificationBoard* module. We explicitly use this module to inform hosts of node position changes, but it is also used implicitly by module *Ieee80211NicAdhoc* to learn of wireless channel state changes.

NotificationBoard is accessed via direct C++ method calls and not message exchanges. Modules subscribe to categories of changes (e.g. "routing table change" or "radio channel became empty"). When such a change occurs, the corresponding module will inform the *NotificationBoard* of the situation, and the *NotificationBoard* will disseminate this information to all interested modules.

5.2.1.10 Modules BasicMobility and RandomWPMobility

Module *BasicMobility* is a simple module employed from INET, which does not actually provide a mobility model, but is used as a prototype for other mobility models. Each wireless host has an instance of this module, and each instance is assigned the mobility model provided in simple module *RandomWPMobility*, employed from INET.

In the random waypoint mobility model, a node moves in line segments. For each line segment, a random destination position (distributed uniformly over the playground) is chosen and the node moves toward this destination with a given *speed*. The node *speed* value is a module parameter and it may be a constant or a distribution from where a random value is chosen for each line segment. When the node reaches the target position, it waits for the time $waitTime$, which can also be defined as a variate or a constant as is the case with *speed*. After this time the algorithm calculates a new random position, and the cycle continues.

The *RandomWPMobility* module has the following relevant parameters: x, and y, which respectively represent the x and y coordinates of the starting point of the node in the playground. These values may be set as to let the simulator place each node randomly over the playground. Parameters *speed* and $waitTime$ were discussed above.

5.2.1.11 Modules Ieee80211NicAdhoc, Ieee80211Radio, Ieee80211Mac, and Ieee80211MgmtAdhoc

Module *Ieee80211NicAdhoc* represents an IEEE 802.11b network interface card operating in ad hoc mode (see Figure 5.4). It is a compound module, consisting of the simple modules *Ieee80211Radio*, representing the physical radio component, *Ieee80211Mac*, representing the actual MAC layer, and *Ieee80211MgmtAdhoc*, representing the management module operating in ad hoc mode.

Simple module *Ieee80211Mac* provides the implementation of the 802.11b MAC protocol. This module is intended to be used in combination with the *Ieee80211Radio* module as the physical layer. Encapsulation/decapsulation of frames are done in module *Ieee80211MgmtAdhoc*. The following features are not supported by the current INET implementation of the IEEE 802.11b MAC protocol: 1) fragmentation, 2) power management, 3) polling (PCF). Also, physical layer algorithms such as frequency hopping and direct sequence spread spectrum are not modeled directly. Fields related to the above unsupported features are omitted from management frame formats as well (for example, FH/DS/CF

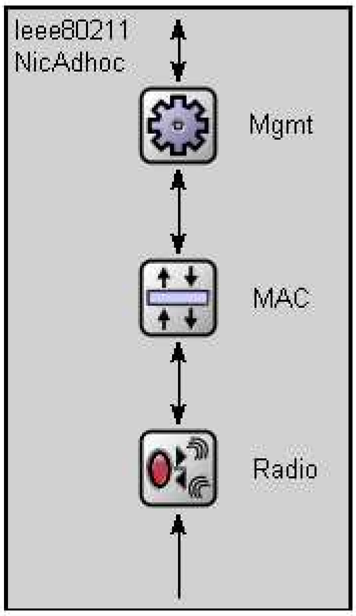

Figure 5.4: Internal representation of the Ieee80211NicAdhoc module.

parameter sets, beacon/probe timestamp which is related to physical layer synchronization, listen interval which is related to power management, capability information which is related to PCF and other non-modeled features). Simple module *Ieee80211Radio* provides the physical layer and is intended to be used in conjunction with the *Ieee80211Mac* module.

5.2.2 Message Types

In this section, we present information regarding the message types used by Elessar. Our protocol currently makes use of the following message types:

1. **Packet** The base class for all messages used by the protocol.
2. **LSMsg** A link-state message.
3. **DataNormal** A normal data packet.
4. **DataRT** A real-time data packet.

5. **SpecialRTMsg** A message informing the nodes along its path of the start of a real-time message.
6. **CostInformMsg** A message informing receiving nodes about a real-time flow.
7. **CostMsg** A cost message, including the link costs of the sending node.

We describe each message type in the following sections.

5.2.2.1 Packet

The *Packet* message type is not used by itself, but forms the base class for all messages used by the protocol. Its definition is provided below:

```
Packet
message Packet
{
    fields:
      int source;
      int dest; // includes LOCAL_BC and GLOBAL_BC
      int hopCount;
      int pktType;
      unsigned long seqNum;
};
```

Its fields are self-explanatory. The source and destination fields (represented by node IDs) of each packet are modified during multi-hop communication by the node forwarding the packet. We would like to note here that the hop count field is not used by all message types. This field is modified (decremented by 1) by a forwarding node if the packet is forwarded based on its hop count. Fields packet type and sequence number are unmodified once they are set by the creator of the packet.

5.2.2.2 LSMsg

The *LSMsg* message type forms the definition of a link-state message. Its definition is provided below:

```
LSMsg
message LSMsg extends Packet
{
    fields:
      int originator;
      int msgType; // incremental vs. whole
      LSMsgContent content;
};
```

The *content* field of a link-state message is of a user-defined type *LSMsgContent*. The header file for this class is provided in the appendix. Basically, the *content* field may contain two different types of data structures based on whether this is an incremental or a whole link-state message. If this is a whole message, then the *content* field contains the IDs of all current neighbors of the message originator in a list. Otherwise, it contains the changes that have occurred in the neighbor list of the originator since the last time that node created a link-state message.

5.2.2.3 DataNormal

DataNormal represents a normal data message, where we use the term "normal" to indicate that this packet does not belong to a real-time flow. The definition of the *DataNormal* message type is given below:

```
DataNormal
message DataNormal extends Packet
{
    fields:
      string data; // carried data
    Route route; // embedded route this pkt should follow
};
```

A *DataNormal* message has two additional fields to the fields inherited from *Packet*. Field *data* is a string, representing the actual data being carried by the packet, and *route* is the embedded route this packet should follow, used in source routing. You may find the header file for the *Route* class in the appendix.

5.2.2.4 DataRT

DataRT is a real-time data packet, with the following definition:

DataRT

```
message DataRT extends Packet
{
    fields:
      string data; // carried data
      Route route; // embedded route this pkt should follow

      double duration; // estimated duration of rt flow (in sec.s)
      int costCount; // no of cost msgs requested during rt flow
      bool periodic; // request periodic cost msgs or not
      double period; // in case of periodic cost msgs, period (in sec.s)
      int costInformHopCount; // hop count for cost inform msgs
      int pathCount; // number of paths special rt msg for
                     // this flow should be sent over

      // QoSReq qosReq;
      // The route object includes a pointer to a QoSReq object.
      // This pointer may also be used here, so there is no need for
      // an explicit QoSReq object here.
};
```

It has all of the fields of a *DataNormal* message, along with some additional ones. *duration* is the estimated duration in seconds of the real-time flow this packet belongs to. *costCount* is the number of cost messages requested during the lifetime of the flow; *periodic* is a boolean representing whether periodic cost messages are requested or not; *period* is the requested period in seconds of cost messages if periodic cost messages are used; $costInformHopCount$ is the hop count imposed on $CostInformMsg$ messages, used during information of nodes of the real-time flow; and *pathCount* is the number of paths requested by the flow source, over which one or more $SpecialRTMsg$ messages are sent to inform nodes of the start of a real-time flow.

5.2.2.5 SpecialRTMsg

A *SpecialRTMsg* represents the special packet sent over the initial path(s) of the real-time flow in order to inform nodes along the path(s) of the start of the real-time flow. This information of nodes is the first part of the cost dissemination mechanism, and we achieve this information through the use of $SpecialRTMsg$ and $CostInformMsg$ messages.

SpecialRTMsg

```
message SpecialRTMsg extends Packet
{
    fields:
      int rtSource; // source of rt flow
      int rtDest; // dest of rt flow
      Route route; // embedded route this pkt should follow

      double duration; // estimated duration of rt flow (in sec.s)
      int costCount; // no of cost msgs requested during rt flow
      bool periodic; // request periodic cost msgs or not
      double period; // in case of periodic cost msgs, period (in sec.s)
      int costInformHopCount; // hop count for cost inform msgs

      // QoSReq qosReq;
      // Note: Route object includes a QoSReq ptr that can be
      // used to identify the QoS req. of the flow this pkt
      // represents, so another QoSReq qosReq field is not added.
};
```

The *SpecialRTMsg* message definition is provided above. Such a message inherits all of the fields of *Packet*, includes all of the real-time specific fields except *pathCount*, presented above on the discussion of message *DataRT*, and also has three additional fields: *rtSource*, which is the source of the flow, *rtDest*, the destination of the flow, and *route*, the route that this packet should follow, used in source routing.

5.2.2.6 CostInformMsg

Message type *CostInformMsg* is used in the information of nodes of the start of a real-time flow. Nodes along the initial path(s) are informed of the start of a real-time flow by the special packet of type *SpecialRTMsg* sent over the path(s) the flow will follow at the start. Such nodes also inform their neighbors of the real-time flow by the use of *CostInformMsg* messages, which are broadcasted limited by the hop count imposed on them.

CostInformMsg

```
message CostInformMsg extends Packet
{
    fields:
      int sendTo; // send your costs to this node
      int type; // send this type of cost

      double duration; // estimated duration of rt flow (in sec.s)
      int costCount; // no of cost msgs requested during rt flow
      bool periodic; // request periodic cost msgs or not
      double period; // in case of periodic cost msgs, period (in sec.s)
      int costInformHopCount; // hop count for cost inform msgs
};
```

A *CostInformMsg* includes all of the real-time flow related fields present in message type *SpecialRTMsg*, as well as two fields of its own, *sendTo*, denoting the source of the flow that should receive the cost messages to be sent, and *type*, denoting the type of the link costs that should be included in the cost messages.

5.2.2.7 CostMsg

A *CostMsg* message is the actual message including the link costs of the originator node, sent to the source of the real-time flow to enable intelligent path selection at the source node.

CostMsg

```
message CostMsg extends Packet
{
    fields:
      Route route; // embedded route this pkt should follow

      // if this msg includes cost for a specific type,
      // corresponding bool is true, and vice versa
      // note that at most two of these bools can be false
      bool hasRTT;
      bool hasBW;
      bool hasLoss;

      // note that at most two of these contents can be null (empty)
      CostMsgContent costRTT;
      CostMsgContent costBW;
      CostMsgContent costLoss;
};
```

It has the following fields: *route*, representing the route this cost message should follow, used in source routing, three boolean variables, used to denote which type of link costs are included in the message, and three data structures of type *CostMsgContent*, each potentially including the corresponding type of link costs. Each *CostMsgContent* data structure includes the current and average link costs of corresponding type of the originator node's links to its current neighbors. Header file for class *CostMsgContent* is given in the appendix.

5.3 Pseudocodes and Algorithms

Pseudocodes and algorithms of the proposed protocol are given here. The reason why we are providing this material in this section, and not in Chapter 4, is for clarity purposes, as we wanted to provide a concise specification of our protocol in Chapter 4 and left more technical details of Elessar for discussion here.

5.3.1 Neighborhood Beaconing

We present the pseudocode for the basics of the neighborhood beaconing in this section. The following notations are used in the presented algorithms:

B A beacon message. It has two fields: *sender*, representing the sender of the message and *type*, representing the type of the beacon message. These fields may be accessed by the usual dot notation (i.e. to access the *sender* field of message B, we use $B.sender$).

nodeId The unique ID of the current node. Each node in the network is given a numeric ID that uniquely represents the node.

nList The neighborhood list of the node. Information on current neighbors (such as IDs, etc.) are kept in this list. We currently assume that the neighbor list only keeps the node IDs of this node's neighbors.

retryLimit The limit for the number of retries in joining the network. The node will retry to join the network at most this many times.

inNetwork Whether the node has joined the network or not.

P A packet. It has at least one field: *sender*, representing the originator of the packet.

msg A link-state message. Please see the topology dissemination section for more information on this.

We use a single timer for the creation and sending of beacon messages. When this beacon timer expires, a new beacon message is created and locally broadcasted in the network. Note that since the wireless medium is inherently a broadcast medium, the node only needs to send the beacon once in order to reach all its neighbors. We also use a timer for each neighbor in the neighbor list ($nList$). When the timer for a neighbor expires, it is removed from the neighbor list.

Since a node detects changes in its neighborhood through the neighborhood beaconing mechanism, and such changes trigger the dissemination of a link-state message, the neighborhood beaconing mechanism and the topology dissemination mechanism are closely related. More specifically, procedure $createLSMsg()$ is called when a change in the neigborhood of the current node is detected through the beaconing mechanism and the created link-state message is broadcasted. Please refer to the section on topology dissemination below for more information on these procedures.

We would like to note here that the neighborhood mechanism presented here is not implemented in our simulations as the simulation environment provides us the neighborhood information.

5.3.2 Topology Dissemination

We present the pseudocode for the topology dissemination mechanism here. Since this mechanism forms the backbone of our protocol, many other components of the protocol rely heavily on it. One such component is the route discovery protocol. We have mentioned that we keep a local route cache at each node. We represent this route cache as $routeCache$ in the algorithms below. There is a one-way interaction between link-state messaging and this route cache in that we need to clear the route cache when we receive a link-state message as mentioned in the section on route discovery. We also maintain a single timer for the route cache, and when this timer expires, we clear all routes in the route cache. As mentioned earlier, this is done to eliminate stale routes from the route cache and to enable the protocol to find QoS routes as it receives cost messages from nodes

Algorithm 2 Neighborhood Beaconing, Part 1

```
1: if $\neg inNetwork$ then
2:     create beacon $B$ with $B.sender \leftarrow nodeId$ and $B.type \leftarrow join$
3:     $retries \leftarrow 0$
4:     local broadcast beacon $B$
5:     create and start beacon timer
6: end if

7: if beacon timer expired and $inNetwork$ then
8:     create beacon $B$ with $B.sender \leftarrow nodeId$ and $B.type \leftarrow normal$
9:     local broadcast beacon $B$
10:    restart beacon timer
11: else if beacon timer expired and $\neg inNetwork$ and $retries < retryLimit$
    then
12:    create beacon $B$ with $B.sender \leftarrow nodeID$ and $B.type \leftarrow join$
13:    $retries \leftarrow retries + 1$
14:    local broadcast beacon $B$
15:    restart beacon timer
16: else if beacon timer expired and $\neg inNetwork$ and $retries \geq retryLimit$
    then
17:    delete beacon timer
18:    abort joining procedure
19: end if

20: if neighbor timer expired for neighbor $i$ then
21:    $nList \leftarrow nList - \{i\}$
22:    $msg \leftarrow createLSMsg()$
23:    broadcast $msg$
24: end if

25: if beacon $B$ received then
26:    $i \leftarrow B.sender$
27:    $t \leftarrow B.type$

28:    if $i \in nList$ then
29:        restart timer for neighbor $i$
30:    else
31:        $nList \leftarrow nList \cup \{i\}$
32:        create and start timer for neighbor $i$

33:        if $t \neq leave$ then
34:            $msg \leftarrow createLSMsg()$
35:            broadcast $msg$
36:        end if
37:    end if
```

Algorithm 3 Neighborhood Beaconing, Part 2

```
38:     if $t = normal$ or $t = join$ then
39:         create reply beacon $B_r$ with $B_r.sender \leftarrow nodeId$ and $B_r.type \leftarrow reply$
40:         local broadcast beacon $B_r$
41:         restart beacon timer
42:     else if $t = leave$ then
43:         $nList \leftarrow nList - \{i\}$
44:         stop and remove timer for neighbor $i$
45:     else if $t = reply$ and $\neg inNetwork$ then
46:         $inNetwork \leftarrow true$
47:         process boot-up information in $B$
48:     end if
49: end if

50: if node wishes to leave network then
51:     create beacon $B$ with $B.sender \leftarrow nodeId$ and $B.type \leftarrow leave$
52:     local broadcast $B$
53:     $inNetwork \leftarrow false$
54:     $nList \leftarrow \emptyset$
55:     leave the network
56: end if
```

Algorithm 4 Passive Neighborhood Beaconing

```
if packet $P$ received then
    $i \leftarrow P.sender$

    if $i \in nList$ then
        restart timer for neighbor $i$
    else
        $nList \leftarrow nList \cup \{i\}$
        create and start timer for neighbor $i$
        $msg \leftarrow createLSMsg()$
        broadcast $msg$
    end if
end if

if sending out packet $P$ then
    restart beacon timer
end if
```

in the network. This periodic route cache cleansing is provided in pseudocode in the section on route discovery.

Each node keeps its own global view of the wireless network in a directed simple graph representation. We represent this graph as $G = (V, E)$, where V = set of all vertices (nodes) in the graph and E = set of all edges (links) in the graph. Each wireless node corresponds to a node in the graph, and if node i can transmit to node j, then edge $(i, j) \in E$. Note that our implementation works on both directed and undirected graph representations, as we may transform an undirected graph to an equivalent directed graph in a very straightforward way. Most wireless networks will probably have bidirectional links due to identical transceivers, meaning that if node i can reach node j, then node j may reach node i (i.e. $(i, j) \in E \Leftrightarrow (j, i) \in E$). However, the characteristics of such links may be quite different from each other. For example, the noise of link (i, j) may be higher/lower than the noise of link (j, i). Therefore, even though the links are bidirectional, they may be asymmetric.

Each node in the graph has a unique integer ID > 0. We represent the neighbors of each node in the adjacency list representation. The adjacency list of a node $v \in V$ is represented as $Adj[v]$.

We use the following notations in the algorithms presented below:

msg A link-state message. It has at least two fields, *type*, representing the type of the received message (incremental vs. whole), and *sender*, the originator of the message.

LSTable The table that keeps track of the most recently seen link-state messages. Please see Section 4.3.2.2 for a detailed description of this table.

nodeId The unique ID of the current node. Each node in the network is given a numeric ID that uniquely represents the node. Note that $nodeId > 0, \forall v \in V$.

wholeK Every $wholeK$-th link-state message is a whole message. This is done in order to increase robustness of the protocol. Please see Section 4.3.2.1

for more information on the incremental link-state messaging scheme.

currentK Keeps track of the count of created link-state messages. Used primarily in order to decide if the new link-state message will be an incremental or a whole message.

lastLSNeighbors This is a snapshot of the neighbor list of the current node at the time of the creation of the last link-state message by the node. It is used in order to create an incremental message.

nList The neighborhood list of the node. Information on current neighbors (such as IDs, etc.) are kept in this list. We currently assume that the neighbor list only keeps the node IDs of this node's neighbors.

routeCache The route cache of the current node. Please refer to Section 4.3.4.1 for a detailed discussion on the use of this data structure.

Please note that we keep the $LSTable$ in FIFO[1] manner. Since the table has a limited capacity, when we say $LSTable \leftarrow LSTable \cup \{msg\}$, if the table can accommodate the new message, msg is added to the table in FIFO manner, without deleting any other message from the table. If there is no space left in the table, then the oldest message in the table (the message at the front of the queue) is deleted and msg is added to the end of the queue.

The current implementation of $LSTable$ is based on a linked-list. Since the table is kept in FIFO manner, addition to and removal from the table each take $O(1)$ time whereas finding an element in the table takes $O(n)$ time through exhaustive search, where n is the size of the table. Please note that the current implementation is for experimental purposes only, and if real-life deployment of the protocol is required, then we may use a more efficient data structure in order to implement the link-state table. The same argument applies to almost all data structures used in the protocol, such as $RTFlowsTable$ (mentioned in Section 5.3.3.3), $routeCache$, $nList$, $lastLSNeighbors$, etc.

[1]FIFO: first-in, first-out data structure, i.e. a queue

We have implemented the incremental message scheme along with inclusion of neighborhood-information-only in the link-state message content. An incremental message includes only information about changes at a node that have occurred since the last link-state message created by that node. A whole message, on the other hand, includes all of the neighbors of the node at the time of the creation of the message.

An incremental link-state message *msg* includes the information in a field named *changeList*, which is a list consisting of changes that have occurred since the last link-state message. Each entry in *changeList* represents a change (removal or addition) in the neighborhood of the originator and is a tuple in the form of (<*action*>, <*nodeId*>), where <*action*> is one of $\{remove, add\}$ and <*nodeId*> is the *nodeId* of the neighbor that has been removed from or added to the neighbor list at the originating node.

A whole link-state message *msg* includes all of the neighbors of the originator node in a list named *nList*, where each entry is the *nodeId* of a current neighbor in the neighbor list of the originator.

The $processLSMsg(msg)$ procedure is called when the node receives a link-state message *msg*. First, the node checks if it has seen the received message before. If it has, it silently discards the message. Otherwise, the received link-state message is recorded in the link-state table, the route cache is cleared, and the graph representation of the network is updated. Furthermore, the timer for the route cache is restarted since it has been cleared and the node locally broadcasts the message in order to continue the global broadcast mechanism. Notice that a node will forward a link-state message only once, and since there are a finite number of nodes in the network, we are guaranteed that the global broadcast will stop eventually. To be more precise, the global broadcast will take at most $d + 1$ steps, where d is the diameter[2] of the network, with step 1 consisting of the originator of the message sending the message, step 2 consisting of the 1-hop neighbors of the originator forwarding the message and so on[3].

[2]Diameter of a network is defined as the number of hops in the shortest path between the furthest pair of nodes.

[3]For the mathematically inclined, this may be expressed as $Step_i$ = forwarding of the message

$updateGraphLS(msg)$ is a helper procedure that is called in $processLSMsg$. It first checks the type of the received link-state message, and proceeds accordingly. The nodes and edges that need to be added to or removed from the graph at the current node are determined by looking at the content of the link-state message.

$createLSMsg$ is called when we need to create a new link-state message. This procedure creates either a whole or an incremental message based on the values of $currentK$ and $wholeK$. $extractDiffNeighbors$ is a helper procedure which is used when we need to create an incremental message and it extracts the difference between the current neighbor list and the neighbor list sent in the last link-state message and returns it as a list in the format of $changeList$.

5.3.3 Route Discovery and Packet Forwarding

It has been noted that each node has its own view of the global network and is therefore able to run a local path finding algorithm in order to find a path from itself to the destination node in the graph. The route discovery mechanism runs in two modes: *normal mode*, and *QoS mode*. The route discovery mechanism of Elessar operates in normal mode for normal data flows, and in QoS mode for real-time data flows. Before we present the pseudocode for the packet handling and route embedding parts of route discovery, we would like to expose the reader to the algorithms at the heart of the mechanism: the actual local path finding algorithms used on the graph representation, which are either modified versions of Dijkstra's well-known single source shortest paths algorithm [36, 38] or a derivative thereof.

5.3.3.1 Local Path Finding Algorithms

Please note that the following notation is used in the given algorithms:

G(V,E) The directed graph representation of the network at a node. V is the

by all nodes which are exactly $i-1$ hops away from the originator.

Algorithm 5 Topology Dissemination, Part 1

```
 1: procedure PROCESSLSMSG($msg$)
 2:     if $msg \in LSTable$ then
 3:         delete $msg$
 4:     else
 5:         $LSTable \leftarrow LSTable \cup \{msg\}$
 6:         do $updateGraphLS(msg)$
 7:         $routeCache \leftarrow \emptyset$
 8:         restart timer for route cache
 9:         broadcast $msg$
10:     end if
11: end procedure

12: procedure UPDATEGRAPHLS($msg$)
13:     if $msg.sender \notin V$ then
14:         $V \leftarrow V \cup \{msg.sender\}$
15:     end if

16:     if $msg.type = incremental$ then
17:         for all $(action, id) \in msg.changeList$ do
18:             if $id \notin V$ then
19:                 $V \leftarrow V \cup \{id\}$
20:             end if

21:             if $action = add$ then
22:                 if $(msg.sender, id) \notin E$ then
23:                     $E \leftarrow E \cup \{(msg.sender, id)\}$
24:                     $E \leftarrow E \cup \{(id, msg.sender)\}$          ▷ needed if links are bidirectional
25:                     $Adj[msg.sender] \leftarrow Adj[msg.sender] \cup \{id\}$
26:                     $Adj[id] \leftarrow Adj[id] \cup \{msg.sender\}$     ▷ needed if links are bidirectional
27:                 end if
28:             else
29:                 if $(msg.sender, id) \in E$ then
30:                     $E \leftarrow E - \{(msg.sender, id)\}$
31:                     $E \leftarrow E - \{(id, msg.sender)\}$          ▷ needed if links are bidirectional
32:                     $Adj[msg.sender] \leftarrow Adj[msg.sender] - \{id\}$
33:                     $Adj[id] \leftarrow Adj[id] - \{msg.sender\}$     ▷ needed if links are bidirectional
34:                 end if
35:             end if
36:         end for
```

Algorithm 6 Topology Dissemination, Part 2

```
37:     else
38:         for all $i \in msg.nList$ do
39:             if $i \notin V$ then
40:                 $V \leftarrow V \cup \{i\}$
41:             end if

42:             if $(msg.sender, i) \notin E$ then
43:                 $E \leftarrow E \cup \{(msg.sender, i)\}$
44:                 $E \leftarrow E \cup \{(i, msg.sender)\}$   ▷ needed if links are bidirectional
45:                 $Adj[msg.sender] \leftarrow Adj[msg.sender] \cup \{id\}$
46:                 $Adj[id] \leftarrow Adj[id] \cup \{msg.sender\}$   ▷ needed if links are bidirectional
47:             end if
48:         end for

49:         for all $j \in Adj[msg.sender]$ do   ▷ check for removed edges
50:             if $j \notin msg.nList$ then
51:                 $E \leftarrow E - \{(msg.sender, j)\}$
52:                 $E \leftarrow E - \{(j, msg.sender)\}$   ▷ needed if links are bidirectional
53:                 $Adj[msg.sender] \leftarrow Adj[msg.sender] - \{j\}$
54:                 $Adj[j] \leftarrow Adj[j] - \{msg.sender\}$   ▷ needed if links are bidirectional
55:             end if
56:         end for
57:     end if
58: end procedure

59: procedure createLSMsg
60:     create $msg$
61:     $msg.sender \leftarrow nodeId$
62:     $msg.nList \leftarrow \emptyset$
63:     $msg.changeList \leftarrow \emptyset$

64:     if $mod(currentK, wholeK) = 0$ then
65:         $msg.type \leftarrow whole$
66:         $lastLSNeighbors \leftarrow \emptyset$

67:         for all $i \in nList$ do
68:             $msg.nList \leftarrow msg.nList \cup \{i\}$
69:             $lastLSNeighbors \leftarrow lastLSNeighbors \cup \{i\}$
70:         end for
71:     else
72:         $msg.type \leftarrow incremental$
73:         $msg.changeList \leftarrow extractDiffNeighbors()$
74:         $lastLSNeighbors \leftarrow \emptyset$
```

Algorithm 7 Topology Dissemination, Part 3

```
75:         for all $k \in nList$ do
76:             $lastLSNeighbors \leftarrow lastLSNeighbors \cup \{k\}$
77:         end for
78:     end if

79:     $currentK \leftarrow currentK + 1$
80:     return $msg$
81: end procedure

82: procedure extractDiffNeighbors
83:     $cList \leftarrow \emptyset$

84:     for all $v \in nList$ do
85:         if $v \notin lastLSNeighbors$ then
86:             $cList \leftarrow cList \cup \{(add, v)\}$
87:         end if
88:     end for

89:     for all $y \in lastLSNeighbors$ do
90:         if $y \notin nList$ then
91:             $cList \leftarrow cList \cup \{(remove, y)\}$
92:         end if
93:     end for

94:     return $cList$
95: end procedure
```

set of vertices (nodes) and E is the set of edges (links).

Adj[i] The adjacency list of node $i \in V$. This list represents the nodes that node i has a link to.

rtt(i,j) The current delay on edge (i, j). Note that an edge (i, j) means that node i has a link to node j in the graph. We have $0 < rtt(i, j), \forall (i, j) \in E$.

rttAvg(i,j) The average delay on edge (i, j). Note that this average is an EWMA[4], which is calculated as $rttAvg(i, j) = \alpha \times rtt(i, j) + (1 - \alpha) \times rttAvg(i, j)$.

loss(i,j) The current loss rate on edge (i, j). Note that $0 \leq loss(i, j) \leq 1, \forall (i, j) \in E$.

lossAvg(i,j) The average loss rate on edge (i, j). Note that this is an EWMA and calculated as $lossAvg(i, j) = \beta \times loss(i, j) + (1 - \beta) \times lossAvg(i, j)$.

bw(i,j) The current available bandwidth on edge (i, j). Note that $0 \leq bw(i, j) \leq cap(i, j), \forall (i, j) \in E$, where $cap(i, j)$ is the maximum capacity of edge (i, j).

bwAvg(i,j) The average available bandwidth on edge (i, j). Note that this is an EWMA and calculated as $bwAvg(i, j) = \omega \times bw(i, j) + (1 - \omega) \times bwAvg(i, j)$.

c(i,j) The cost of edge (i, j). This is primarily used in normal path calculation, so $c(i, j) = \lambda : \lambda \in \{0, 1\}, \forall (i, j) \in E$.

d(v) Current distance estimate from the source to node v. Used in Dijkstra's algorithm.

pred(v) The predecessor of node v along the path from source to itself. Used in Dijkstra's algorithm.

MAXBW A very large value for available bandwidths, used in a role similar to ∞, but producing meaningful results on operations like $\infty - someFiniteValue$. $MAXBW$ is chosen so that $(MAXBW - bw(i, j)) \geq 0, \forall (i, j) \in E$.

[4]EWMA: exponentially weighted moving average

Q The minimum priority queue used in Dijkstra's algorithm. $extractMin(Q)$ removes and returns the minimum element in the queue. The comparison between elements in Q is based on the current distance estimates of nodes $(d(v))$.

R A route. The route is represented as a list of node ids. $addToRouteFront(R, i)$[5] adds node with id $= i$ to the front of route R. $addToRouteEnd(R, i)$[6] adds node i to the end of route R. $getRouteFront(R)$[7] returns the node at the front of the route R.

5.3.3.1.1 Local Path Finding Algorithms in Normal Mode Dijkstra's single source shortest paths algorithm is employed in this mode in order to find a path to the destination. Since link costs are not used in this operation mode, the path found is a minimum-hop path[8]. Due to the mobility of nodes and the dynamic nature of the ad hoc network, the graph representation of the network may not be connected or it may not be up-to-date, so we need to add some checks to the classic Dijkstra algorithm in order to ensure correct operation of the procedure. This slightly modified algorithm is presented in algorithms 8, 9.

5.3.3.1.2 Local Path Finding Algorithms in QoS Mode Whenever a node wants to send a real-time flow, it starts the directed cost dissemination and the route discovery mechanism operates in QoS mode, trying to find paths that fulfill the requirements of the real-time flow. Such intelligent and efficient path selection is enabled through the directed cost dissemination mechanism, which only operates in QoS mode. In order to find efficient paths that meet the QoS requirements, a source node sending real-time data runs a local path finding algorithm on its own view of the network. These local path finding algorithms are either modified versions of Dijkstra's shortest paths algorithm or a derivative of it, and we present them here.

[5]Example: With $R = \{3, 9, 2\}$, after calling $addToRouteFront(R, 4)$, $R = \{4, 3, 9, 2\}$.
[6]Example: With $R = \{4, 3, 9, 2\}$, after calling $addToRouteEnd(R, 1)$, $R = \{4, 3, 9, 2, 1\}$.
[7]Example: With $R = \{3, 9, 2\}$, $getRouteFront(R)$ returns 3.
[8]Each existing edge has a cost of 1.

Algorithm 8 Dijkstra's Shortest Paths Algorithm, with Additional Checks, Part 1

```
1: procedure INITIALIZESS(source)
2:     for all $v \in V$ do
3:         $d(v) \leftarrow \infty$
4:         $pred(v) \leftarrow -1$
5:     end for

6:     $d(source) \leftarrow 0$
7: end procedure

8: procedure RELAX(i,j)
9:     if $i > 0$ and $j > 0$ then                         ▷ check for valid ids
10:        if $d(j) > d(i) + c(i,j)$ then
11:            $d(j) \leftarrow d(i) + c(i,j)$
12:            $pred(j) \leftarrow i$
13:        end if
14:    end if
15: end procedure

16: procedure DIJKSTRASSP(source)
17:    if $source \notin V$ then
18:        return
19:    end if

20:    do $initializeSS(source)$

21:    $Q \leftarrow \emptyset$
22:    for all $v \in V$ do                                 ▷ prepare min-PQ
23:        $Q \leftarrow Q \cup \{v\}$
24:    end for

25:    while $Q \neq \emptyset$ do
26:        $i \leftarrow extractMin(Q)$
27:        for all $j \in Adj[i]$ do
28:            do $relax(i,j)$
29:        end for
30:    end while
31: end procedure
```

Algorithm 9 Dijkstra's Shortest Path Algorithm, with Additional Checks, Part 2

```
32: procedure getDijkstraSSP(source, dest)
33:     $R \leftarrow \emptyset$

34:     if $source \notin V$ or $dest \notin V$ then
35:         return $R$
36:     end if

37:     do $DijkstraSSP(source)$

38:     if $pred(dest) = -1$ then                ▷ unable to reach dest from source
39:         return $R$
40:     end if

41:     do $addToRouteEnd(R, dest)$
42:     $pr \leftarrow pred(dest)$

43:     while $pr \neq -1$ do
44:         do $addToRouteFront(R, pr)$
45:         $pr \leftarrow pred(pr)$
46:     end while

47:     if $getRouteFront(R) \neq source$ then
48:         $R \leftarrow \emptyset$
49:         return $R$
50:     end if

51:     return $R$
52: end procedure
```

There are 6 versions of our QoS path finding algorithms:

1. Delay-sensitive path finding, using current delay values of links
2. Delay-sensitive path finding, using average delay values of links
3. Bandwidth-sensitive path finding, using current available bandwidth values
4. Bandwidth-sensitive path finding, using average available bandwidth values
5. Loss-sensitive path finding, using current loss rates of links
6. Loss-sensitive path finding, using average loss rates of links

We present the three main versions (the ones based on current values of links) of our QoS path finding algorithms as the ones based on average values are almost the same as the current-value-based algorithms. We use the following notation in addition to the ones used above:

R A route. This is now a compound structure, having the following fields: *route*, *cost*, and *type*.

R.route A field of R, representing the route as the list of node ids, as defined previously.

R.type A field of R, representing the type of the route. Type can be one of $Normal$, RTT, BW, and $Loss$, representing normal, delay-sensitive, bandwidth-sensitive, and loss-sensitive traffic, respectively.

R.cost A field of R, representing the cost of the route. This field represents the value of the hop count, end-to-end delay, end-to-end loss rate, or the minimum available bandwidth of a route[9] of type $Normal$, RTT, $Loss$, or BW, respectively.

[9]The notion of the minimum available bandwidth of a route is defined in the section on bandwidth-sensitive path finding.

Before continuing, we would like to note that

$$
\begin{aligned}
rtt(i,j) &= rttAvg(i,j) = \infty, \\
bw(i,j) &= bwAvg(i,j) = \infty, \\
loss(i,j) &= lossAvg(i,j) = \infty, \forall (i,j) \in E
\end{aligned}
$$

initially. Furthermore, as stated in sections 4.3.4.4 and 5.3.5, whenever an edge cost on the graph becomes stale, it is cleared and set back to ∞.

5.3.3.1.2.1 Delay-Sensitive Path Finding Algorithms For delay-sensitive traffic, we try to minimize the end-to-end delay along a path. Let $rtt(i,j)$ denote the last measurement of delay on edge $(i,j) \in E$, and a path P from source node v_1 to destination node v_k be denoted by its edges as $P(v_1, v_k) = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)\}$[10]. The end-to-end delay along a path is calculated by the summation of all delays along the links belonging to the path:

$$Delay(P) = rtt(v_1, v_2) + rtt(v_2, v_3) + \ldots + rtt(v_{k-1}, v_k) = \sum_{\forall (i,j) \in P} rtt(i,j) \quad (5.1)$$

We are then trying to find the path from a given source node u to a destination node v with the least end-to-end delay. We may state our problem as a minimization problem as follows,

$$P' = argmin_{P(u,v)} Delay(P(u,v)), \quad (5.2)$$

where P' is the path from u to v in the current graph that has the least end-to-end delay.

[10]Please note here that we list the edges along the path in order to denote the path. This notation is interchangeable with the notation we previously used to represent routes. For example, if $route = \{3, 9, 2\}$, then the corresponding path notation for the same route is $path = \{(3, 9), (9, 2)\}$.

We use Dijkstra's shortest paths algorithm in order to find the least-cost path in terms of current latencies. The algorithm consists of the procedures *initializeSS*$(source)$, *DijkstraSSPRTT*$(source)$, *getDijkstraSSPRTT*$(source, dest)$, and *relaxRTT*(i, j). Procedure *initializeSS*$(source)$ was provided above. Procedure *DijkstraSSPRTT*$(source)$ is the same as procedure *DijkstraSSP*$(source)$, with calls to procedure *relax* replaced with ones to *relaxRTT*. Procedures *getDijkstraSSPRTT*$(source, dest)$ and *relaxRTT*(i, j) are provided in Algorithm 10.

The local path finding algorithm used to find a least-cost path in terms of average latencies is almost the same as the one using current latencies. We just have to replace any occurrence of the term *rtt*(i, j) in *relaxRTT*(i, j) with the term *rttAvg*(i, j).

5.3.3.1.2.2 Bandwidth-Sensitive Path Finding Algorithms In order to support bandwidth-sensitive traffic, we try to maximize the minimum available bandwidth along edges of different paths. To illustrate the idea, please refer to Figure 5.5.

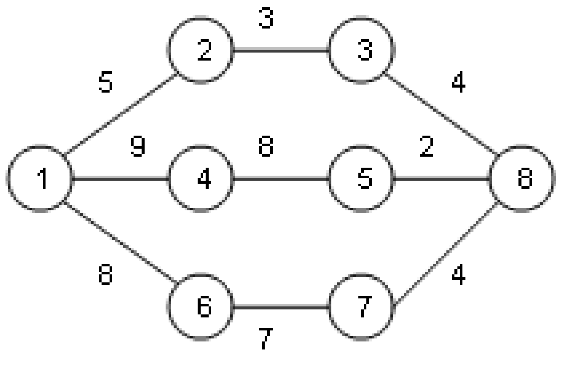

Figure 5.5: Example topology for max-min bandwidth path selection.

In Figure 5.5, a sample topology with current available link bandwidths is presented. With $source = 1$ and $dest = 8$, we have three paths from 1 to 8 on the graph:

Algorithm 10 Dijkstra's Shortest Paths Algorithm, Delay-Sensitive Traffic

```
1: procedure RELAXRTT(i,j)
2:     if $i > 0$ and $j > 0$ then                      ▷ check for valid ids
3:         if $d(j) > d(i) + rtt(i, j)$ then
4:             $d(j) \leftarrow d(i) + rtt(i, j)$
5:             $pred(j) \leftarrow i$
6:         end if
7:     end if
8: end procedure

9: procedure GETDIJKSTRASSPRTT(source, dest)
10:    $R.route \leftarrow \emptyset$
11:    $R.type \leftarrow RTT$
12:    $R.cost \leftarrow 0$

13:    if $source \notin V$ or $dest \notin V$ then
14:        return $R$
15:    end if

16:    do $DijkstraSSPRTT(source)$

17:    if $pred(dest) = -1$ then          ▷ unable to reach dest from source
18:        return $R$
19:    else
20:        $R.cost \leftarrow d[dest]$
21:    end if

22:    do $addToRouteEnd(R.route, dest)$
23:    $pr \leftarrow pred(dest)$

24:    while $pr \neq -1$ do
25:        do $addToRouteFront(R.route, pr)$
26:        $pr \leftarrow pred(pr)$
27:    end while

28:    if $getRouteFront(R.route) \neq source$ then
29:        $R.route \leftarrow \emptyset$
30:        $R.cost \leftarrow 0$
31:        return $R$
32:    end if

33:    return $R$
34: end procedure
```

$$\begin{aligned}
P_1 &= \{(1,2),(2,3),(3,8)\}, \quad minLink(P_1) = (2,3), \quad minBW(P_1) = 3,\\
P_2 &= \{(1,4),(4,5),(5,8)\}, \quad minLink(P_2) = (5,8), \quad minBW(P_2) = 2,\\
P_3 &= \{(1,6),(6,7),(7,8)\}, \quad minLink(P_3) = (7,8), \quad minBW(P_3) = 4,
\end{aligned}$$

where $P_i, i \in \{1,2,3\}$ denotes the i-th path and provides its edges, $minLink(P_i)$ denotes the link with the minimum available bandwidth along all edges of path P_i, and $minBW(P_i)$ gives the value of the minimum available bandwidth along all edges of path P_i. Since we want to choose the path among all available paths that has the *maximum* value as $minBW(P_i)$, we will choose path P_3 in this example, as $minBW(P_3) > minBW(P_1) > minBW(P_2)$.

Let $bw(i,j)$ denote the last measurement of available bandwidth on edge $(i,j) \in E$, and path P be defined as before. The minimum available bandwidth along edges of a path $P(v_1, v_k) = \{(v_1,v_2),(v_2,v_3),\ldots,(v_{k-1},v_k)\}$ is calculated as:

$$MinBW(P) = min[bw(v_1,v_2), bw(v_2,v_3), \ldots, bw(v_{k-1},v_k)] \tag{5.3}$$

The problem of maximizing the minimum available bandwidths of different paths from source u to destination v may be given as:

$$P' = argmax_{P(u,v)} MinBW(P(u,v)), \tag{5.4}$$

where P' is the path from u to v in the current graph that has the maximum of minimum available bandwidths on its edges.

We use a greedy algorithm which is loosely based on Dijkstra's shortest paths algorithm in order to find the maxmin-bandwidth path in terms of current available bandwidths. The algorithm consists of the procedures $initializeSSBW(source)$, $relaxBW(i,j)$, $pathBW(source)$, and $getPathBW(source, dest)$. The $pathBW(source)$ procedure is the same as

$DijkstraSSP(source)$, with the call to $initializeSS(source)$ replaced with the one to $initializeSSBW(source)$ and the calls to $relax(i,j)$ replaced with ones to $relaxBW(i,j)$. The $getPathBW(source, dest)$ procedure is the same as $getDijkstraSSPRTT(source, dest)$ with the call to $DijkstraSSPRTT(source)$ replaced with the one to $pathBW(source)$ and the line $R.type \leftarrow RTT$ replaced with $R.type \leftarrow BW$. The other procedures may be found in Algorithm 11.

Algorithm 11 Max-Min Available Bandwidth Path Finding Algorithm

```
 1: procedure INITIALIZESSBW(source)
 2:     for all $v \in V$ do
 3:         $d(v) \leftarrow \infty$
 4:         $pred(v) \leftarrow -1$
 5:     end for

 6:     $d(source) \leftarrow 0$
 7: end procedure

 8: procedure RELAXBW(i,j)
 9:     if $i > 0$ and $j > 0$ then                          ▷ check for valid ids
10:         if $d(i) \neq \infty$ and $bw(i,j) \neq \infty$ then
11:             if $d(i) > MAXBW - bw(i,j)$ then
12:                 $usedDist \leftarrow d(i)$
13:             else
14:                 $usedDist \leftarrow MAXBW - bw(i,j)$
15:             end if

16:             if $d(j) > usedDist$ then
17:                 $d(j) \leftarrow usedDist$
18:                 $pred(j) \leftarrow i$
19:             end if
20:         end if
21:     end if
22: end procedure
```

The local path finding algorithm used to find a maxmin-bandwidth path in terms of average available bandwidths is almost the same as the one using current available bandwidths. We just have to replace any occurrence of the term $bw(i,j)$ in $relaxBW(i,j)$ with the term $bwAvg(i,j)$.

5.3.3.1.2.3 Loss-Sensitive Path Finding Algorithms For loss-sensitive traffic, we try to minimize the total loss rate along a path. Let $loss(i,j)$ denote the current loss rate of edge $(i,j) \in E$. Please note that $0 \leq loss(i,j) \leq 1, \forall(i,j) \in E$. The availability of link (i,j) is then

$$Av(i,j) = 1 - loss(i,j), \tag{5.5}$$

where the availability of link (i,j) is used here in synonym with the probability of a packet of fixed length being able to be transmitted over link (i,j) successfully (i.e. without being lost). We calculate the loss rate of a path $P(v_1, v_k) = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)\}$ as

$$Loss(P) = 1 - (Av(v_1, v_2) \times Av(v_2, v_3) \times \ldots \times Av(v_{k-1}, v_k)) = 1 - \prod_{\forall(i,j)\in P} Av(i,j). \tag{5.6}$$

Notice that during the calculation of the loss rate of a path, we assume that individual loss rates of edges along the path are independent of each other. We may formulate our path selection problem from source u to destination v as

$$P' = argmin_{P(u,v)} Loss(P(u,v)), \tag{5.7}$$

where P' is the path from u to v with the least loss rate in the current graph.

We use a modified version of Dijkstra's shortest paths algorithm in order to find the least-cost path from a source to a destination in terms of current loss rates. The algorithm consists of the following procedures: $initializeSS(source)$, $relaxLoss(i,j)$, $pathLoss(source)$, and $getPathLoss(source, dest)$. The procedure $initializeSS(source)$ was provided before in Algorithm 8. The procedure $pathLoss(source)$ is the same as procedure $DijkstraSSP(source)$, with the calls to procedure $relax(i,j)$ replaced with the ones to $relaxLoss(i,j)$. Procedure *get-PathLoss*$(source, dest)$ is the same as $getDijkstraSSPRTT(source, dest)$, with the call to $DijkstraSSPRTT(source)$ replaced with the call to $pathLoss(source)$ and the line *R.type* $\leftarrow$ *RTT* replaced with *R.type* $\leftarrow$ *Loss*. The procedure $relaxLoss(i, j)$ is given in Algorithm 12.

Algorithm 12 Path Finding Algorithm with End-to-End Loss Rate Minimization

```
 1: procedure RELAXLOSS(i,j)
 2:     if $i > 0$ and $j > 0$ then                                  ▷ check for valid ids
 3:         if $d(i) \neq \infty$ and $loss(i, j) \neq \infty$ then
 4:             if $d(j) > 1 - ((1 - d(i)) \times (1 - loss(i, j)))$ then
 5:                 $d(j) \leftarrow 1 - ((1 - d(i)) \times (1 - loss(i, j)))$
 6:                 $pred(j) \leftarrow i$
 7:             end if
 8:         end if
 9:     end if
10: end procedure
```

The local path finding algorithm used to find a least-cost path in terms of average loss rates is almost the same as the algorithm using current loss rates. We just have to replace any occurrence of the term $loss(i, j)$ in $relaxLoss(i, j)$ with the term $lossAvg(i, j)$.

5.3.3.2 Packet Forwarding

We have discussed the route discovery mechanism in detail in the previous chapter. We provide in this section pseudocodes for procedures that handle packet forwarding. The following notation is used in the given pseudocodes:

pkt A packet. Each packet has at least the following fields, which may be accessed using the dot notation: *source*, *dest*, *R*, *type*.

pkt.source The creator of packet pkt.

pkt.dest The destination node of packet pkt.

pkt.R The route structure embedded in the header of packet pkt. We have the following procedure that works on routes: $getNextNode(R.route, id)$, which gives the id of the next node following node with nodeId *id* in the route. For example, if $R.route = \{3, 4, 7, 1, 9\}$, then $getNextNode(R.route, 4)$ returns 7, $getNextNode(R.route, 2)$ returns -1, since node 2 is not in the route, and $getNextNode(R.route, 9)$ returns -1 as node 9 is the last node in the route.

pkt.type The packet type. Note that this may have one of the following values: $\{DataNormal, DataRT, LSMsg, SpecialRTMsg, CostInformMsg, CostMsg, Beacon, RouteFailureMsg, RouteFailureReverseMsg\}$. Please refer to Section 5.2.2 for more information on these packet types.

nodeId The unique identification number of the current node.

nList The neighbor list of current node.

We present three procedures here. Procedure $receivePkt(pkt)$ is called when the current node receives a packet that uses source routing[11]. $forwardPktWithRM(pkt)$ and $forwardPktWithoutRM(pkt)$ are helper procedures that are used in $receivePkt(pkt)$, which try to forward the given packet to its next hop on the route given in its header. Among the two variants, while the former procedure starts the route maintenance mechanism when it is unable to forward the packet to its next hop, the latter simply discards (drops) the packet. Route maintenance is initiated for packets of types $DataNormal$, $DataRT$, $SpecialRTMsg$, and $CostMsg$. Please note that there is no "real" unicasting in wireless networks as the wireless medium is inherently a broadcast medium, so when we say *unicast* in the pseudocode, we mean that the current node actually broadcasts the packet locally, but only the intended next hop node processes the message.

The procedure $processPkt(pkt)$ is a generic procedure, variants of which are specified in their related sections. The real thing with $processPkt(pkt)$ is that the packet *pkt* has reached its destination, and the destination node now needs to

[11]Packets of types $DataNormal$, $DataRT$, $SpecialRTMsg$, $CostMsg$, $RouteFailureMsg$, and $RouteFailureReverseMsg$ use source routing.

process the packet locally (i.e. perform necessary actions). Since this local processing is dependent on the received packet's type, each variant of the procedure *processPkt*(*pkt*) is provided in its related section (e.g. *processLSMsg*(*msg*) previously specified in Algorithm 5) and we omit here the details of the dispatching of the received packet to these *processPkt*(*pkt*) variants based on its packet type. The pseudocode for the procedure *routeMaintenance*(*pkt*) is provided in Section 5.3.4.

Algorithm 13 Packet Forwarding

```
 1: procedure RECEIVEPKT(pkt)
 2:     if $pkt.dest \neq nodeId$ then                ▷ pkt has not reached its dest
 3:         if $pkt.type = DataRT$ or $pkt.type = DataNormal$ or $pkt.type =$
    $SpecialRTMsg$ or $pkt.type = CostMsg$ then
 4:             do $forwardPktWithRM(pkt)$
 5:         else
 6:             do $forwardPktWithoutRM(pkt)$
 7:         end if
 8:     else
 9:         do $processPkt(pkt)$                      ▷ pkt has reached its destination
10:     end if
11: end procedure

12: procedure FORWARDPKTWITHRM(pkt)
13:     $next \leftarrow getNextNode(pkt.R.route, nodeId)$
14:     if $next \notin nList$ then        ▷ next node in route not found in neighbor list
15:         do $routeMaintenance(pkt)$
16:     else
17:         unicast $pkt$ to $next$
18:     end if
19: end procedure

20: procedure FORWARDPKTWITHOUTRM(pkt)
21:     $next \leftarrow getNextNode(pkt.R.route, nodeId)$
22:     if $next \notin nList$ then        ▷ next node in route not found in neighbor list
23:         drop $pkt$
24:     else
25:         unicast $pkt$ to $next$
26:     end if
27: end procedure
```

5.3.3.3 Route Discovery

In this section, pseudocodes for route discovery using local path finding algorithms and the route cache are provided. Route discovery has two different modes, operating in *normal mode* for normal data traffic and in *QoS mode* for real-time traffic requiring QoS constraints. We use here the following notation in addition to the ones presented in Section 5.3.3.2.

pkt.qosType A field of a packet, representing the type of the real-time flow. This field may take one of the values of $\{RTT, BW, Loss\}$, representing delay-sensitive, bandwidth-sensitive, and loss-sensitive traffic respectively.

pkt.qosReq A field of a packet, representing the QoS requirement of the real-time flow. This field represents the value for the maximum tolerable end-to-end delay or loss rate for delay-sensitive or loss-sensitive traffic, respectively, and the value for the minimum tolerable available bandwidth for bandwidth-sensitive traffic.

lastNode(route) Procedure that returns the last node on the route (the destination of the path). For example, if $route = \{5, 6, 4, 2, 9\}$, then $lastNode(route)$ would return 9.

routeCache The route cache of the current node. Please refer to Section 4.3.4.1 for a detailed explanation on the use of the route cache.

rtMsg A special packet that informs the nodes along the initial path from the source of a real-time flow to its destination about the real-time flow. Please see Section 4.3.3.1 for more information on this special packet. This packet has the following fields: *source*, *dest*, *type*, *R*, *duration*, *costCount*, *periodic*, *period*, and $costInformHopCount$.

rtMsg.source The source of the real-time flow.

rtMsg.dest The destination of the real-time flow.

rtMsg.type The type of this message, which is always $SpecialRTMsg$.

rtMsg.R The route that this message should follow (used in source routing).

rtMsg.duration The estimated duration of the real-time flow.

rtMsg.costCount The number of cost messages requested by the source during the real-time flow.

rtMsg.periodic Whether the source has requested periodic cost messages or not.

rtMsg.period If the source has requested periodic cost messages, the period of these cost messages.

rtMsg.costInformHopCount The hop count imposed on $costInform$ messages[12].

We utilize *single paths* in the current implementation of Elessar, meaning that *multipaths* are not used at the moment. We have implemented the infrastructure needed to find and employ such multipaths, but our present emphasis is on the investigation of our protocol under the single paths case. Therefore, data streams and special flow information packets are sent over single paths. Please note that when we say single paths in this discussion, we mean the use of a single path for a (specific) packet at any given time point, so we use single paths in the *spatial dimension*. However, due to the dynamic nature of the underlying network and our protocol's ability to adapt to such changes and employ different paths as the topology alters, the path(s) that packets belonging to the same data stream are sent over may be different, so it may be said that we are actually using multipaths in the *temporal dimension*.

Route discovery has the following procedures: *receivePktFromUpperLayer*(*pkt*), which is called when the routing layer receives a packet to be sent into the network from the upper layer (i.e. the application or transport layer); $findNormalRoute$(*pkt*), which finds a normal route (path) from the current node to the destination of the given packet; $findRTRoute$(*pkt*), which finds a QoS route from the current node to the destination of the given

[12]See Section 5.2.2 for information on the $costInform$ message type.

packet (the found route supports the QoS type present in parameter *pkt*); and *inCache*(*pkt*), which finds and returns the route to destination *pkt.dest* with type *pkt.qosType* in the route cache. Procedure $sendSpecialRTMsg(pkt)$ creates and sends a special packet in order to inform the nodes on the normal path from the current node to the destination of the packet[13]. Procedure $createSpecialRTMsg(pkt)$ is a helper procedure which creates and returns a special packet according to the information provided in parameter *pkt*. Procedure $assignSpecialRTFields(rtMsg, pkt)$ is a stub procedure, which fills in the remaining fields of packet $rtMsg$ that have not been yet assigned. These fields are specific to each real-time flow and determined by the application creating the real-time data packets. Procedures $forwardPktWithRM(pkt)$, *startCost-*$Information(pkt)$, and $findInRTFlows(pkt)$ are presented in algorithms 13, 20, and 19, respectively.

We would like to note that we do not check the lengths of real-time flows in our implementation to see if the directed cost dissemination mechanism should be enabled for them, as stated in Section 4.3.3.1, as all our real-time flows are long enough to grant operation in QoS mode. We investigate the problem of how long a stream must be in order to start the cost dissemination mechanism in Chapter 6.

5.3.4 Route Maintenance

In this section, pseudocodes for the route maintenance mechanism of Elessar are presented. It has been discussed in Section 4.3.5 that the route maintenance mechanism has two different operation modes depending on the traffic type for which route maintenance is done. Furthermore, different schemes for route maintenance are available based on whether route failure messages are employed or not. In our current implementation, we do not make use of route failure messages for two reasons. The first reason is that with the current simulations, we did not

[13]One thing to note here is that although we mentioned in Section 4.3.3.1 that such a special packet must be sent reliably, in our current implementation, we are sending this special packet unreliably.

Algorithm 14 Route Discovery, Part 1

```
 1: if route cache timer expired then
 2:     $routeCache \leftarrow \emptyset$
 3:     restart route cache timer
 4: end if

 5: procedure receivePktFromUpperLayer(pkt)
 6:     if $pkt.type = DataNormal$ then
 7:         $success \leftarrow findNormalRoute(pkt)$
 8:         if $success$ then
 9:             do $forwardPktWithRM(pkt)$
10:         else
11:             drop $pkt$
12:         end if
13:     else if $pkt.type = DataRT$ then
14:         $success \leftarrow findRTRoute(pkt)$
15:         if $\neg success$ then
16:             $success \leftarrow findNormalRoute(pkt)$
17:         end if

18:         if $success$ then
19:             if $\neg findInRTFlows(pkt)$ then
20:                 do $sendSpecialRTMsg(pkt)$
21:             end if

22:             do $startCostInformation(pkt)$
23:             do $forwardPktWithRM(pkt)$
24:         else
25:             drop $pkt$
26:         end if
27:     end if
28: end procedure

29: procedure findNormalRoute(pkt)
30:     $R \leftarrow inCache(pkt)$                       ▷ try to find route in cache

31:     if $R.route = \emptyset$ then            ▷ run the local path finding algorithm
32:         $R.route \leftarrow getDijkstraSSP(nodeId, pkt.dest)$
33:         $R.type \leftarrow Normal$
34:         $R.cost \leftarrow -1$                ▷ value not used in normal routes
35:     end if
```

Algorithm 15 Route Discovery, Part 2

```
36:     if $R.route = \emptyset$ then                    ▷ no route from current node to dest
37:         $pkt.R \leftarrow R$
38:         return false
39:     else
40:         if $R \notin routeCache$ then                  ▷ add route to route cache
41:             $routeCache \leftarrow routeCache \cup \{R\}$
42:         end if

43:         $pkt.R \leftarrow R$
44:         return true
45:     end if
46: end procedure

47: procedure findRTRoute(pkt)
48:     $R \leftarrow inCache(pkt)$                        ▷ try to find route in cache

49:     if $R.route = \emptyset$ then                     ▷ run local path finding algorithm
50:         if $pkt.qosType = RTT$ then
51:             $R \leftarrow getDijkstraSSPRTT(nodeId, pkt.dest)$
52:         else if $pkt.qosType = BW$ then
53:             $R \leftarrow getPathBW(nodeId, pkt.dest)$
54:         else if $pkt.qosType = Loss$ then
55:             $R \leftarrow getPathLoss(nodeId, pkt.dest)$
56:         end if
57:     end if
                                           ▷ check if found route satisfies QoS req.
58:     if $pkt.qosType = RTT$ or $pkt.qosType = Loss$ then
59:         if $R.route \neq \emptyset$ and $pkt.qosReq < R.cost$ then
60:             $R.route \leftarrow \emptyset$
61:         end if
62:     else if $pkt.qosType = BW$ then
63:         if $R.route \neq \emptyset$ and $pkt.qosReq > R.cost$ then
64:             $R.route \leftarrow \emptyset$
65:         end if
66:     end if
```

Algorithm 16 Route Discovery, Part 3

```
67:     if $R.route = \emptyset$ then    ▷ no (satisfactory) route from current node to dest
68:         $pkt.R \leftarrow R$
69:         return false
70:     else
71:         if $R \notin routeCache$ then    ▷ add route to route cache
72:             $routeCache \leftarrow routeCache \cup \{R\}$
73:         end if

74:         $pkt.R \leftarrow R$
75:         return true
76:     end if
77: end procedure

78: procedure inCache(pkt)
79:     for all $r \in routeCache$ do
80:         if $pkt.type = DataNormal$ and $lastNode(r.route) = pkt.dest$ and
            $r.type = Normal$ then
81:             return $r$
82:         else if $pkt.type = DataRT$ and $lastNode(r.route) = pkt.dest$ and
            $r.type = pkt.qosType$ then
83:             return $r$
84:         end if
85:     end for

86:     return $\emptyset$
87: end procedure
```

Algorithm 17 Procedure sendSpecialRTMsg(pkt)

```
1: procedure sendSpecialRTMsg(pkt)
2:     $tempR \leftarrow pkt.R$        ▷ needed since $findNormalRoute(pkt)$ may modify $pkt.R$
3:     $success \leftarrow findNormalRoute(pkt)$

4:     if $success$ then
5:         $rtMsg \leftarrow createSpecialRTMsg(pkt)$
6:         do $forwardPktWithRM(rtMsg)$
7:     end if

8:     $pkt.R \leftarrow tempR$        ▷ restore original route on $pkt$
9: end procedure

10: procedure createSpecialRTMsg(pkt)
11:     $rtMsg.source \leftarrow pkt.source$
12:     $rtMsg.dest \leftarrow pkt.dest$
13:     $rtMsg.type \leftarrow SpecialRTMsg$
14:     $rtMsg.R \leftarrow pkt.R$
15:     do $assignSpecialRTFields(rtMsg, pkt)$
16:     return $rtMsg$
17: end procedure
```

observe many benefits of route failure messages. The idea behind the use of route failure messages is early information of the source node of the route breakage and decreasing the number of dropped packets at intermediate nodes. Since our implementation does not have an explicit neighborhood beaconing mechanism, each node is informed of changes in its neighborhood instantaneously, and therefore the topology dissemination mechanism is able to inform the source node of the route breakage in a timely manner. We have also observed that the overhead incurred by route failure messages do not justify the decrease in the number of dropped packets. We believe this to be a result of the timely informing of the source node by the topology dissemination mechanism, as then the source node finds a new path to the destination and the number of packets dropped at intermediate nodes is negligible.

The second reason why we do not employ route failure messages is its complexity. With the inclusion of route failure messages, route discovery and packet handling procedures needed significant changes and we currently think that the benefits of route failure messages do not justify the complexity incurred, at least in our current simulation setup. Therefore, in this section, we provide pseudocodes for the route maintenance mechanism which does not make use of route failure messages.

The notation used in the following pseudocodes are provided below:

pkt A packet. It has at least the following field: R, representing the route embedded in the packet header.

R A compound structure representing a route. It has the following fields: *cost*, *type*, and *route*.

R.route A field of R, representing the route as a list of node ids, as defined previously.

Route maintenance consists of the following procedures: $routeMaintenance(pkt)$, which is the main procedure, and $checkRouteForLoops(route)$, which is a helper procedure that checks whether the given route includes loops. Procedures

$findNormalRoute(pkt)$ and $forwardPktWithRM(pkt)$ were presented in algorithms 13 and 14, respectively.

Algorithm 18 Route Maintenance, No Route Failure Messages

```
1: procedure routeMaintenance(pkt)
2:     $success \leftarrow findNormalRoute(pkt)$

3:     if $\neg success$ then
4:         drop $pkt$
5:     else
6:         $hasLoop \leftarrow checkRouteForLoops(pkt.R.route)$
7:         if $hasLoop$ then
8:             drop $pkt$
9:         else
10:            do $forwardPktWithRM(pkt)$
11:        end if
12:    end if
13: end procedure

14: procedure checkRouteForLoops(route)
15:    for all $i \in route$ do
16:        $count \leftarrow 0$

17:        for all $j \in route$ do
18:            if $i = j$ then
19:                $count \leftarrow count + 1$
20:            end if

21:            if $count \geq 2$ then
22:                return true
23:            end if
24:        end for
25:    end for

26:    return false
27: end procedure
```

5.3.5 Directed Cost Dissemination

Directed cost dissemination is the mechanism that enables Elessar to support real-time traffic. Through the reception of link costs, Elessar is able to find least-cost paths, providing soft QoS guarantees without any resource reservation. Directed cost dissemination is only part of the QoS operation mode, and activated only when there is at least one real-time flow in the network. Each activation of the directed cost mechanism is flow-oriented and costs are sent to the source *directly*, making use of local path finding algorithms. This section presents pseudocodes for the directed cost dissemination mechanism of Elessar.

We have presented various schemes for cost dissemination in Section 4.3.3.3. Among the schemes presented there, Elessar currently supports options 1, 2, and 3, namely schemes labeled *exactly once*, *periodic, with bound*, and *periodic, without bound*. The implementation of the threshold-based cost dissemination scheme is left as future work. Note that the *exactly once* scheme corresponds to the *periodic, with bound* scheme, with bound set to 1.

We would like to note that each data generator knows the duration of the generated flow, and due to this knowledge, the information of nodes sending their costs of the end of the flow can be accomplished based on flow durations. It is also important to note here that link costs received by the source are recorded on the local graph at the source node. Edge costs that have gone stale are cleaned from the graph.

As mentioned in Section 4.3.3.1, we need to keep a "real-time flows table" at each node participating in the cost dissemination mechanism in order to prevent a node to take part in the limited flooding mechanism more than once. Please note that we keep this table in FIFO[14] manner. Since the table has a limited capacity, when we say $RTFlowsTable \leftarrow RTFlowsTable \cup \{(s, d, t)\}$, if the table can accommodate the new item, it is added to the table in FIFO manner, without deleting any other entry from the table. If there is no space left in the table, then the oldest entry in the table (the entry at the front of the queue) is deleted and

[14]FIFO: first-in, first-out data structure, i.e. a queue

the new item is added to the end of the queue. Particulars on this table, along with other notation used in the following algorithms are provided below:

RTFlowsTable This is the real-time flows table that is used to keep information regarding the last recently seen real-time flows by the current node. Each entry in this table has the following form: ($< source >, < dest >, < type >$), where *source*, *dest*, and *type* are the source, destination, and type of the flow, respectively. *type* can be one of $\{RTT, BW, Loss\}$.

msg A cost information message, informing nodes that they should send their costs to the source node given in the message. Please see Section 5.2.2 for information on the $CostInformMsg$ message type. A cost information message has the following fields:

msg.sendTo The node that costs should be sent to.

msg.qosType The type of link costs that should be sent. This can be one of *RTT, BW, Loss.*

msg.type The type of this message, which is always $CostInformMsg$.

msg.duration The estimated duration of the real-time flow.

msg.costCount The number of cost messages requested by the source during the real-time flow.

msg.periodic Whether the source has requested periodic cost messages or not.

msg.period If the source has requested periodic cost messages, the period of these cost messages.

msg.hopCount The hop count imposed on $CostInformMsg$ messages.

pkt A packet. A packet has at least the following fields: *source*, *dest*, *type*, *qosType*, *R*, *duration*, *costCount*, *periodic*, *period.*

pkt.source The source of the packet.

pkt.dest The destination of the packet.

pkt.type The type of the packet.

pkt.R The route that this packet should follow (used in source routing). Note that R is defined in Section 5.3.3.1.2.

pkt.duration The estimated duration of the real-time flow.

pkt.costCount The number of cost messages requested by the source during the real-time flow.

pkt.periodic Whether the source has requested periodic cost messages or not.

pkt.period The period of the requested cost messages.

nodeId The unique ID of the current node. Each node in the network is given a numeric ID that uniquely represents the node. Note that $nodeId > 0, \forall v \in V$.

Edge Costs on Graph We represent edge costs on the local graph as $rtt(i,j)$, $rttAvg(i,j)$, $bw(i,j)$, $bwAvg(i,j)$, $loss(i,j)$, $lossAvg(i,j)$, $\forall(i,j) \in E$. Please refer to Section 5.3.3.1 for more information on edge costs.

rtMsg A message of type $SpecialRTMsg$. Please see Section 5.3.3.3 for more information on the special real-time message.

costTimer A cost timer. This is a timer that expires every $costTimer.period$ seconds. It is used in order to inform the current node that it should now send its link costs to the node given in the timer message.

costTimer.sendTo The id of the node that link costs must be sent to.

costTimer.qosType The type of link costs that should be sent. This can be one of *RTT, BW, Loss.*

costTimer.costCount The remaining number of cost messages to be sent.

costTimer.period The period of the timer. The timer expires every $costTimer.period$ seconds.

costMsg A cost message, including the link costs of the current node.

costMsg.source The source of the cost message.

costMsg.dest The destination of the cost message.

costMsg.type The type of the cost message, which is always $CostMsg$.

costMsg.R The route that this cost message should follow.

costMsg.qosType The type of link costs included in the cost message. This can be one of *RTT, BW, Loss*.

costMsg.costList This is a list containing the link costs sent by the originator of the cost message. Each entry in this *costList* has the form (<*from*>, <*to*>, <*cost*>, <*costAvg*>), where <*from*> is the starting node (head) of the edge, <*to*> is the ending node (tail) of the edge, <*cost*> is the current link cost from node <*from*> to node <*to*>, and <*costAvg*> is the average link cost of the same edge.

RTTCosts A list that keeps the delays of links from the current neighbor to its neighbors. Each entry is in the format for $costMsg.costList$.

BWCosts A list that keeps the available bandwidths of links from the current neighbor to its neighbors. Each entry is in the format for $costMsg.costList$.

LossCosts A list that keeps the loss rates of links from the current neighbor to its neighbors. Each entry is in the format for $costMsg.costList$.

We have the following procedures for the directed cost dissemination mechanism. Procedure $findInRTFlows(pkt)$ returns true if the flow that the given packet belongs to is found in $RTFlowsTable$; it returns false otherwise. Procedure $startCostInformation(pkt)$ initiates the cost information mechanism for the flow the given packet belongs to. Procedure $startCostTransmission(pkt)$ starts the cost transmission mechanism for the flow the given packet belongs to. Please refer to Section 4.3.3.1 for a detailed discussion on the information of nodes during directed cost dissemination. Procedure $assignCostInformFields(msg, pkt)$ is a stub procedure, which fills in the remaining fields of message *msg* that have not been yet assigned. These fields are specific to each real-time flow and determined by the application creating the real-time data packets. $createCostInformMsg(pkt)$ creates and returns a message of type $CostInformMsg$ for the flow that the given packet belongs to. Procedure $scheduleCostTimerAndSendCost(timer)$ restarts the given cost timer and creates and sends the link costs of the current node to the node specified in the given cost timer. $createCostMsg(timer)$ creates and returns a cost message including the link costs of the current node with type given in the cost timer. Procedure $costTimerExpired$ is called when a cost timer expires. Procedures $processCostInformMsg(msg)$, $processCostMsg(costMsg)$, and $processSpecialRTMsg(rtMsg)$ are called when the node receives a packet of type $CostInformMsg$, $CostMsg$, and $SpecialRTMsg$, respectively. Procedures $findNormalRoute(pkt)$ and $forwardPktWithRM(pkt)$ are provided in algorithms 14 and 13, respectively.

Algorithm 19 Procedure findInRTFlows(pkt)

```
1: procedure FINDINRTFLOWS(pkt)
2:     $s \leftarrow pkt.source$
3:     $d \leftarrow pkt.dest$
4:     $t \leftarrow pkt.qosType$

5:     if $(s, d, t) \in RTFlowsTable$ then
6:         return true
7:     else
8:         return false
9:     end if
10: end procedure
```

5.3.6 Link Cost Measurement

We have assumed in Chapter 4 that there is an underlying mechanism that provides link costs to our protocol periodically. In the implementation of our simulations, we had to implement this mechanism ourselves as the simulator was not able to give us such information automatically. This section provides information on the implementation of the link cost measurement mechanism.

We have used a simple beaconing mechanism in order to measure link costs

Algorithm 20 Information of Nodes - CostInform Messages

```
1: procedure startCostInformation(pkt)
2:     $found \leftarrow findInRTFlows(pkt)$

3:     if $\neg found$ then
4:         $msg \leftarrow createCostInformMsg(pkt)$
5:         broadcast $msg$
6:         do $startCostTransmission(pkt)$
7:     end if
8: end procedure

9: procedure createCostInformMsg(pkt)
10:    $msg.sendTo \leftarrow pkt.source$
11:    $msg.qosType \leftarrow pkt.qosType$
12:    $msg.type \leftarrow CostInformMsg$
13:    do $assignCostInformFields(msg, pkt)$
14:    return $msg$
15: end procedure
```

between nodes. A node periodically broadcasts beacon messages locally, and all nodes receiving such a beacon message reply with a beacon reply message. The beacon creator measures the round trip time (RTT) of the beacon and beacon reply messages, and assuming symmetrical links, divides this time by 2 to get the one-way link delay. Loss rates are measured by the use of sequence numbers on beacons and nodes that do not receive a reply to their beacons in a predetermined amount of time consider either the beacon or the beacon reply lost, adding to the loss rate of the link. Beacon replies that are received out-of-order do not contribute to the loss rate. Available bandwidth measurement is as follows. Representing the current available bandwidth of a link (i, j) as $B_{(i,j)}$, the length of a packet p as $len(p)$ and the one-way delay as $delay(p)$, we calculate the current available bandwidth at link (i, j) as

$$B_{(i,j)} = \frac{len(p)}{delay(p)}. \tag{5.8}$$

Of course, this provides a rough estimate of the available bandwidth at the link but it is currently sufficient for our purposes.

Algorithm 21 Directed Cost Transmission

```
1: procedure startCostTransmission(pkt)
2:     $found \leftarrow findInRTFlows(pkt)$

3:     if $\neg found$ then
4:         $tempEl \leftarrow \{(pkt.source, pkt.dest, pkt.qosType)\}$
5:         $RTFlowsTable \leftarrow RTFlowsTable \cup tempEl$

6:         if $nodeId \neq pkt.source$ then    ▷ beware of source node starting cost transmission to self!
7:             if $pkt.periodic$ then    ▷ scheme: periodic, without bound
8:                 $costTimer.sendTo \leftarrow pkt.source$
9:                 $costTimer.qosType \leftarrow pkt.qosType$
10:                $costTimer.costCount \leftarrow (pkt.duration/pkt.period) + 1$
11:                $costTimer.period \leftarrow pkt.period$
12:                do $scheduleCostTimerAndSendCost(costTimer)$
13:            else    ▷ scheme: periodic, with bound
14:                $costTimer.sendTo \leftarrow pkt.source$
15:                $costTimer.qosType \leftarrow pkt.qosType$
16:                $costTimer.costCount \leftarrow pkt.costCount$
17:                $costTimer.period \leftarrow (pkt.duration/pkt.costCount)$
18:                do $scheduleCostTimerAndSendCost(costTimer)$
19:            end if
20:        end if
21:    end if
22: end procedure

23: procedure scheduleCostTimerAndSendCost(timer)
24:    if $timer.sendTo \neq nodeId$ and $timer.costCount > 0$ then
25:        $costMsg \leftarrow createCostMsg(timer)$
26:        $success \leftarrow findNormalRoute(costMsg)$

27:        if $success$ then
28:            do $forwardPktWithRM(costMsg)$
29:        end if

30:        $timer.costCount \leftarrow timer.costCount - 1$
31:        restart $timer$
32:    else
33:        delete $timer$
34:    end if
35: end procedure
```

Algorithm 22 Cost Message Processing

```
1: procedure createCostMsg(timer)
2:     $costMsg.source \leftarrow nodeId$
3:     $costMsg.dest \leftarrow timer.sendTo$
4:     $costMsg.type \leftarrow CostMsg$
5:     $R.type \leftarrow normal$
6:     $R.cost \leftarrow -1$                          ▷ cost not used for normal route
7:     $R.route \leftarrow \emptyset$
8:     $costMsg.R \leftarrow R$
9:     $costMsg.qosType \leftarrow timer.qosType$

10:    if $costMsg.qosType = RTT$ then
11:        for all $c \in RTTCosts$ do
12:            $costMsg.costList \leftarrow costMsg.costList \cup \{c\}$
13:        end for
14:    else if $costMsg.qosType = BW$ then
15:        for all $c \in BWCosts$ do
16:            $costMsg.costList \leftarrow costMsg.costList \cup \{c\}$
17:        end for
18:    else if $costMsg.qosType = Loss$ then
19:        for all $c \in LossCosts$ do
20:            $costMsg.costList \leftarrow costMsg.costList \cup \{c\}$
21:        end for
22:    end if
23:    return $costMsg$
24: end procedure

25: procedure processCostMsg(costMsg)
26:    if $costMsg.dest = nodeId$ then                ▷ pkt reached its destination
27:        for all $c \in costMsg.costList$ do
28:            if $costMsg.qosType = RTT$ then
29:                $rtt(c.from, c.to) \leftarrow c.cost$
30:                $rttAvg(c.from, c.to) \leftarrow c.costAvg$
31:            else if $costMsg.qosType = BW$ then
32:                $bw(c.from, c.to) \leftarrow c.cost$
33:                $bwAvg(c.from, c.to) \leftarrow c.costAvg$
34:            else if $costMsg.qosType = Loss$ then
35:                $loss(c.from, c.to) \leftarrow c.cost$
36:                $lossAvg(c.from, c.to) \leftarrow c.costAvg$
37:            end if
38:        end for
39:        delete $costMsg$
40:    else
41:        do $forwardPktWithRM(costMsg)$
42:    end if
43: end procedure
```

Algorithm 23 Cost Timer, CostInformMsg, and SpecialRTMsg Processing

```
1: procedure processSpecialRTMsg(rtMsg)
2:     do $startCostInformation(rtMsg)$

3:     if $rtMsg.dest = nodeId$ then
4:         delete $rtMsg$                          ▷ pkt reached its destination
5:     else
6:         do $forwardPktWithRM(rtMsg)$
7:     end if
8: end procedure

9: procedure costTimerExpired(timer)
10:     do $scheduleCostTimerAndSendCost(timer)$
11: end procedure

12: procedure processCostInformMsg(msg)
13:     $found \leftarrow findInRTFlows(msg)$

14:     if $found$ then
15:         delete $msg$
16:     else
17:         $msg.hopCount \leftarrow msg.hopCount - 1$
18:         do $startCostTransmission(msg)$

19:         if $msg.hopCount > 0$ then
20:             broadcast $msg$
21:         else
22:             delete $msg$
23:         end if
24:     end if
25: end procedure
```

Chapter 6

Experimental Results

We have done extensive simulation experiments with our protocol on the discrete event simulation system OMNeT++. We have also made use of the INET framework written to support realistic network simulations on OMNeT++. We have especially made use of the mobility support of INET, using the random waypoint mobility model, and of the IEEE 802.11b MAC protocol implementation, along with compatible radio technologies. For more information on OMNeT++ and the INET framework, please refer to Section 5.1.

Experiments on Elessar were conducted in order to analyse the design choices we have made in the development of the protocol, to gain more information on suitable parameter choices, to measure the performance of our protocol under various conditions, and of course, to verify the correctness of our implementation. We have divided our experimental analysis into two sections in this chapter, with the first section focusing on the operation of the protocol in normal mode and the second section focusing more on its QoS mode of operation.

We especially look into the topology dissemination, route discovery and route maintenance mechanisms of Elessar in Section 6.1, whereas we analyse the cost dissemination mechanism, along with route discovery and maintenance in QoS mode in Section 6.2. Before presenting our results, however, we would like to provide information on some simulation parameters in Table 6.1. Table 6.1 presents simulation parameters related with the IEEE 802.11b physical radio and MAC layer, and the wireless channel characteristics of the simulation.

Parameter Name	Value
Signal Attenuation Threshold	-70 dBm
Path Loss Coefficient	2.5
Thermal Noise	-110 dBm
Carrier Freqency	2.4 GHz
Number of Channels	1
WLAN Frame Capacity	10
MAC Address Appointment	“Auto”
MAC Queue Size	14
MAC Bitrate	11 Mbps
MAC RTS Threshold Bytes	500
MAC Retry Limit	7
MAC Min. Con. Win. Size for Unicast	7
MAC Min. Cont. Win. Size for Broadcast	31
Radio Transmit Power	20 mW
Radio SNR Threshold	-40 dBm
Radio Sensitivity	-85 dBm

Table 6.1: Simulation parameters for the wireless channel and physical radio.

With the given physical radio properties and the wireless channel characteristics, the transmission range of a node is approximately 50 m.

6.1 Experimental Results for Normal Mode

In this section, we present simulation results mainly focused on the operation of Elessar in *normal mode*. We investigate the topology dissemination mechanism along with the route discovery and maintenance mechanisms in normal mode. Since we are dealing with normal operation mode in this section, generated traffic are *non-real-time* data.

Our experiments were conducted as follows. We create an ad hoc network consisting of a number of nodes deployed on a playground of variable size. We provide the exact numbers of such simulation parameters in each experiment's related parameters table. We start the experiment and the simulation first creates and sets up necessary objects, modules and data structures. After all objects, data structures and modules are initialized, mobile nodes in the network communicate with the *MACTable* module directly in order to learn of their assigned MAC addresses and with the *ChannelControl* module to learn of their starting locations in the palyground. After this bootup phase, the simulation starts in the sense that nodes start to move and our protocol starts to operate. During the initial 60 seconds of the simulation, only the topology dissemination mechanism is active and no data flows are created by any node. This is done to allow the network to settle and to prevent any initial transient conditions from affecting our analysis during data transmission.

After the initial 60 seconds, the *FlowController* module starts to operate and chooses random nodes in the graph as data sources and destinations, and directs the *TrafficGen* modules of source nodes to create normal data flows to the given destinations. Each flow in the network is created randomly, meaning that the data rates and flow durations of each flow is potentially different and we take extra care not to have more than one flow from a specific source node to a specific destination node at the same time; e.g. there is at most one flow from node u to v in the network at any given time. Please note that this is **NOT** to say that there is at most one flow in the entire network at any given time or that nodes u and v may not carry or forward data of other flows. To illustrate, when we have a flow from node u to v, we may have cases where we have a flow from v to u and/or another flow from u to some other node w and/or another flow from v to some other node x and/or a flow from some node y to some node z passing over nodes u and/or v etc.

The number of flows in the entire network may be variable and the flow number for each experiment in this section, along with information on data rates and durations of flows are provided in Table 6.2.

We fix the number of flows to 15 for each experiment in this section. Each data packet has a payload length chosen uniformly in the integer range [40,600] bytes. Note that the actual length of the packet is higher than the payload due to protocol headers. The number of packets and the data rate of each flow is

Parameter Name	Value
Number of Flows	15
Packet Payload Length	[40,600] bytes
Packet Creation Period	[5,50] ms
Packet Creation Rate	[20,200] packets/sec
Flow Length	[400,4000] packets
Flow Duration	[2,200] sec

Table 6.2: Simulation parameters for data traffic in normal operation mode experiments.

set once at the creation of the flow and does not change during the lifetime of the flow. The number of packets in a flow is chosen uniformly over the integer range [400,4000]. Each of our data flows has a constant packet creation rate, where the packet creation period is chosen uniformly over the real range [5,50] ms. Depending on the packet creation period, the packet creation rate of a flow changes between 20 and 200 packets/sec. According to the packet creation rate and the number of packets in the flow, the overall duration of a flow changes between 2 and 200 seconds.

We use the *random waypoint mobility model* to simulate the mobility of nodes. Parameters on node speeds and waiting times between location changes are provided in each experiment's related parameters table.

For each experiment in this section, we have used the table sizes and timer periods[1] provided in Table 6.3.

Parameter Name	Value
Route Cache Timer Period	60 sec
LSTable Size	15

Table 6.3: Miscellaneous simulation parameters for normal operation mode experiments.

We run each experiment for a total duration of 11 simulated minutes, where we have data flows in the network for the last 10 minutes and allow the network to settle in the first minute.

[1]The route cache at a node is cleared periodically according to the route cache timer. This is actually done for QoS purposes, but we include the information on the timer here due to its relevance.

We examine experimental results regarding the protocol overhead in normal operation mode in Section 6.1.1 whereas we look into the performance of our protocol in terms of routing success in Section 6.1.2.

6.1.1 Examination of the Protocol Overhead

In this section, we provide results on the overhead induced by Elessar. We specifically examine the overhead of the link-state topology dissemination mechanism and the overhead caused by source routing. Both of these types of overhead are also incurred in exactly the same format and amount by Elessar in QoS operation mode, therefore we investigate these types of overhead only in this section. We will investigate other types of protocol overhead specific to QoS operation mode in Section 6.2.

The link-state overhead of Elessar is independent of the amount of data traffic in the network, so the more there is data traffic, the less is the ratio of link-state overhead to data traffic. In light of this observation, we can say that our protocol can support high traffic intensities as its primary source of overhead is independent of the volume of data carried in the network.

The topology overhead of our protocol is highly dependent on the mobility rates of nodes in the network since we employ an event-driven link-state mechanism, where changes in node neighborhoods are the events of interest. As the mobility rates of nodes increase, we incur higher overhead due to the increase in the number of events in the system. The link-state overhead is also dependent on the size of the network due to its global broadcast property. When we say “size of the network”, we mean a couple of metrics, including network diameter, node count, average node degree and node density.

The diameter of a network is defined as the length in terms of hop count of the longest path among all shortest paths between all node pairs in the network. The diameter of the network in Figure 6.1 is 4, which may be dtermined by the path $\{(5,6),(6,7),(7,8),(8,9)\}$ between nodes 5 and 9, which is of length 4. Note that even though we have a longer path $\{(5,6),(6,2),(2,1),(1,4),(4,8),(8,9)\}$ of length 6 between these two nodes, we do not consider this path in the determination of the network diameter as it is not one of the shortest paths between these two nodes. The node count of a network is simply the number of nodes in the network, so the node count of the network in Figure 6.1 is 13.

The degree of a node i is defined as the number of its neighbors in an undirected graph or as the sum of its in-degree and out-degree in a directed graph, where the in-degree of a node i is the number of incoming edges to i and the out-degree of i is the number of outgoing edges from i. For example, the degree of node 7 in Figure 6.1 is 4. The average node degree in the network is the average of all node degrees in the network. For the mathematically inclined, this may be denoted as

$$AD_G = \frac{\sum_{u \in V} degree(u)}{|V|}, \tag{6.1}$$

where a graph G is as defined before in terms of the set of its nodes V and the set of its edges E as $G(V,E)$, AD_G is the average node degree of G, and $degree(u)$ is the degree of node $u \in V$.

We define the node density of a network as the number of nodes that fall within an area of fixed size. Denoting the node count of a network N as C_N and the deployment area of the network as A_N, node density is given by $ND_N = \frac{C_N}{A_N}$. If the network given in Figure 6.1 spans an area of 260 m^2, then the node density of the network is $13/260 = 0.05$ $nodes/m^2$.

Changes in the values of these various definitions of "network size" may affect the link-state overhead of our protocol in different ways. We examine the effects of such metrics below.

Figures 6.2 and 6.3 presents the total link-state (LS) overhead in terms of bytes vs. $wholeK$, while mobility rate changes. By total LS overhead, we mean all of the LS messages created and forwarded by all nodes in the network. $wholeK$ is the parameter used in the incremental LS messaging scheme, introduced in

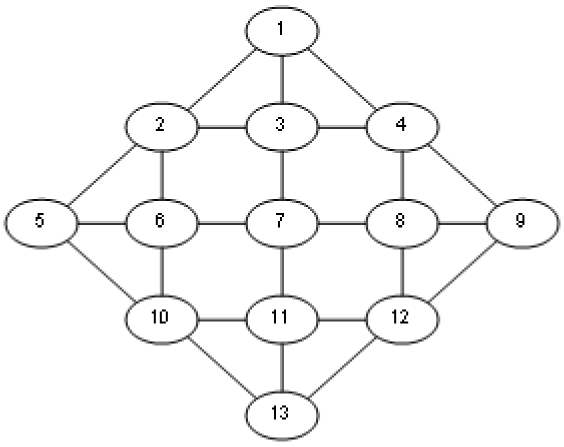

Figure 6.1: An example network.

Section 4.3.2.1. In this scheme, every *wholeK*-th LS message is a whole message, where the other LS messages are incremental messages. Table 6.4 gives the corresponding values for each mobility rate[2].

Mobility Rate	Node Speed	Waiting Time
Mobility Rate 1	[1,2] m/sec	[5,10] sec
Mobility Rate 2	[2,4] m/sec	[5,10] sec
Mobility Rate 3	[5,10] m/sec	[0,5] sec
Mobility Rate 4	[10,20] m/sec	[0,0] sec

Table 6.4: Mobility rates.

We see from these figures that as *wholeK* increases, the total LS overhead in bytes decreases as predicted, since each incremental LS message is generally smaller in size than a whole LS message. We especially see a great decrease in LS overhead between the values of 1 and 2 of *wholeK*. $wholeK = 1$ corresponds to the non-incremental LS scheme, whereas with $wholeK > 1$, we use the incremental LS scheme. We do not observe any significant improvements in the LS overhead above values of 4 and 8 for *wholeK*.

[2]According to the random waypoint mobility model, each time a node changes state from stationary to mobile, it selects a speed uniformly from the given real number range. Please note that the given ranges correspond to the following ranges in km/h (kph): $[1,2]$ m/s = $[3.6,7.2]$ kph; $[2,4]$ m/s = $[7.2,14.4]$ kph; $[5,10]$ m/s = $[18,36]$ kph; $[10,20]$ m/s = $[36,72]$ kph. Each time a node finishes moving along its linear segment, it uniformly chooses a waiting time during which it is stationary from the given real number range.

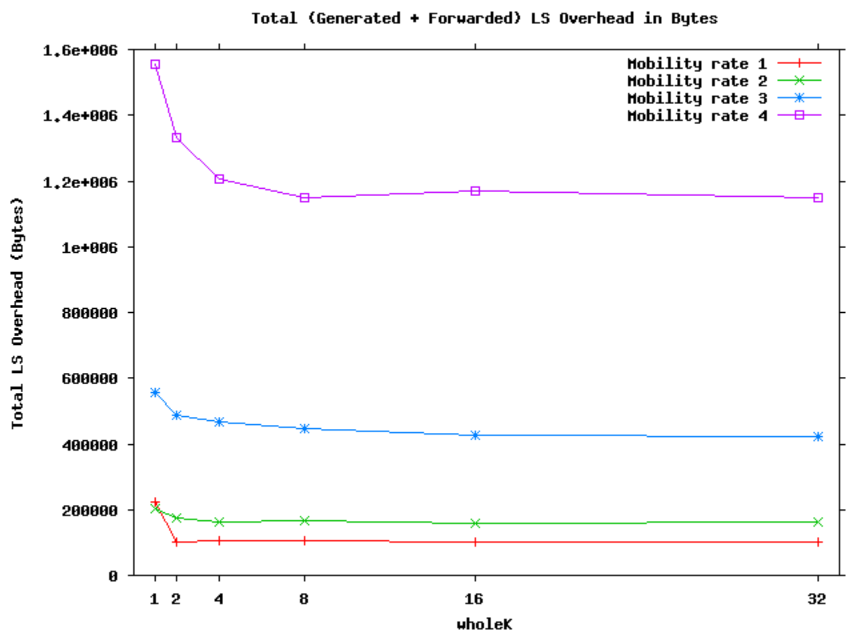

Figure 6.2: LS overhead in bytes vs. wholeK, for different mobility rates.

We can also realize by looking at these figures that as the mobility rate increases, the overall LS overhead increases as predicted. As mentioned before, this is due to our event-based LS mechanism.

Figure 6.4 presents total LS overhead in terms of packet count vs. *wholeK*, as mobility changes. We can see from this figure that the choice *wholeK* has no effect on the LS overhead in terms of packet count, as the incremental LS scheme only decreases the size of LS packets, but not the number of packets created. As before, the LS overhead increases significantly as the mobility rate increases.

Figure 6.5 presents the ratio of the total LS overhead in bytes to the total successfully received data packets in bytes vs. *wholeK*, while mobility changes. From this graph we observe that for the lowest mobility rate of 1, we have an LS overhead of approximately 0.3%; for mobility rate 2, 0.5% overhead; for mobility rate 3, 1.5% overhead, and for the highest mobility rate of 4, 3.8% overhead. We are generally expecting mobile nodes in conventional ad hoc networks to have mobility rates lower than or similar to rates 1 and 2, so our protocol incurs a very low overhead for such real-life mobility rates[3]. Even with high mobility rates, the

[3]Considering that the average walking, jogging, running, and cycling speeds of a human are

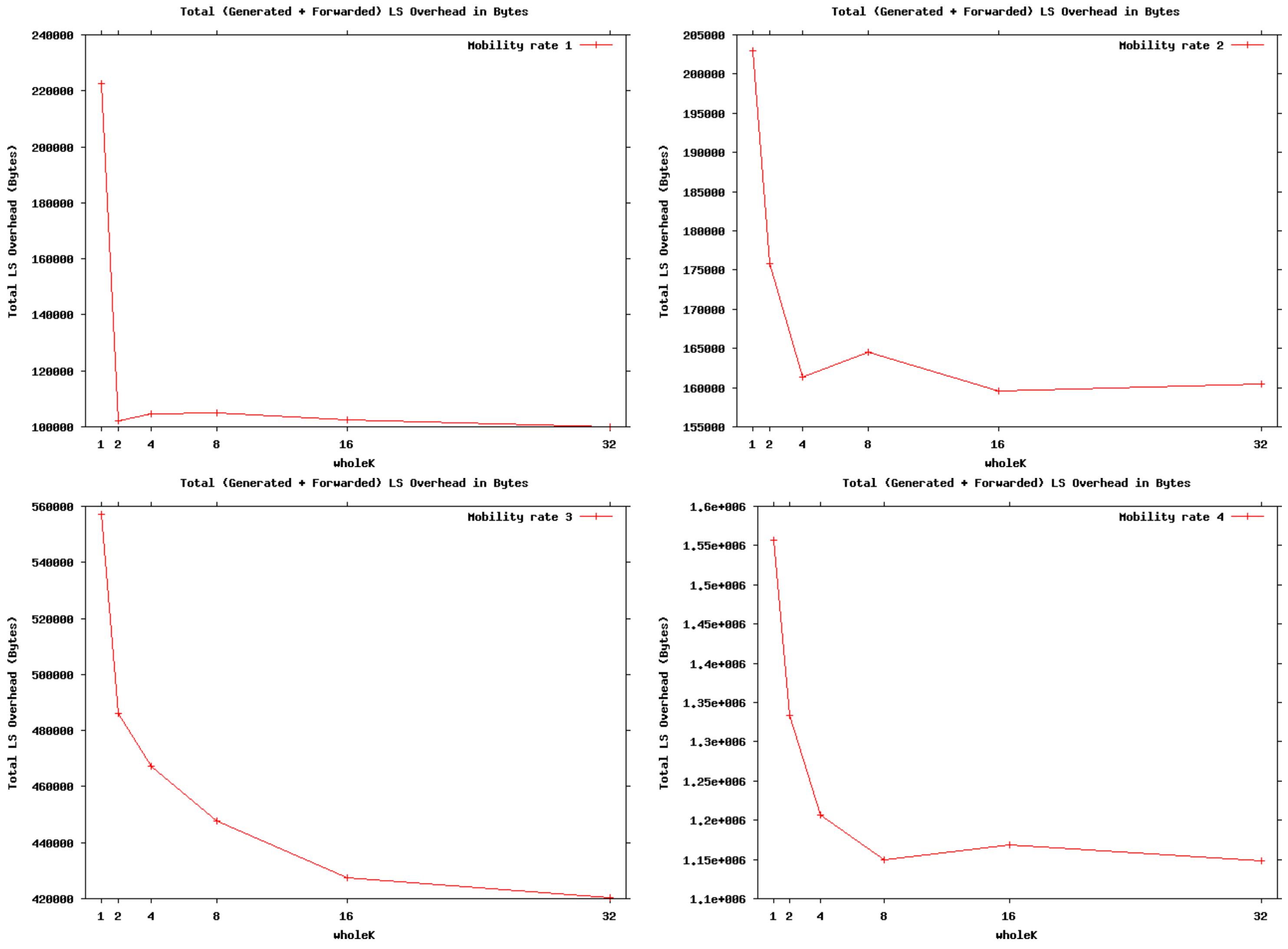

Figure 6.3: LS overhead in bytes vs. wholeK, for different mobility rates - individual cases.

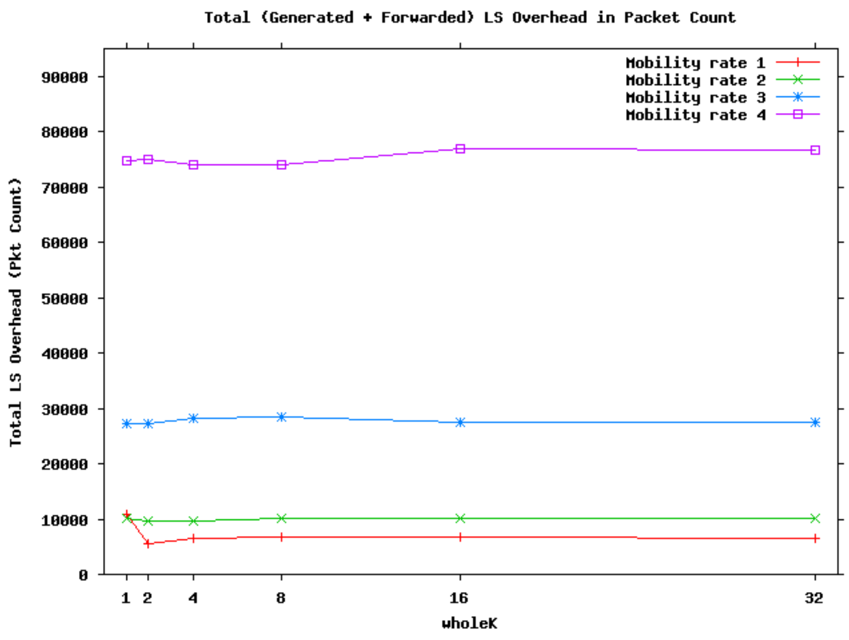

Figure 6.4: LS overhead in packet count vs. wholeK, for different mobility rates.

overhead of Elessar is acceptable. The effect of the incremental scheme cannot be observed in this graph due to the low overhead ratio, with the exception of the obvious improvement between values of 1 and 2 of $wholeK$ for mobility rate 4. We can observe such an improvement at mobility rate 4 since it incurs the highest overhead ratio among all mobility rates presented in the graph.

Figure 6.6 presents the results for LS overhead in bytes vs. wholeK, with changing network sizes. In the rest of the discussions in this section, whenever we say "changing network sizes", we mean that we increase the number of nodes in the network and the diameter of the network while keeping the node density and the average node degree the same. From the presented graph, we see that the LS overhead in bytes decreases as $wholeK$ increases, as expected, and that the LS overhead increases with increasing network sizes. We can observe roughly a 10 times increase in total LS overhead in terms of bytes when we change the node count from 10 to 25, corresponding to a 2.5 times increase in node count.

Figure 6.7 presents the same results for LS overhead in packet count. The

around 5, 10, 18, and 30 km/h, respectively, we believe that we are justified in our assumption of mobility rates closer to rates of 1 and 2 in real-life.

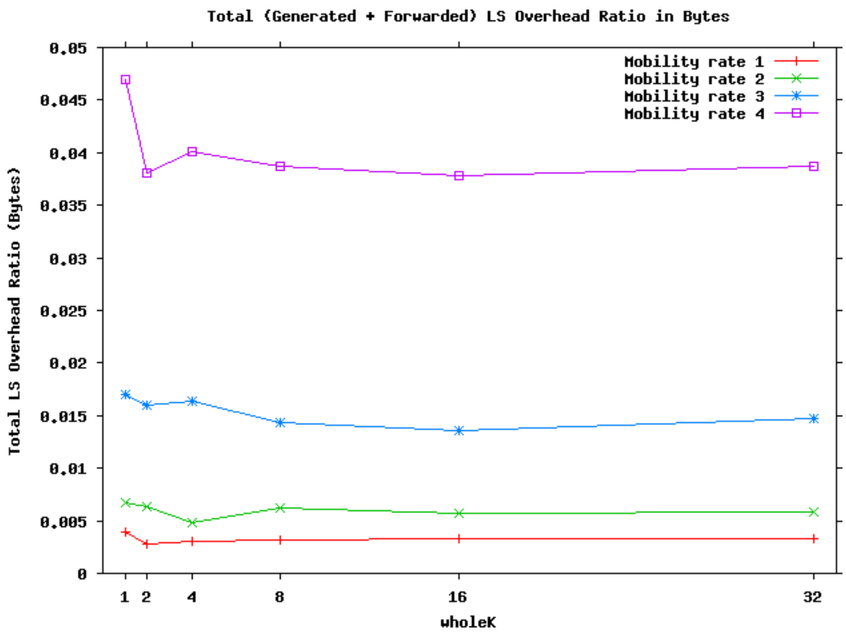

Figure 6.5: LS overhead ratio in bytes vs. wholeK, for different mobility rates.

LS overhead in terms of packet count shows a very similar behavior as the LS overhead in bytes. We can see that when we increase the node count by 2.5 times, the LS overhead increases roughly 10 times, from an average value of $12,000$ to $120,000$ packets, as also observed for the LS overhead in bytes case in Figure 6.6. Parameter $wholeK$ does not affect the number of LS packets created and only affects the sizes of LS packets and this phenomenon is also observed in Figure 6.7, as average LS overhead in packet count does not change as $wholeK$ changes.

Figure 6.8 presents the ratio of the LS overhead in bytes to the total received data size in bytes vs. $wholeK$, as node count changes. We can see from the figure that when node count $= 10$, our LS overhead is around 1% and when node count $= 25$, our LS overhead is around 8%, which corresponds to an 8 times increase in LS overhead ratio for an increase of 2.5 times in network size. As $wholeK$ increases, we observe a decrease in the LS overhead ratio since the total amount of LS packet size decreases with increasing $wholeK$. We do not observe such a behavior for node count $= 10$ in the graph due to the scale of the y-axis. Of course, the decrease in LS overhead ratio shows diminishing-returns, and we do not see significant gains for values of $wholeK > 8$.

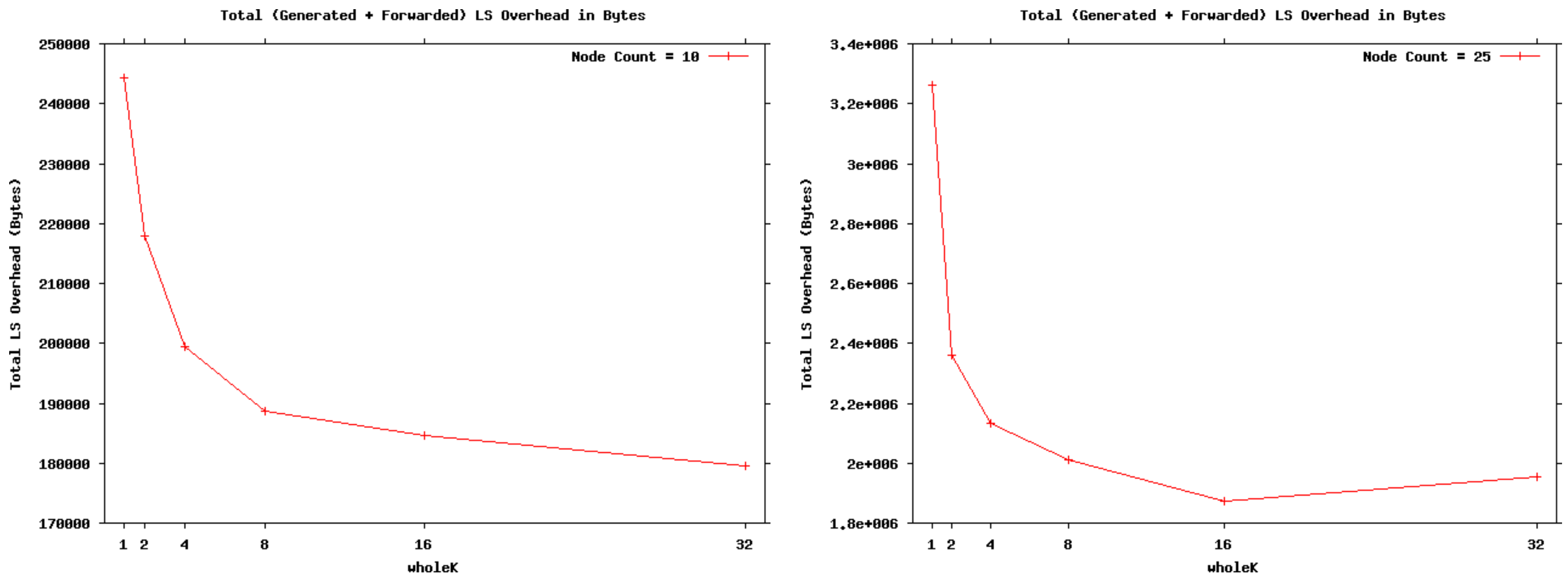

Figure 6.6: LS overhead in bytes vs. wholeK, for different network sizes - individual cases.

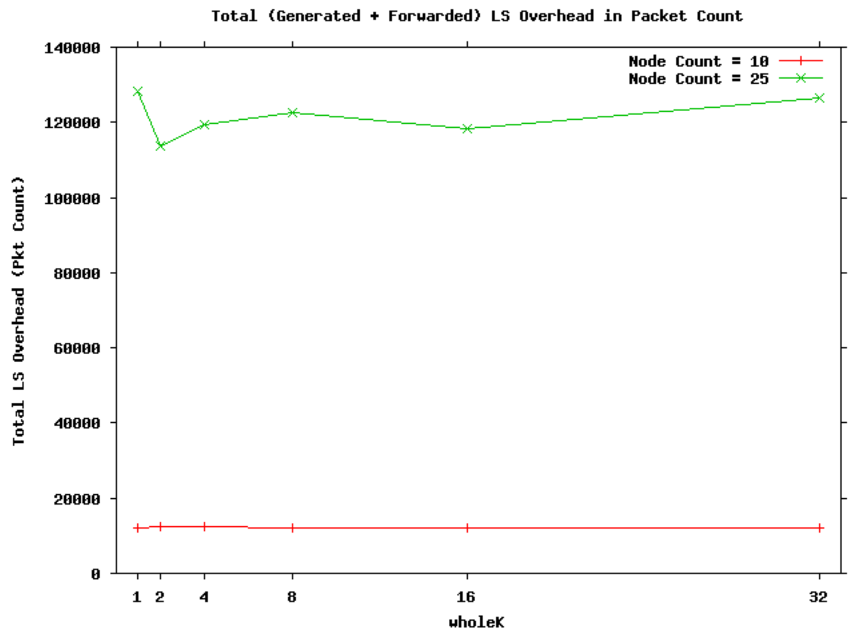

Figure 6.7: LS overhead in packet count vs. wholeK, for different network sizes.

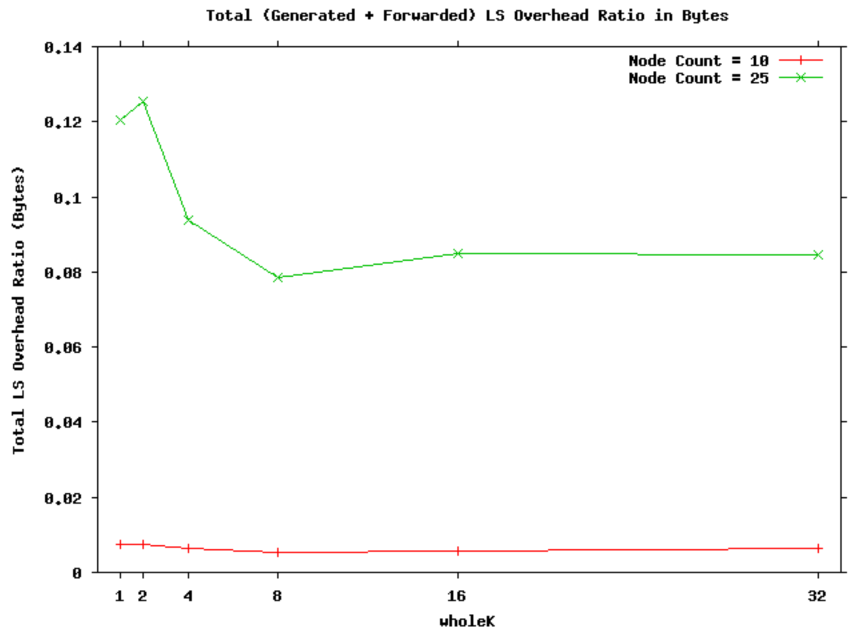

Figure 6.8: LS overhead ratio in bytes vs.wholeK, for different network sizes.

We provide results of total LS overhead in bytes vs. $wholeK$, as the node density changes[4] [5] in Figure 6.9. We can see from the graph that as $wholeK$ increases, we see a decrease in total LS overhead in bytes, as expected. This decrease is especially significant for higher values of node densities. We again observe that for values of $wholeK > 8$, the decrease in LS overhead diminishes. As the node density increases, LS overhead also increases, due to the increase in the node count and the average node degree in the network.

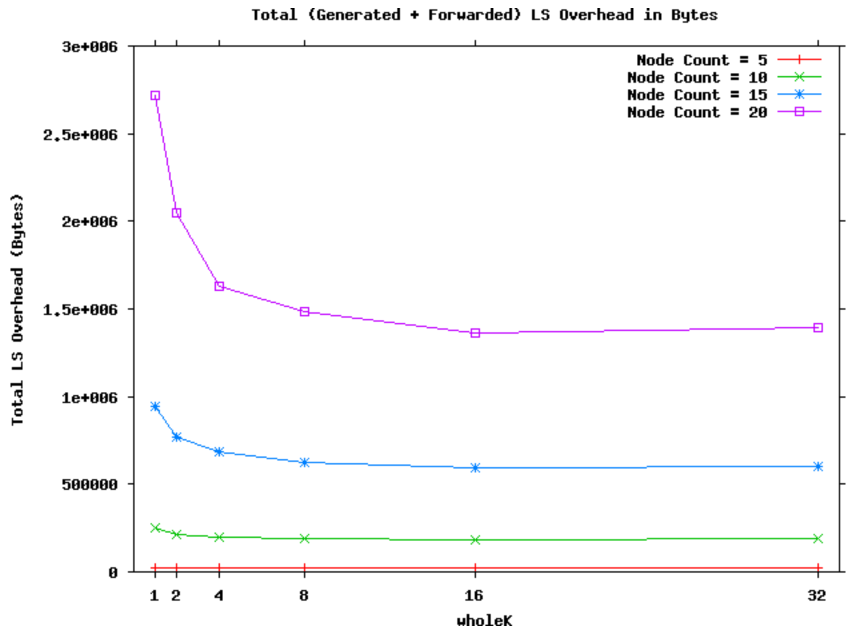

Figure 6.9: LS overhead in bytes vs. wholeK, for different node densities.

We can see similar results in terms of packet count in Figure 6.10. As the node density increases, LS overhead in terms of packet count also increases. We observe that parameter $wholeK$ does not affect the number of LS packets, as expected.

Figure 6.11 shows total LS overhead ratio in bytes vs. $wholeK$, as node density changes. We can see from the graph that as node density increases, LS overhead ratio also increases due to the increase in total LS overhead. Increasing $wholeK$ decreases the LS overhead ratio since it lowers total LS overhead in bytes.

[4]Please note that node counts of 5, 10, 15, and 20 correspond to node densities of 0.0005, 0.0010, 0.0015, and 0.0020 $nodes/m^2$, respectively.

[5]As we increase the node density of the network, we also increase the average node degree in the network.

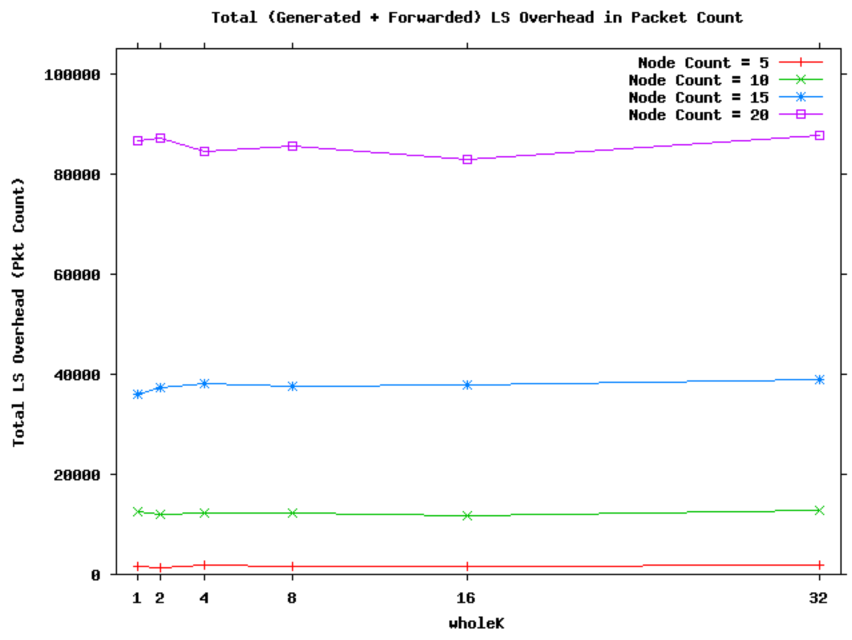

Figure 6.10: LS overhead in packet count vs. wholeK, for different node densities.

A value between 8 and 16 seems to be a good choice for $wholeK$ as for values of $wholeK > 16$, the decrease in the overhead ratio is not very significant. Even for the highest node density with node count = 20, the average overhead ratio is around 5%, and the overhead decreases to values of almost 0% as node density decreases.

Figures 6.12, 6.13, 6.14, and 6.15 present results on total LS overhead in bytes, total LS overhead in packet count, LS ratio in bytes, and LS ratio in packet count, respectively, vs. node densities, for differing mobility rates. The behavior observed in each graph is almost the same, as we see that LS overhead increases as node density and/or mobility rate increase, as expected. The LS overhead ratio in bytes is $\leq 4\%$ for almost all cases, with the exception of node count = 20 and mobility rate of 4, which corresponds to the highest node density and mobility rate in the presented graph. A similar scenario is present when we look at LS overhead ratio in packet count, where we observe ratios $\leq 80\%$ of LS packets to data packets for almost all cases, with the exception of the combination of the highest node density and the highest mobility rate.

We present in figures 6.16, 6.17, 6.18, and 6.19 the average LS packet sizes

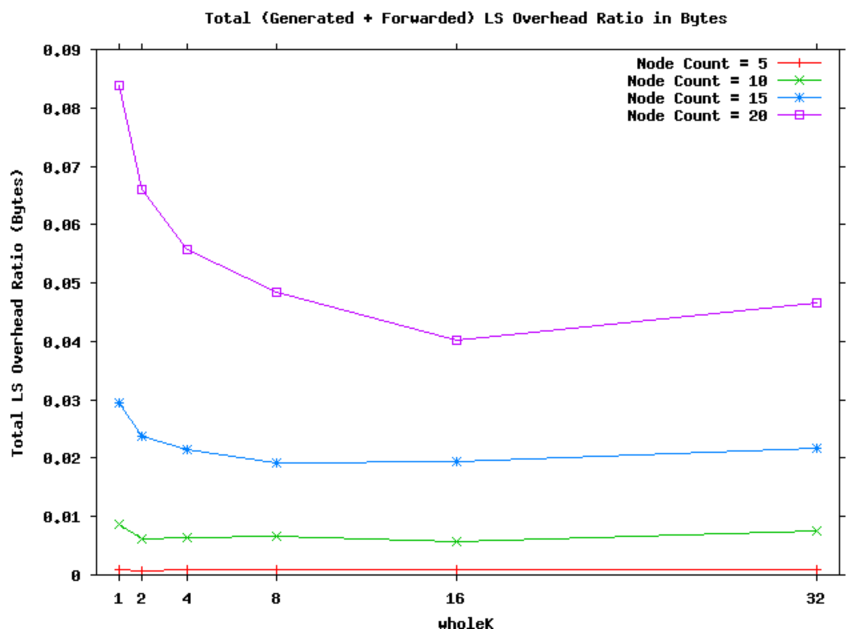

Figure 6.11: LS overhead ratio in bytes vs. wholeK, for different node densities.

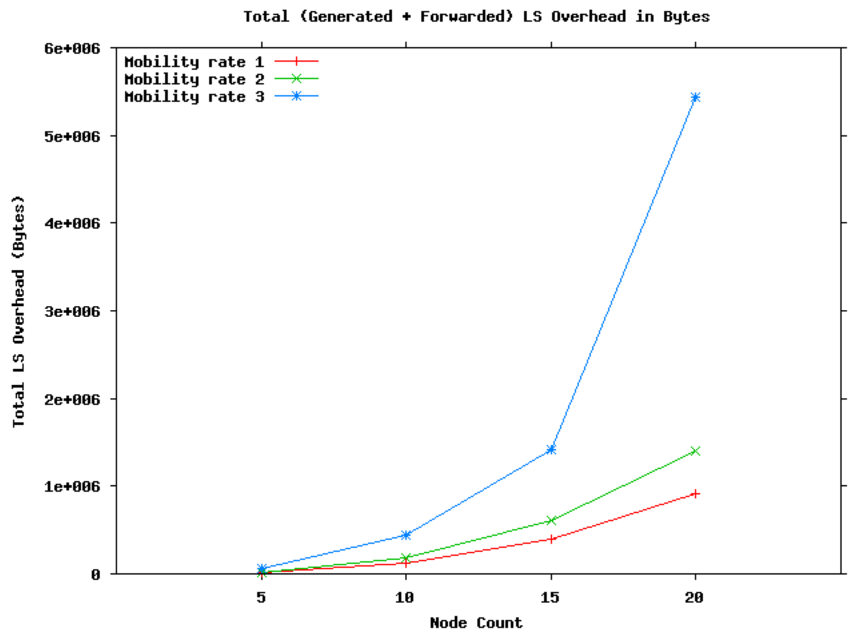

Figure 6.12: LS overhead in bytes vs. node density, for different mobility rates.

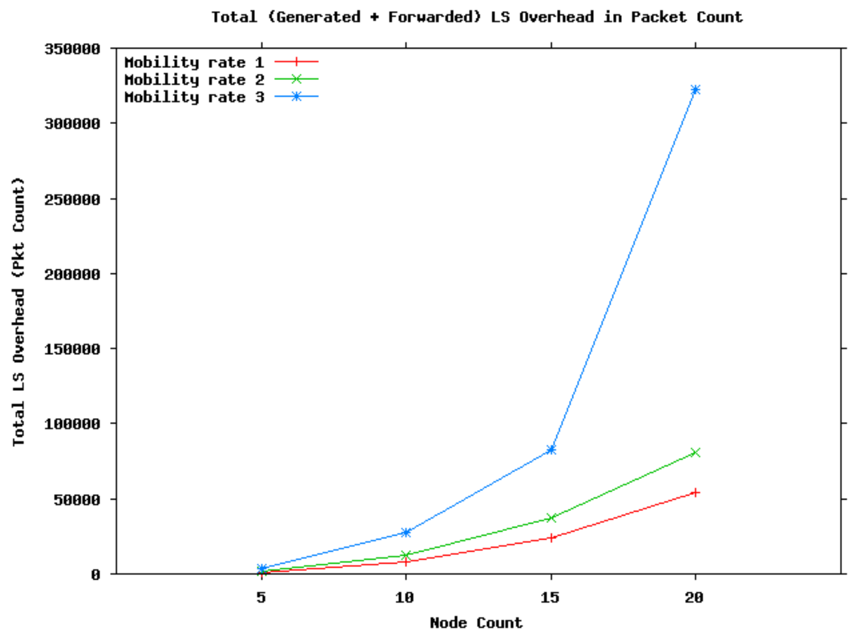

Figure 6.13: LS overhead in packet count vs. node density, for different mobility rates.

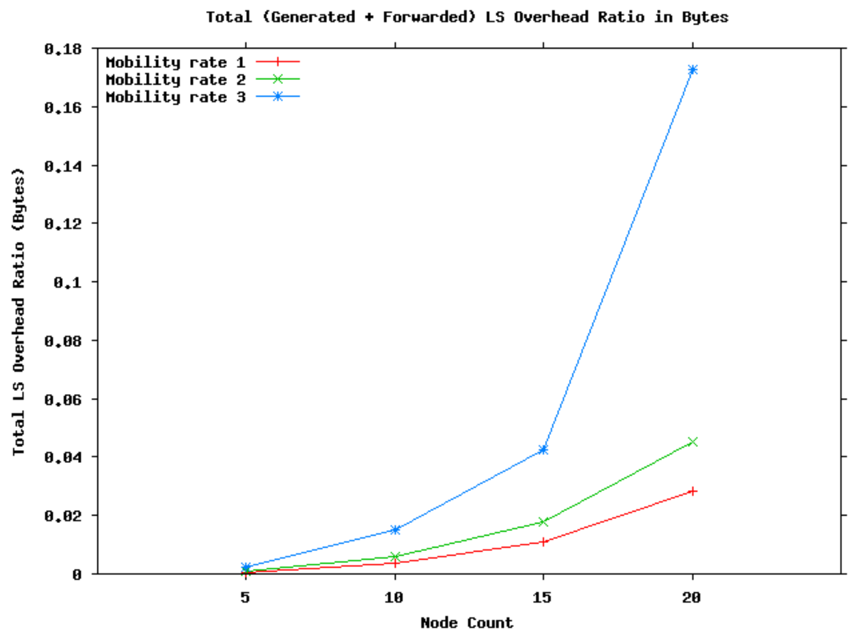

Figure 6.14: LS overhead ratio in bytes vs. node density, for different mobility rates.

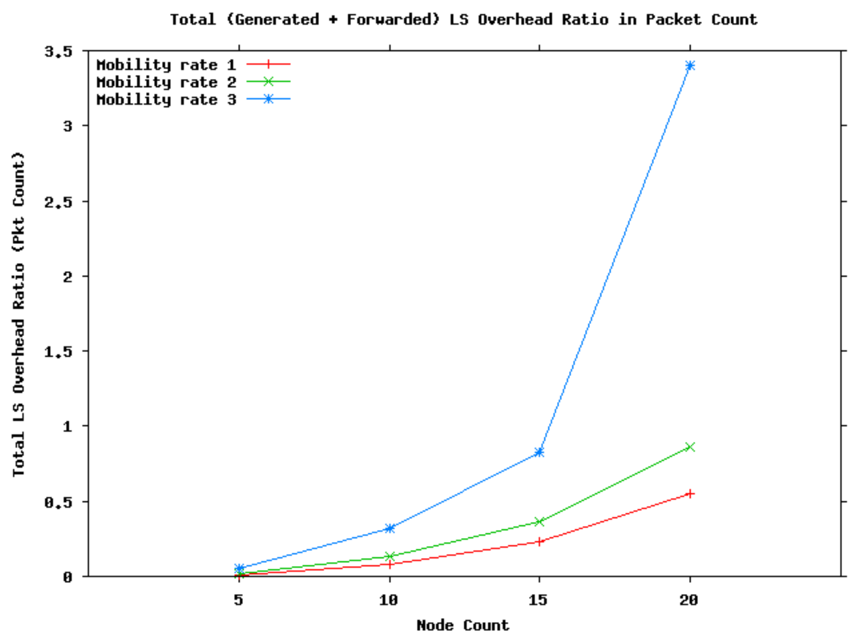

Figure 6.15: LS overhead ratio in packet count vs. node density, for different mobility rates.

in bytes. In Figure 6.16, we see that the average LS packet length in bytes is independent of the mobility rate since the sizes of individual LS packets do not change as mobility rate changes, but rather the number of LS packets created and forwarded in the network changes as mobility rate changes. In the same figure, we observe the effect of $wholeK$ on the average LS packet size, where increasing $wholeK$ decreases the average LS packet length as expected. As we had observed before, we again see that we have a diminishing-returns and values of $wholeK$ between 8 and 16 seem to be the most suitable in terms of average LS packet lengths.

Figure 6.17 presents the average LS packet length vs. $wholeK$, for differing network sizes. We have a higher average packet length for larger networks, since the number of nodes and therefore the information included in LS packets increase with increasing network size. The effect of $wholeK$ on average LS packet length is as before.

We can see the effect of node density on average LS packet length in Figure 6.18. We see in this graph that average packet length increases as node density

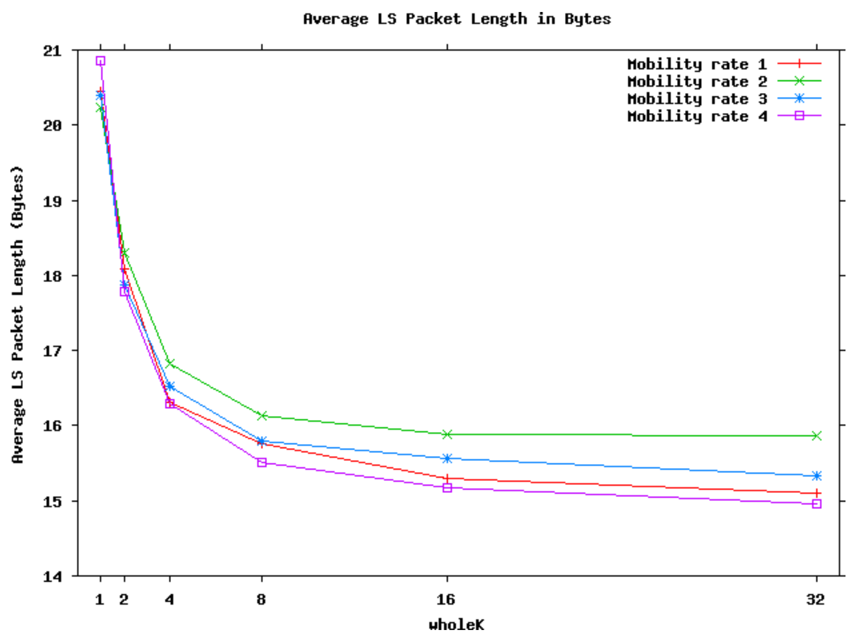

Figure 6.16: Average LS packet lengths vs. wholeK, for different mobility rates.

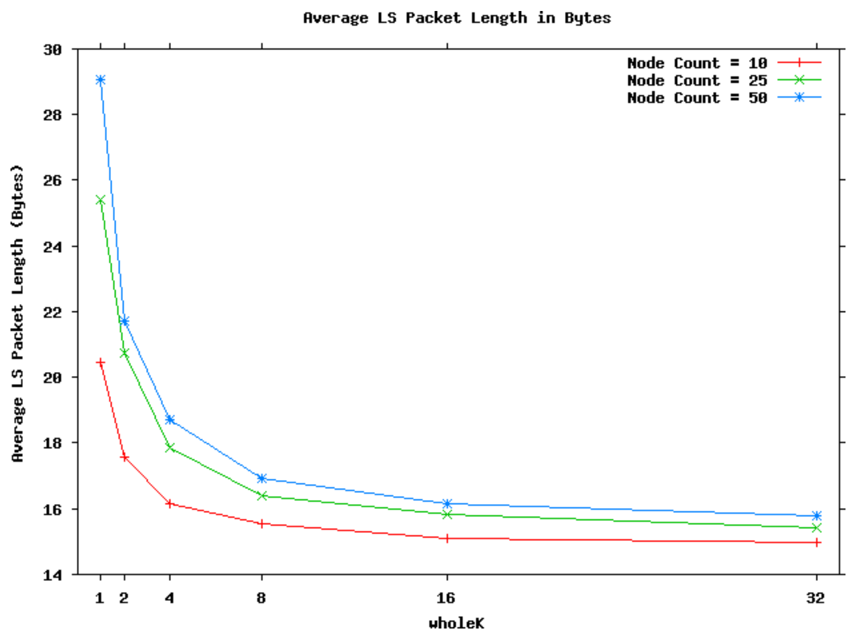

Figure 6.17: Average LS packet lengths vs. wholeK, for different network sizes.

increases. Remembering that increasing node density means increasing average node degree, it is expected for average LS packet length to increase as node density increases since the number of neighbors of a node and respectively the amount of information carried in an LS packet increases. The effect of *wholeK* on average LS packet length is as before.

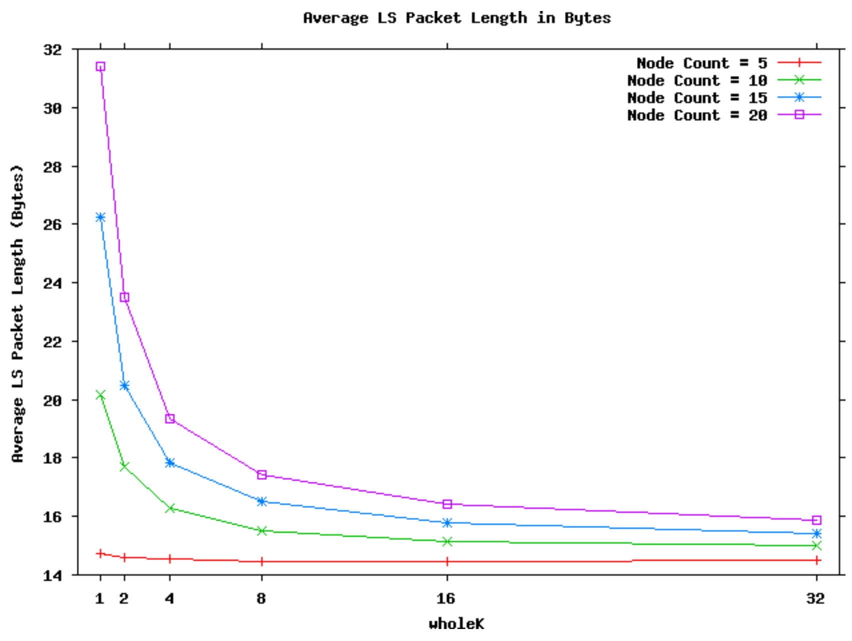

Figure 6.18: Average LS packet lengths vs. wholeK, for different node densities.

Figure 6.19 shows average LS packet length vs. node density, for different mobility rates. We observe that average LS packet length is independent of mobility rate, as was stated before, and that average LS packet length increases with increasing node density.

We present results on the overhead of our protocol due to source routing (the inclusion of routes in packet headers) in figures 6.20, 6.21, 6.22, 6.23, and 6.24.

From Figure 6.20, we see that the overhead ratio[6] due to source routing is independent of both the mobility rate and *wholeK*. We seem to have high variance in this graph, but this is due to the scale of the y-axis and if we look at the

[6]Letting R_G and R_F denote the total amount of space in bytes occupied by source routes in data headers of all generated and forwarded data packets, respectively, and D_G and D_F denote the total sizes (including headers) of all generated and forwarded data packets in bytes, respectively, we calculate the overhead ratio due to source routing as $\frac{R_G+R_F}{D_G+D_F}$.

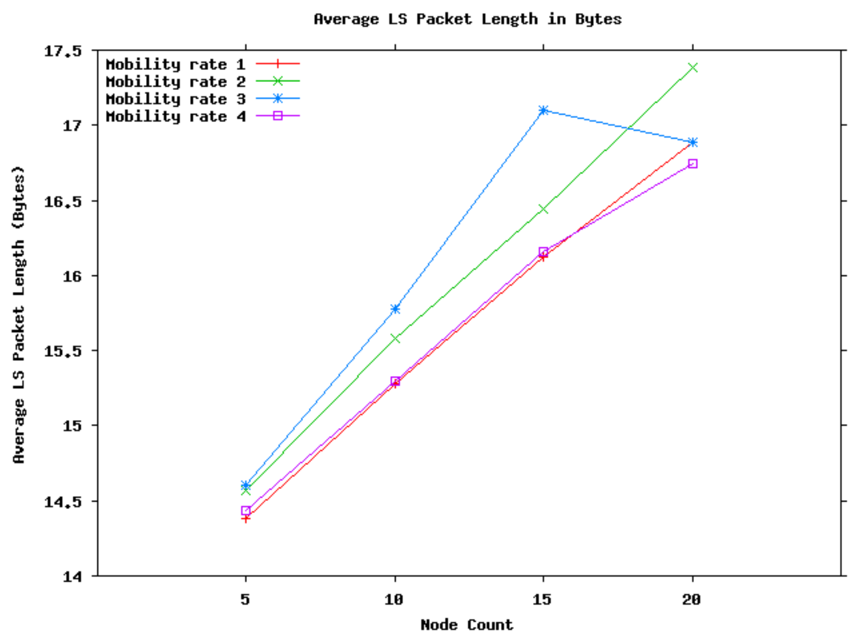

Figure 6.19: Average LS packet lengths vs. node density, for different mobility rates.

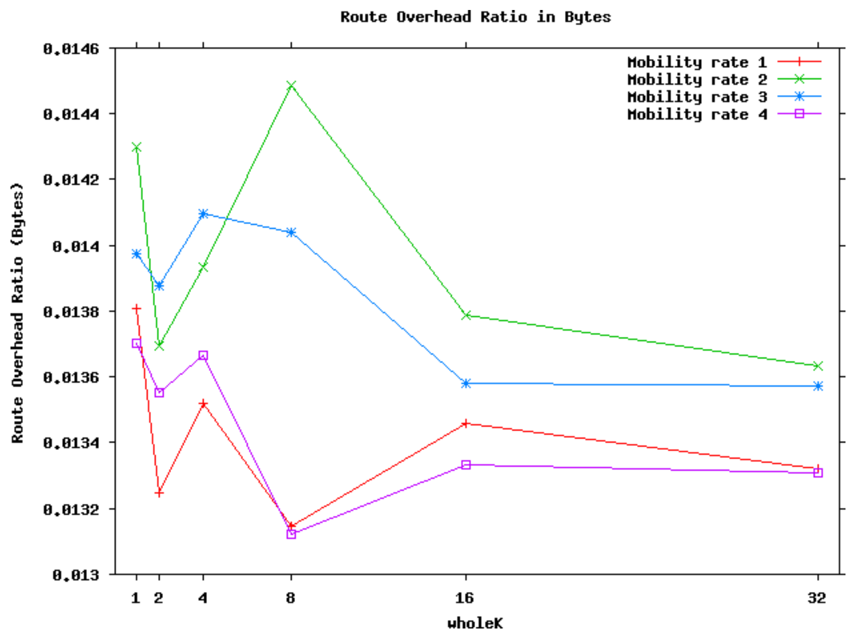

Figure 6.20: Source routing overhead ratios in bytes vs. wholeK, for different mobility rates.

actual values, we can see that the highest and the lowest values for source routing overhead are 1.45% and 1.31%, respectively.

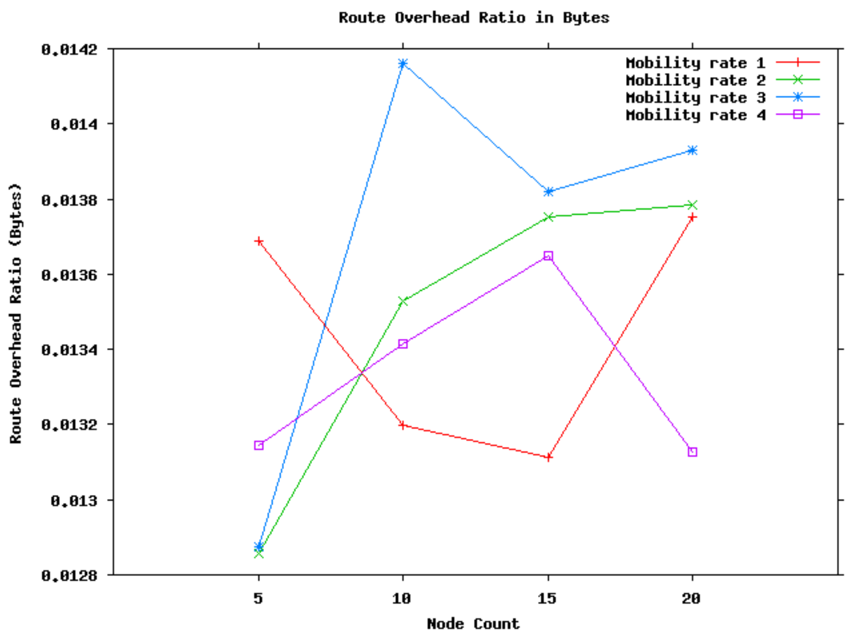

Figure 6.21: Source routing overhead ratios in bytes vs. node density, for different mobility rates.

We observe a similar case when we plot the route overhead vs. node density, for changing mobility rates. The source routing overhead seems to be independent of the mobility rate and the node density, which is as expected, since changing the mobility rate does not change the length of the routes and changing the node density does not change the diameter of the network, which is the major factor influencing the average route length. Figure 6.21 also seems to present a lot of variance, but this is again due to the scale of the y-axis, and we can see from the graph that the highest and lowest overhead ratios are 1.42% and 1.29%, respectively.

We can see from Figure 6.22 that as the network size increases, source routing overhead increases from around 1.4% to 2.0%, since when we increase the network size, we also increase the network diamater, and therefore the average route length between any two nodes in the network.

Figure 6.23 shows the source routing overhead in bytes per data packet for

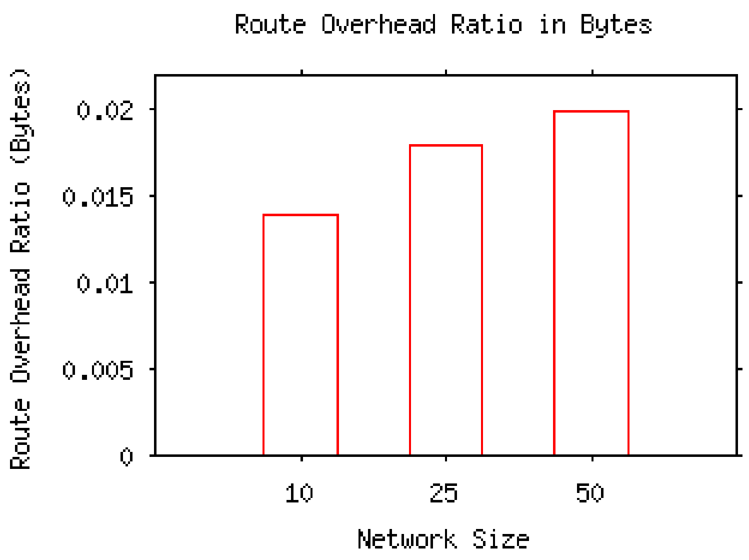

Figure 6.22: Source routing overhead ratios in bytes vs. network size.

different values of network size. As expected, the average amount of space occupied by a source route in the data header increases with increasing network size, since when we increase the network size, we also increase the network diameter, resulting in longer routes.

Figure 6.24 shows the source routing overhead in bytes per data packet for different values of node density and we see from the graph that there are slight increases in route overhead per packet for increasing node densities. As the node density increases in the network, the number of available paths between any two nodes in the network increases. Since we have enabled route maintenance in these experiments, with increasing numbers of available paths between any two nodes, intermediate nodes have a greater chance in finding alternate but possibly longer paths to the destinations in cases of route breakages, and these longer paths are used until the source node is informed of the situation and finds a better/shorter path. Due to the greater use of these non-optimal (longer) paths in cases of route breakages, the average amount of space occupied by a source route in a data packet increases with increasing node density, however slight the increase may be.

After going over these results, we can see that the overhead caused by the

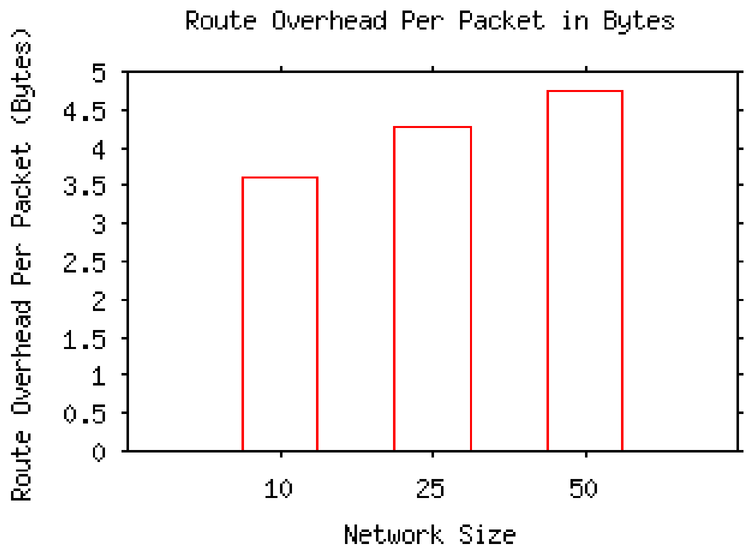

Figure 6.23: Source routing overhead per packet in bytes vs. network size.

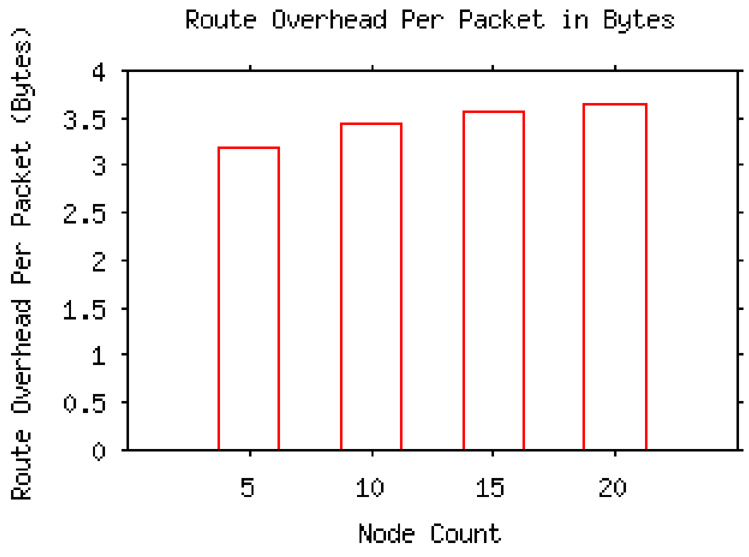

Figure 6.24: Source routing overhead per packet in bytes vs. node density.

topology dissemination mechanism is acceptable for small-to-medium sized networks and for larger networks with a high number of data flows in the network. Protocol overhead due to source routing seems to be quite low and therefore acceptable for all network sizes. In light of these results, we may say that our protocol is best suited, in terms of protocol overhead, for small-to-medium sized networks having low-to-high data traffic densities, and for larger networks having medium-to-high data traffic densities. Elessar is able to support considerably high mobility rates, easily supporting node speeds of up to 10 m/s, where nodes are constantly in motion, without incurring excessive overhead. We observe that our incremental message scheme for topology dissemination is able to lower LS overhead significantly, and desirable $wholeK$ values are generally between 8 and 16.

The simulation parameters used in obtaining the results presented in this section are given in Table 6.5. Please note that the playground size parameter denotes the height and width of the network deployment area in meters.

In the following section, we examine the performance of Elessar in routing normal data.

6.1.2 Protocol Performance in Normal Data Routing

We present results on the performance of our protocol in routing normal data. We specifically look at the data loss ratios and the route find success rates for normal data flows. Whenever an intermediate node is unable to forward a data packet to its next hop, it drops the packet[7], constituting a data packet loss. We also include dropped data packets at source nodes due to unreachable destinations (the destination node and the source node are in disconnected sections of the network) in the lost data packets counter, and we calculate the data loss ratio as the number of lost data packets to the number of all generated and forwarded data packets. Aside from trying to send a data flow to a disconnected destination,

[7]If route maintenance is enabled, the intermediate node will first try to find an alternate path to the destination of the packet, and if it is unable to find such a path, it will then drop the packet.

Parameter Name / Figure Number	wholeK	Playground Size	Node Count	Mobility Rate
Figures 6.2, 6.3, 6.4, 6.5, 6.16	varies	100×100	10	varies
Figures 6.9, 6.10, 6.11, 6.18	varies	100×100	varies	Rate 2
Figures 6.6, 6.7, 6.8, 6.17	varies	varies	varies	Rate 2
Figures 6.12, 6.13, 6.14, 6.15, 6.19	8	100×100	varies	varies
Figure 6.20	varies	100×100	10	varies
Figure 6.21	8	100×100	varies	varies
Figure 6.22	16	varies	varies	Rate 2
Figure 6.23	16	varies	varies	Rate 2
Figure 6.24	32	100×100	varies	Rate 2

Table 6.5: Simulation parameters for overhead experiments in normal mode operation.

data losses will be caused by bit errors, collisions, and node mobility. The MAC layer is responsible for handling packets with bit errors and corrupted packets due to collisions, so we ignore such packets in our lost data calculations to focus more on the performance of our protocol in the face of topology changes.

When a node needs to find a route for a packet, it first looks into its route cache, and if it cannot find a suitable route in the route cache, it then initiates the local path finding algorithm in order to find a route. We calculate route find success rate as the number of successful route lookups (including both route cache lookups and runs of the local path finding algorithm) to the total number of route lookups. Note that when a node tries to find a route to a destination when the two nodes are in disconnected sections of the network, this will cause an unsuccessful route lookup. Aside from such disconnections, unsuccessful route lookups will be caused by inconsistencies between the actual network and the local graph representation of the network at the nodes. Such inconsistencies may be caused due to bit errors in LS packets and/or corrupted LS packets due to collisions.

Figure 6.25 presents the route find success rate vs. $wholeK$, for changing mobility rates. We observe that the mobility rate does not have a strong effect on the route find success rate, whereas increasing parameter $wholeK$ decreases the route find success rate, from around 98% success rate for $wholeK = 1$, down to rates of 84% for $wholeK = 32$. Remembering that inconsistencies in local graph representations of the network at nodes is one strong factor responsible for unsuccessful route lookups, the latter result is as expected. As we increase the distance between whole LS messages by increasing $wholeK$, we decrease the robustness of the protocol, since the inconsistency caused by a lost incremental LS message will be resolved by a whole LS message, and as the distance between whole messages increases, the inconsistency in the local graph will persist longer, causing more unsuccessful route lookups. We can conjure from the presented graph that parameter $wholeK$ has a much stronger effect on route find success rates than the mobility rate of nodes, and choosing a value between 1 and 4 for $wholeK$ seems to be a reasonable choice.

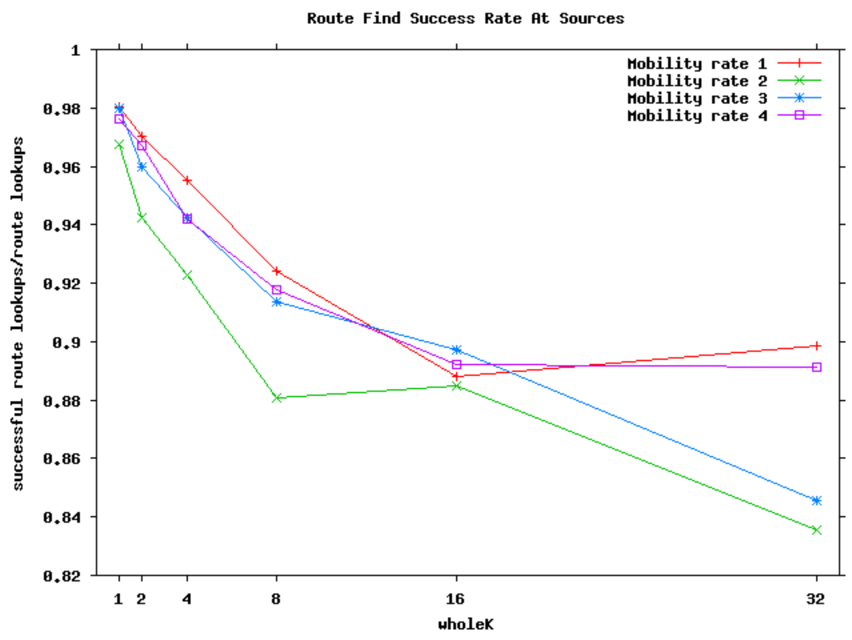

Figure 6.25: Route finding success rates vs. wholeK, for different mobility rates.

We can see the effect of network size on route find success rate in Figure 6.26. The minimum route find success rate of our protocol is around 90% at a network size of 10 nodes and the success rate increases with increasing network sizes. As the network size increases, the number of paths between any two nodes in the network increases, and even when a route breaks between such two nodes, our protocol is able to find an alternate path in a greater percentage of such cases, increasing the route find success rate to a value of around 97% when the network size is 50 nodes.

Route find success rate vs. $wholeK$, for changing node densities is given in Figure 6.27. We can see that there is a general tendency for the route find success rate to fall with increasing $wholeK$, and this is due to the decreased robustness of the protocol to lost LS packets in the incremental messaging scheme, with increasing $wholeK$. We also observe that as the node density increases, the route find success rate increases, due to the availability of more paths between any two nodes in a more densely-deployed network that has more nodes. The low route find success rates for the node count = 5 case is due to the high number of disconnected components in the network.

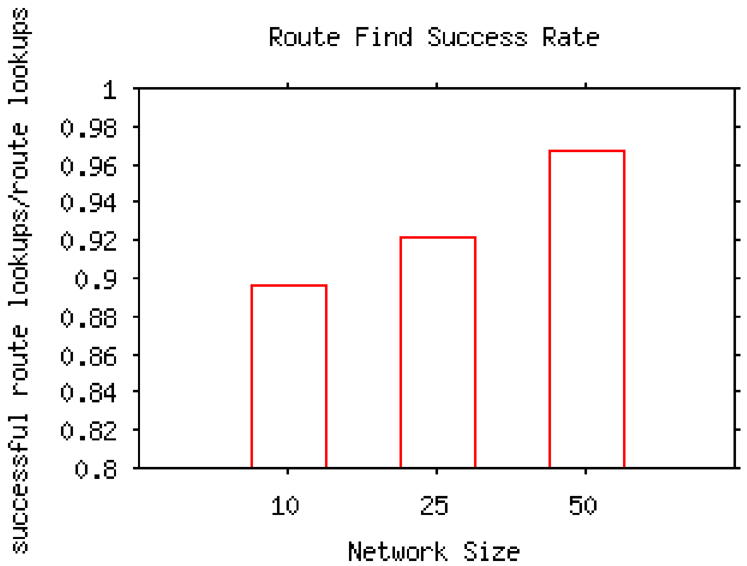

Figure 6.26: Route finding success rates vs. network size.

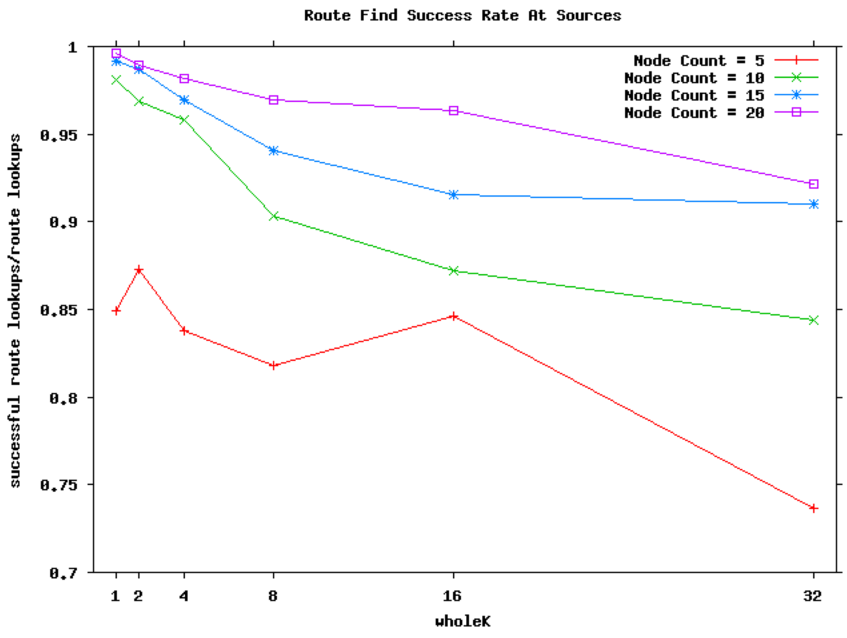

Figure 6.27: Route finding success rates vs. wholeK, for different node densities.

Figure 6.28 gives results on the route find success rate vs. node density, for changing mobility rates. We observe that the mobility rate has a negligible effect on route find success rate compared to the node density, and we see that there is a direct relationship between node density and route find success rate, due to reasons mentioned before. We again observe low route find success rates for the node count = 5 case, due to the high number of disconnected components in the scarcely-deployed network.

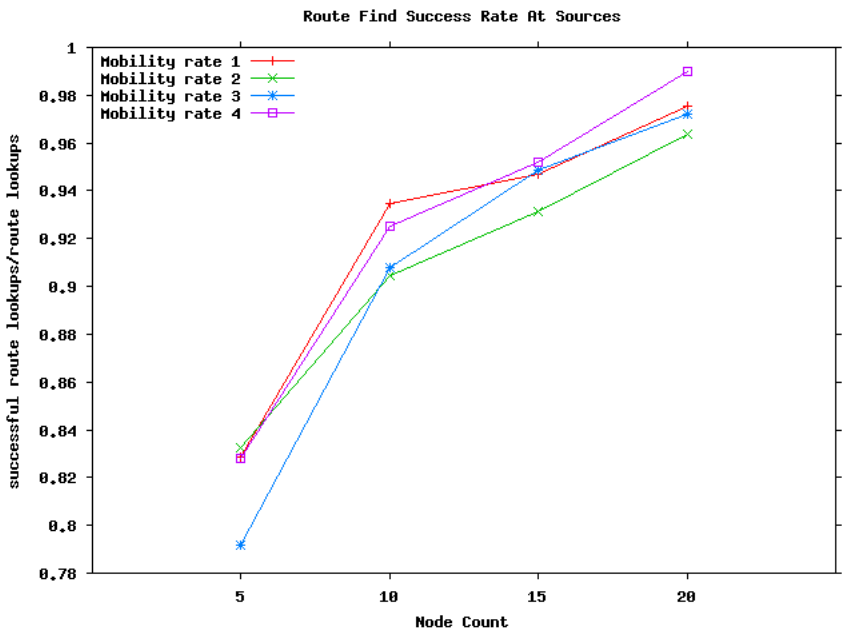

Figure 6.28: Route finding success rates vs. node density, for different mobility rates.

From these results on route find success rates, it can be conjured that values between 1 and 4 seem to be reasonable choices for *wholeK*. The rest of the presented graphs in this section provide results on data loss ratios.

Figure 6.29 presents results on data loss ratio vs. *wholeK*, for different mobility rates. We observe that the mobility rate has a negligible effect on data loss ratios compared to *wholeK*. A direct relationship between data loss ratio and *wholeK* may be observed in this graph, where data loss ratios are as low as 2% for *wholeK* = 1 and go up to 14% for *wholeK* = 32. This observed behavior is due to the decreased robustness of the protocol and the increased number of nodes with discrepancies in their local network views with increasing $wholeK$, as mentioned before.

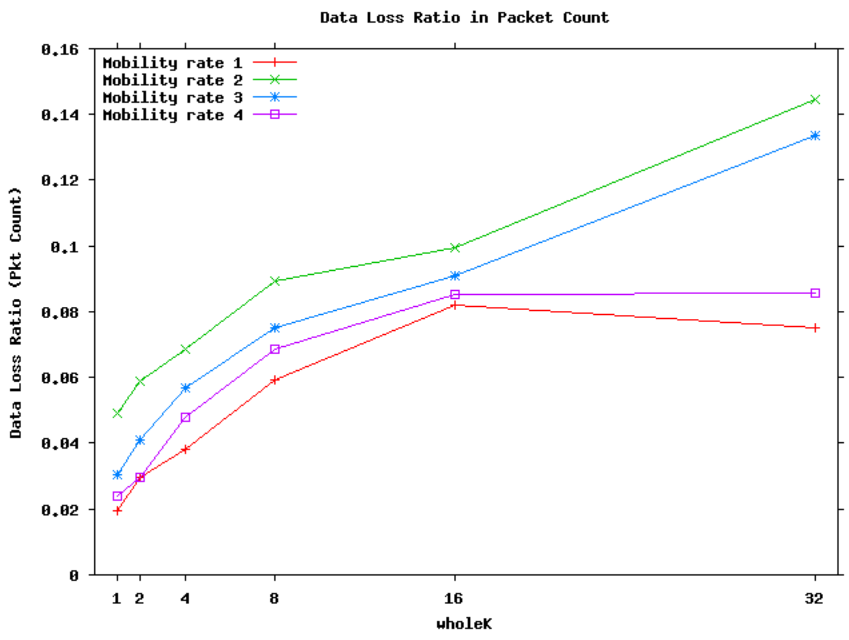

Figure 6.29: Data loss ratios in packet count vs. wholeK, for different mobility rates.

Figure 6.30 shows data loss ratios for different network sizes. It can be seen from the graph that the data loss ratio decreases with increasing network size. This is due to the increased number of paths between any two nodes in the network with increased network size.

Figure 6.31 gives data loss ratios vs. $wholeK$, for changing node densities. As oberved before, with increasing $wholeK$, data loss ratio increases. We also observe an inverse relationship between data loss ratio and node density, which is due to the increased number of paths between any two nodes in the network with increasing node density and node count. The high data loss ratios observed when node count $= 5$ is due to the high number of disconnected components in the graph in this case.

We can see from Figure 6.32 that mobility rate has a negligible effect on data loss ratio compared to node density, and we observe an inverse relationship between data loss ratio and node density, as before.

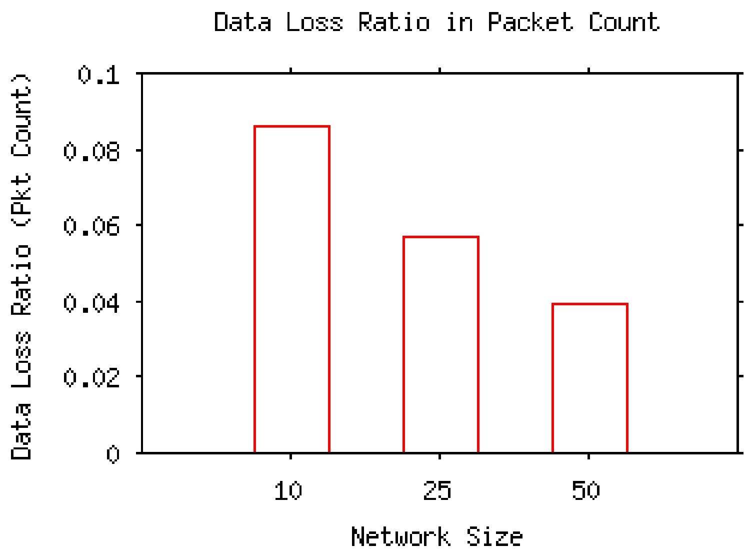

Figure 6.30: Data loss ratios in packet count vs. network size.

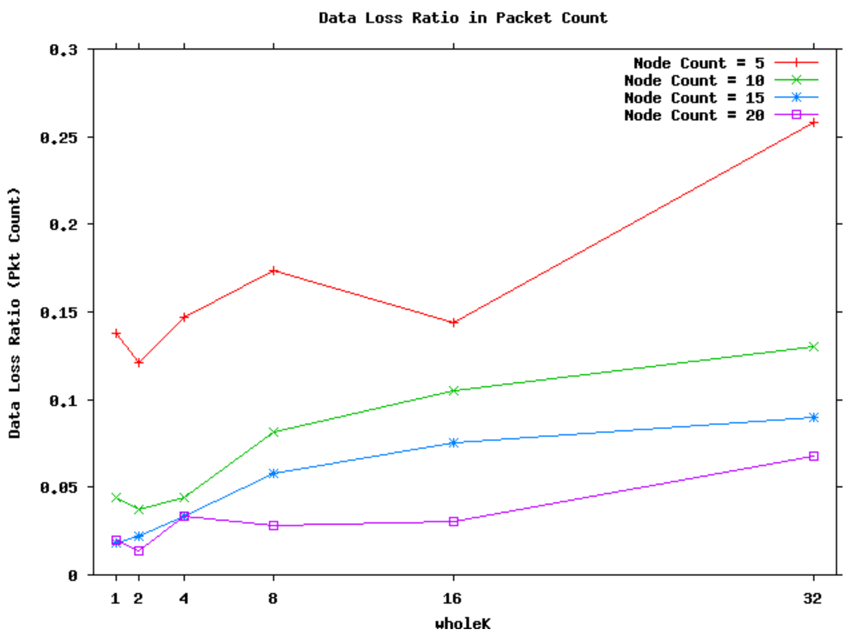

Figure 6.31: Data loss ratios in packet count vs. wholeK, for different node densities.

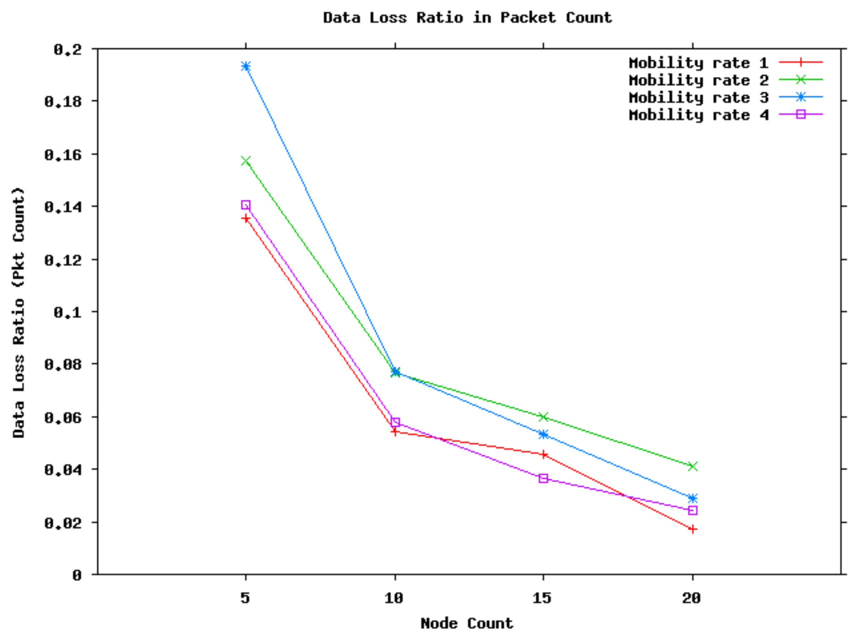

Figure 6.32: Data loss ratios in packet count vs. node density, for different mobility rates.

Figures 6.33 and 6.34 show results on the performance of the route maintenance mechanism of Elessar. We observe from these graphs that our route maintenance mechanism is able to improve data loss ratios by 1% to 3%.

After looking at the performance of Elessar in normal data routing, we see that our protocol is especially suited for densely-deployed, medium-sized networks in terms of routing performance. We generally observe data loss ratios lower than 8% for connected networks (node count $\geq$ 10) with intelligent selection of $wholeK$. Values between 1 and 4 for $wholeK$ seem to provide the best routing performance. We had found earlier that values between 8 and 16 for $wholeK$ were the most appropriate choices for protocol overhead. In light of this information, it seems that setting $wholeK$ to a value between 4 and 8 would provide a nice compromise between protocol overhead and routing performance. If routing performance is of more importance, we should have $2 \leq wholeK \leq 4$; on the other hand, if protocol overhead is of importance, we may have $6 \leq wholeK \leq 12$.

Simulation parameters used in obtaining the results presented in this section are given in Table 6.6.

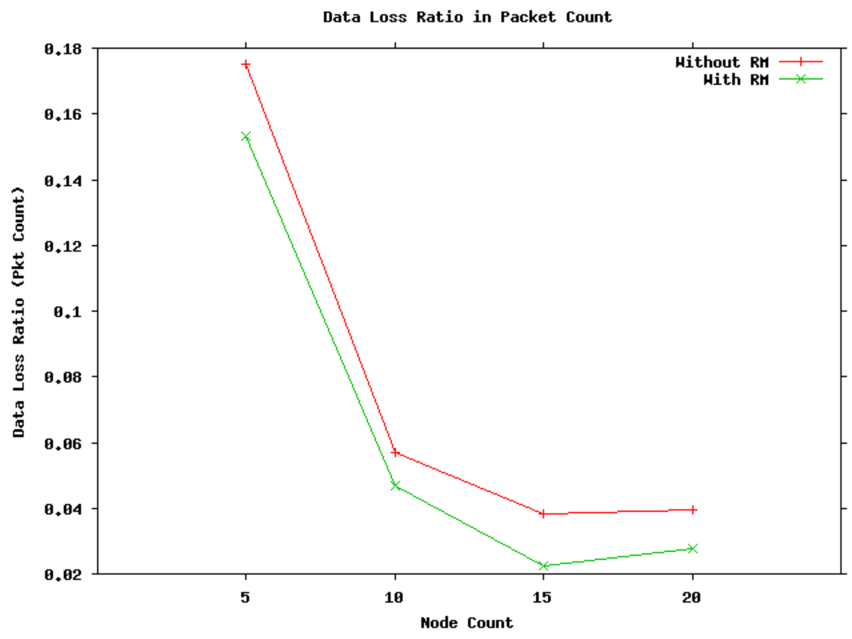

Figure 6.33: Data loss ratios in packet count vs. node density, with/without route maintenance.

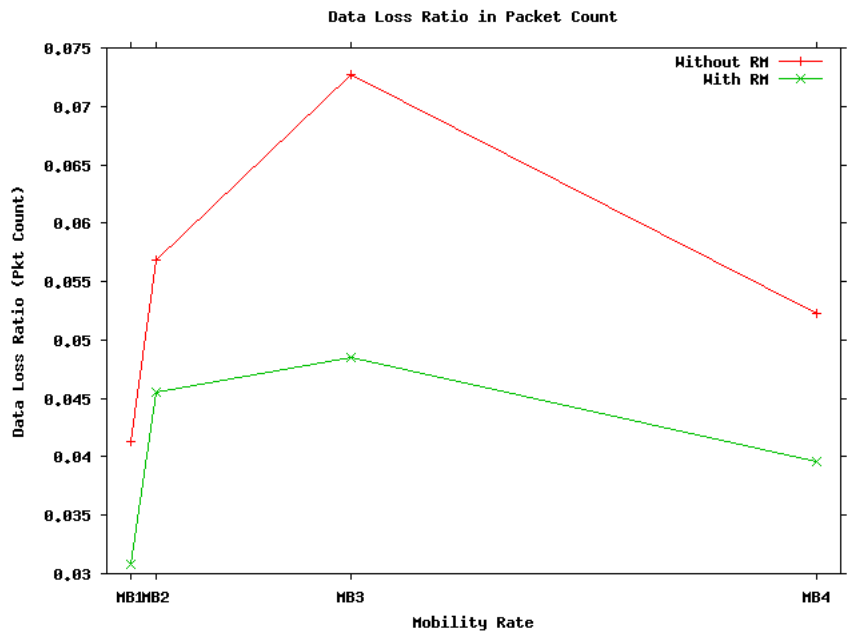

Figure 6.34: Data loss ratios in packet count vs. mobility, with/without route maintenance.

Parameter Name / Figure Number	wholeK	Playground Size	Node Count	Mobility Rate
Figures 6.25, 6.29	varies	100×100	10	varies
Figures 6.27, 6.31	varies	100×100	varies	Rate 2
Figures 6.28, 6.32	8	100×100	varies	varies
Figures 6.26, 6.30	16	100×100	varies	Rate 2
Figure 6.33	4	100×100	varies	Rate 2
Figure 6.34	4	100×100	10	varies

Table 6.6: Simulation parameters for routing performance experiments in normal mode operation.

In this section, we have presented results on the main sources of our protocol overhead, namely the topology dissemination mechanism and source routing. We also looked at the performance of Elessar in normal data routing for different scenarios. The results provided in this section were generic and largely independent of the types of data flows. We present in the next section experimental results specific to the QoS operation mode of Elessar, looking into how well our protocol is able to support real-time traffic.

6.2 Experimental Results for QoS Mode

In this section, we present simulation results regarding operation of Elessar in QoS mode in the presence of real-time flows. We present results on the overhead of our protocol due to the cost dissemination mechanism, along with total induced overhead. We also look at the performance of the proposed protocol in supporting real-time flows. Elessar is currently has built-in support for delay-sensitive, loss-sensitive, and bandwidth-sensitive traffic. We present results on each type of traffic support separately.

The cost overhead of our protocol is independent of the supported traffic type, therefore we only present total overhead and cost mechanism related overhead results for the delay-sensitive traffic case. Elessar is able to support different types of real-time and normal traffic concurrently, but we focus on real-time flows only in this investigation, without mixing real-time and normal data flows. We also do not have different types of real-time traffic running concurrently in the network in these experiments, in order to correctly analyse the performance of our protocol for specific real-time flow types and for the sake of clarity.

Simulation parameters regarding the wireless channel, network interface card, and the IEEE 802.11 MAC layer were presented in Table 6.1. We use those same parameters in the experiments discussed in this section.

Each QoS experiment was conducted in the same manner of normal data experiments. For each experiment, we create an ad hoc network of fixed size, where nodes do not join or leave during the lifetime of the network. Of course, nodes in a network are mobile and move according to the random waypoint mobility model. Note that the network size may be different for each experiment. All nodes in the network are deployed on a fixed size area, called the “playground”, and nodes cannot move outside the playground. The dimensions of the playground for each experiment are provided in their related parameter table.

After setting up the ad hoc network, the simulator creates the necessary objects, communication channels, tables, timers and messages. After all such objects and modules are created and set-up, our simulation begins and the simulation time begins to advance. We do not create any data messages in the first 60 seconds of the simulation in order to allow the network to settle. After this initial 60 seconds, the *FlowController* module assigns real-time flows to source nodes and any necessary actions required for the successful delivery of data packets to their destinations are taken by the protocol. We would like to note here that the topology dissemination and link cost measurement mechanisms are active during the whole duration of the simulation, including the initial 60 seconds.

Table 6.7 provides information on the traffic parameters used in the QoS experiments.

Parameter Name	Value
Number of Flows	25
Packet Payload Length	[40,600] bytes
Packet Creation Period	[5,50] ms
Packet Creation Rate	[20,200] packets/sec
Flow Length	[1000,4000] packets
Flow Duration	[5,200] sec

Table 6.7: Simulation parameters for data traffic in QoS operation mode experiments.

The meaning and usage of these parameters are as given in Section 6.1. We use the *random waypoint mobility model* to simulate the mobility of nodes. Node speeds and waiting times corresponding to the mobility rates used in the experiments are as given in Table 6.4.

For each experiment in this section, we have used the table sizes and timer periods provided in Table 6.8.

Parameter Name	Value
Route Cache Timer Period	5 sec
Edge Cost Timer Period	45 sec
wholeK	4
LSTable Size	15
RTFlowsTable Size	20

Table 6.8: Miscellaneous simulation parameters for QoS operation mode experiments.

We clear the route cache at a node periodically according to the "route cache timer". This is done to allow the local path finding algorithm to be activated regularly in order to find potentially better paths than the ones currently used by the node. The "edge cost timer" is used to clear stale edge costs from the local graph representations of nodes. Parameter *wholeK* controls the incremental message scheme used in topology dissemination. *LSTable* is the table that keeps track of the last recently seen link-state messages, used to prevent a node from participating more than once in a network-wide broadcast. *RTFlowsTable* is essentially used for the same purpose, preventing a node from participating more than once in a limited cost information broadcast.

We run each experiment for a total duration of 11 minutes, where the first initial minute is used to improve the accuracy of our results.

Results on protocol overhead of Elessar in QoS operation mode are presented in Section 6.2.1 and results on real-time support performance are given in Section 6.2.2.

6.2.1 Results on Protocol Overhead in QoS Operation Mode

In this section, we provide experimental results on the overhead of our protocol, both due to the cost dissemination mechanism and due to the topology dissemination mechanism.

The overhead of the cost dissemination mechanism is dependent on various mechanism parameters. Depending on whether the periodic or non-periodic cost sending scheme is used, parameters affecting cost overhead are different. The primary parameter affecting cost overhead in the periodic scheme is the period of the cost messages. The number of cost messages (cost count) is the equivalent parameter for the non-periodic case. The cost overhead of our protocol is also dependent on the number of nodes sending their costs, which is largely determined by the hop count imposed on the cost information messages (CostInform hop count). We now look into the effect of these protocol parameters on protocol overhead.

Figure 6.35 presents the cost overhead ratio in bytes[8] vs. cost count for the non-periodic case. We can see from the figure that as exptected, when the cost count increases, cost overhead increases. We also observe that our cost overhead is very low, accounting for 0.25% overhead even in the highest case.

Figure 6.36 presents results on cost overhead as cost period changes in the periodic cost scheme. The figure portrays that lower periods incur higher overhead due to the larger number of created cost messages. Even so, we see that the cost overhead is as low as 0.13% even for the lowest cost period case.

As the hop count imposed on messages of type *CostInformMsg* increases, we generally expect the cost overhead to increase due to higher number of nodes informed of the real-time flow and sending their costs. This behavior may be observed in Figure 6.37. We again observe very low cost overheads, where overheads change between values of 0.06% and 0.08%.

[8]We calculate the cost overhead ratio as follows. Let $genSpecialRT$, $fwdSpecialRT$, $genCost$, $fwdCost$, $genCostInform$, $fwdCostInform$ denote the number of bytes of total generated and forwarded packets of type $SpecialRT$, $CostMsg$, and $CostInformMsg$, respectively, and let $genDataRT$, $fwdDataRT$, $genDataNormal$, $fwdDataNormal$ denote the total number of bytes of generated and forwarded packets of type $DataRT$ and $DataNormal$. The cost overhead ratio in bytes is then equal to $(genSpecialRT + fwdSpecialRT + genCost + fwdCost + genCostInform + fwdCostInform)/(genDataRT + fwdDataRT + genDataNormal + fwdDataNormal)$.

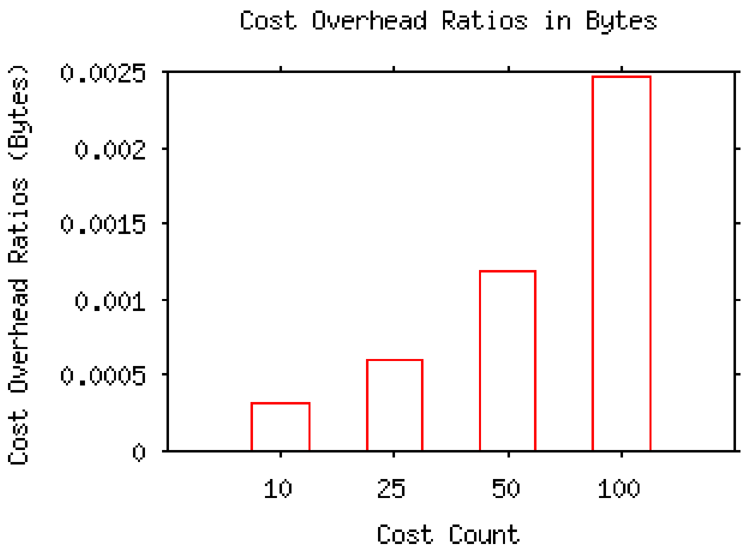

Figure 6.35: Cost overhead vs. cost count, non-periodic case.

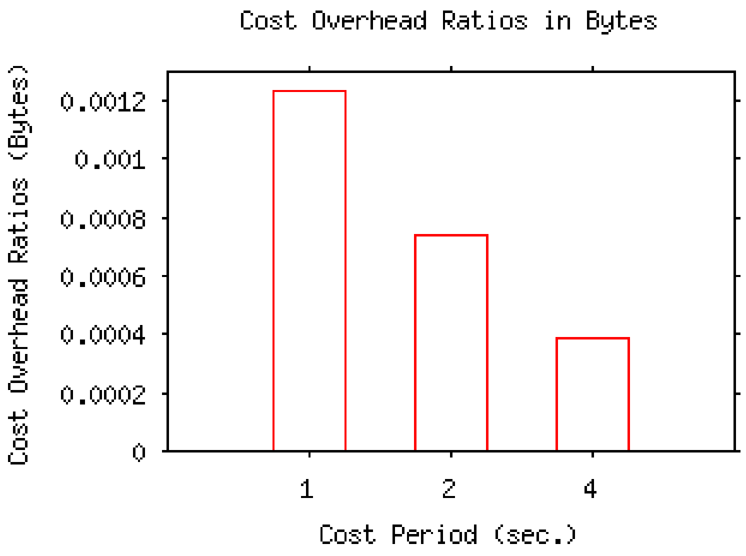

Figure 6.36: Cost overhead vs. cost period, periodic case.

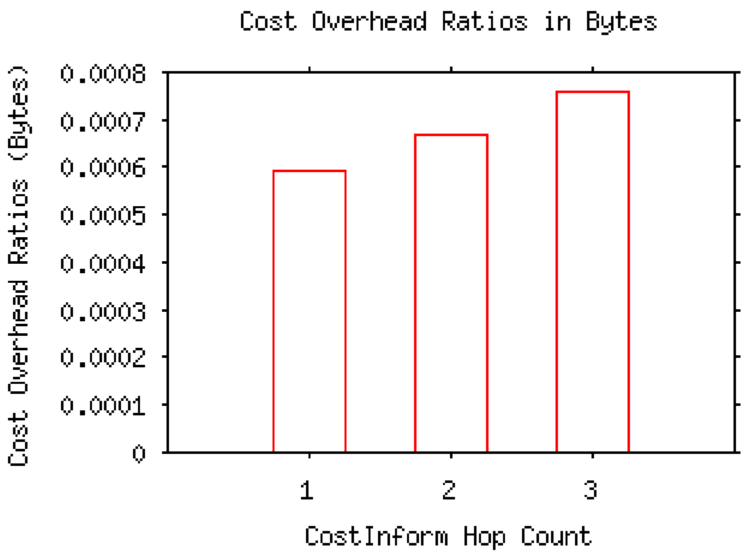

Figure 6.37: Cost overhead vs. cost inform hop count.

In Figure 6.38, we observe that as the mobility rate increases, the cost overhead increases. When a previously used route breaks and a new route is established between the source and the destination, the new nodes that now relay real-time packets also send their costs to the source to allow more optimal path selection at the source. As the mobility rate in the network increases, route breakages increase, and due to this cost sending behavior of “new” nodes on the new paths, our cost overhead increases. However, we can see from the figure that our cost overhead is still at a low of 0.075% even for the highest mobility rate.

Figures 6.39 and 6.40 show the behavior of our protocol’s cost overhead with increasing network size and node density, respectively. We can see from these figures that as the network size and the node density increase, our cost overhead increases due to the increased number of nodes in the network, and in parallel, the increased number of cost sending nodes.

Figure 6.41 shows results on the total overhead vs. cost count, for the non-periodic case. It can be inferred from the figure that as the cost count increases, the ratio of the cost overhead to the total overhead increases significantly. However, we observe that the topology dissemination overhead forms the greatest portion of our total protocol overhead. This phenomenon may also be observed

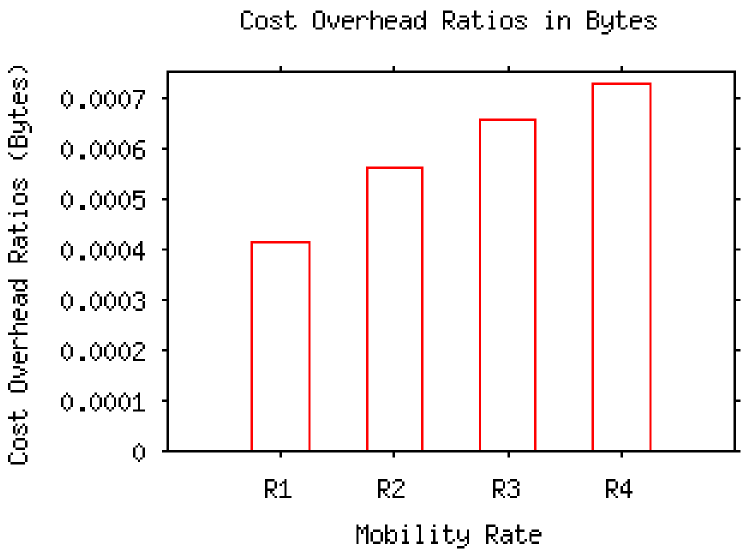

Figure 6.38: Cost overhead vs. mobility rate.

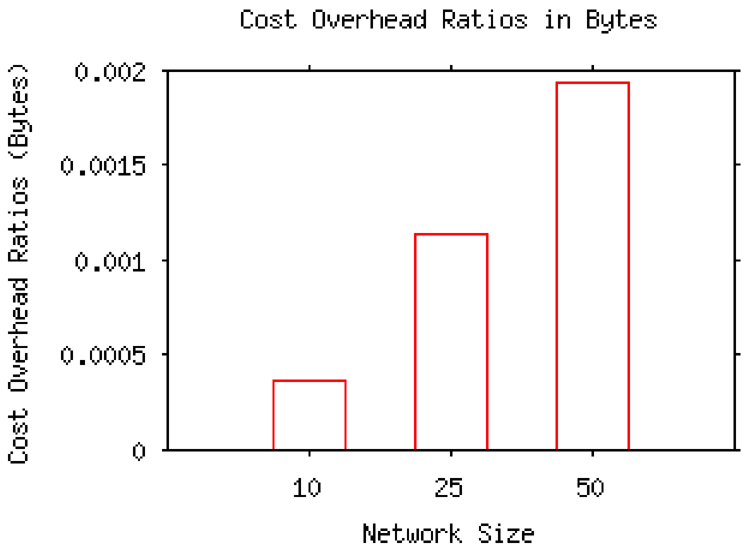

Figure 6.39: Cost overhead vs. network size.

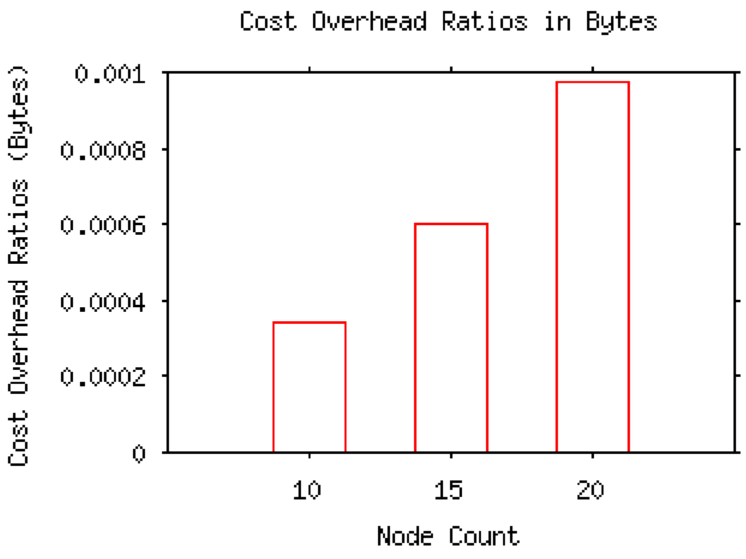

Figure 6.40: Cost overhead vs. node density.

in figures 6.42, 6.43, 6.44, 6.45, and 6.46. For the figures regarding changes in the cost dissemination parameters, we see that the total overhead is at most 0.8%. Of course, as we increase the mobility rate, network size or the node density in the network, our total overhead increases to values of 4% or more, as such parameters influence the overhead of the link-state mechanism more directly.

Table 6.9 provides the simulation parameters used for the experiments presented in this section.

From the experimental results presented above, we observe that our protocol overhead due to the cost dissemination mechanism does not form a significant portion of the total overhead of our protocol and that the overhead due to the link-state dissemination mechanism dominates the overhead of Elessar in almost all cases.

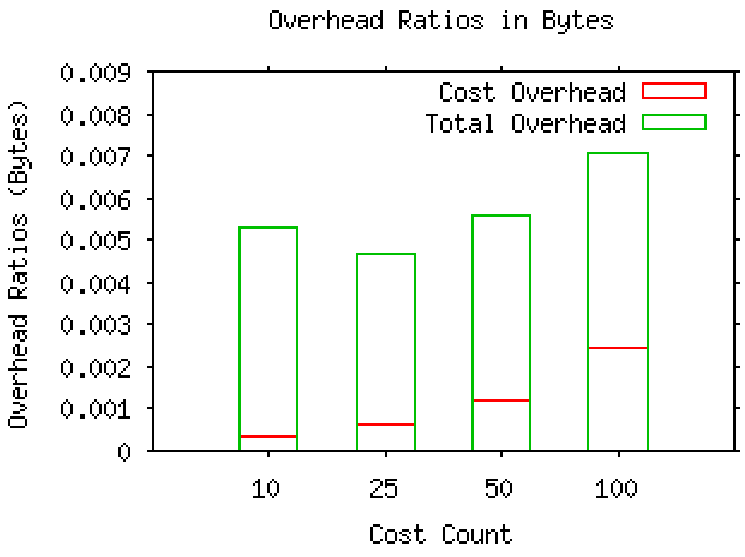

Figure 6.41: Total overhead vs. cost count, non-periodic case.

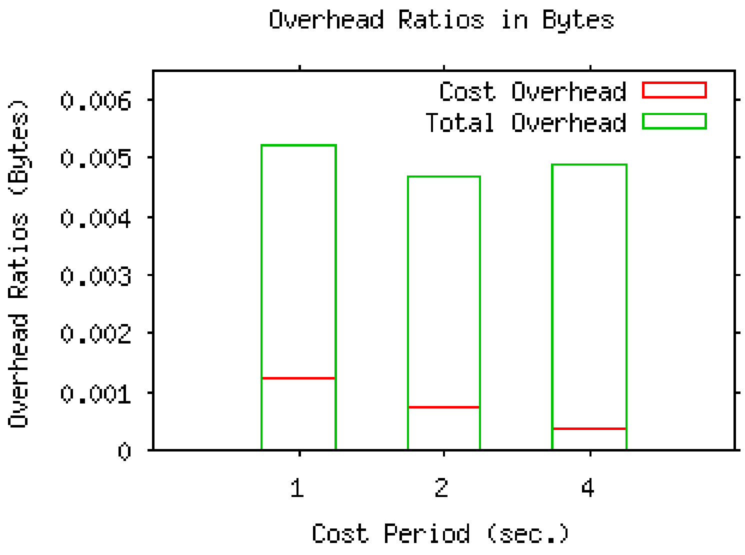

Figure 6.42: Total overhead vs. cost period, periodic case.

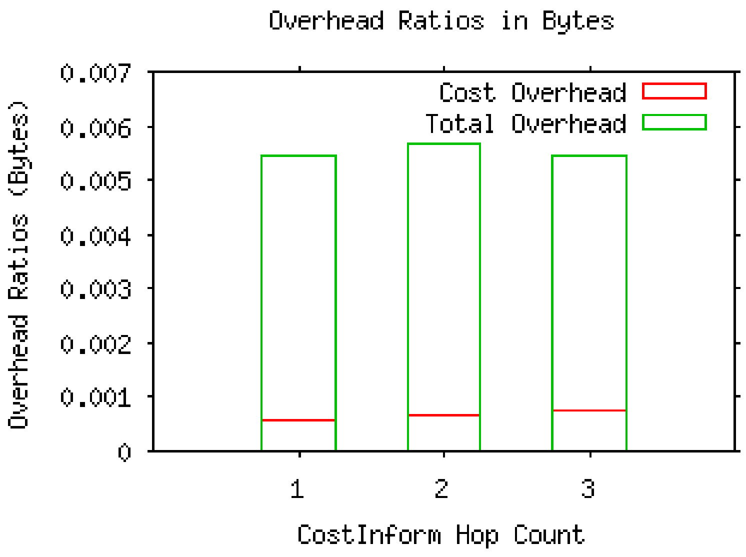

Figure 6.43: Total overhead vs. cost inform hop count.

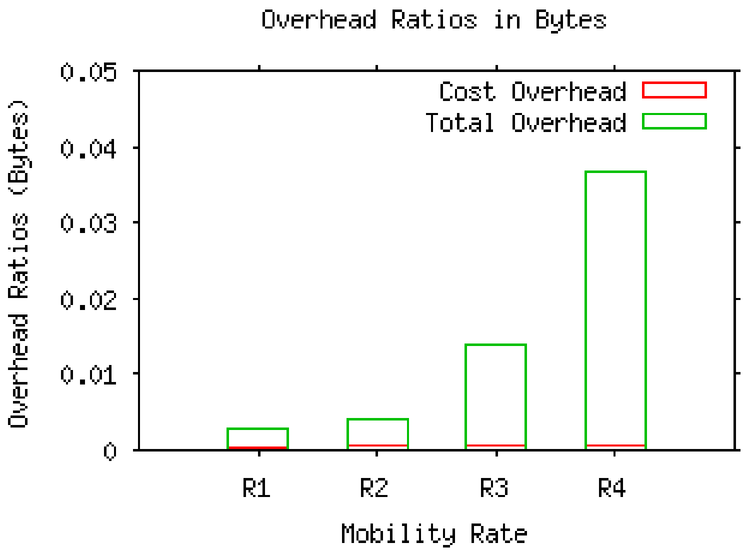

Figure 6.44: Total overhead vs. mobility rate.

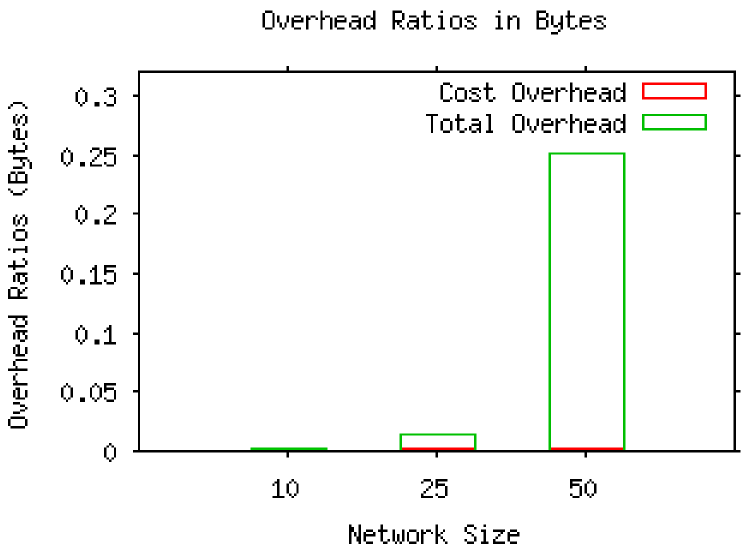

Figure 6.45: Total overhead vs. network size.

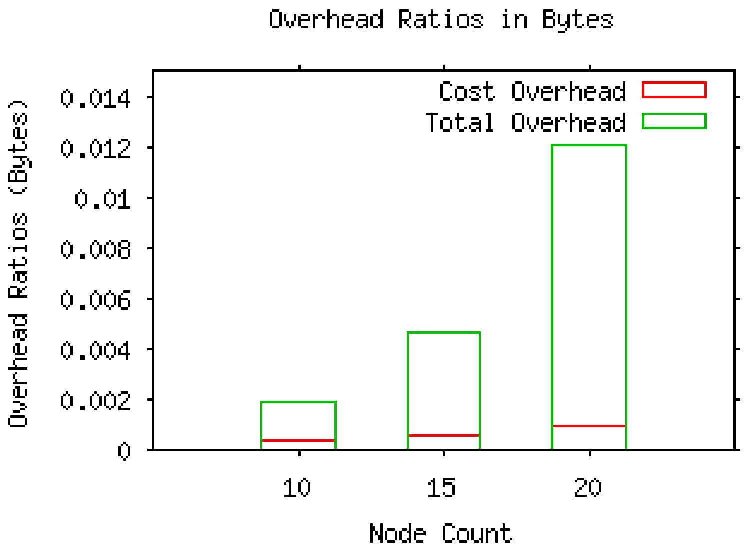

Figure 6.46: Total overhead vs. node density.

Parameter Name / Figure Number	Node Count	Playground Size	Mobility Rate	Hop Count	Cost Period	Cost Count
Figures 6.35 and 6.41	15	115 × 115	Rate 2	2	not used	varies
Figures 6.36 and 6.42	15	115 × 115	Rate 2	2	varies	not used
Figures 6.37 and 6.43	15	115 × 115	Rate 2	varies	not used	25
Figures 6.38 and 6.44	15	115 × 115	varies	2	not used	25
Figures 6.39 and 6.45	varies	varies	Rate 2	2	not used	25
Figures 6.40 and 6.46	varies	115 × 115	Rate 2	2	not used	25

Table 6.9: Simulation parameters for overhead experiments in QoS mode operation.

6.2.2 Results on Protocol Performance in QoS Operation Mode

In this section, we present results on the performance of Elessar for real-time support for different traffic types. Sections 6.2.2.1, 6.2.2.2, and 6.2.2.3 present results for delay-sensitive, loss-sensitive, and bandwidth-sensitive traffic, respectively. We would like to note here that we try to minimize the end-to-end delay for delay-sensitive traffic, to minimize the end-to-end loss rate (or equivalently the end-to-end path availability) for loss-sensitive traffic, and to maximize the end-to-end minimum available bandwidth for bandwidth-sensitive traffic.

6.2.2.1 Results on Protocol Performance in QoS Mode: Delay-Sensitive Traffic

Figure 6.47 shows how the number of cost messages sent during the lifetime of a flow affects protocol performance in the non-periodic cost scheme. We see that as the cost count increases, the average end-to-end packet delay[9] decreases, since the source nodes are able to find more optimal paths with increased frequency, with increasing cost count. We also observe that with QoS routing, the average end-to-end packet delay is always lower than without QoS routing, where we lower the average delay from around 180 ms to values between 95 ms and 165 ms. We can thus observe that our QoS routing scheme is successful in lowering the average end-to-end packet delay compared with the non-QoS case.

Figure 6.48 presents average end-to-end packet latency vs. cost period, for the periodic cost dissemination case. We would expect an inverse relationship between cost period and average latency, since with higher cost periods, sources will get a lesser number of cost messages, with longer delays in between, and this will lower the accuracy with which sources are able to find optimal paths. We can see that our expectation is met in Figure 6.48. We can also see from the figure that average end-to-end latency with QoS routing is lower than average end-to-end latency without QoS routing, where our QoS routing scheme is able

[9]We calculate the end-to-end delay of data packets only.

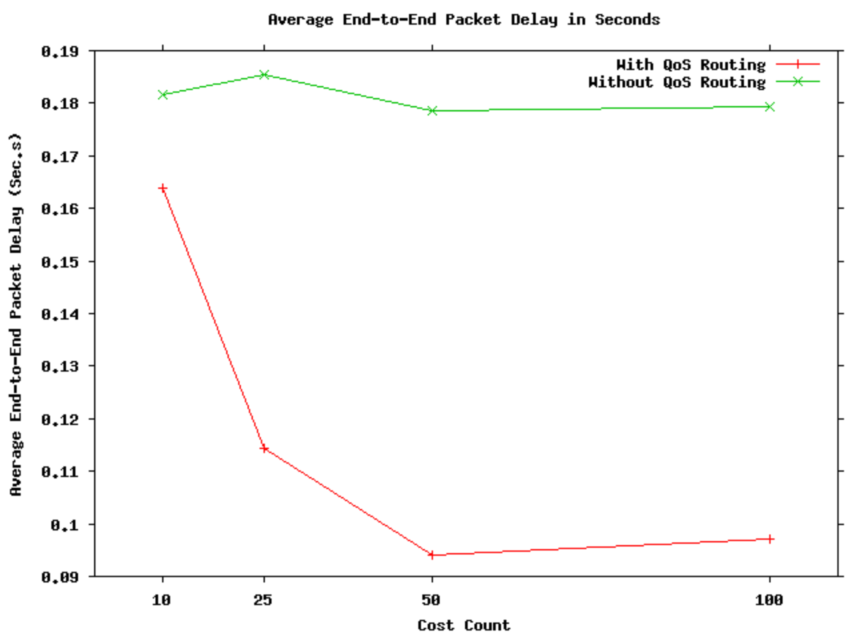

Figure 6.47: Average end-to-end delay vs. cost count.

to lower the latency from a value of around 180 ms to values between 90 ms and 140 ms.

As the hop count imposed on cost information messages is increased, a greater number of nodes in the network are informed of a real-time flow, and the source nodes receive cost messages from a larger percentage of the network. With increasing percentage of the network a source node has information on, it is potentially able to make smarter decisions during path finding, resulting in more optimal paths. We can observe the effect of changing the hop count on average end-to-end packet latency in Figure 6.49, where we see the expected behavior. We can also see that with QoS routing, average end-to-end packet delays are in between 100 ms and 150 ms, whereas average delays without QoS routing are around 180 ms.

Figure 6.50 shows the effect of increasing network size on the performance of our protocol, and we see that with increasing network size, average end-to-end packet delay decreases, both with and without QoS routing. This is due to the availability of more paths between any two nodes in a larger network, and with a larger amount of paths to choose from, our protocol is able to find more optimal

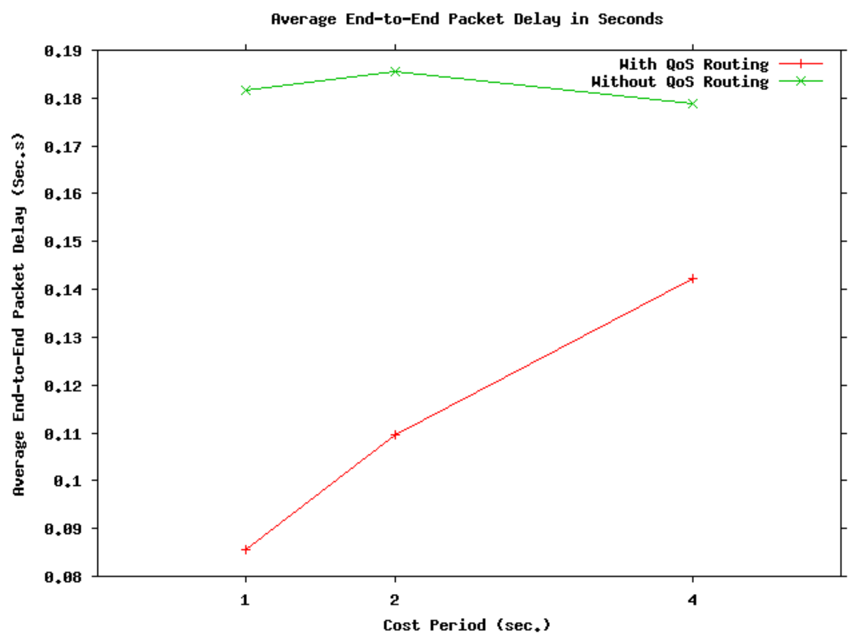

Figure 6.48: Average end-to-end delay vs. cost period.

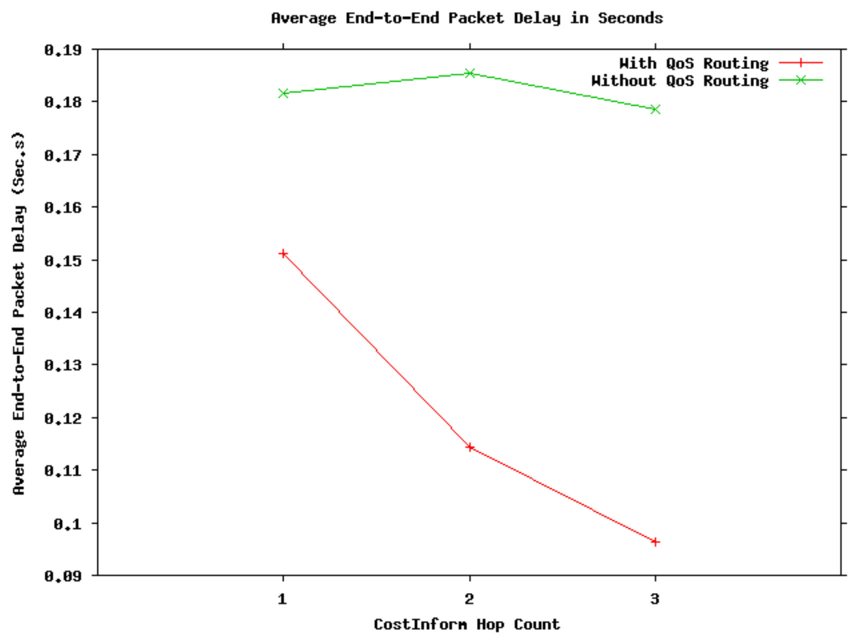

Figure 6.49: Average end-to-end delay vs. cost inform hop count.

paths. We also employ route maintenance in our protocol, so when a route breaks, with a greater number paths between nodes, there is a higher probability that an intermediate node will be able to find an alternate path. We again observe that our QoS routing scheme incurs lower end-to-end delays than our non-QoS routing scheme.

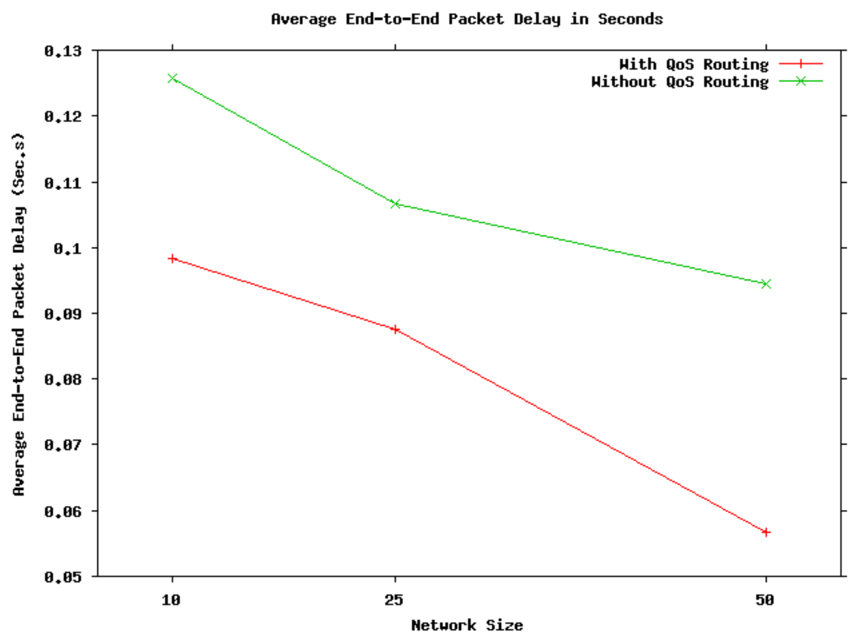

Figure 6.50: Average end-to-end delay vs. network size.

We observe a direct relationship between node density and average end-to-end latency, as with increasing node density, interference between nodes increase, also increasing link latencies. This effect is observed with both QoS and non-QoS routing schemes, where the QoS routing scheme performs better than the non-QoS routing scheme. These results may be observed in Figure 6.51.

Figures 6.52, 6.53 and 6.54 present results on the ratio of packets with QoS requirements satisfied[10] for different QoS requirements. In Figure 6.52, we observe that as the cost inform hop count increases, the number of packets with QoS requirements satisfied increases, since average end-to-end latency decreases with increasing hop count. We also observe that as we relax the end-to-end latency

[10]We calculate the ratio of packets with QoS satisfied as the number of received packets with requirements satisfied to the number of all received packets.

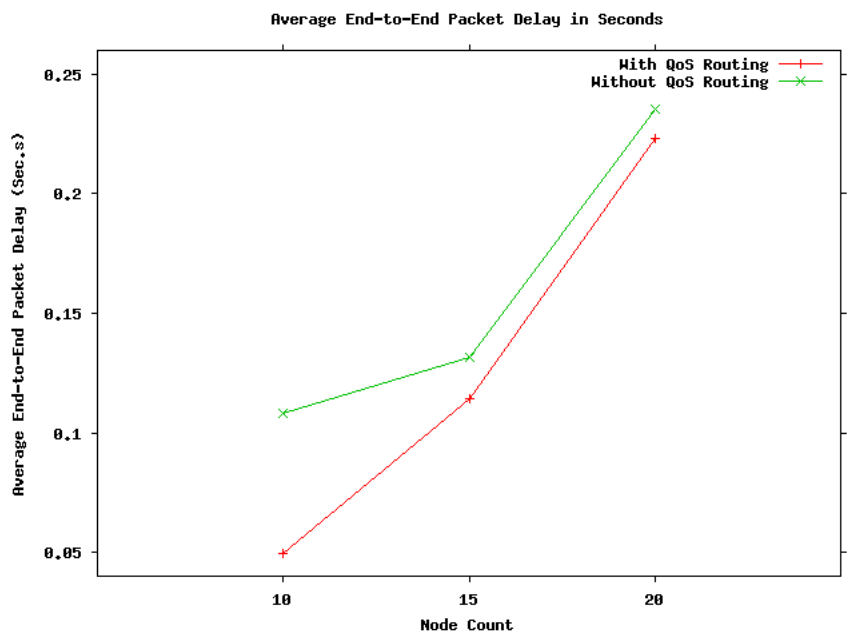

Figure 6.51: Average end-to-end delay vs. node density.

requirement on packets, a greater percentage of packets meet their requirements, a natural result. We can see from the figure that our QoS satisfaction ratio is between 76% and 91%.

Figure 6.53 presents results on QoS satisfaction ratio vs. node density and we can see from the figure that there is an inverse relationship between the two. With increasing node density, average end-to-end delay increases, therefore the number of packets that meet their requirements decrease. We observe that our QoS satisfaction ratio changes between 76% and 97% for different QoS requirements.

Figure 6.54 presents QoS satisfaction ratios for different simulation parameters. You may find the parameters used in all of the figures in this section in Table 6.10.

We can see from Figure 6.54 that our QoS satisfaction ratios vary between 81% and 94%, which are quite high values, showing that our protocol is able to support delay-sensitive traffic satisfactorily.

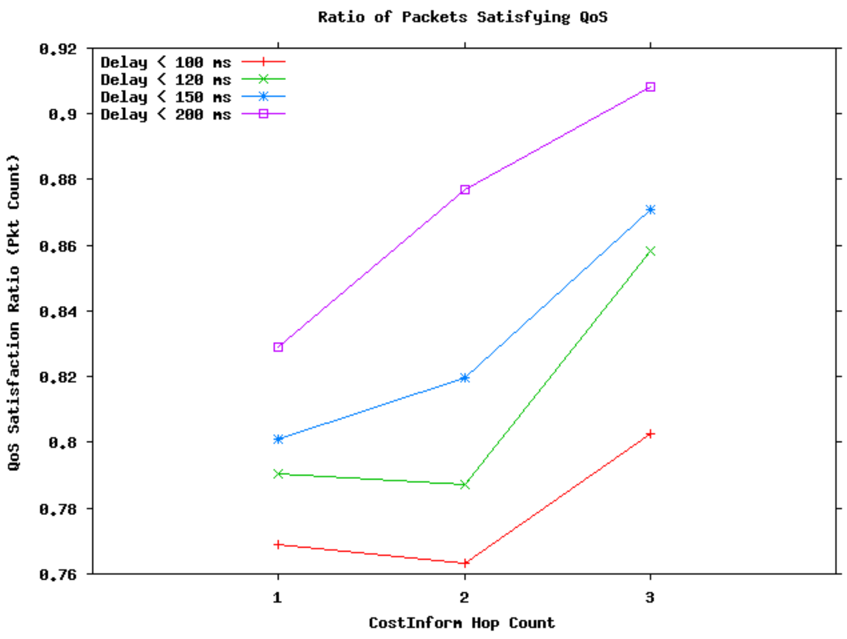

Figure 6.52: QoS satisfaction ratio vs. cost inform hop count for delay-sensitive traffic.

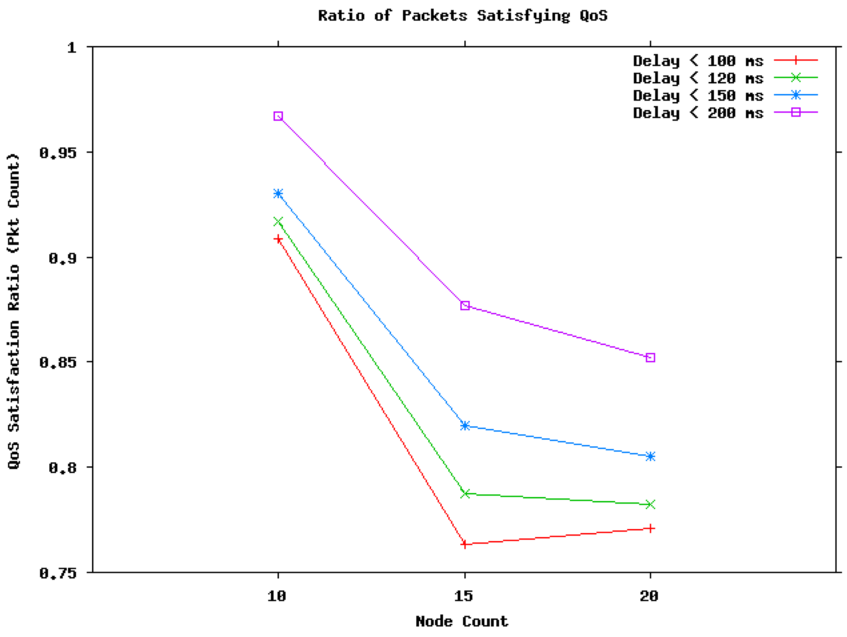

Figure 6.53: QoS satisfaction ratio vs. node density for delay-sensitive traffic.

Parameter Name / Figure Number	Node Count	Playground Size	Mobility Rate	Hop Count	Cost Period	Cost Count
Figure 6.47	15	115×115	Rate 2	2	not used	varies
Figure 6.48	15	115×115	Rate 2	2	varies	not used
Figures 6.49 and 6.52	15	115×115	Rate 2	varies	not used	25
Figure 6.50	varies	varies	Rate 2	2	not used	25
Figures 6.51 and 6.53	varies	115×115	Rate 2	2	not used	25
Figure 6.54, Exp. 1	15	115×115	Rate 2	2	not used	50
Figure 6.54, Exp. 2	15	115×115	Rate 3	2	not used	25
Figure 6.54, Exp. 3	50	225×225	Rate 2	2	not used	25
Figure 6.54, Exp. 4	15	115×115	Rate 2	2	1 sec.	not used

Table 6.10: Simulation parameters for delay-sensitive traffic experiments in QoS mode operation.

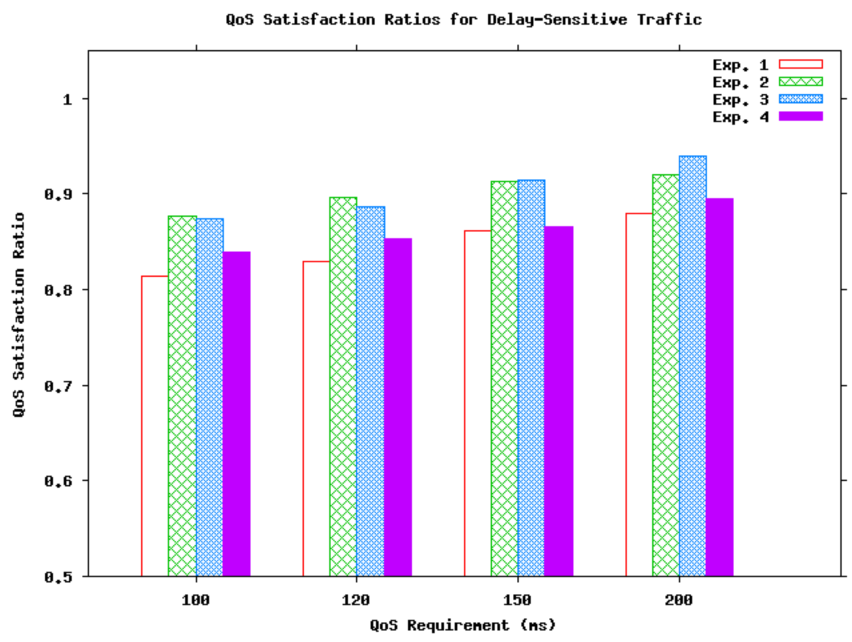

Figure 6.54: QoS satisfaction ratios for delay-sensitive traffic for different simulation parameters.

6.2.2.2 Results on Protocol Performance in QoS Mode: Loss-Sensitive Traffic

Figure 6.55 shows end-to-end packet loss rate vs. cost count, for the non-periodic cost dissemination scheme. We see from the figure that as cost count increases, average end-to-end loss rate decreases. One thing to note in this figure is that we have lower end-to-end packet loss rates with QoS routing for all cases, except the case where cost count = 10. We understand from these results that our protocol is not able to lower end-to-end loss rates with a small number of cost messages, but it rather requires the cost count to be $>$ 15 for effective support of loss-sensitive traffic. We also observe that end-to-end loss rates are between 23% and 28% with QoS routing and around 26.5% without QoS routing. At first glance, these numbers may seem high, but this is because we are using the accumulated end-to-end loss rates of links that packets have traversed and not the averaged loss rates along these links. Please refer to Section 5.3.3.1.2 for more information on how our protocol calculates end-to-end loss rates.

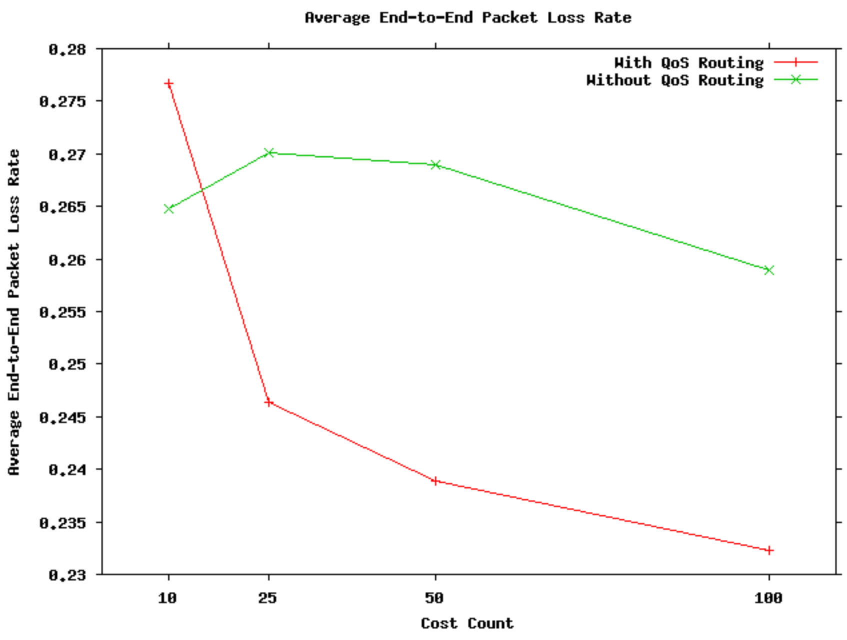

Figure 6.55: Average end-to-end loss rate vs. cost count.

Figure 6.56 presents results on packet loss rates for changing cost periods when the periodic cost dissemination scheme is employed. We would expect an inverse relationship between the cost period and loss rate, and we can observe this behavior in the presented graph. One interesting thing to note here is that our QoS routing scheme provides paths with lower end-to-end loss rates for almost all periods, with the exception of period = 4 sec. We can conjure from this fact that like the case of cost count in the non-periodic cost scheme, our protocol requires a certain period with the periodic scheme in order to be effective in supporting loss-sensitive traffic. This period seems to be around 3 sec. in the presented graph.

We can see the effect of the hop count imposed on cost information messages on loss rates in Figure 6.57. We observe that as the hop count increases, loss rate decreases, as would be expected. We also observe that our QoS routing scheme has lowered the end-to-end loss rates from around 26.8% to values between 26.2% and 25.4%, compared with the non-QoS routing scheme.

Figure 6.58 draws end-to-end loss rate vs. node density. From the figure, we see that as the node density increases, the loss rate increases, both with and

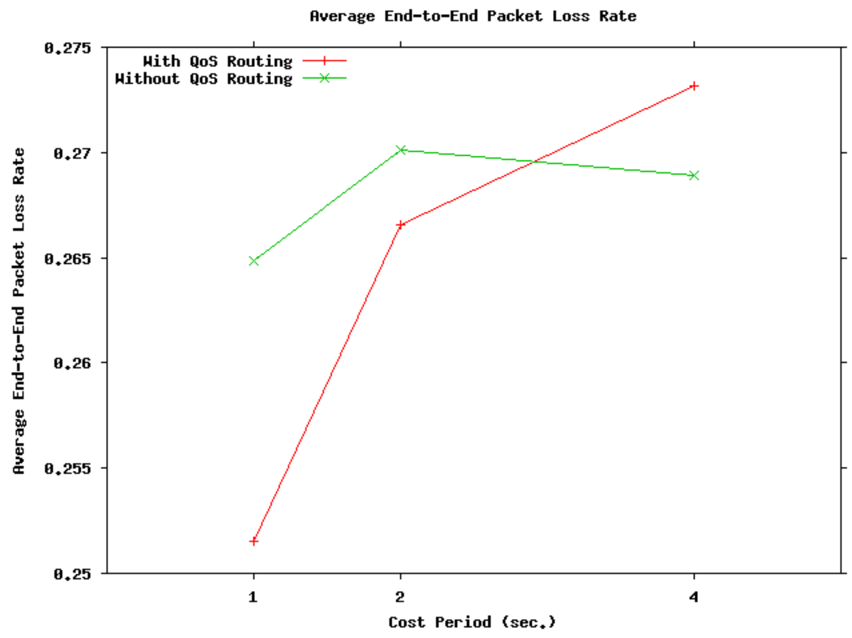

Figure 6.56: Average end-to-end loss rate vs. cost period.

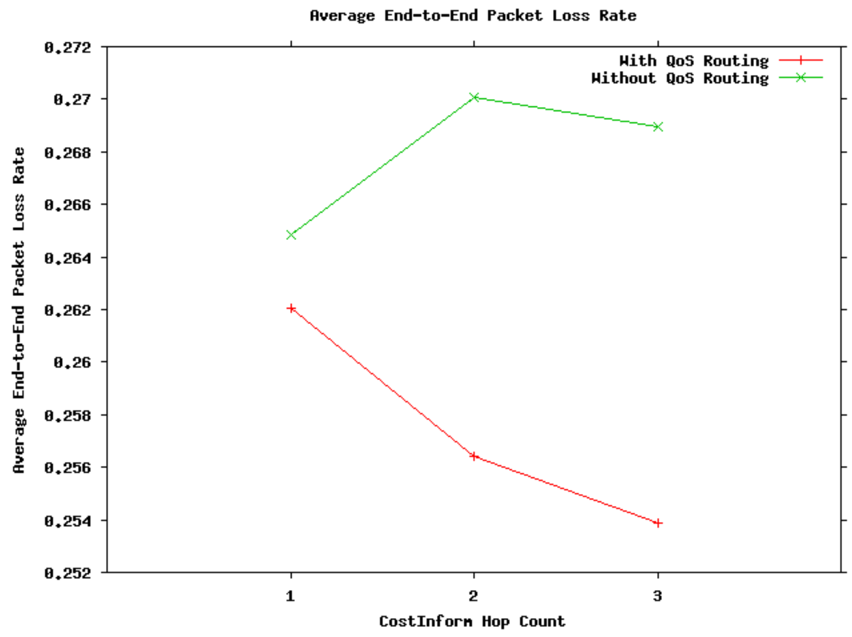

Figure 6.57: Average end-to-end loss rate vs. cost inform hop count.

without QoS routing. The reason for this is that with increasing node density, interference between nodes and collisions during transmissions increase, causing more packets to be lost. Our QoS routing scheme improves end-to-end loss rates by 1% in general, when compared with the non-QoS scheme.

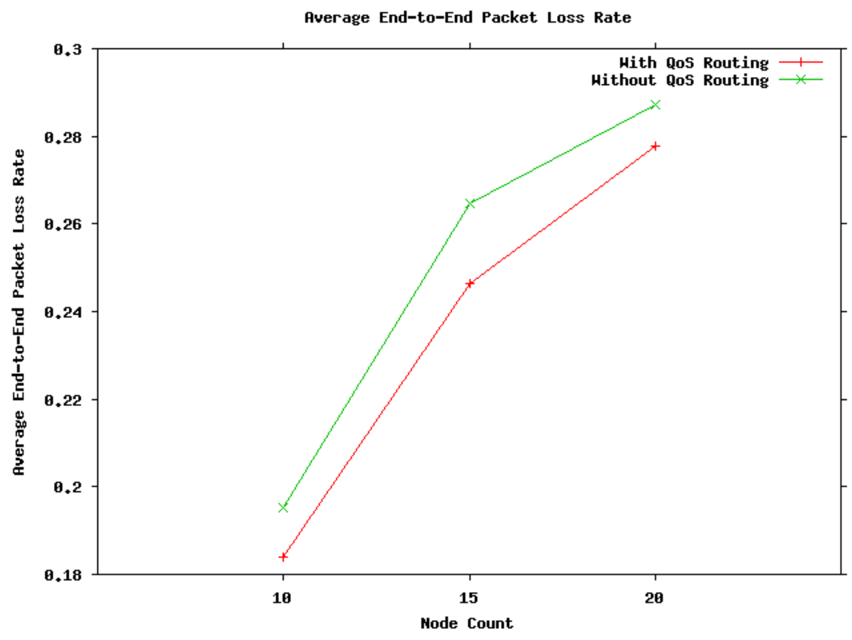

Figure 6.58: Average end-to-end loss rate vs. node density.

Figures 6.59, 6.60, 6.61, and 6.62 present QoS satisfaction ratios for loss-sensitive data flows, for various loss rate requirements. We see that our QoS satisfaction ratios are high, ranging between 65% and 93%. When we look at the loss rate requirements, we see that all requirements are lower, and therefore more strict, than the average end-to-end loss rates observed with QoS routing. Even when this is the case, we observe high QoS satisfaction ratios. We can infer from this fact that there is high deviation in our loss rate samples, where we have a high number of samples with low-to-medium loss rates, and a small number of samples with very high loss rates.

These figures on QoS satisfaction ratios present behaviors that are parallel to the observed behaviors in average end-to-end loss rate graphs.

Table 6.11 presents the simulation parameters used in QoS experiments for loss-sensitive traffic.

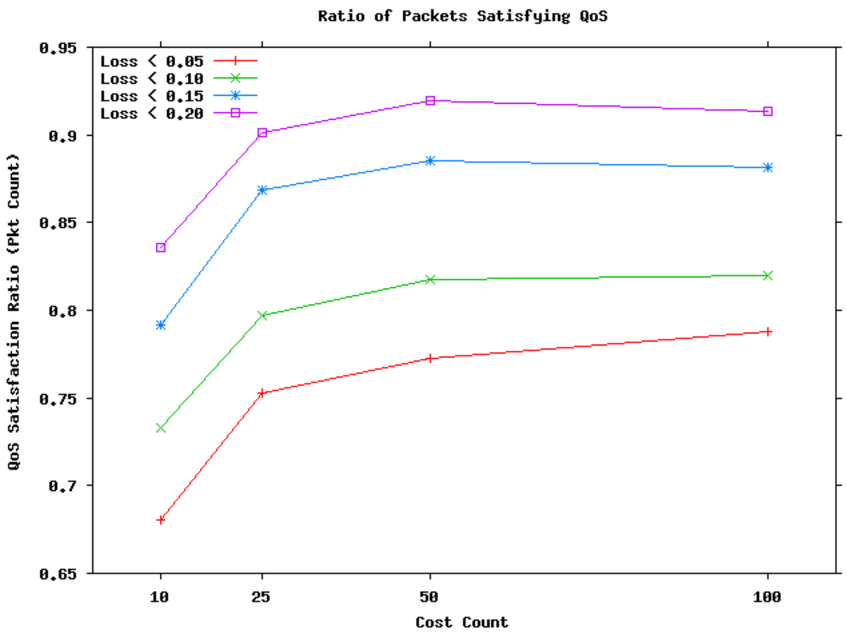

Figure 6.59: QoS satisfaction ratio vs. cost count.

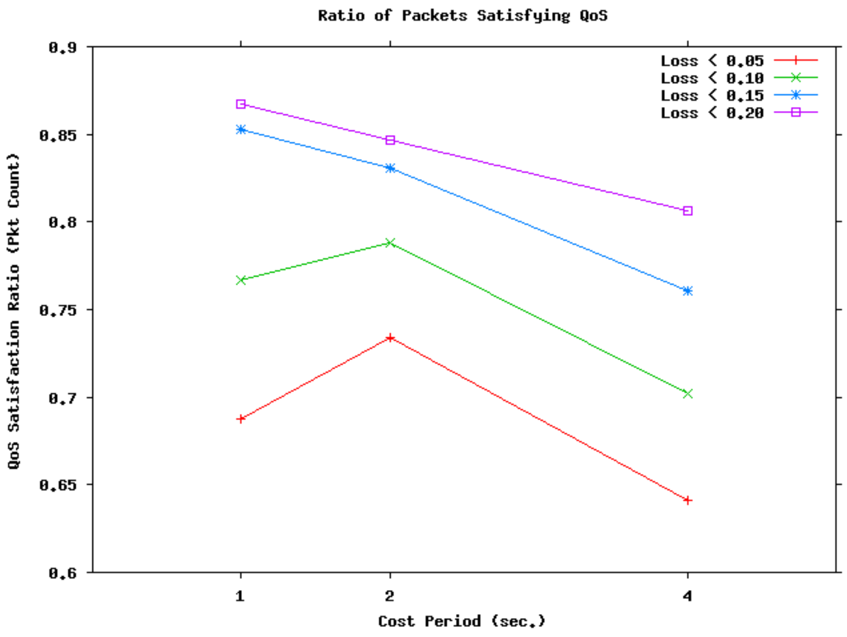

Figure 6.60: QoS satisfaction ratio vs. cost period.

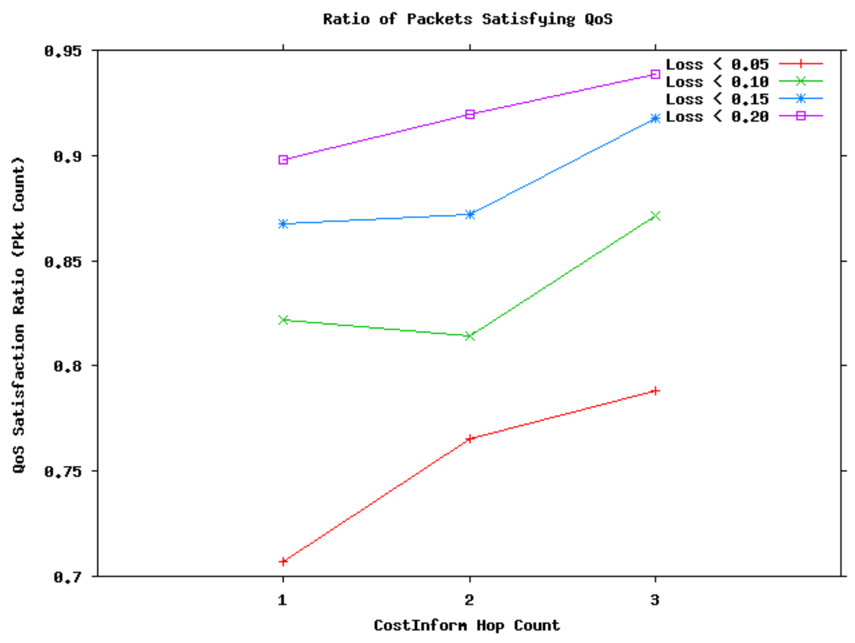

Figure 6.61: QoS satisfaction ratio vs. cost inform hop count.

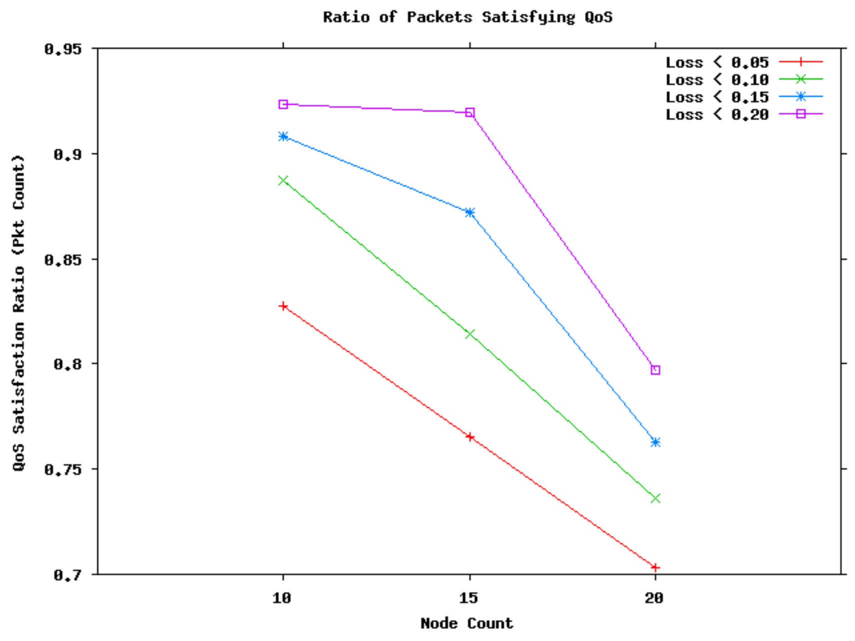

Figure 6.62: QoS satisfaction ratio vs. node density.

Parameter Name / Figure Number	Node Count	Playground Size	Mobility Rate	Hop Count	Cost Period	Cost Count
Figure 6.55 and 6.59	15	115×115	Rate 2	2	not used	varies
Figure 6.56 and 6.60	15	115×115	Rate 2	2	varies	not used
Figures 6.57 and 6.61	15	115×115	Rate 2	varies	not used	25
Figures 6.58 and 6.62	varies	115×115	Rate 2	2	not used	25

Table 6.11: Simulation parameters for loss-sensitive traffic experiments in QoS mode operation.

6.2.2.3 Results on Protocol Performance in QoS Mode: Bandwidth-Sensitive Traffic

Figure 6.63 presents max-min available bandwidth vs. cost count, for the non-periodic cost scheme. From the figure, we can see that as cost count increases, maxmin bandwidth also increases. Our QoS routing scheme finds paths with larger available bandwidth values for almost all cases, except the case count = 10. As was observed for loss-sensitive traffic, it seems that a low number of cost messages does not allow the route discovery mechanism to find better paths in the bandwidth-sensitive traffic case.

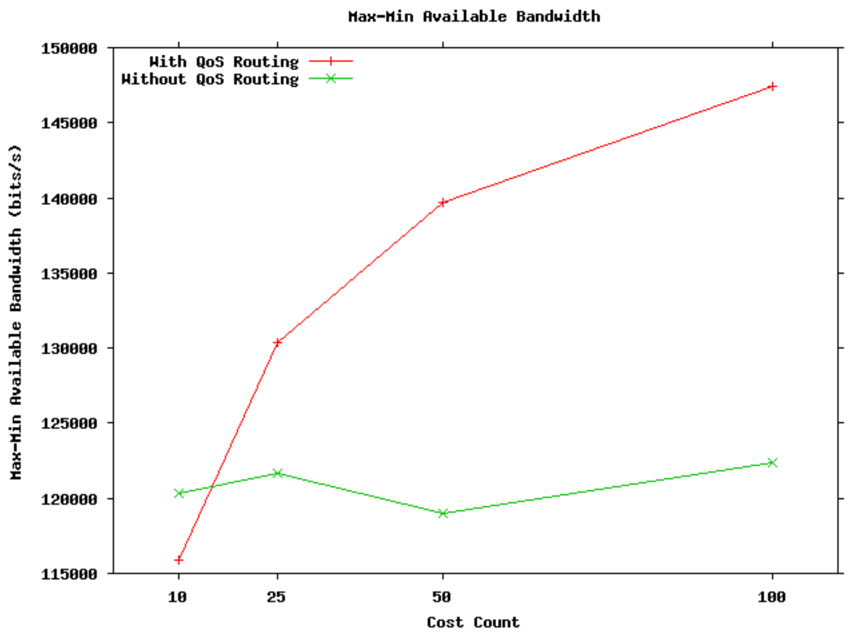

Figure 6.63: Max-Min available bandwidth vs. cost count.

Figure 6.64 plots max-min available bandwidth against cost period, for the periodic cost scheme. As the cost period increases, max-min available bandwidth decreases. We again observe that with a high cost period, Elessar is not able to support bandwidth-sensitive traffic satisfactorily.

Figures 6.65 and 6.66 show how max-min available bandwidth changes with changing cost inform hop count and node density, respectively.

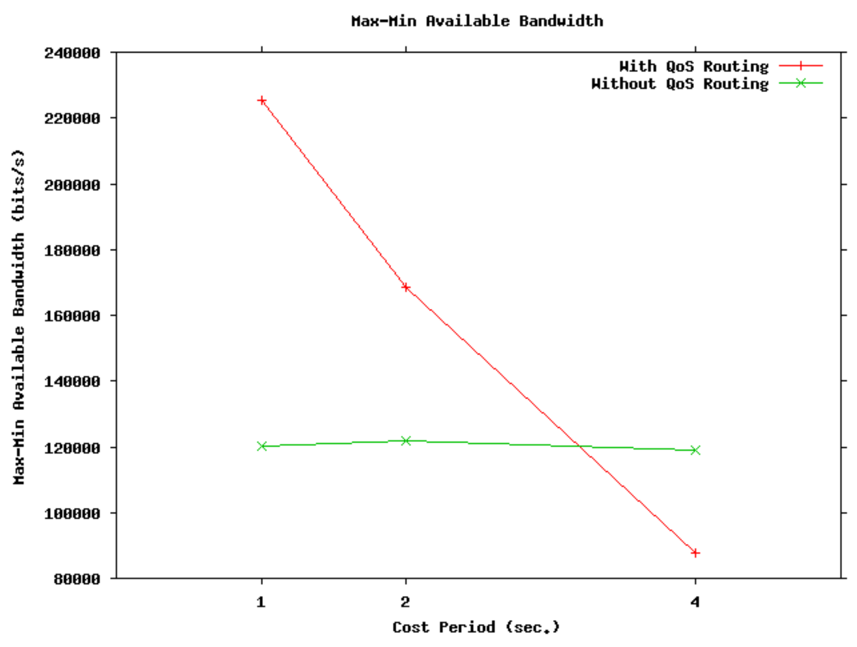

Figure 6.64: Max-Min available bandwidth vs. cost period.

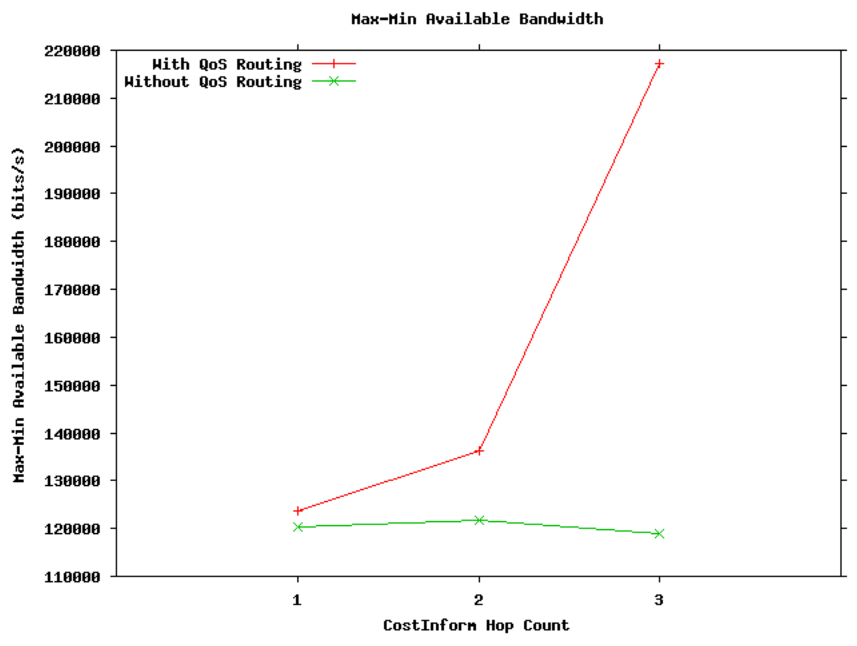

Figure 6.65: Max-Min available bandwidth vs. cost inform hop count.

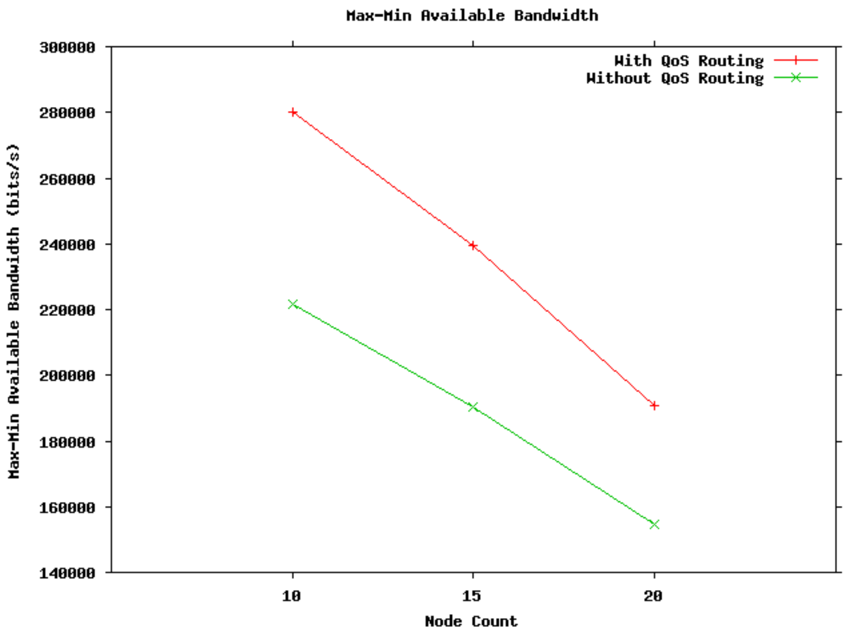

Figure 6.66: Max-Min available bandwidth vs. node density.

Table 6.12 presents the simulation parameters used in QoS experiments for bandwidth-sensitive traffic.

Parameter Name	Node Count	Playground Size	Mobility Rate	Hop Count	Cost Period	Cost Count
Figure Number						
Figure 6.63	15	115×115	Rate 2	2	not used	varies
Figure 6.64	15	115×115	Rate 2	2	varies	not used
Figures 6.65	15	115×115	Rate 2	varies	not used	25
Figures 6.66	varies	115×115	Rate 2	2	not used	25

Table 6.12: Simulation parameters for bandwidth-sensitive traffic experiments in QoS mode operation

Chapter 7

Conclusion and Future Work

In this thesis, we have presented Elessar, a link-state based on-demand routing protocol supporting real-time traffic in wireless mobile ad hoc networks. Instead of the more conventional periodic message dissemination, our protocol employs event-driven link-state updates, where topology changes (mostly due to node mobility) are the events of interest. We use an incremental message scheme during topology dissemination, where each k-th link-state message is a whole message, and other messages include incremental information regarding changes that have occurred since the last link-state message. Through the use of such link-state messages, which are globally broadcasted, each node in the network can construct its own view of the network and therefore run local algorithms on the local graph representation.

Elessar is able to support various different types of real-time traffic, along with normal data traffic. Such different types of traffic may exist concurrently in the network. We currently support bandwidth-, delay-, and loss-sensitive traffic types explicitly, while our protocol has the potential to accommodate other types of real-time traffic with ease. Of course, the ability of the protocol to support such types of traffic depend on the underlying link cost measurement mechanism.

Support for real-time traffic is based on an on-demand mechanism, which is initiated only when there are one or more real-time flows in the network. In light of this, we may say that Elessar operates reactively for real-time traffic and proactively for normal data traffic. We achieve real-time traffic support thorugh intelligent path selection at the source node, without using any resource reservations. We therefore provide only soft QoS guarantees. However, our protocol may easily incorporate a reservation mechanism that reserves resources along least-cost paths found by the route discovery mechanism of our protocol.

Our protocol is targeted towards small-to-medium sized wireless MANETs, having a diameter between 5 and 10. Our protocol supports dynamism resulting from node mobility and dynamic node joins and leaves and it is completely distributed, with no need for centralized components. Elessar is self-adapting to the current conditions in the network and provides QoS throughout the lifetime of a real-time flow, even in the face of node mobility.

From experimental results, we have observed that the protocol overhead of our protocol is very low for low mobility rates and acceptable for high mobility rates. The main overhead of Elessar is due to the link-state based topology dissemination mechanism, with the cost dissemination mechanism taking second place as a source of overhead. Since our link-state mechanism is event-driven, the mobility rate has a significant effect on our protocol overhead. The network size in terms of network diameter and average node degree also play an important role in affecting the protocol overhead due to the topology dissemination mechanism.

We have been able to decrease the protocol overhead in terms of bytes by the incremental message scheme. However, several optimizations may be employed in order to decrease link-state overhead even lower. One such optimization may be the use of multi-point relays for global broadcasting, instead of the straightforward, and potentially costly, flooding method currently employed by Elessar. Another scheme that may lower the link-state overhead is the hybrid use of periodic and event-driven approaches to update dissemination. When the event rate (i.e. the mobility rate) is high, periodic link-state messages may be used instead of event-driven messages. Of course, there are various issues with the use of this hybrid scheme, such as how to decide when to switch between the two approaches, how to set the period, etc. One final optimization may be the use of aggregated link-state messages, where a node creating and/or forwarding a link-state message waits a certain amount of time, and aggregates other link-state messages received during that time into a single message. All of these approaches seem viable and more work is definitely needed to see which ones are more effective for lowering link-state overhead. We leave the investigation of such optimizations as future work.

Our protocol operates with single-paths at the moment, but it may easily be adapted to work with multiple paths. We envision two different cases for the use of multipaths; during the information of nodes of a real-time flow, and during actual data transmission. Instead of using a single path to inform nodes of the start of a real-time flow, multiple paths may be used to inform nodes and therefore to receive link costs from such informed nodes. These multiple paths do not all need to be from the source to the destination, but may as well be from the source to some other node in the network. Through the use of such intelligent selection of multipaths during node informing, the source node may received more complete information on link costs without incurring excessive overhead, and with this higher level of information, it may be able to find better paths for real-time traffic flows.

The second case for the use of multipaths is during actual data transmission, where more than one path between source and destination is employed. Use of multipaths during data transmission may be done for load balancing issues, in order to increase the reliability of the transmission, or to support bandwidth-sensitive applications better, through a higher aggregate bandwidth of multiple paths. The use of multipaths in such scenarios seems to be very attractive, but care must be taken in the choice of the paths. Since wireless ad hoc networks operate in a broadcast medium, paths must be chosen so that transmissions and receptions of nodes do not interfere with each other. Due to each node's global knowledge of the network through Elessar's topology dissemination mechanism, such intelligent multipath selection may be performed with ease in our protocol by the use of local multipath algorithms. Our next step in the development of this protocol will be to look at multipath scenarios, to develop intelligent multipath selection algorithms, and to integrate such algorithms into our protocol.

During the discovery and selection of optimal paths for real-time flows under certain constraints, we currently find the best available path in the network and then check the flow's QoS requirement to see if the found path is satisfactory. Another approach may be to take into account the requirement of the flow during the path finding and selection phase. This second approach may be able to satisfy requirements of more flows in the presence of real-time flows with very heterogeneous constraints. To illustrate the idea more clearly, consider this case: there are two new flows in the network, F_1 and F_2, with requirements of 50 and 300, respectively. What these requirements represent is irrelevant at the moment. Let's assume that F_2 issues a route request before F_1, and our protocol finds a path of cost 25, which is the best path from source to destination at the moment, satisfying the requirement of F_2 since $25 < 300$. Continuing the example, F_2 also issues a route request, and our protocol is now able to find a path of cost 60, which is again the best path from source to destination at the moment, but now, the requirement of F_1 is not satisfied. However, if we take into account the requirements of flows *during* route finding, then we may make a more intelligent path assignment, where we assign the path with cost 60 to F_2 and the path with cost 25 to F_1, satisfying the requirements of both flows, even though the route request order is not changed. Of course, in order to accomplish such intelligent route finding, different algorithms have to be adopted as our local path finding algorithms in QoS mode, and we leave the discussion and investigation of such algorithms as future work.

Bibliography

[1] Tarek Abdelzaher, John Stankovic, Sang Son, Brian Blum, Tian He, Anthony Wood, and Chanyang Lu. A communication architecture and programming abstractions for real-time embedded sensor networks. In *Workshop on Data Distribution for Real-Time Systems (in conjunction with ICDCS 2003)*, May 2003.

[2] F. Sidi Abed, M. Gueroui, and J.P. Claude. Providing QoS in ad hoc networks with behaviours nodes algorithm (BNA). In *Proceedings of Computer, Science, and Technology*, May 2003.

[3] K. Akkaya, M. Younis, and M. Youssef. Efficient aggregation of delay-constrained data in wireless sensor networks. In *Proceedings of Internet Compatible QoS in Ad Hoc Wireless Networks*, 2005.

[4] Kemal Akkaya and Mohamed Younis. An energy-aware QoS routing protocol for wireless sensor networks. In *Proceedings of the IEEE Workshop on Mobile and Wireless Networks (MWN 2003)*, May 2003.

[5] Kemal Akkaya and Mohamed Younis. Energy-aware routing of time-constrained traffic in wireless sensor networks. *International Journal of Communication Systems, Special Issue on Service Differentiation and QoS in Ad Hoc Networks*, 17(6):663–687, 2004.

[6] Kemal Akkaya and Mohamed Younis. Energy-aware and QoS routing in wireless sensor networks. *Springer Cluster Computing Journal*, 8:179–188, 2005.

[7] Kemal Akkaya and Mohamed Younis. A survey on routing protocols for wireless sensor networks. *Elsevier Ad Hoc Network Journal*, 3(3):325–349, 2005.

[8] Ian F. Akyildiz, WellJan Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–114, August 2002.

[9] Ian F. Akyildiz, WellJan Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: A survey. *Computer Networks (Elsevier) Journal*, 38(4):393–422, March 2002.

[10] Ian F. Akyildiz and Xudong Wang. A survey on wireless mesh networks. *IEEE Communications Magazine*, 43(9):S23–S30, September 2005.

[11] Ian F. Akyildiz, Xudong Wang, and Weilin Wang. Wireless mesh networks: A survey. *Computer Networks (Elsevier) Journal*, 47:445–487, March 2005.

[12] J. N. Al-Karaki and A.E. Kamal. Routing techniques in wireless sensor networks: A survey. *IEEE Wireless Communications*, 11(6):6–28, 2004.

[13] David G. Andersen, Alex C. Snoeren, and Hari Balakrishnan. Best-path vs. multi-path overlay routing. In *Internet Measurement Conference*, October 2003.

[14] Hakim Badis and Khaldoun Al Agha. A distributed algorithm for multiple-metric link state QoS routing problem. In *International Workshop On Mobile and Wireless Communications Networks*, October 2003.

[15] Hakim Badis and Khaldoun Al Agha. QOLSR multi-path routing for mobile ad hoc networks based on multiple metrics. In *Vehicular Technology Conference*, May 2004.

[16] Hakim Badis and Khaldoun Al Agha. QOLSR, QoS routing for ad hoc wireless networks using OLSR. *European Transactions on Telecommunications*, 15(4), 2005.

[17] Hakim Badis, Anelise Munaretto, Khaldoun Al Agha, and Guy Pujolle. QoS for ad hoc networking based on multiple-metric: Bandwidth and delay. In *International Workshop On Mobile and Wireless Communications Networks*, October 2003.

[18] Hakim Badis, Anelise Munaretto, Khaldoun Al Agha, and Guy Pujolle. Optimal path selection on a link state QoS routing. In *Vehicular Technology Conference*, May 2004.

[19] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[20] Dimitri P. Bertsekas and Robert G. Gallager. Distributed asynchronous Bellman-Ford algorithm. In *Data Networks*, chapter 5.2.4, pages 325–333. Prentice Hall, Englewood Cliffs, 1987.

[21] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. *An Architecture for Differentiated Services (RFC 2475).* IETF, December 1998.

[22] Azzedine Boukerche and Mirela Sechi M. Annoni Notare. Routing in mobile and wireless ad hoc networks. *Journal of Parallel and Distributed Computing*, 63(2):107–109, February 2003.

[23] R. Braden, D. Clark, and S. Shenker. *Integrated Services in the Internet Architecture: an Overview (RFC 1633).* IETF, June 1994.

[24] brian P. Crow, Indra Widjaja, Jeong Geun Kim, and Prescott T. Sakai. IEEE 802.11 wireless local area networks. *IEEE Communications Magazine*, 35(9):116–126, September 1997.

[25] Raffaele Bruno, Marco Conti, and Enrico Gregori. WLAN technologies for mobile ad hoc networks. In *Proceedings of the 34th Hawaii International Conference on System Sciences*, January 2001.

[26] Dazhi Chen and Pramod K. Varshney. QoS support in wireless sensor networks: A survey. In *Proceedings of the 2004 International Conference on Wireless Networks (ICWN 2004)*, June 2004.

[27] S. Chen and K. Nahrstedt. Distributed quality-of-service routing in ad hoc networks. *IEEE Journal on Selected Areas in Communications*, 17(8):1488–1505, August 1999.

[28] Tsu-Wei Chen and Mario Gerla. Global state routing: A new routing scheme for ad hoc wireless networks. In *IEEE International Conference on Communications*, pages 171–175, June 1998.

[29] Yuh-Shyan Chen, Yu-Chee Tseng, Jang-Ping Sheu, and Po-Hsuen Kuo. On-demand, link-state, multi-path QoS routing in a wireless mobile ad-hoc network. *Computer Communications*, 27(1):27–40, 2004.

[30] Ching-Chuan Chiang, Hsiao-Kuang Wu, Winston Liu, and Mario Gerla. Routing in clustered multihop, mobile wireless networks with fading channel. In *IEEE Singapore International Conference on Networks, SICON'97*, pages 197–211, April 1997.

[31] T. Clausen, P. Jacquet, A. Laouiti, P. Muhlethaler, A. Qayyum, and L. Viennot. Optimized link state routing protocol for ad hoc networks. In *Proceedings of IEEE INMIC 2001*, pages 62–68, December 2001.

[32] Douglas E. Comer. *Internetworking with TCP/IP Vol. 1 - Principles, Protocols, and Architectures*, chapter 29, pages 539–551. Prentice Hall, 4 edition, 2000.

[33] Douglas E. Comer. *Internetworking with TCP/IP Vol. 1 - Principles, Protocols, and Architectures*, chapter 14, pages 253–167. Prentice Hall, 4 edition, 2000.

[34] Douglas E. Comer. *Internetworking with TCP/IP Vol. 1 - Principles, Protocols, and Architectures*, chapter 16, pages 293–315. Prentice Hall, 4 edition, 2000.

[35] Douglas E. Comer. *Internetworking with TCP/IP Vol. 1 - Principles, Protocols, and Architectures*, chapter 8, pages 115–121. Prentice Hall, 4 edition, 2000.

[36] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 24, pages 580–619. MIT Press and McGraw-Hill, 2 edition, 2001.

[37] Sylwia Van den Heuvel-Romaszko and Chris Blondia. A survey of MAC protocols for ad hoc networks and IEEE 802.11. In *Proceedings of 4th National Conference MISSI '04*, pages 23–33, 2004.

[38] Edsger Wybe Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[39] Rohit Dube, Cyinthia D. Rais, Kuang-Yeh Wang, and Satish K. Tripathi. Signal stability based adaptive routing (SSA) for ad hoc mobile networks. *IEEE Personal Communications*, 4(1):36–45, February 1997.

[40] E.S. Elmallah, H. S. Hassanein, and H. M. AboElFotoh. Supporting QoS routing in mobile ad hoc networks using probabilistic locality and load balancing. In *Global Telecommunications Conference*, volume 5, pages 2901–2906, 2001.

[41] Afshin Fallahi, Ekram Hossain, and Attahiru S. Alfa. QoS and energy trade off in distributed enegy-limited mesh/relay networks: A queuing analysis. *IEEE Transactions on Parallel and Distributed Systems*, 17(6):576–592, June 2006.

[42] Karoly Farkas, Dirk Budke, Oliver Wellnitz, Bernhard Plattner, and Lars Wolf. QoS extensions to mobile ad hoc routing supporting real-time applications. In *Proceedings of the 4th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA '06)*, pages 54–61, March 2006.

[43] Erina Ferro and Francesco Potorti. Bluetooth and Wi-Fi wireless protocols: A survey and a comparison. *IEEE Wireless Communications*, 12(1):12–26, February 2005.

[44] J. J. Garcia-Luna-Aceves and Marcelo Spohn. Source-tree routing in wireless networks. In *Proceedings of the Seventh Annual International Conference on Network Protocols*, page 273, 1999.

[45] Zygmunt J. Haas and Marc R. Pearlman. The performance of a new routing protocol for the reconfigurable wireless networks. In *IEEE International Conference on Communications*, pages 156–160, June 1998.

[46] Tian He, John A. Stankovic, Chenyang Lu, and Tarek Abdelzaher. SPEED: A stateless protocol for real-time communication in sensor networks. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 46–55, May 2003.

[47] Yan He and Hussein Abdel-Wahab. HQMM: A hybrid QoS model for mobile ad-hoc networks. In *Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC'06)*, pages 194–200, 2006.

[48] C. Hedrick. *Routing Information Protocol (RFC 1058).* IETF, June 1988.

[49] Y.-K. Ho and R.-S. Liu. On demand QoS-based routing protocol for ad hoc mobile wireless networks. In *Proceedings of the 5th IEEE Symposium on Computers and Communications (ISCC)*, pages 560–565, 2000.

[50] Chia-Hao Hsu, Yu-Liang Kuo, E.H.-K. Wu, and Gen-Huey Chen. QoS routing in mobile ad hoc networks based on the enhanced distributed coordination function. In *Vehicular Technology Conference*, volume 4, pages 2663–2667, September 2004.

[51] Chenn-Jung Huang, Wei Lai, Yi-Ta Chuang, and Sheng-Yu Hsiao. A dynamic alternate path QoS enabled routing scheme in mobile ad hoc networks. *International Journal of Wireless Information Networks*, 14(1):1–16, March 2007.

[52] Atsushi Iwata, Ching-Chuan Chiang, Guangyu Pei, Mario Gerla, and Tsu-Wei Chen. Scalable routing strategies for ad hoc wireless networks. *IEEE Journal on Selected Areas in Communications*, 17(8):1369–1379, August 1999.

[53] P. Jacquet, A. Laouiti, P. Minet, and L. Viennot. Performance analysis of OLSR multipoint relay flooding in two ad hoc wireless network models. Technical Report 4260, INRIA, 2001.

[54] Philippe Jacquet, Anis Laouiti, Pascale Minet, and Laurent Viennot. Performance of multipoint relaying in ad hoc mobile routing protocols. In *NETWORKING 2002*, pages 387–398, April 2002.

[55] Mario Joa-Ng and I-Tai Lu. A peer-to-peer zone-based two-level link state routing for mobile ad hoc networks. *IEEE Journal on Selected Areas in Communications*, 17(8):1415–1425, August 1999.

[56] David B. Johnson. Routing in ad hoc networks of mobile hosts. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 158–163, December 1994.

[57] David B. Johnson, David A. Maltz, and Josh Broch. DSR: The dynamic source routing protocol for multihop wireless ad hoc networks. In Charles E. Perkins, editor, *Ad Hoc Networking*, chapter 5, pages 139–172. Addison-Wesley, 2001.

[58] Lestor R. Ford Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.

[59] Thanasis Karapantelakis and Giorgos Iacovidis. Experimenting with real time applications in an IEEE 802.11b ad hoc network. In *Proceedings of the The IEEE Conference on Local Computer Networks*, pages 554–559, 2005.

[60] S. Kumar, V. S. Raghavan, and J. Deng. Medium access control protocols for ad-hoc wireless networks: A survey. *Elsevier Ad Hoc Networks Journal*, 4(3):326–258, May 2006.

[61] Sathiyamurthy Kuppuswamy, N. Sreenath, and S. Amith. A distance vector routing protocol for real-time traffic in ad hoc networks. In *2nd International Conference on Technology, Knowledge, and Society*, page December, 2005.

[62] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*, chapter 7, pages 565–643. Pearson Education, Inc., 3 edition, 2005.

[63] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*, chapter 4, pages 351–370. Pearson Education, Inc., 3 edition, 2005.

[64] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*, chapter 6, pages 503–529. Pearson Education, Inc., 3 edition, 2005.

[65] A. Laouiti, A. Qayyum, and L. Viennot. Multipoint relaying: An efficient technique for flooding in mobile wireless networks. In *35th Annual Hawaii International Conference on Systems Sciences (HICSS '02)*, 2002.

[66] Seoung-Bum Lee, Gahng-Seop Ahn, Xiaowei Zhang, and Andrew T. Campbell. INSIGNIA: An IP-based quality of service framework for mobile ad hoc networks. *Journal of Parallel and Distributed Computing*, 60(4):374–406, April 2000.

[67] Sung-Ju Lee and Mario Gerla. Split multipath routing with maximally disjoint paths in ad hoc networks. In *IEEE International Conference on Communications*, pages 3201–3205, June 2001.

[68] R. Leung, J. Liu, E. Poon, A. Chan, and B. Li. MP-DSR: A QoS-aware multi-path dynamic source routing protocol for wireless ad-hoc networks. In *Proceedings of the 26th Annual IEEE Conference on Local Computer Networks (LCN 2001)*, pages 132–141, 2001.

[69] W.-H. Liao, Y.-C. Tseng, J.-P. Sheu, and S.-L. Wang. A multi-path QoS routing protocol in a wireless mobile ad hoc network. In *Proceedings of IEEE ICN '01: International Conference on Networking, Part II*, pages 158–167, July 2001.

[70] C.-R. Lin. On-demand QoS routing in multihop mobile networks. In *Proceedings of IEEE INFOCOM 2001*, pages 1735–1744, April 2001.

[71] C.-R. Lin and J.-S. Liu. QoS routing in ad hoc wireless networks. *IEEE Journal on Selected Areas in Communications*, 17(8):1426–1438, August 1999.

[72] Chenyang Lu, Brian Blum, Tarek Abdelzaher, John Stankovic, and Tian He. RAP: A real-time communication architecture for large-scale wireless sensor networks. In *Real-Time Technology and Applications Symposium*, September 2002.

[73] G. Malkin. *RIP Version 2 (RFC 2453).* IETF, November 1998.

[74] Mahesh K. Marina and Samir R. Das. On-demand multipath distance vector routing in ad hoc networks. In *IEEE International Conference on Network Protocols*, pages 14–23, 2001.

[75] Prasant Mohapatra, Jian Li, and Chao Gui. QoS in mobile ad hoc networks. *IEEE Wireless Communications*, 10(3):44–52, June 2003.

[76] Stephen Mueller, Rose P. Tsang, and Dipak Ghosal. Multipath routing in mobile ad hoc networks: Issues and challenges. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems - Tutorials '03*, pages 209–234, 2003.

[77] Anelise Munaretto, Hakim Badis, Khaldoun Al Agha, and Guy Pujolle. A link-state QoS routing protocol for ad hoc networks. In *International Workshop On Mobile and Wireless Communications Networks*, September 2002.

[78] Shree Murthy and J. J. Garcia-Luna-Aceves. An efficient routing protocol for wireless networks. *ACM Mobile Networks Applications Journal, Special Issue on Routing in Mobile Communications*, 1(2):183–197, November 1996.

[79] Dang-Quan Nguyen and Pascale Minet. QoS support and OLSR routing in a mobile ad hoc network. In *5th International Conference on Networking and the International Conference on Systems (ICN / ICONS / MCL 2006)*, page 74, April 2006.

[80] Vincent D. Park and M. Scott Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *Proceedings of the IEEE Conference on Computer Communications, INFOCOM'97*, pages 1405–1413, April 1997.

[81] Charles E. Perkins, Elizabeth M. Belding-Royer, and Samir Das. *Ad Hoc On Demand Distance Vector (AODV) Routing (RFC 3561).* IETF.

[82] Charles E. Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *Proceedings of the ACM SIGCOMM '94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, August 1994.

[83] Charles E. Perkins and Elizabeth M. Royer. Ad hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, February 1999.

[84] Charles E. Perkins, Elizabeth M. Royer, and Samir Das. *Quality of Service for Ad hoc On-Demand Distance Vector Routing*, July 2000. IETF Internet Draft, draft-ietf-manet-aodvqos-00.txt. Work in progresss.

[85] Dmitri D. Perkins and Herman D. Hughes. A survey on quality-of-service support for mobile ad hoc networks. *Wireless Communications and Mobile Computing*, 2(5):503–513, September 2002.

[86] J. Raju and J. J. Garcia-Luna-Aceves. Efficient on-demand routing using source-tracing in wireless networks. In *IEEE Globecom*, 2000.

[87] Ram Ramanathan and Jason Redi. A brief overview of ad hoc networks: Challenges and directions. *IEEE Communications Magazine*, 10(5):20–22, May 2002.

[88] S. Ramanathan and Martha Steenstrup. A survey of routing techniques for mobile communications networks. *Baltzer/ACM Mobile Networks and Applications*, 1(2):89–104, 1996.

[89] Elizabeth M. Royer and Chai-Keong Toh. A review of current routing protocols for ad hoc mobile wireless networks. *IEEE Personal Communications*, 6(2):46–55, April 1999.

[90] Cesar Santivanez, Ram Ramanathan, and Ioannis Stavrakakis. Making link-state routing scale for ad hoc networks. In *Proceedings of the ACM*

International Symposium on Mobile Ad Hoc Networking and Computing (ACM MOBIHOC '01), pages 22–32, October 2001.

[91] Nityananda Sarma and Sukumar Nandi. Enhancing QoS support in mobile ad hoc networks. In Khaled Elleithy, Tarek Sobh, Ausif Mahmood, Magued Iskander, and Mohammad Karim, editors, *Advances in Computer, Information, and Systems Sciences, and Engineering: Proceedings of IETA 2005, TeNe 2005, EIAE 2005*, pages 268–274. Springer Netherlands, 2006.

[92] Yan-Tai Shu, Guang-Hong Wang, Lei Wang, and Oliver W. W. Yang. Provisioning QoS guarantee by multipath routing and reservation in ad hoc networks. *Journal of Computer Science and Technology*, 19(2):128–137, March 2004.

[93] Raghupathy Sivakumar, Prasun Sinha, and Vaduvur Bharghavan. CEDAR: A core-extraction distributed ad hoc routing algorithm. *IEEE Journal on Selected Areas in Communications*, 17(8):1454–1465, August 1999.

[94] Chai-Keong Toh. Associativity based routing for ad hoc mobile networks. *Wireless Personal Communications Journal, Special Issue on Mobile Networking and Computing Systems*, 4(2):130–139, March 1997.

[95] Andras Varga. Parametrized topologies for simulation programs. In *Proceedings of the Western Multiconference on Simulation (WMC '98) and Communication Networks and Distributed Systems (CNDS '98)*, January 1998.

[96] Andras Varga. The OMNeT++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM 2001)*, June 2001.

[97] Andras Varga. OMNeT++. *IEEE Network Interactive*, 16(4), July 2002. In the column Software Tools for Networking.

[98] Andras Varga. *OMNet++ Community Site*, 2007. URL:<http://www.omnetpp.org/> (accessed 27 Jul 2007).

[99] Lei Wang, Yantai Shu, Miao Dong, Lianfang Zhang, and Oliver W. W. Yang. Adaptive multipath source routing in ad hoc networks. In *IEEE International Conference on Communications*, pages 867–871, June 2001.

[100] Kui Wu and Janelle J. Harms. Performance study of a multipath routing method for wireless mobile ad hoc networks. In *9th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 99–107, 2001.

[101] Hannan Xiao, Kee Chaing Chua, and Khoon Guan Winston Seah. Quality of service models for ad hoc wireless networks. In Mohammad Ilyas and Richard C. Dorf, editors, *The Handbook of Ad Hoc Wireless Networks*, pages 467–482. CRC Press, Inc., 2003.

[102] Hannan Xiao, Khoon Guan Winston Seah, Anthony Lo, and Kee Chaing Chua. A flexible quality of service model for mobile ad-hoc networks. *Vehicular Technology Conference Proceedings*, 1:445–449, May 2000.

[103] Shugong Xu and Tarek Saadawi. Does the IEEE 802.11 MAC protocol work well in multihop wireless ad hoc networks? *IEEE Communications Magazine*, 39(6):130–137, June 2001.

[104] Qi Xue and Aura Ganz. Ad hoc QoS on-demand routing (AQOR) in mobile ad hoc networks. *Journal of Parallel and Distributed Computing*, 63(2):154–165, 2003.

[105] Zhenqiang Ye, Srikanth V. Krishnamurthy, and Satish K. Tripathi. A framework for reliable routing in mobile ad hoc networks. In *IEEE INFOCOM 2003 - The Conference on Computer Communications*, pages 270–280, March 2003.

[106] Chi-Hsiang Yeh. TSD-CDMA: A QoS MAC protocol for 4G integrated mobile wireless systems. In *The IEEE Symposium on Computers and Communications (ISCC '03)*, pages 930–935, 2003.

[107] Chi-Hsiang Yeh and Tiantong You. A QoS MAC protocol for differentiated service in mobile ad hoc networks. In *Proceesings of the 32nd International Conference on Parallel Processing (ICPP '03)*, pages 349–, 2003.

[108] Tiantong You, Hossam S. Hassanein, and Chi-Hsiang Yeh. Controllable fair QoS-based MAC protocols for ad hoc wireless networks. In *Proceedings of the 33rd International Conference on Parallel Processing (ICPP '04)*, pages 21–28, 2004.

[109] Tiantong You, Chi-Hsiang Yeh, and Hossam Hassanein. DRCE: A high throughput QoS mac protocol for wireless ad hoc networks. In *Proceedings of the 10th IEEE Symposium on Computers and Communications (ISCC'05)*, pages 671–676, 2005.

[110] M. Younis, K. Akkaya, M. Eltowiessy, and A. Wadaa. On handling QoS traffic in wireless sensor networks. In *Proceedings of the International Conference HAWAII International Conference on System Sciences (HICSS-37)*, January 2004.

[111] Xukai Zou, Byrac Ramamurthy, and Spyros Magliveras. Routing techniques in wireless ad hoc networks - classification and comparison. In *Proceedings of the 6th World Multiconference on Systemics, Cybernetics, and Informatics*, pages 352–357, July 2001.

Appendix A

Module Definitions

You may find NED definitions of the three most important modules in our simulation model here.

Module AdhocNetwork

```
import
        "MyMobileHost",
        "MACTable",
        "ChannelControl",
        "FlowController",
        "CentralNode";

module AdhocNetwork
        parameters:
                sizeX : numeric,
                sizeY : numeric,
                nodeCount : numeric;

        submodules:
                channelcontrol : ChannelControl;
                        parameters:
                                playgroundSizeX = sizeX,
                                playgroundSizeY = sizeY;
                        display:
                                "p=60,50;i=block/transport";

                hosts : MyMobileHost[nodeCount];
                        parameters:
                                hostId = index + 1;

                macTable : MACTable;
```

```
                        parameters:
                                size = nodeCount;
                        display:
                                "p=140,50;i=block/table2";

                flowController : FlowController;
                        parameters:
                                hostCount = nodeCount;
                        display:
                                "p=220,50;i=block/dispatch";

                centralNode : CentralNode;
                        display:
                                "p=300,50;i=block/join";

        connections nocheck:
endmodule

network adhocnetwork : AdhocNetwork
endnetwork
```

Module MyMobileHost

```
// import the necessary .ned files
import
        "NotificationBoard",
        "BasicMobility",
        "Ieee80211NicAdhoc",
        "MyRouter",
        "TrafficGen";

module MyMobileHost
        parameters:
                mobilityType : string,
                hostId : numeric const;
        gates:
                in: radioIn, trafficIn;
        submodules:
                notificationBoard: NotificationBoard;
                        display: "i=block/control";
                mobility: mobilityType like BasicMobility;
                        display: "i=block/wheelbarrow";
                wlan: Ieee80211NicAdhoc;
                        display: "i=device/card";
                router: MyRouter;
                        parameters:
                                nodeId = hostId;
                        display: "i=abstract/router2";
                trafficGen: TrafficGen;
                        parameters:
                                myId = hostId;
                        display: "i=block/source";
        connections nocheck:
                trafficGen.outGate --> router.fromUpper;
                router.toLower --> wlan.uppergateIn;
                router.fromLower <-- wlan.uppergateOut;
                radioIn --> wlan.radioIn;
                trafficIn --> trafficGen.inGate;
endmodule
```

Module MyRouter

```
simple MyRouter
        parameters:
                wholeK : numeric const,
                nodeId : numeric const;
        gates:
                in: fromUpper, fromLower;
                out: toLower;
endsimple
```

Appendix B

Data Types used in Message Definitions

Several header files for data types used in message definitions are provided here.

LSMsgContent.h

```
// File: LSMsgContent.h

// Header file for LSMsgContent class

// Copyright (C) Gokce Gorbil, 2007
// Bilkent University, Computer Engineering Dept.

#pragma once

#ifndef LSMSGCONTENT_H_
#define LSMSGCONTENT_H_

#include "IncrementalMsg.h"
#include <vector>

using std::vector;

class LSMsgContent
{
public:
        // data members
        static const int INCREMENTAL = 4;
        static const int WHOLE = 5;

        // contructors etc
        LSMsgContent();
        LSMsgContent(const int s, const int t);
        LSMsgContent(const LSMsgContent &);
        virtual ~LSMsgContent();

        // methods
        int getSource() const;
        int getType() const;
        long getBitLength() const;
        vector<int>* getNeighborsPtr() const;
        vector<IncrementalMsg>* getMsgsPtr() const;

        void setSource(const int s);
        void setType(const int t);
        void addToNeighbors(const int n);
        void addToMsgs(IncrementalMsg &msg);

        // overloaded operators
        const LSMsgContent &operator= (const LSMsgContent &);
        bool operator== (const LSMsgContent &) const; // using source and type
        bool operator!= (const LSMsgContent &) const; // using source and type

private:
        int source;
        int type;
        vector<int> *neighborsPtr;
        vector<IncrementalMsg> *msgsPtr;
};

#endif /*LSMSGCONTENT_H_*/
```

IncrementalMsg.h

```
// File: IncrementalMsg.h

// Header file for IncrementalMsg class

// Copyright (C) Gokce Gorbil, 2007
// Bilkent University, Computer Engineering Dept.

#pragma once

#ifndef INCREMENTALMSG_H_
#define INCREMENTALMSG_H_

class IncrementalMsg
{
public:
        // data members
        const static int ADD = 454;
        const static int REMOVE = 455;

        // constructors etc
        IncrementalMsg();
        IncrementalMsg(int a, int s, int d);
        IncrementalMsg(const IncrementalMsg &);
        virtual ~IncrementalMsg();

        // methods
        int getAction() const;
        int getSource() const;
        int getDest() const;

        void setAction(const int a);
        void setSource(const int s);
        void setDest(const int d);

        // overloaded operators
        const IncrementalMsg &operator= (const IncrementalMsg &);
        bool operator== (const IncrementalMsg &) const; // using all fields
        bool operator!= (const IncrementalMsg &) const; // using all fields

private:
        int action;
        int source;
        int dest;
};

#endif /*INCREMENTALMSG_H_*/
```

Route.h

```
// File: Route.h

// Header file for Route class

// Copyright (C) Gokce Gorbil, 2007
// Bilkent University, Computer Engineering Dept.

#pragma once

#ifndef ROUTE_H_
#define ROUTE_H_

#include <vector>

class QoSReq; // forward class declaration

using std::vector;

class Route
```

```
{
public:
        // constructors etc
        Route();
        Route(const int t);
        Route(const int t, const QoSReq &r);
        Route(const Route &);
        virtual ~Route();

        // methods
        int getSource() const;
        int getDest() const;
        int getType() const;
        int getLength() const;
        long getBitLength() const;
        int getNextNode(const int id) const;
        QoSReq* getQoSReq() const;
        vector<int>* getPath() const;

        void setType(const int t);
        void setQoSReq(QoSReq &r);
        void addToPathEnd(const int id);
        void addToPathFront(const int id);
        void clearPath();

        void reversePath(Route &) const;
        bool reversePath(Route &route, const int id) const;
        void getSubPath(const int startNode, const int endNode, Route &subPath) const;
        void append(const Route &route);
        bool checkForLoop() const;
        int findNodePlace(const int id) const;
        bool hasSubPath(const int fromNode, const int toNode) const;

        // overloaded operators
        const Route &operator= (const Route &);
        bool operator== (const Route &) const; // using source, dest, and type
        bool operator!= (const Route &) const; // using source, dest, and type

private:
        int type;
        QoSReq *qosReqPtr;
        vector<int> *pathPtr;
};

#endif /*ROUTE_H_*/
```

QoSReq.h

```
// File: QoSReq.h

// Header file for QoSReq class

// Copyright (C) Gokce Gorbil, 2007
// Bilkent University, Computer Engineering Dept.

#pragma once

#ifndef QOSREQ_H_
#define QOSREQ_H_

class QoSReq
{
public:
        // data members
        static const int NORMAL = 1983;
        static const int RTT = 1984;
        static const int LOSS = 1985;
        static const int BW = 1986;
        static const int RTT_AVG = 1987;
        static const int LOSS_AVG = 1988;
        static const int BW_AVG = 1989;

        static const float MIN_LOSS_RATE = 0.0; // probability
        static const float MAX_LOSS_RATE = 1.0; // probability
        static const float MIN_RTT = 0.0; // in ms
        static const float MAX_RTT = 60000.0; // in ms
        static const float MIN_BW = 0.0; // in bits/ses
        static const float MAX_BW = 110000000.0; // in bits/sec

        // constructors etc
        QoSReq();
        QoSReq(const int t);
        QoSReq(const int t, const float l, const float u);
        QoSReq(const int t, const float limit);
        QoSReq(const QoSReq &);
        virtual ~QoSReq();

        // methods
        float getLowerLimit() const;
        float getUpperLimit() const;
        int getType() const;

        void setLowerLimit(const float f);
        void setUpperLimit(const float f);
        void setType(int t);

        // overloaded operators
        const QoSReq& operator= (const QoSReq &c);
        bool operator== (const QoSReq &c) const; // using all fields
        bool operator!= (const QoSReq &c) const; // using all fields

        bool operator< (const QoSReq &c) const; // using lowerLimit
        bool operator<= (const QoSReq &c) const; // using lowerLimit
        bool operator> (const QoSReq &c) const; // using lowerLimit
        bool operator>= (const QoSReq &c) const; // using lowerLimit

private:
        float lowerLimit;
        float upperLimit;
        int type;
};

#endif /*QOSREQ_H_*/
```

CostMsgContent.h

```
// File: CostMsgContent.h

// Header file for CostMsgContent class

// Copyright (C) Gokce Gorbil, 2007
// Bilkent University, Computer Engineering Dept.

#pragma once

#ifndef COSTMSGCONTENT_H_
#define COSTMSGCONTENT_H_

#include <vector>
#include "LinkCost.h"

using std::vector;

class CostMsgContent
{
public:
        // constructors etc
        CostMsgContent();
        CostMsgContent(const int s);
        CostMsgContent(const CostMsgContent &);
        virtual ~CostMsgContent();

        // methods
        int getSource() const;
        long getBitLength() const;
        vector<LinkCost>* getCostsPtr() const;

        void setSource(const int id);
        void addToCosts(const LinkCost &);

        // overloaded operators
        const CostMsgContent &operator= (const CostMsgContent &);
        bool operator== (const CostMsgContent &) const; // using source
        bool operator!= (const CostMsgContent &) const; // using source

private:
        int source;
        vector<LinkCost> *costsPtr;
};

#endif /*COSTMSGCONTENT_H_*/
```

LinkCost.h

```
// File: LinkCost.h

// Header file for LinkCost class

// Copyright (C) Gokce Gorbil, 2007
// Bilkent University, Computer Engineering Dept.

#pragma once

#ifndef LINKCOST_H_
#define LINKCOST_H_

class LinkCost
{
public:
        // constructors etc
        LinkCost(); // default constructor
        LinkCost(const LinkCost &c); // copy constructor
        LinkCost(const int type, const int dest, const float alpha);
        virtual ~LinkCost();

        // methods
        int getType() const;
        int getDest() const;
        float getCurrentVal() const;
        float getAvgVal() const;
        float getFactorAlpha() const;
        double getLastUpdateTime() const;

        void setType(const int t);
        void setDest(const int d);
        void setCurrentVal(const float v);
        void setFactorAlpha(const float f);
        void setLastUpdateTime(const double t);

        // overloaded operators
        const LinkCost &operator= (const LinkCost &);
        bool operator== (const LinkCost &) const; // according to dest and type
        bool operator!= (const LinkCost &) const; // according to dest and type

        bool operator< (const LinkCost &) const;  // according to avgVal
        bool operator> (const LinkCost &) const;  // according to avgVal
        bool operator<= (const LinkCost &) const; // according to avgVal
        bool operator>= (const LinkCost &) const; // according to avgVal

private:
        // data members
        int type;
        int dest;
        float currentVal;
        float avgVal;
        float factorAlpha;
        double lastUpdateTime;

        // methods
        void calculateAvgVal();
};

#endif /*LINKCOST_H_*/
```