MASTER'S THESIS

Scalable Route Caching Methods for Networks with Many Mobile Nodes

by Theodoros Salonidis
Advisor: Leandros Tassiulas

CSHCN M.S. 99-8
(ISR M.S. 99-13)

The Center for Satellite and Hybrid Communication Networks is a NASA-sponsored Commercial Space Center also supported by the Department of Defense (DOD), industry, the State of Maryland, the University of Maryland and the Institute for Systems Research. This document is a technical report in the CSHCN series originating at the University of Maryland.

Web site http://www.isr.umd.edu/CSHCN/

ABSTRACT

Title of Thesis: SCALABLE ROUTE CACHING METHODS FOR NETWORKS WITH MANY MOBILE NODES

Degree Candidate: Theodoros Salonidis

Degree and year: Master of Science, 1999

Thesis directed by: Associate Professor Leandros Tassiulas

Department of Electrical Engineering

A mobile, ad hoc network (MANET) is a collection of wireless mobile hosts forming a network without the aid of any established infrastructure or centralized administration. Generally a MANET may consist of many portable devices that are characterized by processing and memory size limitations and in practice it will not be possible for a host to keep routing information for all the nodes in a large network. This thesis attempts to address the scalability issue by introducing a framework and strategies to quantify the concept of destination caching. The observation that a source host can augment its cache's routing table by using the caches of other closely situated hosts forms the basis of our approach. We propose algorithms that determine a host's cached information by taking into account the host's memory capacity, the network size and the number and identity of the destinations this host needs to cache information about.

Mainly two classes of algorithms are introduced. The class of "Best State/Best Cost" algorithms (BSBC) tries to minimize the flooding cost per route discovery by

keeping the most "expensive" destinations in each host's cache. However it does not impose any flooding constraints for the non-cached destinations. The second class of LEADERS algorithms adopts a different view by relaxing on the flooding cost optimality and taking into account a maximum flooding constraint for each node. In this way, the worst flooding case is controlled since any node is guaranteed to find information about any destination within a pre-specified maximum distance.

SCALABLE ROUTE CACHING METHODS FOR NETWORKS WITH MANY MOBILE NODES

by

Theodoros Salonidis

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland at College Park in partial fulfillment
of the requirements for the degree of
Master of Sciences
1999

Advisory Committee:

Associate Professor Leandros Tassiulas, Chair
Assistant Research Scientist M. Scott Corson
Assistant Professor Donald Yeung

DEDICATION

Dedicated to my family Nikos, Anastasia, Maria, Eleftheria and Penny

ACKNOWLEDGEMENTS

First of all I would like to thank my advisor Dr Leandros Tassiulas for proposing the topic of this thesis, and for the enlightening advice and guidance he provided during the elaboration of this work. Many thanks to Dr Donald Yeung and Dr Scott Corson, who were in my committee and provided with a lot of insightful comments during the presentation and after the completion of this thesis.

I am grateful to my roommates and friends Kyriakos Manousakis, Iordanis Koutsopoulos and Nikos Kanlis for their friendship and support during all this period. I also thank Kostas Tsoukatos and Tassos Michail for their useful general advice as senior colleagues, Alexandros Lambrinidis for our discussions on clustering algorithms and all my friends in the Greek Graduate students association (DIGENIS) for creating such a supportive, joyful and friendly environment.

Last but not least I would like to thank my family. Their love, guidance and support is the main factor of success throughout all my years of studies. This work is dedicated to them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

1.1 Introduction

The recent advances in wireless communications technologies combined with the introduction of small portable computing devices have created the demand of realizing the ubiquitous networking dream: "Seamless communication and network access anytime and anywhere".

In order to fulfill these increasingly rising expectations, extensive work nowadays focuses on the integration of mobile computers within the fixed network infrastructure of the Internet. The extension of the Internet to mobile domains and hosts is implemented by the Mobile IP technology. Mobile IP is intended to enable nodes to move from one IP subnet to another either this subnet is in a form of a wireless LAN or an Ethernet segment. One can think of Mobile IP as solving the "macro" mobility management problem, since mobility in this case is happening in the granularity of IP subnets. It is less well suited for more "micro" mobility management applications like the handoff amongst wireless transceivers, each of which covers only a very small geographic area. Supporting this form of host mobility requires address management and protocol interoperability enhancements, but core network functions such as hop-by-hop routing still presently rely upon pre-existing routing protocols operating within the fixed network.

The concept of mobile ad-hoc networking tries to extend mobility within autonomous, mobile, wireless domains. In contrast to the existing cellular networks, there is no centralized administration of each wireless domain (e.g. by the use of base stations), but rather the set of mobile hosts form by themselves the network routing infrastructure

in an ad hoc fashion. Each node participates in an ad-hoc routing protocol that allows it to discover "multi-hop" paths through the network to any other node.

Mobile ad-hoc networks are ideally suited for military and other tactical applications such as emergency rescue or exploration missions, where an established (e.g. cellular) infrastructure is unavailable or unreliable. But apart from the emergency situations, there are a lot of emerging ubiquitous computing technologies that naturally fit in the "infrastructureless" networking model: On-the-fly conferencing applications, networking intelligent devices, wearable computing and smart sensor networks, are commercial applications of ubiquitous computing that are to be deployed in the near future.

There are no assumptions or limitations on the size of the wireless domains in ad-hoc networks. This means that ad-hoc networks may consist of hundreds or thousands of nodes. The sheer number of nodes that one might find in a ubiquitous computing network or a smart mobile sensor network underscores the need for a level of scalability not commonly present in most approaches to network routing and management.

An important question that needs to be addressed in MANETs concerns how scalability is achieved : Since a MANET may consist of many portable devices that are characterized by processing and memory size limitations, in practice it will not be possible for a host to keep routing information for all the nodes in a large network. Some existing routing protocols (DSR [14], AODV [10]) pinpoint the need for caching routing information about some destinations and flooding the network in case a source node's cache does not have the route to a specific destination. However they do not address the question of which destinations to cache information about. In the case of small to

medium sized networks this question is not so important, since the flooding cost for finding a destination is not very large. However, in a large MANET a bad choice of destinations to cache may result in flooding a great portion of the network just for route discovery. If this route traffic adds up to the already existing data traffic then there is a huge congestion problem in the network. Thus, as the MANET becomes larger, the identity of destinations a host keeps in its cache becomes a critical issue. This issue becomes even more critical if we consider the wireless nature of MANETs which is generally characterized by severe bandwidth restrictions for data transmission.

This thesis attempts to address the scalability issue described above. We introduce a framework and strategies to quantify the concept of caching information about destinations. The observation that a source host can augment its cache's routing table by using the caches of other closely situated hosts forms the basis of our approach. We propose algorithms that determine a host's cached information by taking into account the **host's memory capacity**, **the network size** and the **number** and **identity of the destinations this host needs to cache information about**. The main objective is to minimize the flooding cost per route discovery by keeping the most "expensive" destinations in the host's cache.

In the subsequent sections, we will define the notion of MANETs and the issues involved in their design, and will review and compare routing algorithms that have been implemented so far in the context of large MANETs. At the end of the chapter we will point out the need for route caching when the network gets big, and propose a way to do that in order to achieve scalability. In chapter 2 we will define the framework upon which we try to quantify the notion of caching destinations. A NxN torroid mesh will be

introduced, where nodes are considered to be hosts with small memory capacity. Each node will try to address all other nodes in the network with equal probability. Optimal caching strategies are found for this case and at the end of the chapter there will be a performance evaluation of each of the strategies employed. In chapter 3 we will consider the more interesting and realistic case of each node addressing only a subset of the mesh network. We will see that the resulting caching algorithms in this case are more general and they do not depend on the mesh structure of the network, rendering them more applicable to the case of a MANET. Mainly two classes of algorithms were invented. The first class, takes into account the available cache capacity per host but does not impose any flooding constraints for the non-cached destinations after the algorithm is run. The second class adopts a different view by relaxing on the flooding cost optimality and taking into account a maximum flooding constraint in addition to the cache size. At the end of the chapter there will be a performance evaluation of the algorithms and discussion on the trade-offs governing them. The final chapter will consist of a summary with conclusions and discussion.

1.2 Mobile Ad-hoc networks (M.A.NETs)

A mobile, ad hoc network (MANET) is a collection of wireless mobile hosts forming a network without the aid of any established infrastructure or centralized administration. A MANET may be considered an autonomous system of mobile nodes. Such networks have dynamic, sometimes rapidly-changing, random, multihop topologies which are likely composed of relatively bandwidth-constrained wireless links.

MANETs have several salient characteristics that have to be taken into account when considering their design and deployment:

- **Dynamic topologies:** Nodes are free to move arbitrarily; thus, the network topology (which is typically multihop) may change randomly and rapidly at unpredictable times, and may consist of both bidirectional and unidirectional links.
- **Bandwidth-constrained, variable capacity links:** Wireless links typically have significantly lower capacity than their hardwired counterparts. In addition, the realized throughput of wireless communications (after accounting for the effects of multiple access, fading, noise, and interference conditions) is often much less than a radio's maximum transmission rate.
- **Energy-constrained operation**: Some or all of the nodes in a MANET may rely on batteries or other exhaustible means for their energy. For these nodes, a finite energy capacity may be the most significant performance constraint, and thus its utilization should be viewed as a primary network control parameter.
- **Limited physical security:** Mobile wireless networks are generally more prone to physical security threats than are fixed-cable nets. The increased possibility of eavesdropping, spoofing, and denial-of-service attacks should be carefully considered. Existing link security techniques are often applied within wireless networks to reduce security threats.

While the above characteristics are common to any kind of wireless network, MANETs are further distinguished by their **"Infrastructureless" property** : There is no centralized administration or preexisting infrastructure that takes care of the network management and existence. Mobile nodes are themselves responsible for establishing and maintaining connection between them. In such an environment, it is necessary for one

mobile host, to enlist the aid of other hosts in forwarding a packet to its destination, due to the limited range of each mobile host's wireless transmissions. Thus, robust and efficient operation in mobile wireless networks is supported by incorporating routing functionality into all the mobile nodes, in contrast to fixed networks such as the Internet where only some nodes in the network perform the routing function.

It is obvious that the routing function is of utmost importance for the viability of an ad-hoc network It is also a big challenge since all the characteristics of the mobility and wireless channel previously mentioned, must be taken into account when designing such protocols. There has been extensive research and work in the field of routing in MANETs. The routing protocols can be separated in two main classes, namely proactive or reactive, according to the way routes are created and maintained.

1.3 Proactive Routing Protocols

Proactive routing protocols for mobile ad-hoc networks are built on the philosophy of traditional routing protocols used in packet switched networks. So before we describe some representatives of this class we review the two main ways of routing in wired networks namely "Link State" and "Distance Vector" routing algorithms.

In link state protocols each node maintains its own view of the network topology, including link costs of all its outgoing links. To keep views up to date, each node periodically broadcasts the link costs of all its neighbors to all the nodes in the network using flooding. This is done whenever there is a change in link costs. As a node receives this information, it updates its view of the network topology and applies a shortest path algorithm (e.g. Dijkstra's algorithm [2]) to choose the next hop to a destination. Asynchronous link cost updates may give rise to short-lived routing loops; however they

disappear by the time update messages have propagated throughout the network [1]. A very popular link state routing protocol used in wired networks is OSPF (Open Shortest Path First)[12].

In the Distance Vector approach, for each destination *i*, every node *j* maintains a set of distances or costs, $d_{ik}(j)$, where *k* ranges over the neighbors of *i*. Node k^* is treated as the next hop node for a data packet destined for *j*, if $d_{ik^*}(j) = \min_k \{d_{ik}(j)\}$. To keep these distances up-to date, whenever there is any change of this minimum distance because of link cost changes, the new minimum distance is reported to the neighbor nodes. If, as a result, a minimum distance to any neighbor changes, this process is repeated until the network reaches eventually a steady state. This technique is the classical distributed Bellman Ford algorithm (DBF) [2]. Compared to the link state method, it is comptationally more efficient, easier to implement and requires much less storage space.

Besides the advantages over link state, DBF suffers from the problem of short lived or long lived routing loops. The primary cause for this is that nodes make uncoordinated modifications to their routing tables based on some information which could be incorrect. Routing loops result in packets circulating meaningless in the network consuming bandwidth and resources. Especially in the case of the low bandwidth wireless environment we want to avoid the creation of such loops at any cost. This problem is alleviated by employing internodal coordination mechanisms [4][5]. However these are complex methods and they might be effective when topological changes are rare. In the context of mobile networks, a simpler approach to solve this problem using sequence numbers was followed in the DSDV algorithm, which we will review later.

There is also a possibility of the "counting to infinity" problem, where it takes a very large number of update messages to detect that a node is unreachable. This performance problem arises from the fact that DBF does not have an inherent mechanism to determine when a network node should stop incrementing its distance to a given destination [2].

On the other hand, Link State algorithms are free of the "counting to infinity" problem. However they need to maintain the up-to-date version of the entire network topology at every node, which may constitute excessive storage and communication overhead in a highly dynamic network. Also no link state algorithm implemented so far has been able to totally eliminate the creation of temporary routing loops.

Distance Vector and Link State algorithms have as a common characteristic that they are both shortest path approaches : They allow a host to find the next hop neighbor to reach the destination via the "shortest path". By "shortest path" we usually mean the number of hops; however other suitable cost measures such as link utilization or queuing delay can also be used.

Although the methods of distance vector and link state are good ideas and have been successfully used in many dynamic packet switched networks, they cannot be used in their native form in MANETs. The flooding techniques used in link state protocols create excessive traffic in a multihop radio network with dynamic topology. On the other hand, the routing protocols based on DBF take a long time to converge and the frequent topology changes in a wireless network with mobile nodes make the looping problem of DBF unacceptable.

Proactive Routing protocols try to match the link state and distance vector ideas to the wireless environment by taking the above limitations into account and trying to reduce or eliminate them. The two prominent protocols in the proactive class are the Destination Sequenced Distance Vector Protocol (DSDV) [9] and the Wireless Routing Protocol (WRP) [11].

1.3.1 Destination Sequenced Distance Vector Protocol (DSDV)

The Destination Sequenced Distance Vector protocol (DSDV) [9] has been specifically targeted for mobile networks. Its key advantage of DSDV over traditional distance vector protocols is that it guarantees loop freedom. This protocol extends on the classical DBF by tagging each distance entry $d_{ik}(j)$ by a sequence number (SN) that originated by the destination node *j*. In this way, nodes can quickly distinguish stale routes from the new ones and thus avoid formation of routing loops.

To maintain the consistency of routing tables in a dynamically varying topology, each station periodically transmits updates immediately when significantly new information is available. This update must be made often enough to ensure that every mobile computer can almost always locate any other mobile computer of the collection. Also this update is in the form of incremental packets that reflect only the changes and not the whole routing table of the node. So the data broadcast by each mobile computer will contain its **new incremented sequence number** and the following information for each **new** route : The **destination's address**, the **number of hops required to reach the destination** and the **sequence number of the information received regarding the destination,** as originally stamped by the destination.

When a mobile host receives new routing information, that information is compared to the information already available from previous routing information packets. A route R is more favorable than R' if R has a greater sequence number or if the two routes have equal sequence numbers but R has a lower metric. The metric used in the algorithm to determine shortest paths is the hop count. The metrics for routes chosen from the newly broadcast information are each incremented by one hop. Newly recorded routes are scheduled for immediate advertisement to the current mobile host's neighbors. Routes received in broadcasts are also advertised by the receiver when it subsequently broadcasts its routing information; the receiver adds an increment to the metric before advertising the route, since incoming packets will require one more hop to reach the destination (namely the hop from the transmitter to the receiver).

A broken link is described by an infinite metric (i.e. any value greater than the maximum allowed metric). When a node A decides that its route to a destination D has broken, it advertises the route to D with an infinite metric and a sequence number one greater than its sequence number for the route that has broken (making an odd sequence number). This causes any node B which is routing packets through A to incorporate the infinite metric route into its routing table until node B hears a route to D with a higher sequence number.

In addition to loop elimination it has been shown that DSDV avoids the counting to infinity problem as well.

1.3.2 Wireless Routing Protocol (WRP)

The Wireless Routing Protocol (WRP) is based on a broader class of distributed shortest path algorithms that utilize information regarding the length and the second-to-

last hop (predecessor) of the shortest path to each destination to eliminate the counting to infinity problem of DBF. Each node maintains the shortest path spanning tree reported by its neighbors. A node uses this information along with the cost of adjacent links to generate its own shortest path spanning tree. An update message exchanged among neighbors consists of a vector of entries that report updates to a sender's spanning tree; each update entry contains a destination identifier, the distance to the destination, and the second-to-last hop of the shortest path to the destination.

The general class of PFAs eliminate the counting to infinity problem but still incurs temporary loops in the paths specified by the predecessor before they converge. Without proper precautions this can lead to slow convergence or incur substantial processing if a node is required to update its entire routing table for each input event. WRP remedies this by limiting routing table updates to include only these entries affected by a network change.

1.4 Reactive Routing Protocols

As we saw in the previous paragraph, proactive protocols try to keep the shortest path routes and routes are maintained to all potential destinations (possibly all nodes in the network) all the time, whether or not all such routes are actually used. Route maintenance is obtained by route update traffic, and this can be a lot of overhead, especially for large networks.

Reactive routing protocols or "on demand" routing protocols create and maintain routes only in an "as needed" basis. When a route is needed, a global route discovery procedure is initiated to find the path for the specific information that has not been

cached. The route discovery is usually done by employing classical flooding mechanisms.

1.4.1 Dynamic Source Routing (DSR)

The Dynamic Source Routing protocol (DSR) [14], uses source routing – a technique where the source of a data packet determines the complete sequence of nodes through which to forward the packet. The source explicitly lists this route in the packet's header and then transmits the packet over its wireless network interface to the first hop identified in the source route. When a host receives the packet, if this host is not the final destination of the packet, it strips from the packet header its own address and simply transmits the packet to the next hop identified in the packet header. This process goes on until the packet reaches its destination. The two basic operations supported by DSR are Route Discovery and Route Maintenance.

DSR builds routes on demand by flooding the network in a controlled manner (for example by using a Time To Live (TTL) field in the packet header). In Route Discovery, the source node sends a query in the form of a *route request* packet that carries the sequence of hops it passed through. Once a query reaches the destination, the destination replies with a *route reply* packet that simply copies the route from the query packet and traverses it backwards. To reduce the cost and frequency of the Route Discovery, each node has a route cache, where complete routes have been stored as learned from the reply packets. These routes are used by the data packets until they fail as determined by the failure of attempted message transmissions.

Route Maintenance procedure monitors the operation of the routes and informs the sender of any routing errors. Error detection is supported in the data link layer. If the

data link layers reports a transmission problem for which it cannot recover, this host sends a *route error* packet to the original sender of the packet. The route error packet contains the addresses of the hosts at both ends of the hop in error : the host that detected the error and the host to which it was attempting to transmit this packet on this hop. When a route error packet is received by the source host, the hop in error is removed from this host's route cache and all routes which contain that hop are truncated at that point. Then, a new route discovery for the destination is initiated by the sender.

DSR has a unique advantage by virtue of source routing. The key advantage of source routing is that intermediate nodes need not maintain up-to date routing information in order to route the packets they forward, since the packets themselves already contain all the routing decisions. This fact coupled with the on demand nature of the protocol eliminates the need for the periodic route advertisement and neighbor detection packets present in other protocols. Another nice feature of DSR is that since the route is part of the packet itself, routing loops, either short or long lived, cannot be formed, as they are immediately detected and eliminated. This property opens up the protocol to a variety of useful optimizations. For example, a flooded query can be *"quenched early"* by having any non-destination host reply to the query if that host has a route to the intended destination. A node can learn a route to a destination while passing on route reply packets. Also routes can be improved by having nodes promiscuously listen to conversations between other nodes in proximity.

1.4.2 Ad-Hoc On Demand Distance Vector Protocol (AODV)

AODV is an on-demand variation of distance vector protocols. It is essentially a combination of DSR and DSDV : It borrows the basic on demand mechanism of Route

Discovery and Route Maintenance from DSR plus the use of hop-by-hop routing, sequence number and periodic beacons from DSDV.

AODV uses sequence numbers maintained at destinations to determine freshness of routing information. In AODV, a query flood is used to create a route, with the destination responding to the first such query, much as in DSR. However, AODV maintains routes in a distributed fashion, as routing table entries on all intermediate nodes of the route. Routing table entries are tuples in the form of <destination, next hop, distance>. Nodes propagating query packets "remember" the earlier hop taken by such a query packet. This hop is used to forward the reply packet back to the source. The reply packet, in turn, sets the routing table entries on the nodes in its path.

Distributing routing tables in such a fashion makes them smaller than the case of DSR where each node must keep the whole path in its cache. However, in order to maintain routes, AODV normally requires that each node periodically transmit a HELLO message, with a default rate of one per second. Failure to receive three consecutive HELLO messages from a neighbor is taken as an indication that the link to the destination in question is down. When a link goes down, any upstream node that has recently forwarded packets to a destination using that link is notified via an UNSOLICITED ROUTE REPLY containing an infinite metric for that destination. Upon receipt of such a ROUTE REPLY, a node must acquire a new route to the destination using Route Discovery as described above.

AODV maintains the addresses of the neighbors through which packets destined for a given destination were received. A neighbor is considered *active* (for a destination) if it originates or relays at least one packet for that destination, within the past *active*

timeout period. A routing table entry is active if it is used by an active neighbor. The path from a source to destination via the active routing table entries is called an *active path*. On a link failure, all routing table entries for which the failed link is on the active path, are erased. This is accomplished by an error packet going backwards to the active neighbors, which forward them to their active neighbors and so on. This technique effectively erases the route backwards from the failed link.

Much like DSR, AODV advocates use of *"early quenching"* of request packets, i.e, any node having a route to the destination can reply to a request. AODV also uses a technique called *route expiry*, where a routing tale entry expires after a predetermined period, after which fresh route discovery must be initiated.

Neither DSR nor AODV guarantee shortest path. This is particularly true if *early quenching* is used. However, earlier performance evaluation shows that the lengths of the routes discovered are usually very competitive with the ones found in shortest path protocols [16].

1.4.3 Temporary Ordered routing Algorithm (TORA)

The unique feature of TORA [6] is that it is a protocol designed to minimize reaction to topological changes. This is accomplished by maintaining multiple routes to a specific destination, so that many topological changes need no reaction at all, unless all routes to a specific destination are lost. In that case routes are re-established via a temporary ordered sequence of diffusing computations, which are essentially link reversals that eventually establish a path to the destination.

TORA does not maintain routes between a given source/destination pair at all times but creates new routes on demand. Route optimality (shortest paths) is considered

secondary importance, and longer routes are often used to avoid the overhead of discovering newer routes. This ability to initiate and react infrequently serves to minimize the communication overhead at the expense of multiple non-optimal (shortest path) routes to the destination. In order to select one of the multiple routes two alternatives are suggested : choosing a neighbor randomly so that the loads are more or less evenly distributed or choosing the "lowest" neighbor.

TORA is based in part on algorithms that try to maintain the destination oriented property of a directed acyclic graph [3]. A DAG is defined to be *destination oriented* if there is always at least one path to a specific destination. The DAG becomes *destination disoriented* when one or more link fails. In this case by employing link reversals these algorithms ensure that the DAG will become again destination oriented in a finite time. TORA uses the notion of node "height" to maintain the destination oriented DAG. Each node maintains a height and exchanges this value with each neighbor. The significance of the height is that a link is always directed from a "higher" node to a "lower" node. The notion of "height" and link reversals are destination specific. This means that each node of the network runs a logically separate copy of TORA for each destination.

The **route discovery** process for a specific destination creates a destination oriented graph for this destination in a source initiated fashion: The source node broadcasts a QUERY packet containing the address of the destination for which it requires a route. This packet propagates through the network until it reaches the destination or an intermediate node having a route to the destination. The recipient of the QUERY then broadcasts an UPDATE packet listing its height with respect to the destination. As this packet propagates through the network, each node that receives the

UPDATE sets its height to a value greater than the height of the neighbor from which the update was received. This has the effect of creating a series of directed links from the original sender of the QUERY to the node that initially generated the update.

Route maintenance is achieved as follows : when a node *i* discovers that a route to a destination is no longer valid, it adjusts its height so that it is a local maximum with respect to its neighbors and transmits an *UPDATE* packet. This is effectively a link reversal and it means that now all links emanating from node *i* are directed from *i* towards its neighbors (since the neighbors have now lower heights). This has as an effect all traffic entering *i* to flow back out of *i* towards the neighbor nodes that had previously been routing packets to the destination via *i*. If the node has no neighbors of finite height with respect to this destination, then the node attempts to discover a new route via the route discovery process.

Finally, in the event of network partitions, the protocol is able to detect the partition and erase all invalid routes : when a node detects a network partition, it generates a CLEAR packet that resets routing state and removes invalid routes from the network.

1.5 The scalability issue

Most envisioned MANET networks (e.g. mobile military networks or highway networks) may be relatively large (e.g. tens or hundreds of nodes per routing area). An interesting question would be how would proactive and reactive protocols compete in terms of scalability. Even if there has not been a performance comparison between the two classes, we argue in favor of the reactive protocols when scalability is considered. As we saw in the previous sections, traditional routing protocols using link state or distance vector have been successfully used in the deployment of dynamically fixed

packet switched networks. However their success does not include efficiency with respect to scalability : When the number of network nodes becomes very large, routing loops, the counting to infinity problem and very slow convergence times make the network performance unacceptable. Even if proactive protocols like DSDV and WRP have eliminated or reduced the formation of routing loops and the counting to infinity problem, each one has still its own problems.

DSDV requires selection of the following parameters : periodic update interval, maximum value of the settling time for a destination and the number of the update intervals which may transpire before a route is considered "stale". This sensitivity of parameter selection is even more pronounced when we have a huge network. Another drawback of DSDV is that a node has to wait until it receives the next update message originated by the destination in order to update its distance-table entry for that destination. This implicit destination-centered synchronization suffers from the same latency problems as similar algorithms based on explicit synchronization. Also DSDV uses both periodic and triggered updates for updating routing information, which could cause excessive communication overhead. Of course such latency and excessive communication overhead become more severe when the network becomes bigger.
In WRP, there is a significant amount of overhead associated with maintaining the shortest path spanning tree reported by each neighbor and reactions to failures may be far-reaching (i.e every node which includes the link in its shortest path spanning tree must participate in the failure reaction). This high overhead does not seem to make this protocol viable in the case of a large number of nodes.

Finally and most importantly, for a large number of mobile nodes, both proactive protocols will converge very slowly, since shortest path convergence time always depends on the network size.

Fortunately, fixed networks addressed the scalability problem by utilizing hierarchical structures, and using aggregation : Only some specialized nodes called routers or gateways perform the routing function over clusters (LANs) of hosts that support their own more efficient (and more expensive) routing protocols. This approach does not seem to fit in the case of MANETs and proactive algorithms used in them, since **each** node must support a routing functionality and we are required to have a flat structure (every node has the same functionality with the rest).

On the other hand, reactive protocols seem to be a more attractive solution in terms of scalability in MANETs. The property of creating and maintaining routes on demand eliminates the need to keep constantly routing information on every destination, and caching the most expensive routes reduces communication overhead and latency a lot.

However the problem of scalability strikes back in a different way. Each host is a portable device which usually has a limited processing power and memory capacity and in general it will not be able to keep routing information for **all** the destinations it has traffic for : If the destination is in the source node's cache and is up to date, then it is used to address the destination. If it is not, the node must flood the network to find information about the destination.

1.6 Intelligent Caching Algorithms for “On Demand” Routing Protocols

In all the reactive protocols discussed, the authors pinpoint the need for caching of some desirable destinations. However, there is no formulation or criteria as to which destinations each node must cache. The protocols have been simulated either under the assumption that each node can cache ALL the destinations it has traffic for, or by running the algorithms with a few number of nodes where the issue of which destinations to keep in the cache is not so critical. Of course these simplifying assumptions may be accepted in order to see the protocol actually working, and see how mobility affects the protocol performance, but the answer to the scalability problem is not actually addressed.
Our objective is to find algorithms for intelligent caching of destinations in each mobile host. These algorithms run on top of on demand routing protocols rendering them intelligent and viable for large network sizes. The aim of the intelligent caching algorithms is fairly simple *: “For each source node at a specific time instant, keep in the node’s cache the destinations that are considered to be more expensive in terms of discovering them by flooding.”*

Furthermore, if the destination paths that are NOT cached by a node are (optimally) placed in other nodes **nearby**, we can create the illusion of a large **"virtual"** routing table that includes information about **all** the destinations, rather than the small one that is physically attached to this source node. This effect can be accomplished by the use of the "early quenching" mechanism : upon a route discovery, when a non-destination node contains the destination in its own routing table, it stops the flooding message propagation and sends its own information to the source node without interfering the destination.

There are two benefits of early quenching that make its use critical in large mobile ad-hoc networks. The first is significant flooding cost reduction : If early quenching were not used and the destination is very far from the source node, the source would have to flood a great portion of the network in order to find the destination's path and in a large network of nodes this flooding cost is unacceptable due to excessive congestion. The second benefit of early quenching is higher speed for retrieval of a specific destination path that is not cached by the host.

Unlike AODV and TORA, DSR has some desirable characteristics that may make it more suitable in a large network context. Routing loops are avoided, periodic advertisements for routes are no longer needed and the packets themselves already contain the route to the destination, thus taking off the load from the intermediate nodes to make the routing decision. Furthermore routing tables are not distributed as in the previous cases, so it's easier to perform caching of destinations. However the source routing nature of DSR is its potential doom as well, since there is a scalability problem : As the network becomes larger, control packets (which collect node addresses from each node visited) become larger and constitute a large proportion of the total traffic generated. Clearly this has a negative impact due to the limited available bandwidth. Early quenching is the only way to make DSR a scalable protocol and enjoy all its benefits that were previously mentioned. In this way, routing messages are suppressed by intermediate non-destination nodes before they become too large.

Of course one could argue that early quenching does not produce shortest routing paths. Actually this is the price all reactive protocols have to pay in order to favor from their "on demand" behavior. Shortest path constraints are many times relaxed in favor of

multiple path information, or more robust paths according to the wireless channel's current state and stringent bandwidth requirements. As a matter of fact, in many comparative studies performed [15][16], on demand protocols have exhibited competitive routes when compared to the shortest paths provided by proactive algorithms.

We proceed in the next chapter by introducing a framework and methodology for intelligent caching algorithms running on top of a DSR-like routing algorithm. This does not mean that AODV and TORA could not use similar techniques. We prefer using DSR because the routing tables are not distributed and the caching algorithms are simpler and more straightforward to understand and implement in this case.

Chapter 2

2.1 General Framework

We now introduce a framework upon which we try to quantify the notion of destination address caching in a network of nodes that form a NxN mesh network. Each node in this mesh is considered to be a host equipped with a processor, a network interface, and a memory (cache) that can hold information for up to K destinations. The nodes can only communicate directly only in a horizontal or vertical fashion, and there can be no message broadcasting along diagonals in a single step. This type of communication is often called "Manhattan Style", and the distance associated with it, is called **Manhattan Distance**.

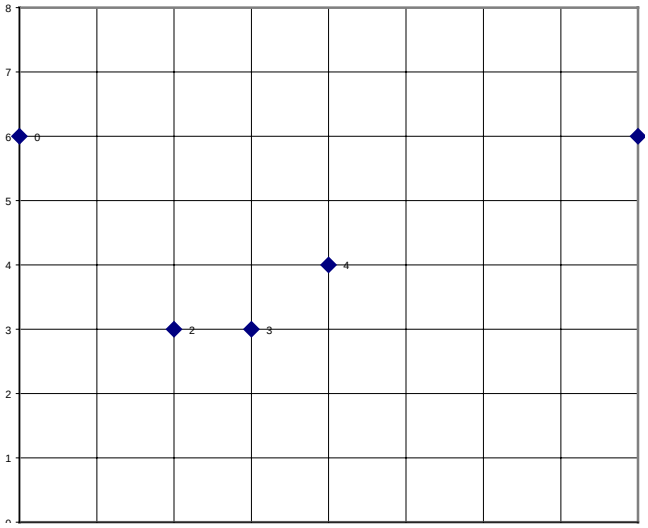

Figure 2.1. : A 9x9 mesh

The Manhattan Distance between two nodes is defined as a hop count and is the minimum number of nodes that form a path between them, including the destination node. In order to eliminate discrimination for nodes placed at the sides of the mesh and

treat the mesh network in a uniform manner, we also assume that the mesh is organized as a torroid, that is the two farmost opposite sides communicate directly with each other producing a "wrap-around" effect : the nodes situated on the same line and are at the farmost opposite sides they have a distance of one. For example in Fig. 2.1, nodes 3 and 4 have a distance of 2, nodes 2 and 3 a distance of 1, 0 and 8 a distance of 1 (due to the wraparound effects), and nodes 2 and 0 a distance of 5.

In this configuration, each node may wish to address any other node in the network with an equal probability. The cache memory of each node can hold information for only *K* out of the N^2 nodes in the network. Destinations that are not cached by the node are discovered by flooding the network.

The flooding cost between two nodes *x* and *y* with respect to their Manhattan Distance *d(x,y)* is :

$$Cost(x,y) = f(d(x,y)) = 1 + \sum_{i=1}^{d} 4i = 2d^2 + 2d + 1 \qquad (2.1)$$

This is mainly a diamond of a diagonal equal to *2d+1* nodes centered at the source node.

A node is called *"Aware"* of a specific destination if it has info in its routing table about this destination and *"Unaware"* if it doesn't. A destination is of course considered "Aware" of itself.

Each entry of the routing table of each node is kept in the form ***<destination,PathToDestination>***, where *PathToDestination* is a minimum path (sequence of nodes) to the destination. Furthermore, we assume that a mobile ad-hoc routing protocol like DSR is used in order to keep the routes consistent and taking care of the new route creations upon a node's request.

2.2 The Flooding Mechanism

If a node wants to communicate with a specific destination and does not have routing info about it, then it must initiate a route discovery and flood the network to find a path to the desired destination. Route discovery is accomplished in the following way : The source node broadcasts a flooding message ("needle packet") containing the destination address to all its neighbor nodes. Each neighbor, if it has NOT routing info on the destination, it propagates the flooding message to all its neighbors (4 of them) by appending it's address to the needle message, else it stops the flooding message propagation, appends the path that is found in its cache to the needle message constructed so far and sends this path back to the source node. This technique of a node responding with the routing path of its own cache without interfering the destination in the route discovery process, is called "early quenching" in the mobile ad-hoc networks literature, and we believe that it is an indispensable element for an ad-hoc network with many hosts. There are two variations of flooding that we will take into account in our study.

Infinite horizon flooding (IHF)

In Infinite horizon flooding, flooding is continuing endlessly, unless it is suppressed by nodes who have routing info on the destination.

Finite horizon flooding (FHF)

In Finite Horizon Flooding there is a mechanism that constrains the flooding generated by a node. This mechanism is usually implemented by a TTL (Time To Live) field on each flooding message, which is initialized by the desired *Horizon* value and is decremented when it is forwarded by the next neighbour who does not have routing info on the destination. When the counter reaches zero, the flooding message is no longer forwarded and is discarded. If a node starting with a specific horizon does not

find any info on the destination (that is, a destination aware node or the destination itself) it increments the horizon by a step (typically by 1), and retries to flood with the hope to find info on the destination. This process goes on until info about the destination is found.

2.3 The problem at hand

Early quenching seems to be a good technique to cope with small routing tables with respect to a large network. However this technique has by no means been quantified in the literature and is usually proposed as an optimization. Using early quenching, the identity of the destinations that are cached in each node is critical since this directly affects the flooding cost in the network. Confining the flooding cost as much as possible, is our primary interest. This can be accomplished by finding the best strategy of arranging the routing information at each node.

In this chapter, we assume the simplest case where **each node may wish to address any other node in the network with an equal probability**. The cache memory of each node can hold information for only *K* out of the N^2 nodes in the network. So we want to find the best *K* destination entries to be kept in each node's cache, so that the flooding cost is minimum.

Of course there will be different things to take into consideration for the two types of flooding mentioned in the previous paragraph :

In **IHF** it is obvious that the nodes containing routing info for a specific destination, should surround in some way the ones who don't, so that the flooding will eventually be stopped. Also the arrangement of the nodes with routing info for a specific destination should be symmetric in some way. In this way, *the number of the entries of*

the routing table K of each node will be equal to the number of nodes with routing info in the mesh network. In our elaboration we consider two symmetric strategies :

Diamonds strategy : Starting from the main diagonals of the mesh, routing info for a specific destination is placed diagonally in the mesh and the diagonals have a fixed distance r **units** of each other. In this way we have diamond-like regions, with “Aware” nodes on the diamond perimeter and “Unaware” nodes in the interior.

Squares strategy : Starting from the middle row and column of the mesh, routing info is placed on rows and columns which have a fixed distance r **nodes** of each other. In this way, we have rectangle and square regions, with “Aware” nodes on the perimeters and “Unaware” nodes in the interiors.

Note that the torroid structure of the mesh allows us to use the above strategies for any destination, no matter where this destination's place is in the mesh. Thus the strategies assume that the destination is the center node of the mesh and build their diamond or square structures around it.

In **FHF,** The problem of how to arrange the “Aware” nodes in the mesh for a specific destination is still at hand. Questions arise like : “Should the structure be symmetric or not? What is the optimal arrangement?” have to be confronted. Of course the “Aware” nodes for a specific destination need not be so many as in the IHF case, since there is an inherent mechanism for suppressing the flooding eventually. However, the *starting flooding horizon h1*, (which is common for all “Unaware” nodes in the mesh*) for a specific arrangement* (N,r) should be chosen in such a way so that the

average flooding cost in the mesh be minimized. Another consideration that should be taken into account is the following : If for an initial horizon no node with routing info is found, what is the *flooding step increment* that would finally yield the minimum average flooding cost?

2.4 The Infinite Flooding Horizon (IHF) problem

In this case we assume that the use of a symmetric strategy like diamonds or squares is best. We want to find which of these two routing policies yields the minimum flooding cost, for approximately the same number of K "Aware" nodes in the mesh. To do this, we first find *K* and the average flooding cost C_{avg} for a given ***N*** (number of nodes at each side of the square mesh) and a given ***r***. ***r*** is the distance between diagonals in the diamonds strategy and the distance between columns or rows in the squares strategy).

2.4.1 Diamond Strategy

For a fixed **N**, the parameter that changes is actually **r**, the distance between diagonals. Here we have to distinguish between distance that is measured in **units** and in **number of nodes.** By saying **units**, we mean that two consecutive nodes on a diagonal have an **r= 2 units**, and the midway between the two nodes is one unit. The units are introduced because a diamond routing policy may have an **r=3** for example where r includes two nodes on the diagonal plus the halfway distance between the second and third node.

In general, a diamond-shaped grid in our NxN mesh may include three types of regions :

Region A : Diamonds having side equal to **r.**

Region B : Triangles of side **(in units)** $s = \begin{cases} b, & b \neq 0 \\ r, & b = 0 \end{cases}$ **,** where $b = REM\left(\frac{N-1}{r}\right)$ (2.2)

Region C : Truncated diamonds of side **r** where one corner has been truncated by the sides of the NxN mesh.

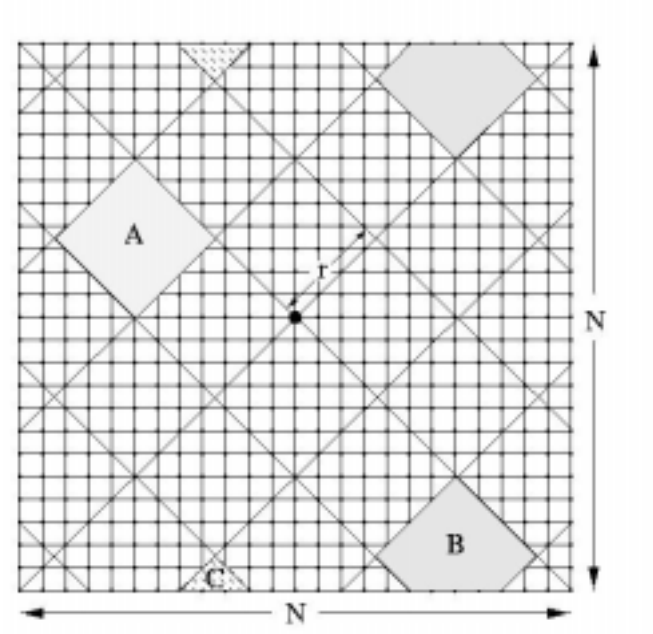

Figure 2.2. : Diamonds strategy for N=25 and r = 7

Figure 2.2 shows the regions A, B and C for the case N=25 and r=7, and what happens in case a node without routing information in each of the regions floods the network.

To distinguish between all the possible cases of (N,r), we have to take into account the following parameters :

$$b = REM\left(\frac{N-1}{r}\right), \qquad D = QUOT\left(\frac{N-1}{r}\right) \tag{2.3}$$

If $b \neq 0$, then our diamond strategy is going to have all the above types of regions, whereas if $b = 0$, the mesh is going to have regions of type A and type B only. We will

consider these two cases separately. For each case, we calculate the average flooding cost $\overline{C}$ and the capacity of each node, *K*.

Finally, we denote by :

- N_i : The number of regions of type $i, \quad i = \{A, B, C\}$
- C_i : The flooding cost of region of type $i, \quad i = \{A, B, C\}$
- l_i : The perimenter of region of type $i, \quad i = \{A, B, C\}$
- $P_i = \frac{C_i - l_i}{N^2}$: The probability that an "Unaware" node is in region of type $i, \quad i = \{A, B, C\}$
- $C_{avg} = \sum_{i \in \{A,B,C\}} N_i C_i P_i$: Average Flooding Cost
- *K* : the number of "aware" nodes about the destination which is equal to the number of cache routing table entries for each node.

CASE A : $b \neq 0$

- **r is even**

Flooding Cost Computation:

Type A : $N_A = 2D(D-1)$

$$C_A = \frac{r^2}{2} + r + 1$$

$$l_A = 2r$$

$$P_A = \frac{C_A - l_A}{N^2}$$

Type B : $N_B = 4(D+1)$

$$C_B = 2\left(\frac{b}{2}+1\right)^2$$

$$l_B = b+1$$

$$P_B = \frac{C_B/2 - l_B}{N^2}$$

Type C : $N_C = 4D$

$$C_C = \left[\left(\frac{r}{2}+1\right)^2 + \frac{br}{2} - \frac{b^2}{4}\right]$$

$$l_C = r+b+1$$

$$P_C = \frac{C_C/2 - l_C}{N^2}$$

$$C_{avg} = \sum_{i\in\{A,B,C\}} N_i C_i P_i$$

Node capacity :

$$K = 4DN + 2N - 2(r+1)D(D+1) + 3$$

- ***r is odd***

b is odd

Flooding Cost Computation:

Let $L = \frac{b+1}{2}$

Type A : $N_A = 2D(D-1)$

$$C_A = 2\left(\frac{r+1}{2}\right)^2$$

$$l_A = 2r$$

$$P_A = \frac{C_A - l_A}{N^2}$$

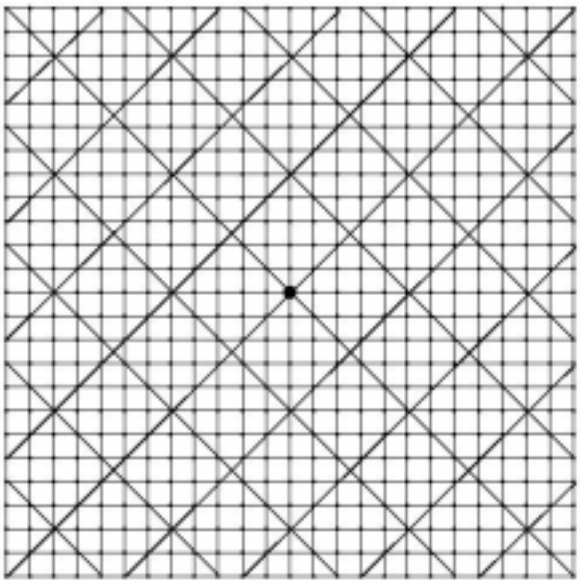

Figure 2.3. : r=5(odd) and b=5(odd).

Type B : $N_B = 4(D+1)$

$$C_B = 2L(L+1)$$

$$l_B = 2L$$

$$P_B = \frac{C_B/2 - l_B}{N^2}$$

Type C : $N_C = 4D$

$$C_C = 2\left[\left(\frac{r+1}{2}\right)^2 + Lr - L(L-1)\right]$$

$$l_C = 2r$$

$$P_C = \frac{C_C/2 - l_C}{N^2}$$

$$C_{avg} = \sum_{i \in \{A,B,C\}} N_i C_i P_i$$

Node capacity :

$$K = \begin{cases} 2N(2d+1) - (2r+1)D^2 - 2(r+1)D + 3 & , D \text{ even} \\ 2N(2d+1) - (2r+1)D^2 - 2rD + 4 & , D \text{ odd} \end{cases}$$

b is even

Flooding Cost Computation:

Let $L = \frac{b}{2}$

Type A : $N_A = 2D(D-1)$

$$C_A = 2\left(\frac{r+1}{2}\right)^2$$

$$l_A = 2r$$

$$P_A = \frac{C_A - l_A}{N^2}$$

Type B : $N_B = 4(D+1)$

$$C_B = 2(L+1)^2$$

$$l_B = 2L+1$$

$$P_B = \frac{C_B/2 - l_B}{N^2}$$

Type C : $N_C = 4D$

$$C_C' = \frac{(r+1)(r+3)}{4} + rL - L^2$$

$$C_C = \begin{cases} 2C_C' & , L \neq \frac{r-1}{2} \\ C_C' & , L = \frac{r-1}{2} \end{cases}$$

$$l_C = 2r$$

$$P_C = \frac{C_C' - l_C}{N^2}$$

$$C_{avg} = \sum_{i \in \{A,B,C\}} N_i C_i P_i$$

Node capacity :

$$K = \begin{cases} 2N(2d+1) - (2r+1)D^2 - 2(r+1)D + 3 & , D \text{ even} \\ 2N(2d+1) - (2r+1)D^2 - 2rD + 4 & , D \text{ odd} \end{cases}$$

CASE B : $b = 0$

- ***r is even***

Flooding Cost Computation:

Type A : $N_A = 2D(D-1)$

$$C_A = \frac{r^2}{2} + r + 1$$

$$l_A = 2r$$

$$P_A = \frac{C_A - l_A}{N^2}$$

Type B : $N_B = 4D$

$$C_B = 2\left(\frac{r^2}{2} + r + 2\right)$$

$$l_B = r + 1$$

$$P_B = \frac{C_B / 2 - l_B}{N^2}$$

$$C_{avg} = \sum_{i \in \{A,B,C\}} N_i C_i P_i$$

Node capacity :

$$K = 4DN - 2rD^2 - 2D^2 - 2D + 3$$

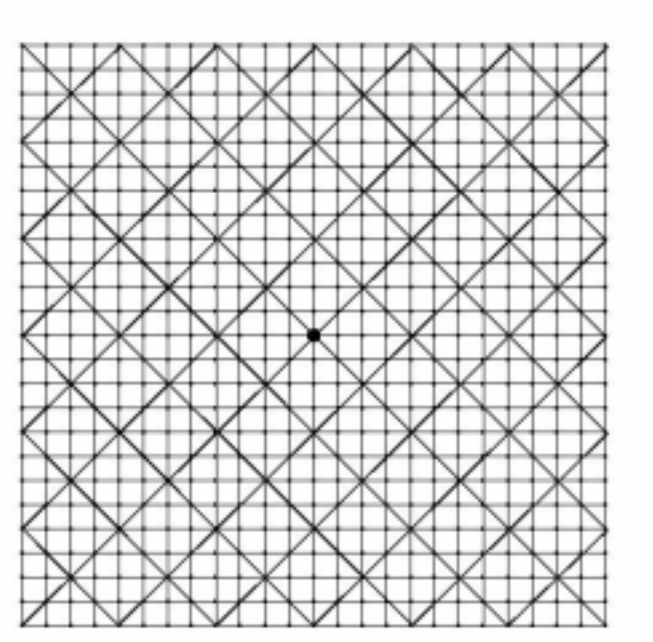

Figure 2.4. : b=0, r =4 (even) : We have only regions of type A and B.

- ***r is odd***

Flooding Cost :

Let $L = \frac{r-1}{2}$

Type A : $N_A = 2D(D-1)$

$$C_A = 2L^2 + 4L + 2$$

$$l_A = 2r$$

$$P_A = \frac{C_A - l_A}{N^2}$$

Type B : $N_B = 4D$

$$C_B = 2(L+1)(L+2)$$

$$l_B = 2(L+1)$$

$$P_B = \frac{C_B/2 - l_B}{N^2}$$

$$C_{avg} = \sum_{i \in \{A,B,C\}} N_i C_i P_i$$

Node capacity :

$$K = 4DN - 2rD^2 - D^2 - 2D + 3$$

2.4.2 Square Strategy

For a fixed **N**, the parameter that changes here is **r**, the distance between rows or columns. Here we do not use **units** but **number of nodes** to denote distance.

In general, a diamond-shaped grid in our NxN mesh may include three types of regions :

Region A : Squares having side equal to **r.**

Region B : Rectangles at the mesh boundaries that communicate with each other via the mesh wrap-around.

Region C : The 4 corner squares that are formed for every choice of **r** .

To distinguish between all the possible cases of (N,r), we have to take into account the following parameters :

$$b = REM\left(\frac{N-1}{2r}\right) \qquad D = QUOT\left(\frac{N-1}{2r}\right)$$

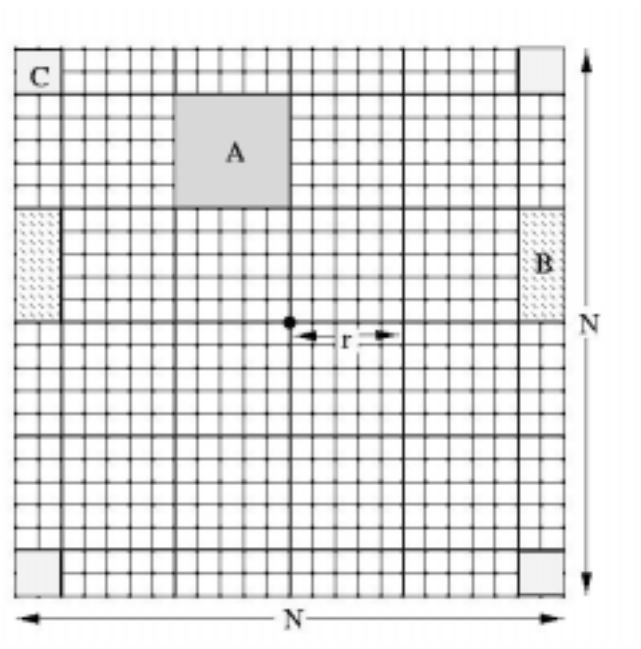

Figure 2.5. : The diamond strategy. In this case,

r=2, D = 2, b = 4

CASE A : $b \neq 0$

Flooding Cost Computation:

Type A : $N_A = 4D^2$

$C_A = (r+1)^2$

$l_A = 4r$

$P_A = \frac{C_A - l_A}{N^2}$

Type B : $N_B = 8D$

$C_B = 2\left(\frac{N-1}{2} - Dr + 1\right)(r+1)$

$$l_B = N - r(2D-1)$$

$$P_B = \frac{C_B/2 - l_B}{N^2}$$

Type C : $N_C = 4$

$$C_C = 4\left(\frac{N-1}{2} - Dr + 1\right)^2$$

$$l_C = N - 2rD$$

$$P_C = \frac{C_C/4 - l_C}{N^2}$$

$$C_{avg} = \sum_{i\in\{A,B,C\}} N_i C_i P_i$$

Node capacity :

$$K = 2N(2D+1) - (2D+1)^2$$

CASE B : $b = 0$

Flooding Cost Computation:

Type A : $N_A = 4(D-1)^2$

$$C_A = (r+1)^2$$

$$l_A = 4r$$

$$P_A = \frac{C_A - l_A}{N^2}$$

Type B : $N_B = 8(D-1)$

$$C_B = 2(r+1)^2$$

$$l_B = 3r + 1$$

$$P_B = \frac{C_B/2 - l_B}{N^2}$$

Type C : $N_C = 4$

$$C_C = 4(r+1)^2$$

$$l_C = 2r+1$$

$$P_C = \frac{C_C/4 - l_C}{N^2}$$

$$C_{avg} = \sum_{i \in \{A,B,C\}} N_i C_i P_i$$

Node capacity :

$$K = 2N(2D-1) - (2D-1)^2$$

2.4.3 Comparison of the diamonds and squares strategies

We performed a comparison between the two strategies analyzed above. We used a mesh of *N=257* (i.e 66049 nodes). The graphs below show for both strategies how the cache capacity *K* and average flooding cost per route discovery change for different values of *r*. In the diamonds strategy, as the diamond size *r* decreases, the flooding cost decreases at the expense of an increase in the cache size *K*. As *r* decreases, the cache capacity seems to increase exponentially with a threshold value of *r=20* approximately. The flooding cost is decreasing in a more mild manner. This shows that one can pick up a value *r=50* approximately, and get a flooding cost of under 2% of the total network per route discovery, with a cache which has a capacity of less than 5000 destinations (this accounts for the $5000 \times \frac{100}{257^2}\% = 7.5\%$ of the whole network). We will see later that by using Finite Horizon Flooding (FHF) we can achieve even smaller caches.

The squares strategies curves seem stranger than the diamonds ones. In the capacity graph for example, we see that for many values of *r*, the capacity remains the same.

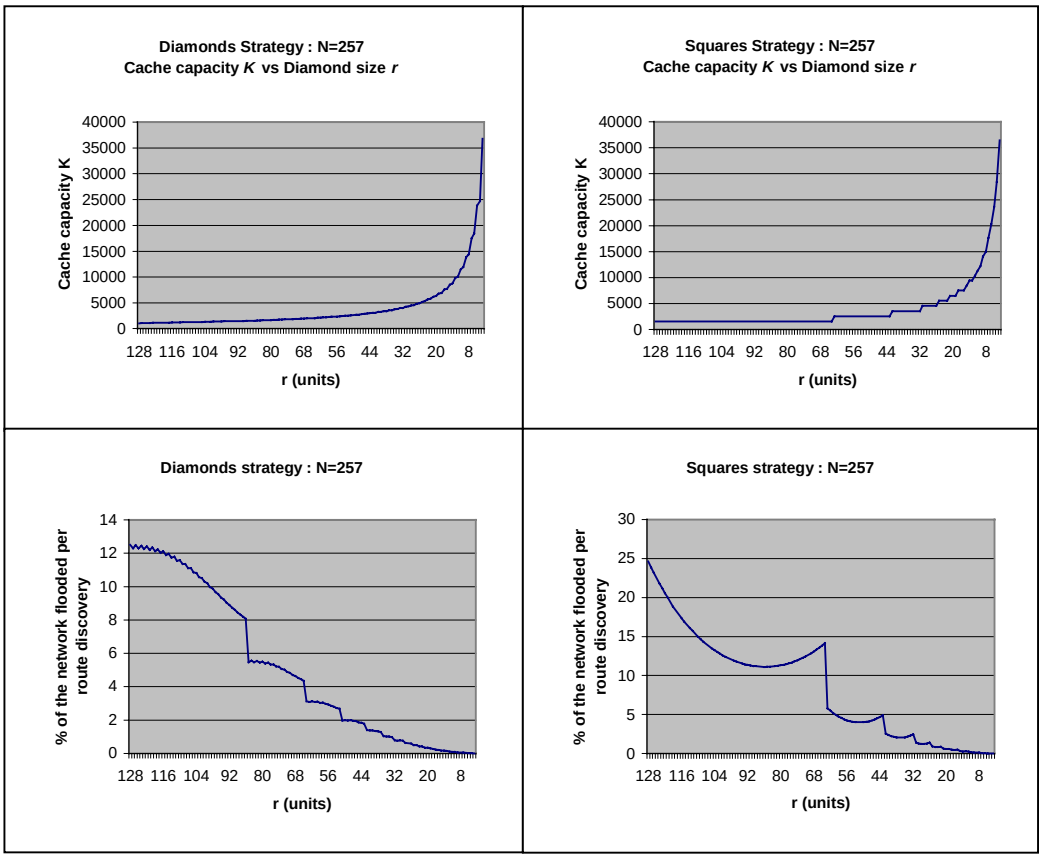

Figure 2.6. : Diamonds v.s Squares

This happens because of the way squares strategy is implemented. If we go back to the formulas of the squares, we see that the capacity *K* is defined by:

$$K = 2N(2D+1) - (2D+1)^2 \text{ and } K = 2N(2D-1) - (2D-1)^2$$

which depend on $D = QUOT\left(\frac{N-1}{2r}\right)$ **only.** So for various successive values of *r* which can be seen as *"batches", D* can be the same and hence the capacity *K* can be the same as well. Now in a batch of *r*'s which have the same K, the cost function is concave up as is shown in the figure. As soon as a new batch of r's begins, the cost takes a sudden dip and it follows again a concave route. So the general tendency of this curve is still decreasing of the cost as *r* decreases. The concave behavior of the curve for each "batch" is happening because of the wrap-around effect : As the square size *r* decreases, the cost decreases up to an "equilibrium" point r, where all the regions are balanced. If *r* decreases further, then regions B and C have an equal flooding cost than a previous flooding cost of region A before the equilibrium point. Effectively, after the equilibrium point, the flooding costs of regions (B,C) and A are interchanged!

In order to compare the two strategies, we first found the equilibrium points of the batches and their corresponding cache capacities *K* and (locally minimum) flooding costs. Then we found in the squares strategies *r*'s with cache capacities approximately equal to the ones obtained by the squares strategies and compared the two strategies flooding costs. The results are in Figure 2.7 :

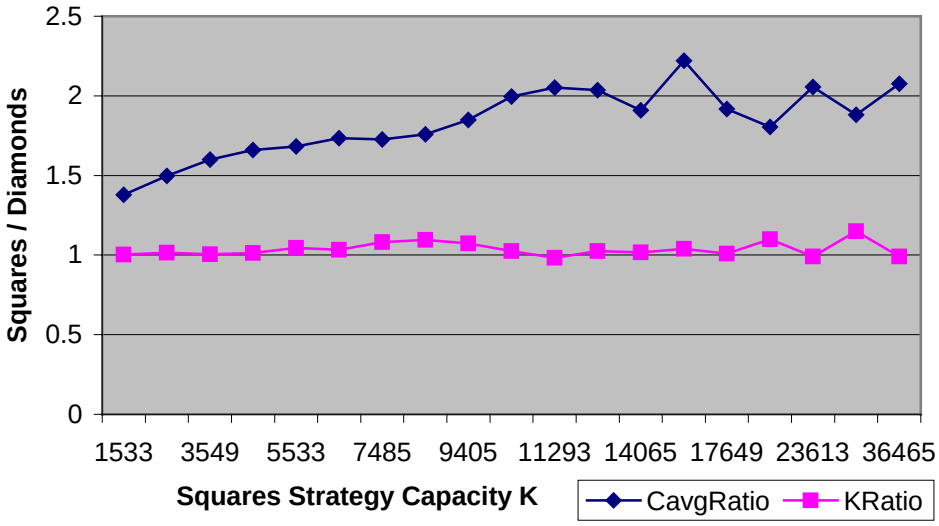

Figure 2.7. : Capacity and Average Cost Ratios for the two strategies (N=257)

For ratios of $\frac{Ksquares}{Kdiamonds} \approx 1$, we see that the ratio $\frac{CavgSquares}{CavgDiamonds}$ ranges from 1.37 to 2.22. Thus in every case, for the same memory capacity, the squares strategy has roughly 1.8 times higher flooding cost than the diamonds strategy on the average.

2.5 The Finite Horizon Flooding (FHF) problem

We now consider the Finite Horizon Flooding case. Suppose that for a specific destination we have a number of "aware" nodes that keep routing info on the destination and some "unaware" that do not. We argue that by placing the aware nodes as centers in diamond regions of side r yields the strategy with the minimum flooding cost. Figure 2.8 shows such a placement where the diamonds have side *r=5* nodes (or equivalently *r=8* units). The nodes shown as larger points on the mesh are the "aware" nodes and the rest

of the nodes are the "unaware" ones. Each "aware" node serves as a centroid for a region of "unaware" nodes.

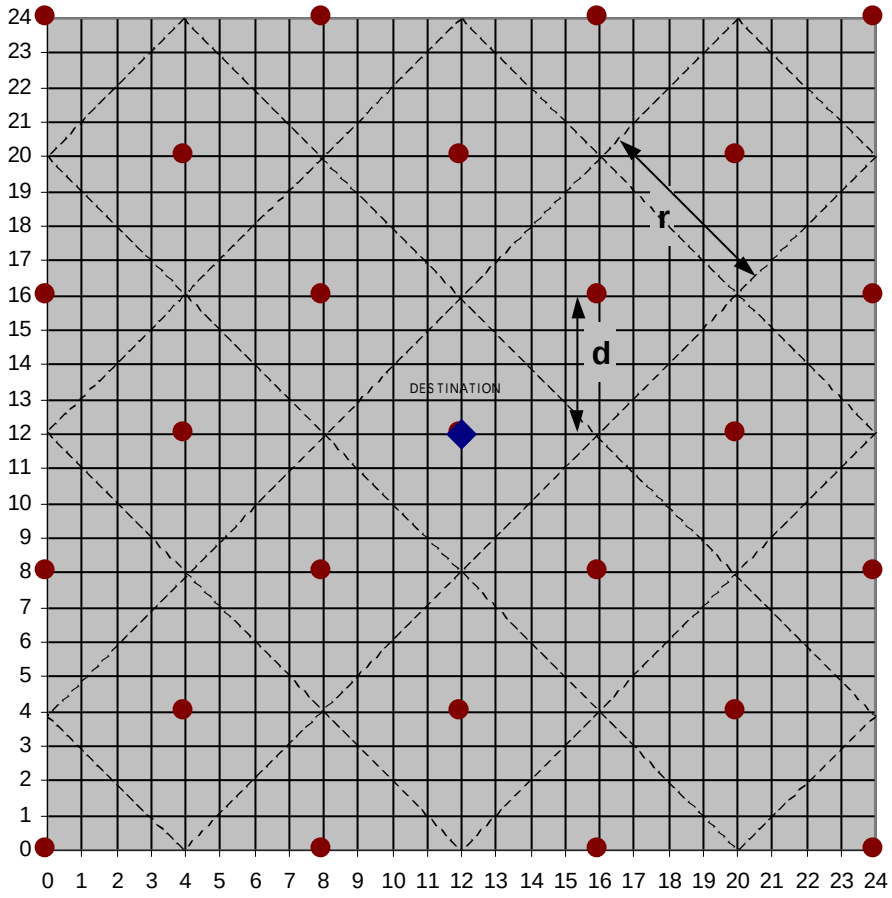

Figure 2.8. Diamond strategy in the finite horizon flooding context.

Each one of the nodes in a region will find routing info about the destination from its corresponding centroid node. By using the such a diamond strategy the regions associated with each aware node are diamonds. However if we placed the nodes with routing information in an arbitrary way, we could have regions of irregular shape. The following theorem shows that for a number of nodes that can belong to a region, the region shape that yields the minimum flooding cost is the diamond one.

Theorem **:** *If we have* $L = 2d^2 + 2d$ *nodes that do NOT contain info for a specific destination, then by arranging them in diamonds of side* $r = d + 1$ ***nodes****, centered at a node with routing info, will minimize the sum of the (manhattan) distances of all the L nodes from the center node.*

Proof : Assume a random arrangement of the L nodes with respect to the reference node with the routing info. Suppose that x_i is the number of nodes that have manhattan distance *i* from the reference node. Then the sum of the distances z is given by :

$$z = \sum_{i=1}^{M} i \cdot x_i$$

Also we have that,

$$\sum_{i=1}^{M} x_i = L$$

Where *M* is the maximum distance where we find an "unaware" node from the reference node. Note that *M* cannot be less than *d* because then we would be able to fit the *L* nodes in a smaller diamond of side *r=M+1* something that is impossible given that $L = 2d^2 + 2d$. The maximum number of nodes that may have manhattan distance *i* from the reference node, are *4i*.

Thus, we have the following constraints :

$$x_i \le 4i, \quad i = 1, \cdots, M \quad \text{where } M \ge d \quad (3)$$

Suppose now that the arrangement that yields the minimum sum of distances from the center node happens for $M > d$. We now start enumerating the x_i s. It is obvious that for this optimum placement $x_i \ge 1, \quad d+1 \le i \le M$. Suppose without loss of generality that $x_{d+1} = 1$, that is there is one node that has a distance *d+1* from the center node. The fact

that $L = 2d^2 + 2d$ means that if the nodes were arranged in a diamond surrounding the center, they would cover the whole diamond. Now since $x_{d+1} = 1$ this means this point lies outside the diamond region and that it should be placed inside it in order for the total sum of the distances to be minimum. Therefore we arrive in a contradiction because we assumed that $M > d$ for a minimum cost strategy. Therefore *M* must be equal to *d*, and since $L = 2d^2 + 2d$ the optimum placement of the unaware nodes is in the diamond around the center node.

2.5.1 Solution to Limited Horizon Problem using the Diamond Strategy

In our study, we assume for simplicity that $REM\left(\frac{N-1}{r}\right) = 0$. This condition ensures that we have only diamonds of side r **units** (regions type A) and triangles of side r at the mesh boundaries, that form diamonds due to the mesh wrap around effect (regions type B). We also assume that **r is even**. The number of nodes with routing info in the mesh (i.e. the cache capacity of each node) is then given by :

$$K = 2D^2 + 2D + 1, \text{ where } D = QUOT\left(\frac{N-1}{r}\right)$$

Figure 2.8. shows the case where *N=25* nodes, *r=8* units, *D=3*. Therefore the number of nodes with routing information is *K=25* (including the destination).

Since we have the limited horizon problem, it suffices to place the routing information only at the center of each diamond. According to the previous theorem, this strategy is the optimal arrangement of routing information in the mesh.

2.5.2 The "best" starting flooding horizon h_1^*

However, even if we found that a diamonds policy is optimal, we must now find what happens for different diamond strategies *r*, and how we can obtain minimal flooding costs. So we consider the following problem :

Given a diamond strategy (N,r) find the best starting flooding horizon ***h_1,*** *so that the flooding cost is minimized on the average.*

We assume that if a source node does not find the destination starting with a specific horizon ***h_1***, it increments the horizon by a *retry step* ***s*** and retries to flood with the hope to find the destination. This process goes on until a node with a routing info on the destination is found.

To solve the above problem we address the following two subproblems :

- Given (N,r), find the number of "unaware" nodes (nodes without routing info) ***N_i*** that can find routing info in AT LEAST ***i*** steps, where $1 \le i \le r/2$.

We start in the mesh by forming diamonds of side r, centered at the "aware" nodes (nodes with routing info) of the mesh. Thus we have non-overlapping diamond regions centered at the "aware" nodes. In each one of these regions the **maximum distance** that a node can reside with respect to the center, is **r/2.** $\mathbf{N_i}$ is then given by the relation :

$$N_i = \begin{cases} (2D^2 - 2D + 1)4i + 4(D-1)(2i+1) + 4(i+1), & 1 \le i \le r/2 - 1 \\ 4\left(N - \frac{r}{2} + N(D-1) - \frac{r}{2}(D^2-1)\right) - 2D^2 - 2D, & i = \frac{r}{2} \end{cases}$$

- Determine the flooding cost for each "unaware" node in the mesh, when starting flooding horizon $\boldsymbol{h_1}$ is used ($1 \le h_1 \le r/2$). For the time being we assume that the *retry step* ***s*** equals to one.

The cost is a function of two variables :

i : All possible distances from a node with routing info. The maximum ***i*** is ***r/2*** since this is the distance (in nodes) from the center of each diamond of side r (in units), to one of its corners.

$\boldsymbol{h_1}$: All possible starting horizons ($1 \le h_1 \le r/2$).

$$Cost(h_1, i) = \begin{cases} 2h_1^2 + 2h_1 + 1, & h_1 \ge i \\ \sum_{h=h_1}^{i} (2h^2 + 2h + 1), & h_1 < i \end{cases}$$

The above equation says that if a node that has distance ***i*** from a node with routing info starts with horizon $\boldsymbol{h_1}$, then if $h_1 \ge i$ then this starting horizon suffices to find routing info. If $h_1 < i$, the node has to restart flooding by incrementing the horizon by *retry step* ***s=1,*** until the starting horizon becomes i and the routing info is eventually found.

Having found $\boldsymbol{N_i}$ and Cost(h_1,i) for a specific (N,r), the average cost (in nodes) for a starting horizon $\boldsymbol{h_1}$, is given by :

$$Cavg(h_1) = \sum_{i=1}^{r/2} N_i \cdot Cost(h_1, i)$$

The preferred starting horizon $\boldsymbol{h_1}$ will be the one that minimizes the average cost :

$$h_1^* = \arg\min\{Cavg(h_1)\}$$

2.5.3 The optimum starting flooding horizon using *retry horizon step s=1*

When the retry step is 1, simulation results based on the elaboration above, showed that for every pair (N,r), the best starting flooding horizon is : $h_1^* = \frac{r}{2} - 1$

This is depicted in the following graph, where we monitored the average flooding cost per route discovery as a function of the starting horizon h_1 for various diamond sizes (represented by *r*).

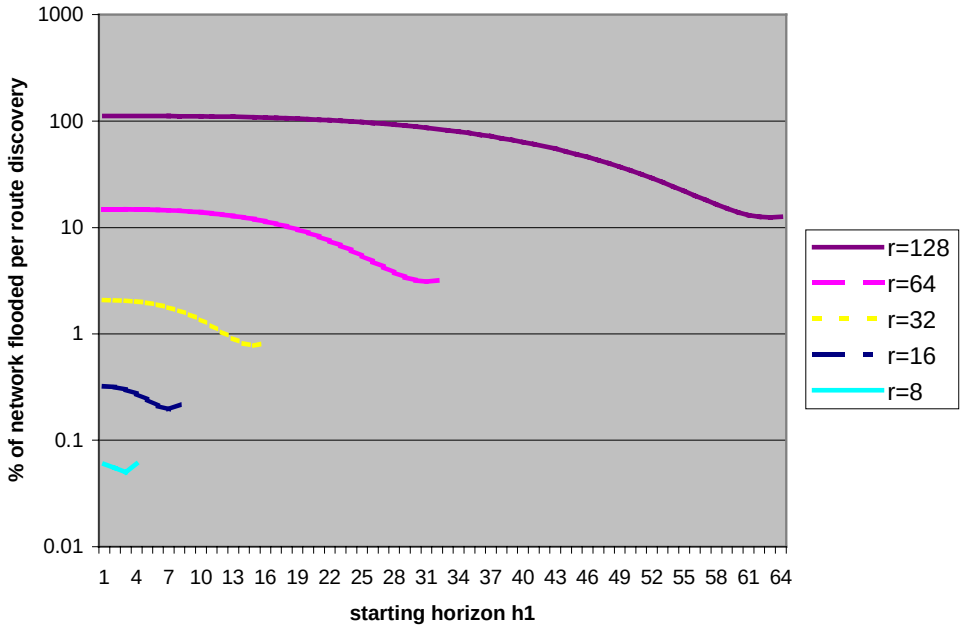

Figure 2.9. : s=1, r and h1 vary

Figure 2.9 shows that for every diamond size r, as we increase h_1, the average flooding cost is relatively stable up to $h_1 = \frac{r}{4}$, and then starts decreasing rapidly achieving the

minimum value always at $h_1^* = \frac{r}{2} - 1$. The above graph also tells us that $r = 8$, achieves the less average flooding cost than other diamond sizes. So one could suggest using a very small value of *r* when deciding on the diamond strategy. The next graph shows why this suggestion would not be a very good idea :

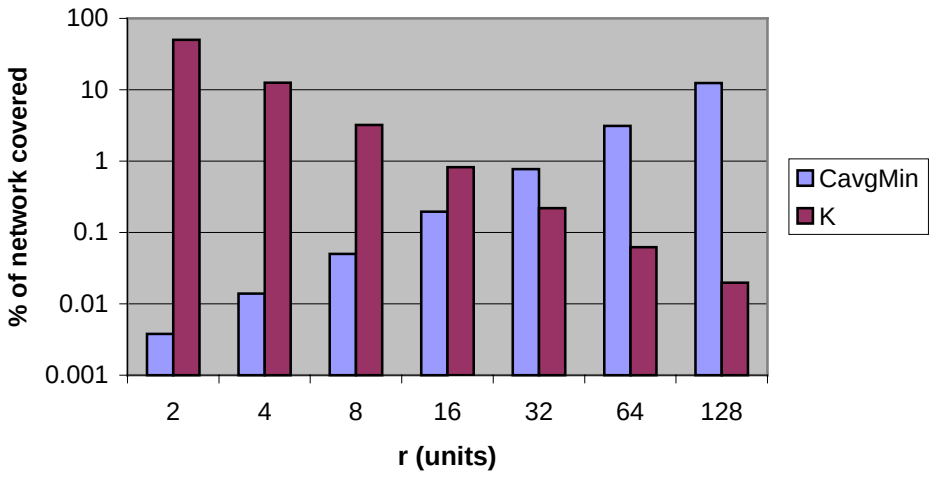

Figure 2.10. : % of network cached (K) and flooded per route discovery (CavgMin)

Figure 2.10 shows **for each strategy *r*** what percentage of the network needs to be cached by each node and what is the minimum average flooding cost achieved by setting the initial flooding horizon to $h_1^* = \frac{r}{2} - 1$ and the horizon step to $s = 1$. We see that choosing $r = 2$, the minimum average flooding cost per route discovery (achieved by setting $(h_1^*, s) = (1,1)$), covers 0.0037% of the whole mesh, but each node must keep in its cache 50% of the total network (this percentage is $\frac{K}{N^2} \times 100\%$). Clearly this is an

unacceptable cache size if we have small cache memories available relatively to the whole network. By setting $r = 8$, the flooding cost is 0.05% and the cache size must track 3.19% of the network. A reasonable choice of strategy seems to be $r = 16$ or $r = 32$ where both the cache size and the flooding cost are kept below 0.1% of the network. Of course by choosing $r = 128$, we get a very small cache, but the flooding cost is unacceptably high (12.4%).

2.5.4 The optimal retry horizon step *s*

When routing information is not found by flooding with a specific horizon *h*, the retry horizon step *s* is defined as the step by which the current horizon is incremented in order to find the routing information with the next flooding. In this experiment, we varied the retry horizon step in order to see how much it affects the flooding cost and if possible, find if there is a specific *s* that achieves the minimum average flooding cost for different strategies *r* and different initial horizons h_1.

Figure 2.11 shows how s affects the flooding cost for different h_1 when *r=16.* First of all we see that for $h_1 = \frac{r}{2} = 8$, the change in the retry step does not affect the flooding cost at all. This is because an $h_1 = \frac{r}{2}$ or more, guarantees for each "unaware" node that the routing information will be found. Suppose we fix $h_1 = 4$, for example. The graph shows that the flooding cost is not an increasing or decreasing function of the step *s*. This is because a small *s* may not result in a successful flooding (by success we mean finding a node that contains the destination in its cache) and a very large *s* guarantees to find the destination but does excessive flooding. This excessive flooding situation is seen for

$h_1 = 4$ after the minimum flooding cost is achieved for *s=5*. After this value, the flooding cost always increases with the flooding step. Also observe that for every h_1 in the graph, the minimum cost is achieved for $s = \frac{r}{2} - h_1 + 1$. However there is another parameter that comes into play, and this is the speed of finding the routing information.

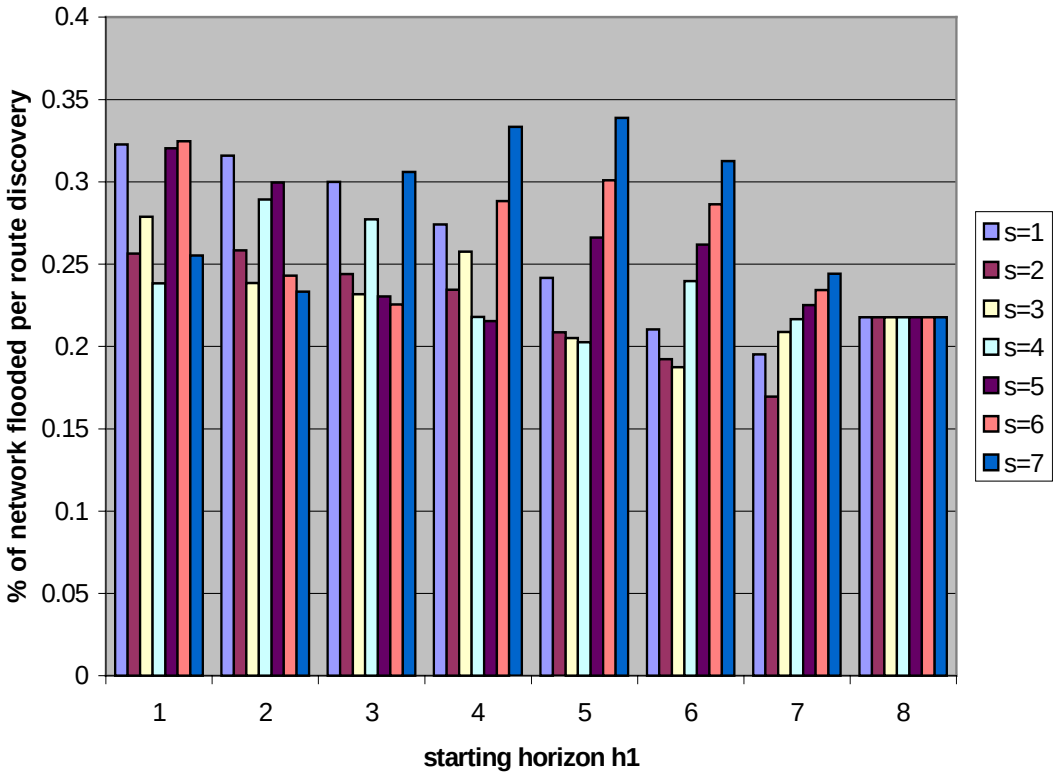

Figure 2.11. : r=16, s and h1 vary

Small *s* are therefore desirable, because if the routing information is found after the increment and new flooding, the cache update of the source node will be faster. Thus we may want to look at smaller values of *s* that produce flooding costs close to the minimum one. For $h_1 = 4$, this value of "desirable small" *s* is *s=2* as is clearly shown in the graph.

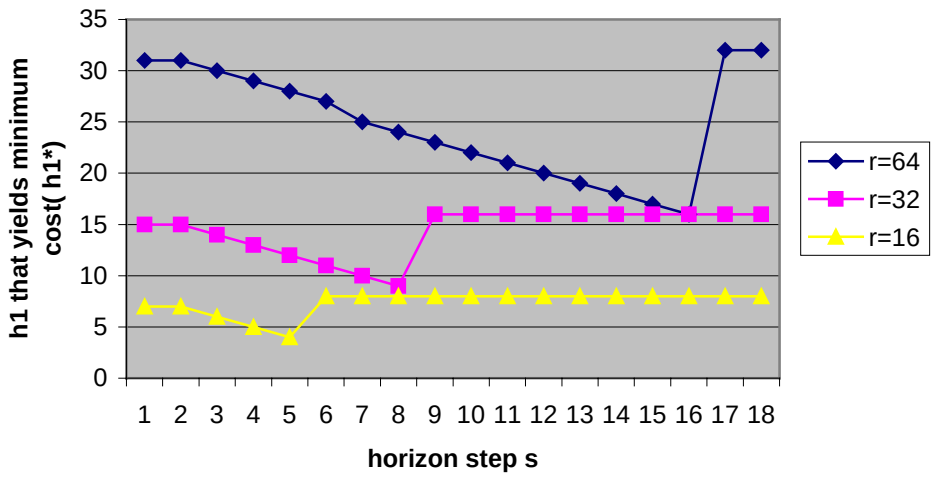

Figure 2.12. : For several (N,r) and at s=2 the best starting horizon is always r/2-1!

Figure 2.12 shows which are the h_1 that yield the minimum flooding cost (i.e the h_1^*) for different values of the step *s*. What we see is that for steps s=1 and s=2 $h_1^* = \frac{r}{2} - 1$ always. Then h_1^* decreases and attains a minimum value of $h_1^* = \frac{r}{2} - s + 1$ at $s = \frac{r}{4} + 1$. After this value h_1^* makes a sudden jump at $h_1^* = \frac{r}{2}$ and remains fixed ever after. Note that this result is compliant with the one of the previous figure where we varied *s* for a fixed h_1. In both graphs we may observe that the minimum cost is attained under the condition: $h_1 + s = \frac{r}{2} + 1$.

Figure 2.13. illustrates the effect of the change in *s* when the optimum h_1^* has been selected. The first thing that we observe is that there is a small variability of the minimum average cost obtained in each case. The second observation is that for ***s=2*** we

obtain THE minimum of all minimum average costs for every value of diamond strategy *r*.

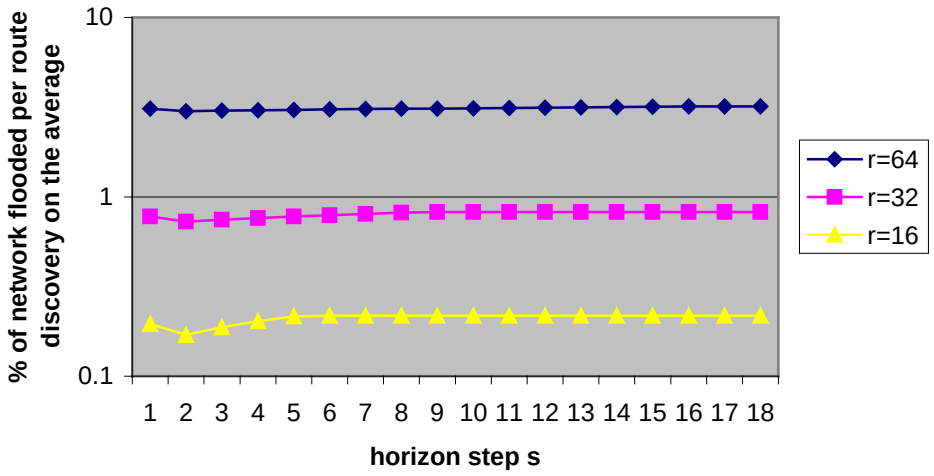

Figure 2.13. : For several (N,r) the best horizon step is s=2

Furthermore, by combining the information in both of the above graphs, we conclude that for *s=2* the best h_1^* is **always** the value $h_1^* = \frac{r}{2} - 1$, for any *r*. This means that, assuming a large network (N is 257 in our experiment), for any diamond strategy *r* the pair h_1, *s* that achieves the minimum flooding cost per route discovery is $h_1^* = \frac{r}{2} - 1$ and *s=2*.

Overall in the FHF case, if we exclude some extreme values of diamond sizes *r*, the diamond strategy achieves in general very small flooding costs per route discovery and very small cache sizes compared to the network size. This is of course due to the optimality of the diamond strategy, the symmetric nature of the torroid structure and the fact that each source node wants to communicate with all the other nodes with equal

probability. In the next chapter, we will see that this "rosy" picture fades away, as we introduce the more realistic case of non-uniformity in each node's access pattern.

Chapter 3

3.1 Introduction

In the previous chapter, we assumed that each node has traffic for all the other nodes in the network, and wants to address each one of its destinations with equal probability. Under this assumption, and based on the symmetric structure of the torroid mesh, we came up with very small memory capacities for very small flooding costs. This was accomplished by using Finite Horizon Flooding and a diamond strategy of placing the nodes with routing information.

However in a practical situation the above assumption will not be the valid, since the network is supposed to be large and each source node may in general generate traffic for a subset of the whole network in a non-uniform way. We generalized the above case even further by assuming that each node generates traffic for only M out of the N^2 nodes in the network. The possible M destinations may be distributed in the network in an arbitrary way, and they need not be close to the source node. It is obvious that the symmetric solutions of chapter 2 do not apply in this case, simply because we cannot treat each destination in a uniform way: Now not all (source) nodes in the network generate traffic for this specific destination. So *the cache size of each node is **not** equal to the number of nodes holding routing information about a specific destination*. Therefore it is obvious that the approaches used will not be able to take into account the symmetry of the torroid structure. This is not so bad as it may sound, because the algorithms are topology independent and can be applied to any network other than a mesh.

We now proceed to formulate the problem by introducing some notions that we are going to use in the algorithm descriptions:

Given a source node *i*, we define:

- $\tilde{D}_i$: The set of destinations node *i* wants to address. Obviously $\left|\tilde{D}_i\right| = M$.
- $D_i^{(t)}$: The set of destinations that are in node *i*'s cache at time instant *t*. Obviously $\left|D_i^{(t)}\right| = K$. We also define the network state by the $N^2 \times 1$ vector $\mathbf{D}^{(t)}$. The elements of this vector are $D_i^{(t)},\ i = 1 \ldots N^2$.
- $\overline{D_i^{(t)}}$: The set of destinations that are not in node *i*'s cache at time instant *t*. Obviously $\left|\overline{D_i^{(t)}}\right| = M - K$.
- Given the above definitions we have : $D_i^{(t)} \cup \overline{D_i^{(t)}} = \tilde{D}_i$.

Given a destination *j*, we define:

- $\tilde{S}_j$: The set of source nodes that wish to address destination *j*.
- $S_j^{(t)}$: The set of source nodes that have destination *j* in their cache at time instant *t*. Generally $S_j^{(t)} \subseteq \tilde{S}_j$.
- $\overline{S_j^{(t)}}$: The set of source nodes that do not have destination *j* in their cache at time instant *t*.

Given the above definitions : $S_j^{(t)} \cup \overline{S_j^{(t)}} = \tilde{S}_j$.

- $d_{ij}^{(t)}$: The minimum distance that node *i* will have to flood in order to find info about destination *j*, given the network state at time *t*. If *d(x,y)* denotes the Manhattan distance between two arbitrary nodes *x* and *y*, $d_{ij}^{(t)}$ is defined as :

$$d_{ij}^{(t)} = \min\left\{d(i,j),\ \min_{k}\left[d(i,k)\right],\ \forall\, k \in S_j^{(t)}\right\}$$

- We define $a_{ij}^{(t)} = \arg\min\left\{d(i,j),\ \min_{k}\left[d(i,k)\right],\ \forall\, k \in S_j^{(t)}\right\}$ to be the corresponding node that achieves $d_{ij}^{(t)}$, i.e. the node that is used by the source node *i* to find information about destination j.

3.2 A Lower Bound on the expected flooding cost of any caching algorithm

Suppose a node *i* wants to find routing information about a destination *j* that is not in its cache. In order to find $d_{ij}^{(t)}$ the source node *i* must search the set $S_j^{(t)}$ and find the node that is closest to it. However, generally $S_j^{(t)} \subseteq S_j$. This means that a node *k* belonging to $\overline{S_j^{(t)}}$ may be closer to *i* than the ones that are currently in $S_j^{(t)}$, but according to its own criteria it has pushed *j* out of its cache. So if **for each** destination $j \in \tilde{D}_i$ we **independently** assume that $S_j^{(t)} = S_j$, then we are going to find the minimum possible $d_{ij}^{(t)}$ that can be achieved. By placing in the cache the destinations that have the *K* maximum $d_{ij}^{(t)}$, we will have the best cache possible for each node *i*.

Of course the above assumption is not realistic since we cannot have $S_j^{(t)} = S_j$ for **every** destination *j*. If this were the case, then the cache size *K* would be equal to *M*, and the problem of caching would not exist at all. Note that the smaller the cache size *K* is, the smaller $S_j^{(t)}$ becomes and the more reality is away from the above assumption. No matter how unrealistic, this rationale will provide a **lower bound on the expected cost** achieved by any of the algorithms we introduce. Thus we will be able to see how good the algorithms perform compared to an "ideal" case.

3.3 The class of BSBC algorithms

We now introduce the class of "Best-State/Best Cost" (BSBC) algorithms that are used to solve the problem of cache entry assignment.

As we saw in the previous section, "the state" of the network at a specific time instant, consists of the contents of the caches of all the nodes at this time instant. The network state is represented by the vector $\mathbf{D}^{(t)}$. The BSBC algorithms are iterative, and in each step they try for each node to keep the K maximum cost (out of M) destinations in the cache, based on the current state of the network. The algorithms terminate when the "best state" has been reached, i.e for each node, *the maximum flooding cost entries are in the cache and the minimum are not.* The difference in the algorithms lies in the "maximum cost" criterion way a node employs to decide which destinations to keep in the cache.

3.3.1 The Local Best State/Best Cost (L-BSBC) algorithm

We call the algorithm of this section *Local BSBC* (*L-BSBC*). It is local in the sense that node *i* decides about which destinations to cache by taking into account only it's cost of finding the destinations. In the next section we will see *Global BSBC* (*G-BSBC*) algorithm which takes into account the global network state and generally achieves slightly better results.

The L-BSBC algorithm iterates over all nodes in the mesh. For each node *i*, and for every entry *y* NOT in Cache of node *i*, we check the following : Suppose *i* flooded the network in order to find *y*, then information about *y* would be found with a flooding cost *costy*. We compare *costy* with the minimum flooding cost entry *z* in the cache. If *costy* is greater than the flooding cost of this cache entry, then we insert *y* in the cache, and drop *z* out of the cache. Next we update the network state as follows : In case there was another

host *j* whose flooding cost to find *z* depended on node *i* (that is, node *i* was closer to *j* than any "aware" node about *z*), then we have to update the flooding cost about *z* for any such host *j*. In addition, since node *i* has now info about destination *y*, there must be an update in the network state that reflects this insertion. So if some node has currently *y* in its cache, it must check whether the addition of *y* in *i*'s cache has an impact on the flooding cost of finding *y*, and do the appropriate cost update.

We denote by $m_i^{(t)}$ the minimum cost in $D_i^{(t)}$, and by $arg(m_i^{(t)})$ the corresponding destination that has this minimum cost.

Initialization:

for each node *i*

{

$D_i^{(0)}$ = any K elements from $\tilde{D}_i$.

$\overline{D_i^{(0)}} = \tilde{D}_i - D_i^{(0)}$

}

t=1*;*

Iterations :

Repeat

{

for each source node *i*

{

for each destination *j* in $\overline{D_i^{(t)}}$

{

if ($d_{ij}^{(t)} > m_i^{(t)}$)

{

$h = arg(m_i^{(t)})$

$\overline{D_i^{(t+1)}}$ = put h in $\overline{D_i^{(t)}}$ in place of j

$D_i^{(t+1)}$ = put j in $D_i^{(t)}$ in place of h

for each node $k \neq i$ in $\tilde{S}_h$

{

if ($a_{kh}^{(t)} == i$) /*node i was used by k to reach destination h*/

{

Find $a_{kh}^{(t+1)}$ and $d_{kh}^{(t+1)}$ according to $\mathbf{D}^{(t+1)}$

}

}

for each node $k \neq i$ in $\tilde{S}_j$

{

if ($d(k,i) < d_{kj}^{(t)}$)

{

$a_{kj}^{(t+1)} = i$

$d_{kj}^{(t+1)} = d(k,i)$

}

}

$t = t + 1$;

}

}

}

} until ($\mathbf{D}^{(t+1)} == \mathbf{D}^{(t)}$) /*"Best State" has been reached*/

Remarks:

An important parameter in the algorithm's operation is the number of iterations needed in order to reach the "Best State". The number of iterations is proportional to N and M and inversely proportional to K. As the size of the network increases, the best state takes more steps to be reached. As M increases and K is fixed, then more checks (M-K) are done per node in each iteration and it takes longer for the network to settle at its "best state". As the size of each node's cache increases, the best state can be reached faster, since there are not as much cache updates per node in each iteration. Finally, the relative topology of the sources and their sets of destinations, is another factor that plays a significant role in the algorithm's convergence. (However in our experiment we have generated M random possible destinations for each node, and thus, this factor will not be taken into account.)

3.3.2 The Global Best State/Best Cost (G-BSBC) algorithm

The previous algorithm does not take into account the global state of the network, in the sense that each source node decides what to place in its cache based on its own cost to find a destination given the current network state. We therefore invented an algorithm where a node decides the cache placement based on what would be the effect to the whole network from its action. This is accomplished by having at each iteration each

source node **speculating** for each destination what would be the global effect of putting it in its cache or not. It then places in the cache the K "best" destinations according to the previous speculations. The "best" destinations are defined in the following way :

"Given the network state at time t $\mathbf{D}^{(t)}$, for each source node i and for each destination $j \in \tilde{D}_i$, find the **difference** in **global** cost of finding destination j incurred, by **speculating** a placement of destination j **in** and **out** of i's cache. Then place in i's cache the K destinations with the maximum speculative differences."

Speculation is performed in the following way : for a given source node i and a destination $j \in \tilde{D}_i$, we create a speculative network state $\hat{\mathbf{D}}^{(t)}$ by taking j off i's cache if j **is** currently in the cache, or by putting j in i's cache if j **is not** currently in the cache. Then $\hat{d}_{ij}^{(t)}$ is the minimum distance that node i would have to flood in order to find info about destination j, according to the previous speculation.

The algorithm is summarized in the following pseudocode fragment:

Initialization :

for each node i

{

$D_i^{(0)}$ = any K elements from $\tilde{D}_i$.

$\overline{D_i^{(0)}} = \tilde{D}_i - D_i^{(0)}$

}

t=1;

Iterations :

Repeat

{

for each source node i

{

for each destination j in $\tilde{D}_i$

{

if ($j \in D_i^{(t)}$) /*j is in cache of node i */

Create speculative network state $\hat{D}_i^{(t)}$ by taking j off cache of node i.

else

Create speculative network state $\hat{D}_i^{(t)}$ by putting j in cache of node i.

$sdiff_{ij}^{(t+1)} = \left| \hat{d}_{ij}^{(t)} - d_{ij}^{t} \right|$

for each source node s in S_j

{

if ($j \in \overline{D_s^{(t)}}$) /*j is not in cache of node s */

{

if ($a_{sj}^{(t)} == i$) /* if s used i to access destination j */

{

$sdiff_{ij}^{(t+1)} = sdiff_{ij}^{(t+1)} + \left| \hat{d}_{sj}^{(t)} - d_{sj}^{(t)} \right|$

}

}

}

}

}

Put in the Cache of node *i* the K destinations with the maximum $sdiff_{ij}^{(t+1)}$.

The new network state $\mathbf{D}^{(t+1)}$ is formed by taking into account the new entries of node *i*'s cache.

t = *t* + 1;

}

} until ($\mathbf{D}^{(t+1)} == \mathbf{D}^{(t)}$) /*"Best State" has been reached*/

Remarks : This algorithm is expected to provide better results than the previous one since it accounts for the whole network state when it decides on the cache entries for the new state. However the global state introduces an increase in computation for decision of each node *i*. In the previous algorithm, for each source node *i* we consider only the M-K destinations that are not in its cache. In this algorithm we consider all M destinations for node i. The computations per destination are also increased. In the previous algorithm we have to consider only the cost for node *i* to the destination *j*. In this algorithm we have to consider the sum of $\overline{S_j^{(t)}}$ sources to the destination *j*, so we have an increase of computations per destination decision of a factor of $\overline{S_j^{(t)}}$. So, a coarse approximation for the total increase in computation complexity for a decision per node *i* for a specific network state $\mathbf{D}^{(t)}$ is :

$$\alpha^{(t)} = \frac{\sum_{j=1}^{M} \overline{S_j^{(t)}}}{\sum_{j=1}^{M-K} (1)} = \frac{\sum_{j=1}^{M} \overline{S_j^{(t)}}}{M-K}$$

This is a ratio that can reach very high values especially for a very large number of destinations per node (which implies a very large set S_j of sources that want to send to the same destination *j*). This increase in computational complexity makes this algorithm slower than the previous one. Actually this is the penalty paid for its generally slightly better performance on the average case. However, for very large *M*, the slightly better performance is offset by the much longer delay, so the first algorithm may be preferred in this case.

3.4 Experiments and simulation of the BSBC algorithms

The mesh network consists of $N^2 = 625$ nodes and **at most** *M=100* destinations for each source node, picked out randomly from the whole network. There are mainly two quantities that we track down and show in the graphs, *Expected **Maximum** Flooding Distance per node*, $ED\max$ and *Expected **Average** Flooding Distance per node*, $EDavg$. The *Expected Maximum Flooding Distance per node* is a measure of the worst case of a route discovery for a node, while the *Expected Average Flooding Distance per node* reflects the average case. Flooding Distance is measured in # of hops.

After a caching algorithm has run and the network has reached a "best cost" state, each node has the "best" K out of the M destinations in its cache. The rest M-K

destinations are not cached by the node, and the node will flood the network in order to find routing information about them.

Let's denote the time where the "best state" has been reached by ∞. Then $\overline{D_i^{(\infty)}}$ is the set of destinations that are **not** in node *i*'s cache when the best state has been reached. The quantities of interest are defined as follows :

Expected Average flooding distance: $EDavg = \dfrac{\sum_{i=1}^{N^2} \dfrac{\sum_{j \in D_i^{\infty}} d_{ij}^{(\infty)}}{M-K}}{N^2} = \dfrac{\sum_{i=1}^{N^2} \sum_{j \in D_i^{\infty}} d_{ij}^{(\infty)}}{N^2 (M-K)}.$

Expected Maximum flooding distance: $ED\max = \dfrac{\sum_{i=1}^{N^2} \max_{j \in D_i^{\infty}} d_{ij}^{(\infty)}}{N^2}.$

In the above equations, we first calculate the **average** and the **maximum distance per node *i*** respectively over all the destinations in $\overline{D_i^{(\infty)}}$. Then we take the expectation of these two quantities over all nodes in the network, obtaining $EDavg$ and $ED\max$.

The quantity that actually interests us is how much part of the network is flooded when a route discovery is initiated. Given a (manhattan) distance *d* (like $EDavg$ and $ED\max$), such a quantity can be found by calculating the associated flooding cost and consecutively finding the percentage of the network covered :

$$\%\ of\ the\ network\ flooded = \frac{2d^2 + 2d + 1}{N^2} \times 100\%.$$

So, in the graphs we will show the **expected percentage of the network flooded per route discovery** in the **worst** and **average** case. These two quantities are generated by $ED\max$ $EDavg$ respectively by application of equation 3.3. The qualitative and quantitative aspects of the BSBC algorithms are described in the four following

experiments. Each experiment is entitled by a natural question that one may ask when considering the parameters of the problem.

3.4.1 Experiment #1 : "How does a BSBC algorithm behave for a fixed cache *K* as the number of desired destinations *M* increases?"

In this experiment we kept the cache size fixed to K=10 positions and varied the number of destinations each source would like to address (that is M) from 20 to 100. The reason for this experiment is to see how well a particular memory size behaves, as the number of destinations per node increases. So for K=10, M=20 we can keep information about half the destinations in each node's routing table, for K = 10 to M=30 for the one third, up to the most limiting case where our memory can only keep 1/10 of the desired destinations (K=10,M=100). The results for *EDavg* and *ED* max are summarized in the following tables :

M	***Davg***			***Dmax***		
	L-BSBC	**G-BSBC**	**"Optimal"**	**L-BSBC**	**G-BSBC**	**"Optimal"**
20	3.215038	3.02544	1.463841	4.8064	4.728	2.6672
30	3.58568	3.228398	1.52214	5.7504	5.8448	2.8352
40	3.692375	3.257334	1.584868	6.1312	6.432	2.9136
50	3.755197	3.297318	1.65736	6.4752	6.8192	2.9648
60	3.790334	3.387359	1.739969	6.72	7.2	2.9888
70	3.805304	3.325121	1.84172	6.8592	6.984	3.0464
80	3.809831	3.379065	1.955467	6.9248	7.2176	3.1696
90	3.818098	3.367638	2.055599	6.9664	7.1712	3.2464
100	3.824996	3.453939	2.08048	7.0192	7.4608	3.1136

Table 3.1. Experiment #1 : M varies while K=10.

Figures 3.1. and 3.2., are derived from Table 3.1. and they show the comparative performance of L-BSBC, G-BSBC and the "optimal" strategy described in paragraph

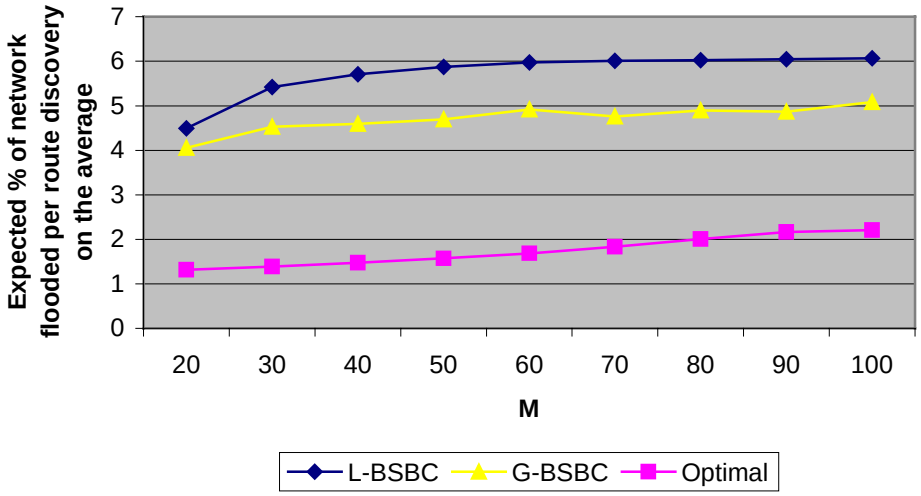

Figure 3.1. : Experiment #1: N=25, K=10, M varies, expected flooding cost in the average case.

3.2. Figure 3.1 shows the **expected percentage of the network flooded per route discovery on the average** (average case).

Both L-BSBC and G-BSBC have a constant difference from the "optimal" (and non-realistic) flooding cost per route discovery of about 4% and 3% respectively. As we can see in the graph, as M increases, the flooding cost in L-BSBC increases as well. For smaller values of M, there is a greater increase in the flooding cost, but after M=70 the increase in M does not seem to affect cost so much. This indicates that for the specific distribution of destinations for each source node, at a cache size of K=10 we are able to cache M>70 destinations at a stable cost of about 6% of the whole network on the average. G-BSBC yields generally a smaller average flooding cost of about 1% less network coverage per route discovery than L-BSBC for values of M greater than 30. For

M=20, we are able to hold in the cache half of the destination per node, so the difference in the two algorithms is not so pronounced. G-BSBC has another interesting characteristic in that the average cost is not a strictly increasing function of M. This is because of the G-BSBC algorithm criterion for choosing the destinations to place in the cache in each iteration. The effect of this difference of destination placement policy between the two algorithms can be more easily seen in Figure 3.2. below :

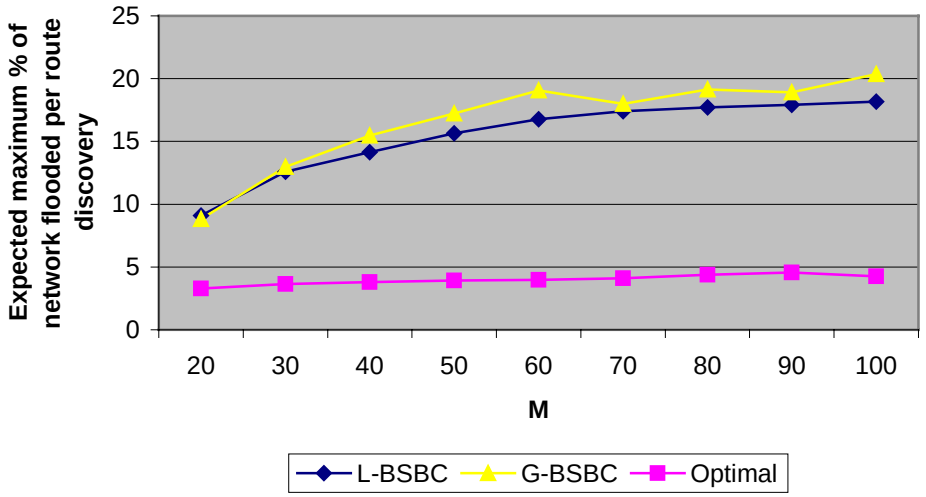

Figure 3.2. : Experiment #1: N=25, K=10, M varies, expected flooding cost in the worst case.

Figure 3.2 shows the **expected maximum percentage of the network flooded per route discovery** (worst case). By comparing G-BSBC and L-BSBC, we see that in this case the flooding cost for the G-BSBC algorithm exceeds the one in the L-BSBC algorithm by a small amount, especially for M=40, 50 and 60. This can be explained by considering the way G-BSBC chooses the destinations it places in a node's cache: It considers the *speculative difference* in the *global* network cost if we placed this destination in the cache

or not. Then the destinations with the largest differences are chosen as the ones to be placed in the cache. In this way, when the algorithm ends, the "best state" is not defined as the state where every node has the most far reached destinations in the cache, but rather as the state where **every node** has the destinations with the maximum speculative differences in its cache. So in G-BSBC it is possible to have **at a specific node** *i*, a destination *j* in $\overline{D_i^{(\infty)}}$ that has a greater $d_{ij}^{(\infty)}$ than the ones of the destinations residing in the cache. Thus, compared to the L-BSBC that always places the most distant destinations **for a specific node** in the cache, G-BSBC may generally have a larger *ED* max for the same (M,K) combination. However as we will see in the next Figures, this difference in *ED* max tends to decrease. Also G-BSBC retains the non-monotonic nature for the maximum flooding cost with respect to *M* in the same way it did in Fig. 3.1. Compared to the optimum algorithm we see a major difference of G-BSBC and L-BSBC from the lower bound curve. In this case, the expected maximum percentage of network flooded per route discovery may even reach 17% to 20% of the whole network, something that would not be tolerable in general.

3.4.2 Experiment #2 : "How big a cache should be used in order to have a tolerable expected flooding cost?"

In this experiment, we varied the Cache size *K*, while *M=100*. As the cache size increases, we may keep information about more destinations in each node. Furthermore, the flooding cost per route discovery (worst case or average) is expected to be reduced by increasing *K*. What we were trying to track is the cache size after which there is not a major improvement in the average or worst case.

K	*Davg*			*Dmax*		
	L-BSBC	**G-BSBC**	**"Optimal"**	**L-BSBC**	**G-BSBC**	**"Optimal"**
10	3.824996	3.453939	1.463841	7.0192	7.4608	2.6672
20	2.76366	2.372781	1.3668	4.9792	5.0992	2.008
30	2.231612	1.954767	1.276159	3.9584	3.9136	2
40	1.886294	1.689974	1.156108	3	2.9968	1.976
50	1.557535	1.480448	1.03008	2.2896	2.0672	1.4016
60	1.4126	1.33552	1.00036	2	2	1.008
70	1.203574	1.100641	1	1.9776	1.6976	1
80	1	1.00168	1	1	1.0336	1
90	1	1	1	1	1	1

Table 3.2. Experiment #2: K varies while M=100.

In Table 3.2., for both BSBC algorithms we see that a change of K/M from 0.1 to 0.2 (K=10 to K=20), induces a dramatic change in the expected **maximum** flooding distance (from 7 hops to 5 hops i.e. from 18% to 9.76% as shown in Fig. 3.4), and then we see a 1-hop decrease every 10 entries of increase in the cache up to K/M=0.5 (K=50). After this, we see the expected maximum flooding distance settling down to 2 hops, and at K=0.8 and K=0.9 the expected max flooding distance goes down to 1 hop, meaning that for every node and every destination that is not in its cache, the routing information is only one hop away. Table 3.2 also shows that **average** flooding distances do not fluctuate as dramatically as the **maximum** ones. Nevertheless they tend to approach the "optimal" algorithm's expected cost. These results will be seen more clearly in the graphs that follow.

Figure 3.3. shows how *expected average flooding cost* varies as *K* increases. Generally *G-BSBC* performs slightly better than *L-BSBC*. The maximum difference in cost is 1% of the network flooded per route discovery for *K=10* (*K/M=1/10*). This difference decreases and for *K>40* the two methods become almost equal in cost. Compared to the lower bounds algorithm, they start approaching it really closely for *K>50*. This means that for the specific destination distribution for each source node,

caching half of the destinations is adequate for providing an optimal flooding cost on the average.

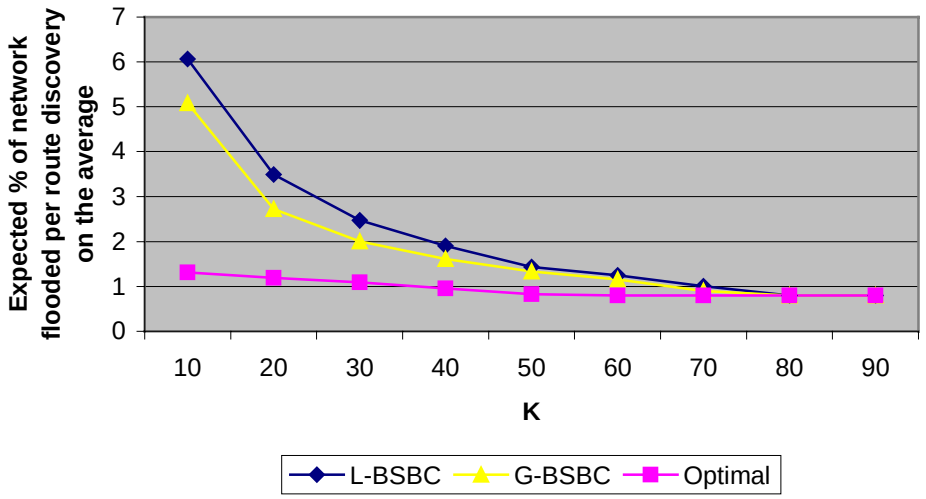

Figure 3.3. : Experiment #2: *N*=25, *M*=100, *K* varies, expected flooding cost on the average

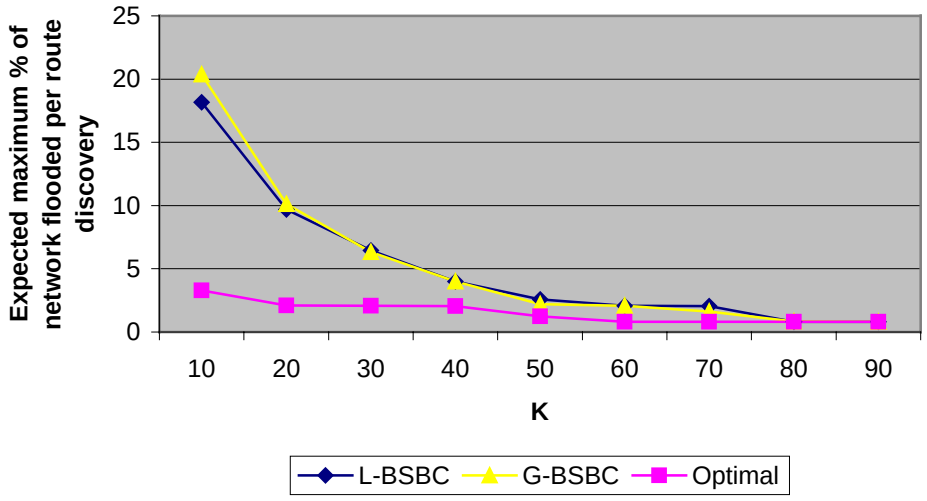

Figure 3.4. Experiment #2: N=25, M=100, K varies, expected maximum flooding cost.

In Figure 3.4, the worst case is depicted. Here *G-BSBC* and *L-BSBC* perform almost identically for every value of *K*. As opposed to the average case in Fig. 3.3, in the worst

case the percentage of the network flooded per route discovery may reach 20% of the network for *K/M=10/100*. However by increasing *K* a little bit, the flooding cost reduces a lot : At *K/M=30/100* both algorithms obtain approximately the same expected flooding cost of 6% as the average case did for K/M=10/100 (see Fig. 3.3). This is a very interesting result showing that a small increase in memory capacity *K* may result in the maximum flooding cost being reduced to the one of the average case. Again for $K / M \geq 40/100$ our algorithms are converging to the lower bound algorithm.

3.4.3 Experiment #3 : "The *K/M* ratio and its robustness to scalability"

The *K/M* ratio mentioned in the previous experiments is the maximum cacheable proportion of the number of destinations *M* a node wants to communicate. What is generally desirable is to achieve low expected flooding costs per route discovery and having a *K/M* ratio as low as possible. However in order to be able to use this ratio, it must be robust. By "robust" we mean that the ratio is in a large degree independent of the specific values of *K* and *M* in terms of the flooding cost associated with them. **For different values of *M*** (*M=50, M=100*) we plotted the expected average and maximum flooding costs **for the same *K/M* ratio**. Figures 3.5 and 3.6 correspond to the *L-BSBC* algorithm and 3.7 and 3.8 to the *G-BSBC*.

The first thing to observe is that for both *L-BSBC* and *G-BSBC* algorithms the ratio behaves almost identically. In both the average and worst case graphs, the cost curves for different values of *M* have a similar decreasing nature for increasing *K/M*. The only difference in the curves is an offset in favor of the *M=100* curve. This happens because, as *M* increases, there are more source nodes sharing a specific destination. So a

node has more chances of finding another node with info about this destination, and therefore the flooding cost will generally go down.

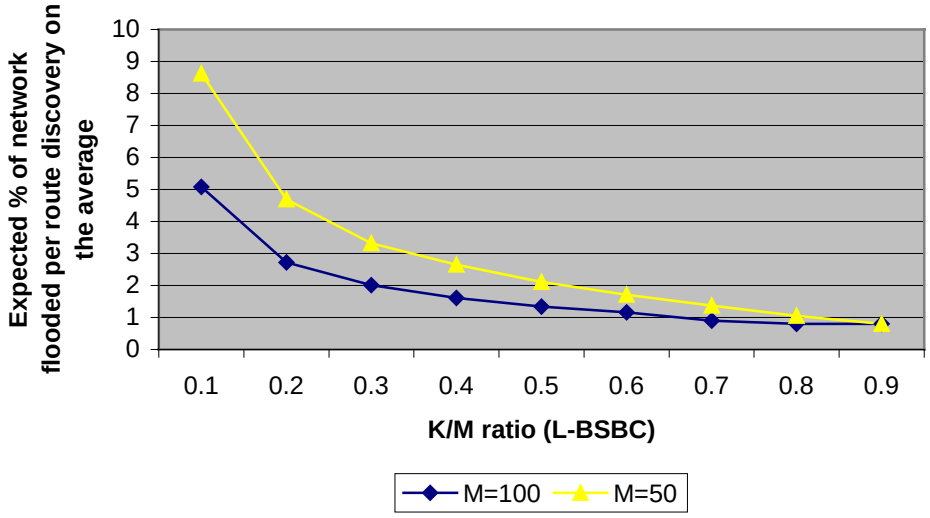

Figure 3.5. Experiment #3: *L-BSBC* algorithm, K/M ratio stability in *average* flooding cost when *M* changes.

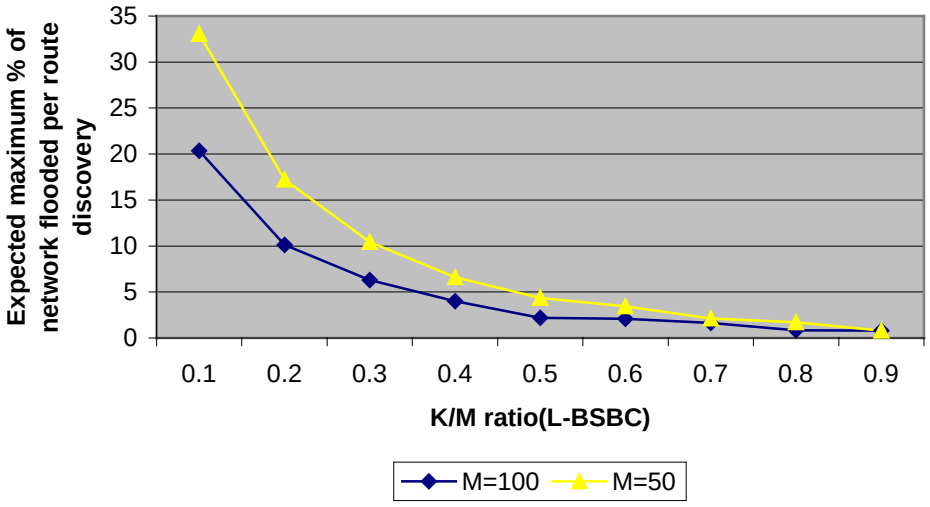

Figure 3.6. Experiment #3: *L-BSBC* algorithm, K/M ratio stability in *maximum* flooding cost when *M* changes.

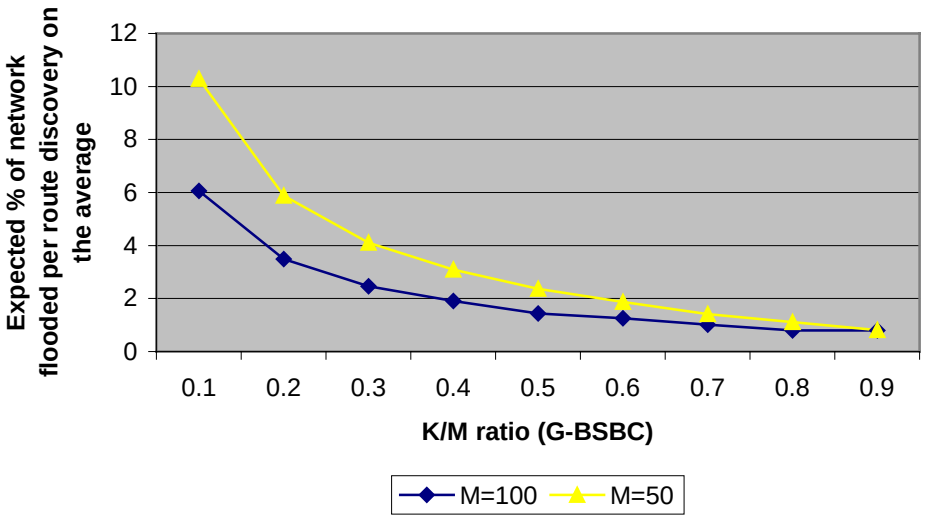

Figure 3.7. Experiment #3: *G-BSBC* algorithm, K/M stability in *average* flooding cost when *M* changes.

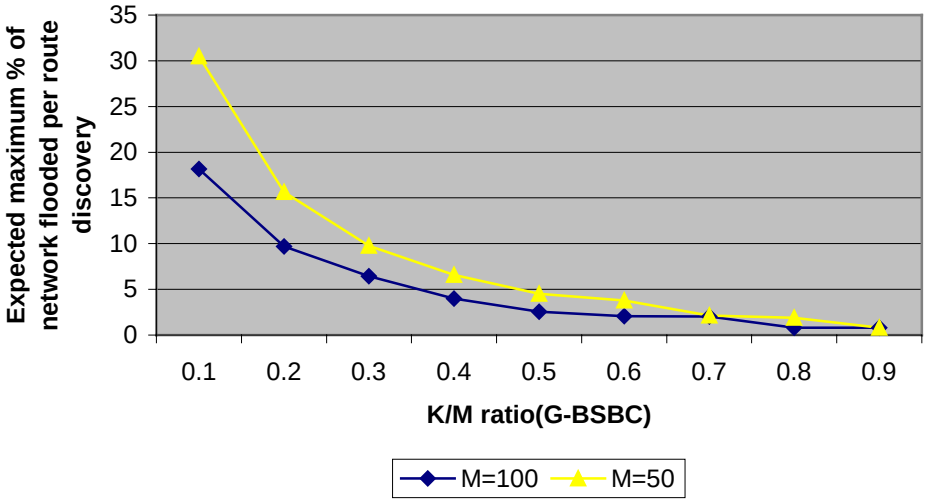

Figure 3.8. Experiment #3: *G-BSBC* algorithm, K/M ratio stability in *maximum* flooding cost when *M* changes.

The robustness of the K/M ratio depends on two factors namely the **shape** of the curves and the **offset** between them. The similar shape of the curves shows that as *K/M* increases the flooding cost decreases in a uniform fashion independently of the specific values of *K* and *M*.

For the same value of *K/M*, the **offset** between the curves is an indication of the cost increase for different values of *M*. **The smaller the offset, the more representative is the K/M ratio for describing the cache efficiency**. We observe that the offset between the two curves generally decreases as *K/M* increases. In all figures 3.5 through 3.8, we identified three regions regarding the offset value:

- $\frac{K}{M} \leq 0.3$: In this range the offset is larger compared with the others. Thus for small memory capacities with respect to the whole network, the *K/M* ratio seems to be lacking robustness with respect to *M*. However as Fig. 3.9 shows, the offset between curves of larger values of *M* like 100 and 200 becomes a lot smaller.
- $0.3 < \frac{K}{M} < 0.7$: In this range, there is some offset between the curves but it is small relative to the one in the previous region. Hence *K/M* can be considered robust in this region.
- $\frac{K}{M} \geq 0.7$: In this range, K/M is almost the same for M=100 and M=50, and K/M is a perfect means of describing cache efficiency.

Figure 3.9 is derived from the data in Figure 3.5. The bar graph named “100 vs 50” is the offset values between the M=10 and M=50 graphs in Fig. 3.5. We also set M=200 and

performed the same experiment for the same K/M ratios. The “200 vs 100” bar graph is the corresponding offsets between M=100 and M=200 curves.

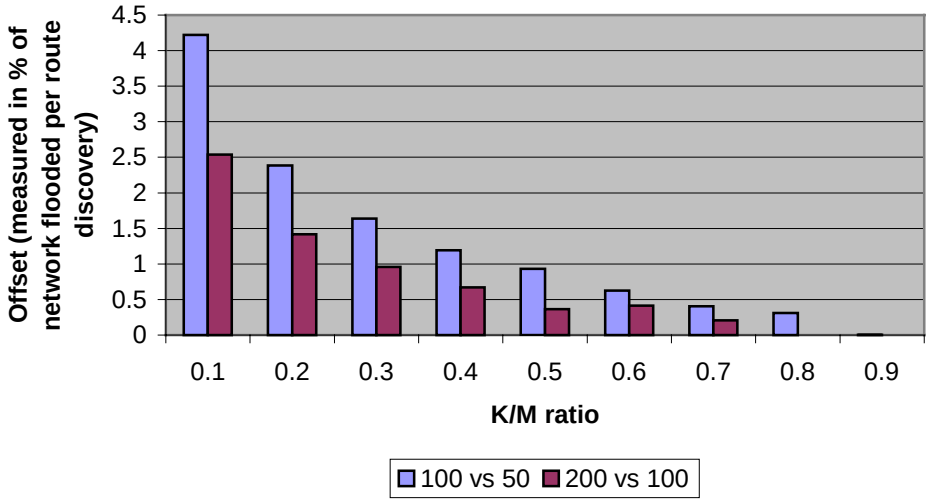

Figure 3.9 : Offsets between different “M-curves”.

The offsets if we compare the M=100 with M=200 cost curves are a lot smaller than the M=100,M=50 case. Especially for the region $\frac{K}{M} \le 0.3$, which had a slight problem when we compared M=100 to M=50, we see that the offsets are reduced. Hence the ratio becomes even more robust as M grows larger. Since M must be generally large in a large network context, we finally conclude that the K/M ratio is generally a meaningful and robust measure of cache efficiency.

3.4.4 Experiment #4: "How do BSBC algorithms perform in the case of uniform traffic?"

In chapter 2 we considered the case of uniform traffic, that is when each node has $\left|\tilde{D}_i\right| = M = N^2 - 1$. We found that by placing routing information for a specific

destination in the vertices of diamonds, the flooding cost is minimized. An interesting twist in the evaluation of the BSBC algorithms would be how they perform under uniform traffic, and whether they finally place routing information according to a diamond-like strategy.

In this experiment we set $\left|\tilde{D}_i\right| = M = N^2 - 1 = 624$. In chapter 2, it was shown that the cache capacity for a diamonds strategy *(N,r)* (r is the side of each diamond in units) is given by the relation :

$$K = 2D^2 + 2D + 1, \text{ where } D = QUOT\left(\frac{N-1}{r}\right)$$

By picking *r = 6*, the above formula yields *D = 4* and a number of nodes with routing information (including the destination) equal to 41. Since the structure is symmetric and the traffic is uniform, the cache size of each node is equal to the number of nodes with routing information, so *K=40*.

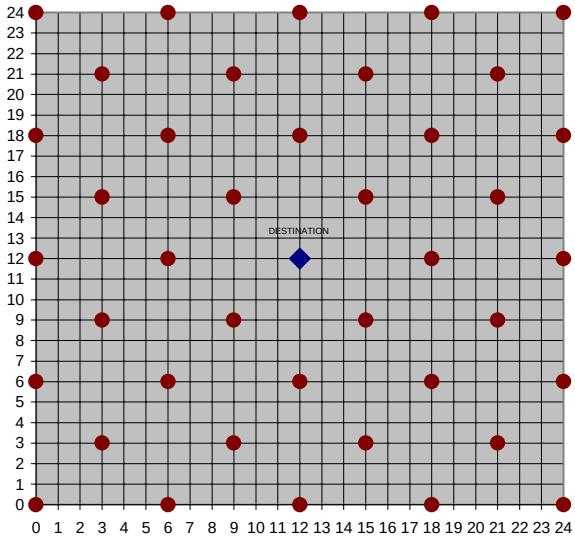

Figure 3.10 : Diamond strategy *(N,r)=(25,6)* for Finite Flooding Horizon (FHF)

Figure 3.10 illustrates the diamond strategy ***(N,r)=(25,6)***.

In this experiment we used the L-BSBC algorithm by setting M=624 (for each source node *i* the set $\tilde{D}_i$ consists of the rest of the nodes in the network) and K=40 (the value

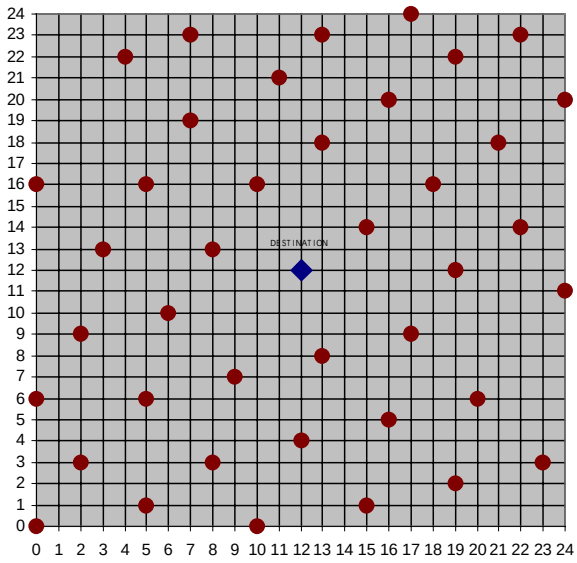

Figure 3.11. : Nodes that have routing information about the center node after the L-BSBC algorithm converges.

computed by the formula above). After the algorithm converged, we placed on the grid the nodes that had information in their cache about the center node.

If we compare figures 3.10 and 3.11 we see a lot of similarity. The L-BSBC algorithm tries to place the nodes with routing information in a diamond-like fashion. By comparing the costs **on the average case**, the diamonds strategy achieves 1.93% of network flooding coverage while the BSBC achieves 1.99%! This fact along with the similarity of the figures above, results in two conclusions **for the uniform traffic case:**

- The optimal diamond strategy performs better than BSBC as expected.

- The BSBC algorithm yields a very close to optimal *expected average flooding cost per route discovery*.

3.5 The LEADERS algorithms

The class of BSBC algorithms tries to obtain a "best state" of the network, after a finite number of iterations. However, there are many such "best states" of the network, and maybe someone would like to find "the best of the best". This is of course impossible, due to the big number of initial network states and the different cache updates during an iteration. Regardless of the non-optimal "best" final state, the BSBC algorithms were shown to be close to optimal **on the average case** for K/M>0.3 (see figure).

The most important "flaw" in the philosophy of BSBC is that upon termination, the algorithms do not guarantee anything about the flooding cost of a specific node. For a specific source node *i* and a specific destination *j* that is not in *i*'s cache after the algorithm ends, even if the cost to find *j* is less than the costs kept in i's cache, **there is no guarantee that this cost is small.**

This fact affects mostly the worst case of a node flooding the network. As Figure 3.4 shows, for medium to small cache capacities K, the % of network flooded per route discovery, can cover even 10% or 20% of the network. In a large mobile ad-hoc network, the number of route discovery initiations at a specific time instant is expected to be large, so a 10% or 20% of the network covered *per route discovery* is clearly unacceptable!

Moreover if we see the whole situation from a user's perspective, the user would like to have a guarantee on the delay of finding routing information. Considering a user *i* the delay for a specific destination *j* is proportional to the distance $d_{ij}^{(t)}$ of the source node to the closest node that holds the destination *j* in its cache.

Given the above considerations, we may want to relax on the concept of "best state" that was fostered by the BSBC algorithms. So now the caching problem can be restated in the following way:

*"Find a state of the network such that, in case some node does not have routing info on some destination, it will **always** be able to find the information at a maximum distance MaxDist (or equivalently with a maximum cost of Cmax = f(MaxDist))."*

In this way there is a restriction on the maximum flooding cost ANY node may generate. Such a policy would be able to restrict the worst case flooding cost of *BSBC* algorithms. So now, we are not looking for a final "best state", but for a state that will satisfy the flooding cost restriction that we impose. Needless to say that such an algorithm will converge faster than the BSBC ones.

The class of LEADERS algorithms is accomplishing this by adopting a destination-based approach. The idea is, for each destination node *x* to separate the sources that wish to reach it into clusters. Each cluster's centroid, must have a (Manhattan) distance from the other nodes in the same cluster, less than or equal to a ***constraint distance MaxDist***. So the regions centered at the centroids have a maximum coverage of *Cmax = f(MaxDist)*. For each centroid node of a region, add destination *x* in its cache. In this way, all the nodes associated with that centroid-leader will be able to access the routing info with a flooding cost less than or equal to *Cmax.*

The *LEADERS* algorithms differ in the underlying clustering algorithms. The clustering algorithms run for every destination *j* in the network, and form clusters containing the sources that have traffic for it (this is the set $\tilde{S}_j$). After finding the centroid nodes for *j*, this destination is placed in the caches of these centroid nodes. Thus

every node in $\tilde{S}_j$ has these nodes as leaders, and is guaranteed to find destination *j* in **at most** MaxDist distance, with **at most** *Cmax = f(MaxDist)* flooding cost. We now introduce the two algorithms that characterize the class of the LEADERS algorithms, namely the Single Pass Leaders Algorithm (SPLA) and the Two Phase Leaders Algorithm (TPLA).

3.5.1 The "Single Pass" Leaders Algorithm (SPLA)

This algorithm is constructive: For each destination *j* we start with the set of source nodes $\tilde{S}_j$. We consider one node at a time, and place it in the appropriate region from the ones that have been constructed so far. If the node cannot be assigned to an existing region, we create a new one with this node being the centroid. If this node can be assigned to an existing region, we choose the region whose centroid is closer to the node. After we add it to this region we recalculate the centroid of this region, by finding which node has the minimum total distance from the rest in this region. The algorithm is called a "single pass", because for each destination *j*, after a single pass of the list $\tilde{S}_j$ all the regions and centroids have been defined.

For destination *j*,

- Let $U_j^{(t)}$ be the set of nodes that have **not** been assigned to any region up to time time *t*. The set $\overline{U_j^{(t)}}$ will be the set of nodes that have been assigned to a region up to time *t* and also: $U_j^{(t)} \cup \overline{U_j^{(t)}} = \tilde{S}_j \quad \forall t$.

- Let $L_j^{(t)}$ be the set of Leaders (centroids) at time *t* and $\left|L_j^{(t)}\right|$ be the cardinality of $L_j^{(t)}$. We also denote by $L_j^{(t)}(m), 1 \le m \le \left|L_j^{(t)}\right|$ a single element (centroid) in $L_j^{(t)}$.
- Also let $R_j^{(t)}(k), 1 \le k \le \left|L_j^{(t)}\right|$ be the k-th region from the ones that have been constructed so far. It is obvious that the number of regions at time *t* is equal to $\left|L_j^{(t)}\right|$.

Initialization:

for each node *j*

{

choose any node *i* in $\tilde{\tilde{S}}_j$

$U_j^{(0)} = \tilde{S}_j - \{i\}$

$R_j^{(0)}(1) = \{i\}$

}

Iterations :

for each destination j

{

t=0*;*

while ($U_j^{(t)} \neq \varnothing$)

{

select a node *i* in $U_j^{(t)}$

$U_j^{(t+1)} = U_j^{(t)} - \{i\}$

$d = \min_m \{d(i, L_j^{(t)}(m))\}$ /*minimum distance from any of the existing centroids*/

$\arg d = \arg \min_m \{d(i, L_j^{(t)}(m))\}$ /*the corresponding centroid*/

if ($d \le D\max$) /*We assign this node to the region $R_j^{(t)}(k)$ that currently has *argd* as a centroid*/

{

$R_j^{(t+1)}(k) = R_j^{(t)}(k) + \{i\}$

Recompute the centroid of $R_j^{(t+1)}(k)$ by choosing the node that has the minimum total distance from the rest in $R_j^{(t+1)}(k)$.

}

else

{

$L_j^{(t+1)} = L_j^{(t)} + \{i\}$

create $R_j^{(t+1)}\left(\left|L_j^{(t+1)}\right|\right)$

}

$t = t + 1$;

}

}

3.5.2 The "Two Phase" Leaders Algorithm (TPLA)

For a destination *j*, the Two Phase Leaders Algorithm consists of two phases :

- First it finds the *centroids* list L_j. L_j is found incrementally by scanning $\tilde{S}_j$ many times and adding to L_j the node that has the following properties :
  - Has a distance from ***each*** existing centroid greater than *MaxDist*.

- Has a maximum total distance from the rest of the existing centroids.
- Its cache is not full yet.

- The second phase assigns the rest of the nodes to the regions according to the centroids found in the first phase.

For destination *j*,

- Let $L_j^{(t)}$ be the set of Leaders (centroids) at time *t* and $\left|L_j^{(t)}\right|$ be the cardinality of $L_j^{(t)}$. We also denote by $L_j^{(t)}(m), 1 \le m \le \left|L_j^{(t)}\right|$ a single element (centroid) in $L_j^{(t)}$.
- $\overline{L_j^{(t)}}$ is the set of nodes that are not centroids. $\overline{L_j^{(t)}}$ is used in phase II, after the centroids have been identified.
- Also let R_j^i be the region that corresponds to centroid *i* after the centroids have been found in phase I.

Algorith Pseudocode:

for each node *j*

{

Phase I:

Initialization:

$L_j^{(0)} = \{j\}$ /*Choose the destination *j* to be the first centroid*/

$T_j^{(0)} = \tilde{S}_j$

Iterations :

t=0*;*

while ($T_j^{(t)} \neq \varnothing$)

{

for every node *i* in $T_j^{(t)}$

{

$TotSum(i) = 0$

for every centroid k currently in $L_j^{(t)}$

{

if($d(i,k) \le MaxDist$) /*node *i* cannot qualify as a centroid*/

{

$T_j^{(t)} = T_j^{(t)} - \{i\}$

$\overline{L_j^{(t)}} = \overline{L_j^{(t)}} + \{i\}$

stop here and consider the next element *i* of $T_j^{(t)}$ by going back to the *i*-loop

}

else

{

$Totsum(i) = Totsum(i) + d(i,k)$

}

}

}

$m = \arg\max_i \{TotSum(i)\}$ /*node *m* is the one that qualifies as a centroid and has the maximum total distance from the centroids in $L_j^{(t)}$.*/

if($T_j^{(t)} \neq \varnothing$)

{

$L_j^{(t+1)} = L_j^{(t)} + \{m\}$

$T_j^{(t+1)} = T_j^{(t)} - \{m\}$

$\overline{L_j^{(t+1)}} = \overline{L_j^{(t)}}$

$t=t+1$

}

}

Phase II: (by $t = \infty$ we mean that phase I has finished)

for every node *i* in $\overline{L_j^{(\infty)}}$

{

$d = \min_m \{d(i, L_j^{(\infty)}(m))\}$ /*minimum distance from any of the existing centroids*/

$\arg d = \arg\min_m \{d(i, L_j^{(\infty)}(m))\}$ /*the corresponding centroid*/

$R_j^{(\infty)}(\arg d) = R_j^{(\infty)}(\arg d) + \{i\}$

}

This algorithm minimizes the number of centroids (and thus the number of regions) created for each destination. Since the number of centroids determines the size of the caches, we expect this algorithm to create as minimum cache sizes as possible.

3.5.3 Experiment #1 : "SPLA vs TPLA"

We performed an experiment by running both *SPLA* and *TPLA* for M=100, K=40 and *MaxDist=5* and tracked down the centroids and the corresponding regions that were

formed corresponding to the destination node *j=265* (i.e the point on the mesh with cartesian coordinates *(x,y)=(15,10)*).

The two algorithms provided similar expected flooding costs per route discovery on the average and maximum case. In terms of speed, SPLA converged faster than TPLA and this is because of the "single pass" nature of SPLA. However TPLA yielded much smaller caches (30% smaller total network cache size) after the algorithm termination. This is because TPLA seeks to minimize the number of centroids per destination. The memory performance difference can be more easily visualized by considering Figures 3.12 and 3.13 which correspond to SPLA and TPLA respectively.

The figures below show the destination and all the sources in the set $\tilde{S}_{265}$ (that is the set of sources that have traffic for this destination). The regions formed for each algorithm are clearly shown as points with different format. Each node in a region finds the information about destination 265 by referencing to the centroid node by an arrow. So the node which is a sink for all the arrows within a region is the centroid of this region. However, there are some points on the mesh that do not initiate or accept arrows. These are "degenerate" regions of a single node-centroid. Generally "degenerate" regions are undesirable, since this single node could have been assigned to an already existing region and not filling its cache with information about a specific destination. Degenerate regions are only justified if their centroid is really far from any other nodes that have traffic for the specific destination.

By observing Figures 3.12 and 3.13 it is obvious that TPLA is superior to SPLA for this specific destination, since a smaller number of regions is formed (12 vs 16), for an initial set $\tilde{S}_{265}$ of 35 nodes. Furthermore in the case of SPLA we see more

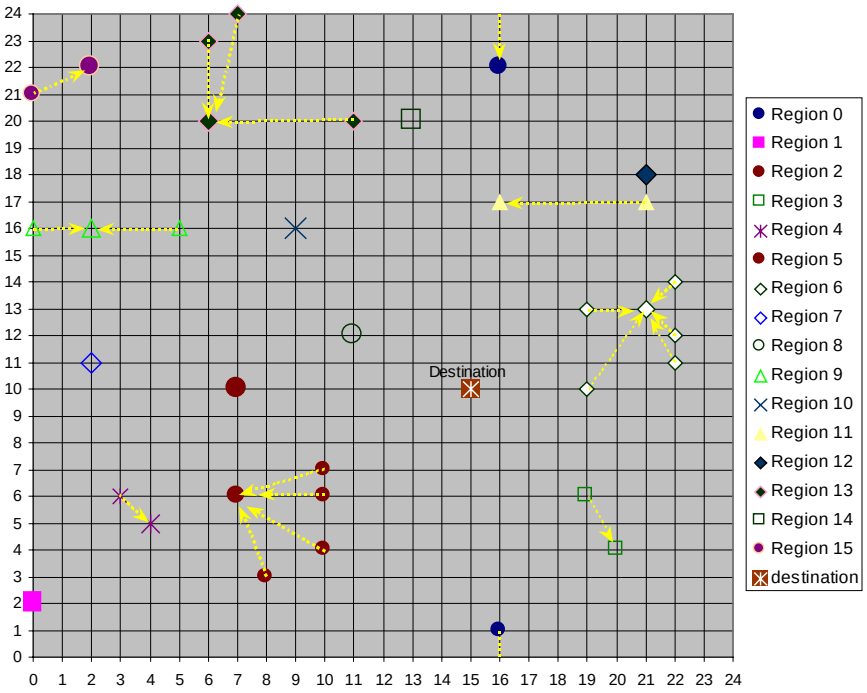

Figure 3.12. SPLA formation of centroids and regions for destination *i*=265

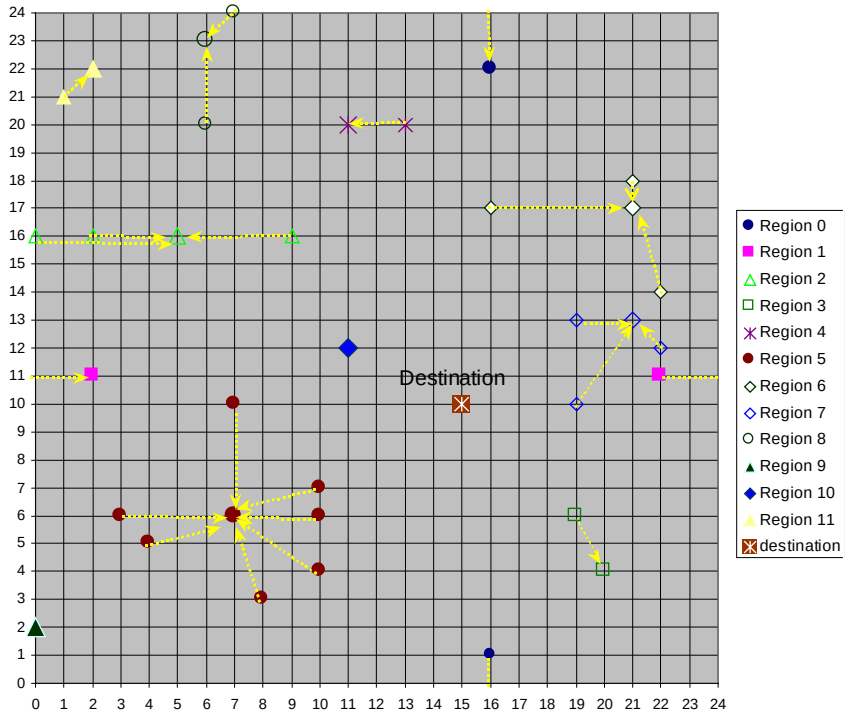

Figure 3.13. TPLA formation of centroids and regions for destination *j*=265

"unjustifiable" degenerate regions. These regions could be assigned to other regions formed nearby. For example Regions 11 and 12 in Fig. 3.12 could be merged in a single one. If we look at figure 3.13 we see that this is accomplished. The most notable difference in the two figures corresponds to larges region 5 in figure 3.13 that is formed by the TPLA algorithm. In this region, 7 nodes refer to node (7,6) in order to reach destination 265. If we look back to figure 3.12, we see that the same points are assigned to three regions namely 3, 4 and the degenerate region 5.

This experiment shows that TPLA is better than SPLA in terms of total network memory capacity achieved while both achieve similar flooding costs. However SPLA converged faster than TPLA and may be more desirable in a lot larger network contexts.

3.5.4 Experiment #2: LEADERS vs BSBC

The two classes of algorithms are based on a different philosophy: The LEADERS algorithms impose a constraint on the maximum flooding cost for each node and after the algorithm runs, the nodes do not have equal cache capacities. On the contrary, BSBC algorithms have the same capacity *K* for each node upon algorithm termination but they do not guarantee anything about the flooding cost per node. The metrics we use to compare to compare the BSBC and the LEADERS algorithms is in terms of the total memory capacity in the network, and the flooding costs per route discovery. In the case of BSBC, the total memory capacity is always $K \times N^2$ while in the LEADERS class it is variable. Because of the non-uniform traffic, some nodes will have a large memory capacity and some others will not. The more destinations a node becomes a centroid for during the clustering algorithm execution, the larger its cache size will be.

In order to do the comparison between the two classes we found "equivalent" experiments for them by consulting Table 3.2. For example for *K=20* we saw that L-BSBC and G-BSBC achieved *Dmax = 4.97* and *Dmax 5.09*. So an "equivalent" experiment for LEADERS is to set *MaxDist*=5, M=100 and K=20.

Table 3.3 shows the results of the "equivalent" experiments for various sizes of *K*. Note that columns 1,2,4,5 of the table were just copied from table 3.2. TPLA was chosen as a representative of the LEADERS algorithms because it is by far better than SPLA. The results in Columns 3 and 5 were obtained by considering *MaxDist* to be the immediately larger integer from the number $\min\{D\max_{L-BSBC}, D\max_{G-BSBC}\}$ for a specific *K*. Thus for *K=40*, *MaxDist* was chosen as the immediately larger integer number of *min{3.9584,3.9136}* which is 4. Also the maximum cache size for the LEADERS algorithm was kept equal to *K*.

K	***MaxDist***	***Davg***			***Dmax***		
		L-BSBC	**G-BSBC**	**TPLA**	**L-BSBC**	**G-BSBC**	**TPLA**
20	5	2.76366	2.372781	2.453118	4.9792	5.0992	4.976
30	4	2.231612	1.954767	2.221376	3.9584	3.9136	3.984
40	3	1.886294	1.689974	1.999272	3	2.9968	2.9904
50	3	1.557535	1.480448	1.752799	2.2896	2.0672	1.9952
60	2	1.4126	1.33552	1.752497	2	2	1.9952
70	2	1.203574	1.100641	1.750456	1.9776	1.6976	1.9952
80	1	1	1.00168	1.471342	1	1.0336	0.9984
90	1	1	1	1.473079	1	1	0.9984

Table 3.3.: MaxCost "Equivalent" experiments for the LEADERS and BSBC algorithms.

The following graphs illustrate the trade-offs between the two algorithm types.

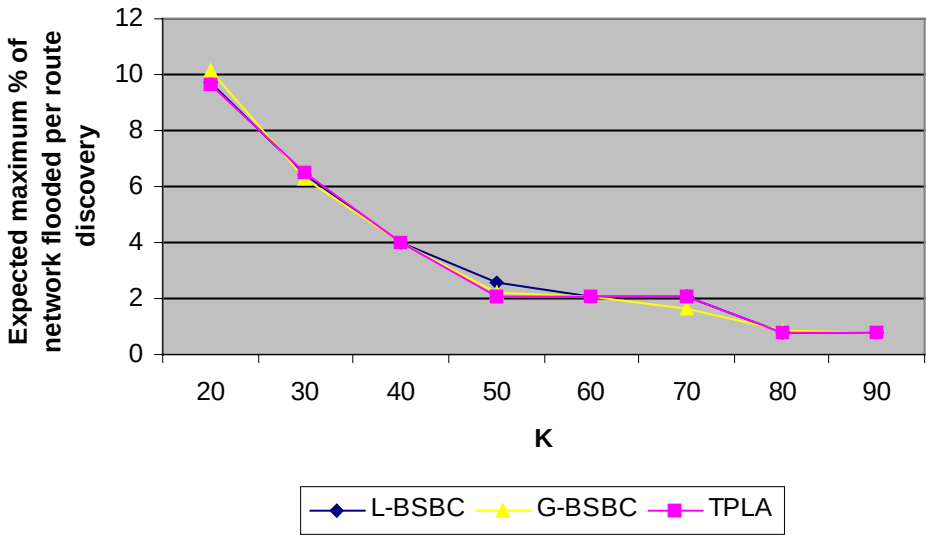

Figure 3.14: M=100, K varies. LEADERS vs BSBC:

Expected flooding cost in the worst case.

Figure 3.14. shows how the worst case behaves in the two classes. We see that the curves match in this case. The reason is that we have "tuned up" our "equivalent" experiments to produce the same worst case by setting MaxDist as was described earlier.

Figure 3.15 is of more interest, since it shows the average case. We see that for small values of *K TPLA* is somewhere between L-BSBC and G-BSBC. For values of *K/M* larger than 0.4, TPLA performs slightly worse. In Figure 3.16.we see the benefit of LEADERS in terms of total network memory utilization. This figure plots the ratio of total network capacity for TPLA over the total network capacity of BSBC, which is always constant and equal to $K \times N^2$.

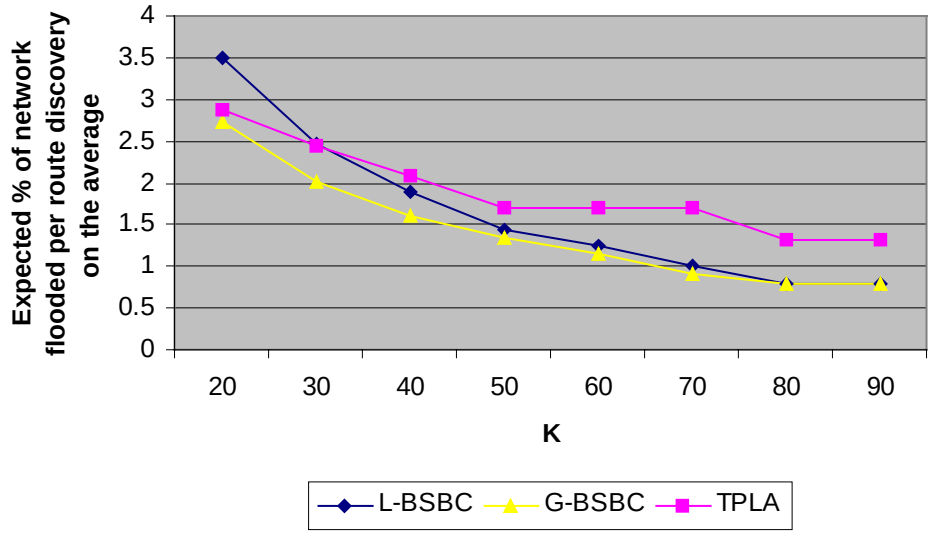

Figure 3.15. M=100, K varies. LEADERS vs BSBC:

Expected flooding cost in the average case

So the value shown in the graph is given by the realation :

$\alpha = \frac{KTotal_{TPLA}}{KN^2} \times 100\%$, where $KTotal_{TPLA}$ is the total network cache capacity after TPLA has run.

Figure 3.16. shows clearly that in almost every case TPLA obtains smaller cache capacity than the BSBC algorithms. This is true because, the capacity of each node depends on how "popular" this node is when it is considered as a centroid by a destination and is not filled by some "best state" destination. Thus TPLA can adapt the memory capacity to the traffic patterns by identifying some nodes that need more memory capacity than others. Thus the LEADERS family of algorithms could be used in

more heterogeneous networks where nodes have different memory capacities and capabilities.

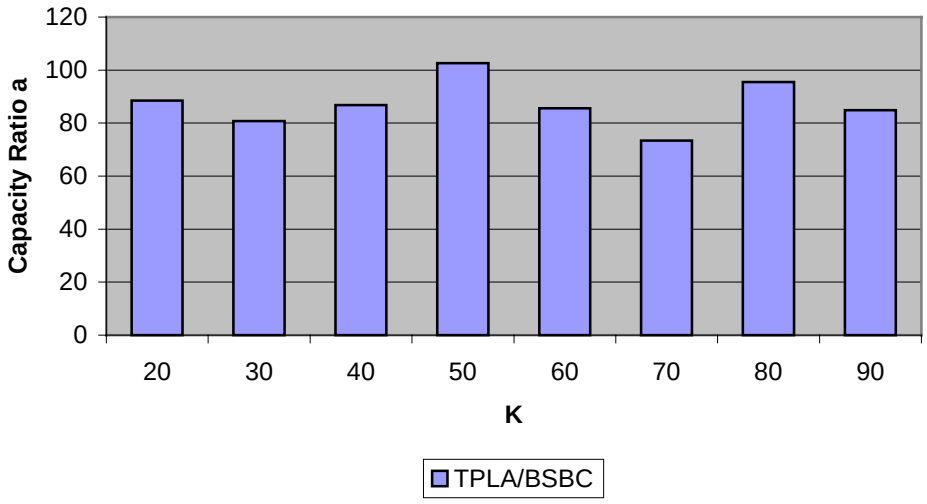

Figure 3.16. M=100, K varies. LEADERS vs BSBC: Capacity Ratio obtained for the same worst case flooding cost.

Chapter 4

4.1 Summary, Conclusions and discussion

Many mobile ad-hoc networks applications are envisioned to support a very large number of nodes under a flat routing structure in a highly mobile and dynamic environment. In this case, we argue that scalability can only be achieved in the context of reactive "on demand" routing protocols as opposed to the proactive routing protocols based on the traditional philosophy of routing algorithms used in fixed packet switched networks. The "on demand" nature of reactive protocols requires the intelligent caching of destinations on each host so that flooding cost on a route discovery be minimized. This motivated us to introduce a framework for quantifying the caching of destinations in a network that is in a form of a torroid mesh. In chapter 2 we initially investigated the case where each source node wants to address all the other nodes in the network with equal probability. This served as a precursor to chapter 3 where the problem of caching was stated in a more general fashion.

In the case of Infinite Horizon Flooding (IHF), we found that a diamonds strategy for placing routing information on the mesh outperforms the square one since for approximately equal cache sizes, the squares strategy has approximately 1.5 times more flooding cost on the average.

If Finite Horizon Flooding (FHF) is used instead of IHF, we need to cache less destinations per node to achieve the same flooding cost. For the limited horizon problem, we proved that a diamonds policy, where routing information is placed at the vertices of each diamond yields the minimum possible average flooding cost for the case of uniform

traffic. However even if optimal, a diamond strategy depends on the starting flooding horizon h_1 and the retry flooding step *s* by which the current horizon is incremented if routing information is not found. By performing experiments we concluded that given a diamond strategy *(N,r)*, in order to have minimum flooding cost per route discovery, the condition $h_1 + s = \frac{r}{2} + 1$ should be satisfied. We furthermore observed that the optimal *s* that yielded the minimum of the minimum average flooding costs was always *s = 2* for different diamond strategies *r*. Also for this value of *s* the optimum h_1 was always found to be $h_1^* = \frac{r}{2} - 1$, something that verified the condition mentioned above.

Overall in the FHF case, if we exclude some extreme values of diamond sizes *r*, the diamond strategy achieved very small flooding costs per route discovery and very small cache sizes compared to the network size. This was due to the optimality of the diamond strategy, the symmetric nature of the torroid structure and the fact that each source node wants to communicate with all the other nodes with equal probability.

In chapter 3 we considered the non-uniform traffic case where each node wishes to address an arbitrarily large and arbitrarily situated subset *M* of the whole network N^2. Two classes of algorithms were proposed in order to solve the above problem. These algorithms are generic in the sense that they do not depend on the mesh structure of the network we use. The torroid mesh structure is convenient only because it provides an easy way to find the distance *d* between two specific nodes in the network, and an even easier way to quantify the flooding cost associated with it ($f(d) = 2d^2 + 2d + 1$ in the case of $h_1 = 1$ and $s = 1$).

The BSBC algorithms are iterative, and in each step they try for each node to keep the K maximum cost (out of M) destinations in the cache, based on the current state of the network. The algorithms terminate when the "best state" has been reached. The difference in the algorithms lies in the "maximum cost" criterion a node employs to decide which destinations to keep in the cache. In the L-BSBC algorithm, each node defines as a cost metric, the distance to reach a specific destination. The source node then tries to cache the destinations with the maximum distance from **itself**. The G-BSBC algorithm refers to the global network state. Each source node speculates for each destination what is the difference on the whole network state of placing the destination in its cache or not. Then it places in its cache the destinations with the *maximum speculative differences*.

The two algorithms were compared in terms of the **expected percentage of the network flooded per route discovery** in the **worst** and **average** case. We found that G-BSBC performs better than L-BSBC on the average case and a little bit worse on the worst case. However the G-BSBC achieves better average performance at the cost of higher computational complexity. Therefore in large networks it may be desirable to use the L-BSBC.

Both of these algorithms were compared with an "ideal" case. This case consists of a node deciding about which destinations to place in its cache assuming **independently** about each destination that all the sources that have traffic for it have this destination in their cache at that instant. Even if this is an unrealistic assumption, it is a lower bound on any algorithm's performance. In the average case, the results of the comparison with the ideal case were fairly good for *K/M* ratios that are over 0.3, and there was an absolute

convergence for *K/M>0.5* (that is if each node is able to cache information about the 50% of the network).

Another experiment that established the validity of the BSBC algorithms is the test under uniform traffic. It was shown that they are generally trying to place routing information according to the optimal diamonds strategy elaborated in chapter 2. In the experiment L-BSBC was very close to the optimal in terms of the average flooding cost per route discovery. More specifically it achieved a 1.99% of network coverage on the average as opposed to the optimal of 1.93%! This is probably due the small diamond size used (r=6 units) but still it is an indication of the BSBC algorithm's power for the **average case**.

Despite their advantages and good performance on medium sized memories with respect to the whole network, the BSBC algorithms after they terminate, do not provide any guarantee about the worst case flooding cost a node may experience. In Figure 3.4, for medium to small cache capacities K, the *expected maximum % of network flooded **per route discovery*** (worst case), may cover even 10% or 20% of the network. Of course a 10% network coverage per route discovery is not acceptable.

In order to alleviate the worst case effect, the LEADERS algorithms class was introduced. The LEADERS algorithms guarantee for each node that it will be able to find a destination that is not cached at a distance no more than *MaxDist*, where *MaxDist* is a parameter of the algorithms. Such a guarantee is a delay guarantee from the user's perspective since the user knows that the routing information will be found in at most *MaxDist* steps. From the network perspective this is a bandwidth guarantee because the network "knows" that each user will never flood more than *Cmax=f(MaxDist)* nodes to

find the destination. The LEADERS algorithms deliver the guarantee mentioned by relaxing on the notion of "best state". They are not seeking an "optimal" network state but a state where the guarantee set by the parameter *MaxDist* is satisfied.

The class of LEADERS algorithms is adopting a destination-based approach. The idea is, for each destination node to separate the sources that wish to reach it into clusters. Each cluster's centroid, must have a (Manhattan) distance from the other nodes in the same cluster, less than or equal to a ***constraint distance MaxDist***. For each centroid node of a region, add the specific destination in its cache. In this way, all the nodes that belong to the same region with the centroid-leader will be able to access the routing info with a flooding cost less than or equal to *Cmax.*

The *LEADERS* algorithms differ in the underlying clustering algorithms. The "*Single Pass" Leaders Algorithm* (*SPLA*) builds and updates the regions incrementally as it sees new nodes, while the "*Two-Phase" Leaders Algorithm* (*TPLA*) constructs the centroids first and then forms the regions based on the centroids found in the first phase. TPLA is designed to minimize the number of centroids needed given the constraint *MaxDist*. Minimizing the number of centroids is a key feature of TPLA that renders it generally superior to SPLA: since the number of times a source node becomes a centroid is actually the size of its cache, and TPLA optimizes on the number of centroids per destination, it will provide the best solution. However, the single pass nature of SPLA makes it attractive in a very large network context where the amount of calculations grows large and TPLA must make a lot of passes through the initial list in order to construct the centroids list.

Finally we attempted to investigate the relative performance of BSBC and LEADERS algorithms in terms of **total network memory capacity** and the **average flooding cost per route discovery**, when we match their **maximum flooding cost per route discovery**. In most of the cases TPLA yielded higher average flooding cost. This is because it does not try to find any "optimal" state but just to provide a viable guarantee in the worst case each node will have to flood the network frequently.

There is a side effect of the *LEADERS* algorithms that may render them desirable in some situations: Each node has generally a different number of cache entries after the algorithm terminates. The experiments showed that this fact usually saves on total network cache capacity, at the penalty of a slight increase in flooding discovery traffic in the average case. What the algorithm does is that it actually adapts the total network cache memory to the given traffic in the network.

We envision extensions to the algorithms developed to be working in a dynamically changing environment within large heterogeneous networks having nodes with different memory and capacity requirements. In such an environment, a pure "optimal state" seeking algorithm like BSBC would not have a lot of meaning. Instead, a *LEADERS*-like algorithm would adapt to the traffic by overloading some specific caches to serve a larger set of destinations, and leaving the caches of other source nodes free to be deciding which entries to keep in the cache on a *BSBC*-like fashion.

REFERENCES

[1] J. M. McQuillan, I. Richer, and E.C. Rosen, "The new routing algorithm for the ARPANET", IEEE Transactions on Communications, COM-28(5):711-719, May 1980.

[2] D. Bertsekas and R. Gallager. "Data Networks", pages 297-333, Prentice Hall Inc, 1992.

[3] E. Gafni and D. Bertsekas, "Distributed algorithms for generating Loop Free Routes in Networks with frequently changing topology," IEEE Trans. In Comm., January 1981.

[4] P.M. Merlin and A. Segall, "A failsafe distributed routing protocol", IEEE Transactions on Communications, COM-27(9):1280-1287, September 1979.

[5] J.M. Jaffe and F.H. Moss, "A responsive distributed routing algorithm for computer networks", IEEE Transactions on Communications, COM-30(7): 1758-1762, September 1982.

[6] V. D. Park, M. S. Corson, "A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks," Proc. INFOCOM '97, pages 1405-1413, April 1997. URL: http://tonnant.itd.nrl.navy.mil/tora/tora.html

[7] V. D. Park and M. S. Corson, "A Performance Comparison of the Temporally-Ordered Routing Algorithm and Ideal Link-State Routing", Proceedings of IEEE Symposium on Computers and Communication '98, Athens, Greece (June 1998), http://tonnant.itd.nrl.navy.mil/tora/tora.html

[8] M.S. Corson and V.D. Park, "An Internet MANET Encapsulation Protocol (IMEP) Specification.", Internet Draft, http://www.ietf.org/internet-drafts/draft-ietf-manet-imep-spec-01.txt., November 1997, Work in progress.

[9] C. Perkins and P. Bhagwat, "Highly Dynamic Destination-Sequenced Distance Vector Routing (DSDV) for Mobile Computers," In proceedings of the SIGCOMM '94 Conference on Communications Architectures, Protocols and Applications, pages 234-244, August 1994. A revised version of the paper is available from http://www.cs.umd.edu/projects/mcml/papers/Sigcomm94.ps.

[10] C. Perkins, "Ad Hoc On Demand Distance Vector (AODV) Routing", Internet Draft, http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-02.txt, November 1997. Work in progress.

[11] S.Murthy and J.J. Garcia-Luna-Aceves, "An efficient Routing Protocol for Wireless Networks," ACM Mobile Networks and Applications Journal, Special issue on Routing in Mobile Communication Networks, (1996). URL: http://www.cse.ucsc.edu/research/ccrg/

[12] J.Moy. OSPF version 2. RFC 1247, July 1991.URL: http://www.ietf.org/rfc/rfc1247.txt

[13] S. Corson, S.Batsel, and J. Macker. "Architectural considerations for mobile mesh networking." http://tonnant.itd.nrl.navy.mil/mmnet/mmnetRFC.txt, May 1996. Request For Comments draft.

[14] D. Johnson and D. Maltz., "Dynamic Source Routing in mobile ad-hoc networks", In *Mobile Computing*, edited by T. Imielinski and H. Korth, chapter 5, pages 153-

181, Kluwer Academic Publishers, 1996. URL: http://www.monarch.cs.cmu.edu/papers.html

[15] J. Broch, D. A. Maltz, D. B. Johnson, Yih-Chun Hu, and J. Jetcheva, "A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols.", In Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking, ACM, Dallas, TX, October 1998. URL: http://www.monarch.cs.cmu.edu/papers.html

[16] S. R. Das, R. Castaneda, J. Yan, R. Sengupta. "Comparative Performance Evaluation of routing protocols for mobile, ad-hoc networks", In 7th International Conference on Computer Communications and Networks (ICSN), October 1998. URL: http://ringer.cs.utsa.edu/~samir/pub.html