A Content-Based Routing Protocol for Mobile Ad-Hoc Networks Using a Distributed Connected k-Hop Dominating Set as a Backbone

Master Thesis

Wouter Klein Wolterink

Date	December 7, 2008
Committee	Dr. ir. G.J. Heijenk
	Dr. ir. M. de Graaf
	Dr. ir. G. Karagiannis
Institution	University of Twente
Faculty	Electrical Engineering, Mathematics and Computer Science (EEMCS)
Group	Design and Analysis of Communication Systems (DACS)
Programme	Telematics (MTE)

Abstract

This report describes the design, implementation and analysis of a content-based routing (CBR) system for a mobile ad-hoc network (MANET) that uses a backbone of flexible size to route its content over. Nodes that are not part of the backbone have a path towards it. By changing the size of the backbone (and thus the length of the paths) an optimum can be found in which routing is at its most effective and efficient. It is shown that for low average node speeds the network is indeed capable of effective and efficient routing, but that at higher speeds the routing paths can no longer be supported.

This backbone used for the CBR system has been created based on a paper by Yang et al. [7]. In their paper they present an algorithm capable of creating and maintaing a connected k-hop dominating set (Ck-HDS). Their algorithm can not be directly applied for a MANET however. This report also describes the design, implementation and analysis of a protocol based on this algorithm.

For both systems a proof of concept has been tested by means of simulation, assuming a network model with symmetric radio links.

Contents

PART I

An introduction to the thesis in general

This part introduces the thesis as a whole. Chapter 1 introduces the concept of content-based routing (CBR) and explains how a CBR system (the subject of Part III) has been designed over a connected k-hop dominating set protocol (the subject of Part II). Chapter 2 discusses the methods of analysis that have been employed in this thesis.

Chapter 1
Introduction

1.1 Mobile ad-hoc networks and the need for content-based routing

A mobile ad-hoc network [2] (or as the acronym goes: MANET) is a type of wireless ad hoc network that consists of a dynamic set of wireless mobile routers (and associated hosts), which we will refer to as nodes. Nodes are interconnected via radio links: if two nodes are within each other's transmission range they are said to share a symmetric radio link; if one can see the other but not vice versa the first node is said to have an asymmetric radio link to the other node. The dark-coloured elements of Figure 1 show the typical graphic representation of a MANET: a graph in which the vertices represent the nodes and the edges the (assumed symmetric) links. They are free to move at will, creating a continuously changing network topology in which links are created and destroyed in often rapid succession[†]. MANETs have no central point of administration, nor are they supported by any fixed infrastructure. Instead, due to the routing capabilities of the nodes themselves, the network is self-organizing. Nodes that are not in each other's radio range may communicate by means of multi-hop routing, in which nodes that interconnect the communication partners aid in the forwarding of data. Nodes that make up the MANET are often battery-powered with little processing power; links tend to be error-prone with only limited bandwidth.

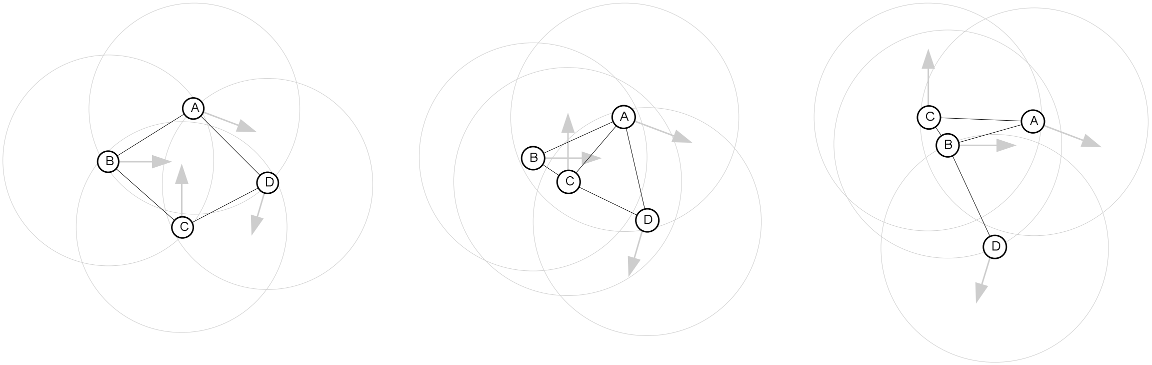


Figure 1.1. A MANET. The circles around the nodes represent the reach of a nodes' radio, the arrows represent the direction in which the nodes are moving. The solid lines between the nodes represent communication links; they change as the nodes change their respective locations. Taken together the nodes and links form the vertices and edges of a connected, undirected graph. □

With the steady growth of consumer products that can act as wireless mobile routers (laptops, mobile phones, PDAs), MANETs are becoming increasingly common and resources are shifting towards the edge of the network (i.e., away from wired backbone routers). With this growth, the need for efficient multi-hop routing protocols rises. One can imagine however that efficient end-to-end communication between any two non-neighbouring nodes in a MANET is a task of some complexity [3]. Due to its volatile nature, communicating over a MANET is unreliable and takes a

[†] Ironically, in Latin 'manet' is the third person simple present verbform of 'manere', which means 'to stay' or 'to remain', making MANET a rather ill-chosen acronym.

relatively long time, forming a highly unattractive networking environment. Although considerable research efforts have been made and there exist quite a number of proposals for routing solutions, so far no solution has seen widespread (if any) adoption.

Part of the problem of traditional routing in MANETs is that traditional protocols (be them in the application layer, network layer or elsewhere) are address-based, and require that all communication partners know each other. Although this does not seem an unreasonable demand, it does in fact place a burden on routing that can often be avoided. Internet radio stations, peer-to-peer filesharing and group conferences are all examples of applications that are mainly interested in *what* they exchange; not with *whom*. Such applications may benefit from content-based routing (CBR).

In CBR there are two actors: someone who has content to offer (usually refered to as a publisher) and someone who wants to receive the content (usually refered to as a subscriber). Both parties do no need to know each other: in stead they rely on the network to route published content to nodes that are interested. The network must therefore base its routing decisions on the information being routed (i.e., the payload of the packets), rather than on the identities of the communicating partners as is the case with address-based routing.

The main problem of any CBR is how to route content both effectively (everyone gets the content they want) and efficiently (content only reaches those parts of a network where it is wanted). Systems are often compared by the percentage of subscribers that get what they want (called the completeness ratio), the percentage of messages that are received that are actually wanted (called the precision ratio) and the overhead that the system's routing scheme places on the network (this includes the forwarding of the content itself).

CBR was conceived as a means of communication better suited to the peer-to-peer model, which is somewhat similar to a MANET. Quite some research has been performed on CBR, albeit that most of this research has so far focused on static networks. But with the increasing number of MANETs and the promise the CBR model holds, the subject seems likely to draw some more attention in the researching community in upcoming years.

1.2 Motivation

This masters assignment began with the motivation to "design a novel CBR system for MANETs using Bloom filters". The object was to apply Bloom filters in a similar way as has been done in for instance [4]. This turned out not to be possible however, so the motivation was reduced to (the somewhat vague notion to) "design a novel CBR system for MANETs". To be further specified based on the results of a preliminary study on the available work on CBR in MANETs, see [5]. A summary of the results of this study has been included in Chapter 9 of this report.

Not suprisingly it turned out that already quite some research has been done on creating solutions intended to solve the problems associated with CBR in MANETs (there are many). The majority of these solutions are however rather static in their nature. A designer will start by making an assumption about the expected number of users for its system, and their respective roles. From that point forward the system is designed and later perfected to meet those conditions.

Inspired by an article by Liu, Huang & Zhang [6], which advocates a design in which two communication parties balance the effort they both make to communicate with each other based on their respective needs, I set out to design a system that would be able to adapt to differing needs and circumstances. How this eventually has been done is described in the next section.

The system presented in this thesis makes use of a backbone, of flexible size, that acts as a medium to publishers and subscribers to route their content over. In this the design closely follows the classical publish/subscribe paradigm, in which publishers and subscribers do not communicate directly with each other, but via a medium that bears all communication responsibilities: here the backbone acts as that medium. Each node outside the backbone has a variable-length path of next hops to the backbone. By parameterizing the maximum length of a node's path to the backbone the backbone's size is controlled.

The routing overhead of this design is twofold:

1. the overhead to maintain the backbone and a path towards it for each actor outside the backbone;
2. the overhead of routing the actual content.

Maintaining the backbone normally gives more overhead than maintaining the paths towards it. Any content published outside the backbone is first routed to the backbone. Inside the backbone routing differs: it is (i) either flooded round the entire backbone or (ii) routed only to those parts of the network that the content is wanted. The first type of backbone is called a dumb backbone, the second type a smart backbone. A dumb backbone has less maintenance overhead but less accurate routing, a smart backbone gives more efficient routing but is harder to maintain (especially as networks become more mobile). The type of the backbone is also parameterized. Finally content is routed to any subscribers outside the backbone. Nodes that are not part of the backbone and are not part of a subscriber's path towards the backbone do not receive any published content at all.

As stated before, CBR systems are compared by their completeness, precision and the overhead they cause. As the overhead for the larger part is made up of the forwarding of the actual content, the key focus of most designs lies on creating a routing scheme that performs as efficient as possible. The efficiency of a scheme may however gravely be influenced by the number of active participants, and the amount of content they have to offer. A flooding scheme can for instance be both effective and efficient when almost every node in the network is a subscriber, but as the number of subscribers drops, do does the scheme's efficiency. A scheme in which every publishers maintains a separate route to every subscriber may lead to efficient routing when there are only a few subscribers, but as their number goes up the amount of effort needed to maintain all routes may easily overshadow those gains. In a similar fashion the efficiency of the design presented here may differ.

Figure 1.2 shows four instances of the same network: two with a small backbone and two with a large backbone. For each type of backbone-size the number of active participants is also varied: one case has only a few participants, in the other all nodes in the network are participating. Defining content load as the rate with which content is published multiplied by the number of participants in the system, the key point of the design is as follows:

> *For a given network and a given content load, there exists a backbone size and a backbone type for which the total overhead is minimal.* (1.1)

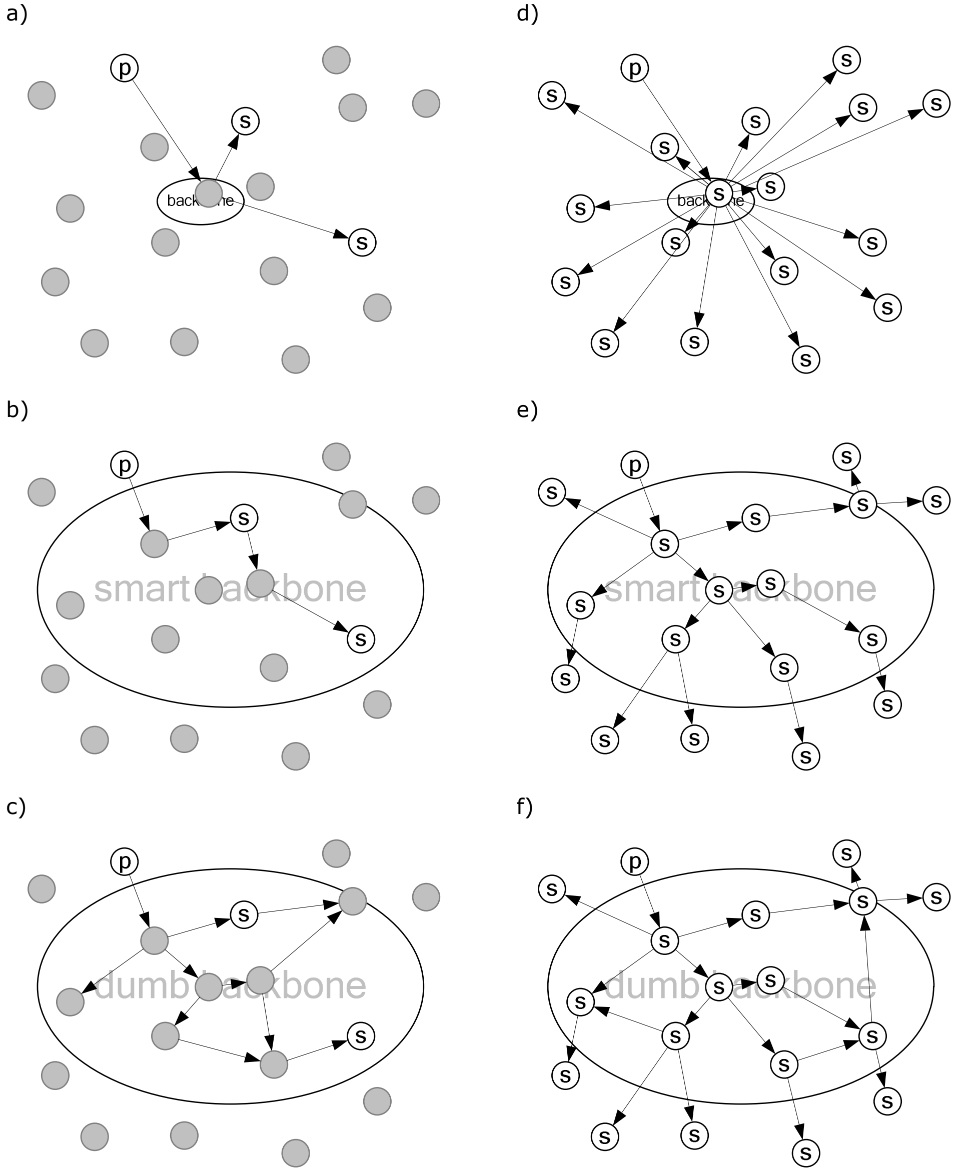


Figure 1.2. Six times the same network but with backbones of different sizes and of different type and different numbers of actors. Nodes labeled 'p' are publishers, nodes labeled 's' are subscribers. Grey nodes are neither. The backbone is visualized by the oval: all nodes in the oval together make up the backbone. The arrows denote content being routed. Whether the small backbones are smart or dumb has been left unspecified as it does not matter in the case of only one node.

Figure 1.2 visualizes this statement: in 1.2.a overhead is low because only a small part of the network is involved in maintaining the backbone and its paths, and messages are forwarded relatively efficient. In 1.2.b and 1.2.c however the backbone is much larger, causing increased overhead for maintaining it, while the content load has not changed. If the backbone is dumb (fig. 1.2.c), content will be routed to a large part of the network, causing unnecessary overhead. If the backbone is smart (fig. 1.2.b) it will be routed only to those parts where it is needed. Whether this will make up for the increased effort needed to maintain such a large backbone is unclear however, and depends on the rate with which content is published (for a static number of actors). Comparing Figure 1.2.d, 1.2.e, and 1.2.f, in which all nodes are active participants, it is expected that a large backbone (fig. 1.2.e and 1.2.f) is the most efficient, as it will enable more efficient routing than is the case with a small backbone (fig. 1.2.d). Whether a dumb or a smart backbone is more efficient depends on the rate with which content is published *and* on whether or not it is at all possible to maintain a smart backbone as mobility increases.

1.3 Approach

The CBR system has been designed using a two-layered approach: the bottom layer is responsible for creating and maintaining a backbone and giving each node that is not part of the backbone a next hop towards it. The topology that is thus created is used by the upper layer to route content over. Figure 1.3 exemplifies the design.

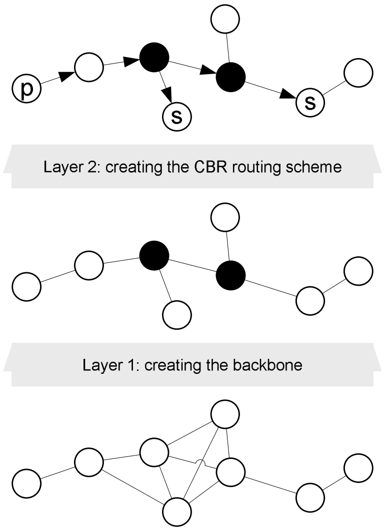

p
s
s
Layer 2: creating the CBR routing scheme
Layer 1: creating the backbone

Figure 1.3. The two-layered design of the CBR system.

Creating a backbone of flexible size in a MANET turned out to be quite the challenge, as (to the best of my knowledge) there are no techniques available that can do this. In the end it was necessary to design such a system myself, based on an algorithm by Yang et al. in [7]. In their paper they present an algorithm for creating a connected k-hop cominating set (Ck-HDS) in a MANET. Or in plain English: their algorithm is capable of creating a backbone in which every node outside the backbone has a path towards the backbone of max *k* hops. This is exactly what is needed here. Their algorithm is however based on a number of assumptions that are unreasonable in a MANET, such as a per-node perfect knowledge of the topology. In part II of this thesis the design, specification and analysis of a novel protocol is presented that is capable of creating a Ck-HDS in a MANET, using only the assumption that nodes have symmetric radio links. Within the whole system this bottom layer is refered to as the Ck-HDS layer.

In part III the CBR system is presented and analysed. The view that the CBR system has of the network is solely determined by the Ck-HDS layer: a node only communicates with nodes that are presented to it as neighbour. Using this topology the CBR layer tries to build a routing layer to route content over.

1.4 Research goals

The two layers have been designed and analysed separately and their research goals are given in a likewise fashion.

The goal of part II is (i) to design a protocol capable of creating a Ck-HDS in a MANET using only the assumption of symmetric wireless links and (ii) to show by means of simulation how well it is capable of maintaining this set in the face of mobility.

The goal of part III is (i) to design a CBR system that is able to make use of the flexibility of the Ck-HDS algorithm and (ii) to show how well it is able to balance the content load while still maintaining high completeness and precision. Chapter 10 gives a slightly more indept view on the CBR system and formulates a number of specific design questions.

1.5 Outline

This part of the thesis is meant as an introduction for the thesis as a whole. In Chapter 2 the methods that have been used for analysis in both part II and III are discussed.

The second part only concerns the design of a novel algorithm capable of creating Ck-HDS algorithm in a MANET. Ck-HDSs are introduces in Chapter 3, and Yang et al.’s algorithm is explained in Chapter 4. In Chapter 5 the design is presented and discussed; the accompanying specification is given in Chapter 6. Chapter 7 gives a performance analysis of the design by means of simulation and Chapter 8 ends part II with conclusions and some ideas for future work.

In part III CBR is introduced in Chapter 9, and the novel CBR design is presented and discussed in Chapter 10. Its specification is given in Chapter 11, a performance analysis by means of simulation in Chapter 12. Chapter 13 concludes the thesis with conclusions on the designed CBR system and tips for future work. ,

Chapter 2
Some notes on analysis

Analysing the performance of a protocol is anything if not hard. It involves choosing a method (construction of a mathematical model, testing or simulation) suited to the protocol and the amount of resources (time, money) at hand. It must have a level of abstraction that covers all (and ideally only) the important details without oversimplifying the design, which could give results that can both be erroneous and ambiguous. The presented results must be statistically valid. Above all it must be repeatable.

Especially in MANETs the number of existing details (multiple layers operating on top of each other, mobility patterns, radio interference) make it hard to perform an analysis that has enough detail to be credible, while still being repeatable. Although live tests will show you whether or not a protocol has any practical value it (i) lacks repeatability, (ii) necessarily encompasses all details making it hard to gauge their effects and (iii) requires a lot of resources. A mathematical model or a simulation, which are both abstractions of reality, will give you full control over all details and perfect repeatability. Mathematically modeling a protocol is often enormously complex however, whereas simulation tools are perfectly suited for just such a case. For these reasons, simulation is the performance analysis method of choice for the MANET community, as well as for this thesis.

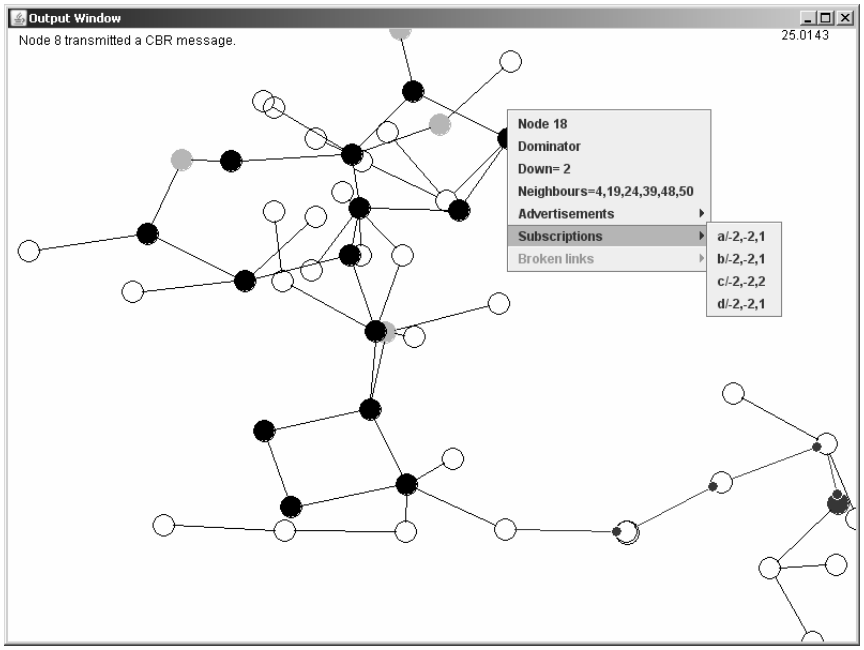
Being the analysis method of choice does not guarantee validity. A survey performed by Kurkowski, Camp & Colagrosso [8] on 114 papers published between 2002 and 2005 in ‘Proceedings of the ACM International Symposium on Mobile Ad Hoc Networking and Computing’ showed, amongst others, that 85% of the presented simulations failed to give enough information to ensure repeatability. Other studies [9,10,11] show even more alarming facts. Perhaps the most worrying of these is the analysis performed in [11], where a simple flooding protocol implemented in a number of simulators gave as much results as there were simulators. Simulations, it should be clear, by no means guarantee results that will prove to be perfectly accurate when held against a real world experiment.

In this thesis the advice voiced by Andel & Yasinsac in [9] – who hold that “MANET simulations [should be used] to provide proof of concept and general performance characteristics” – is followed. Both in the case of the Ck-HDS algorithm and the CBR protocol a conceptual model has been implemented that holds all algorithm/protocol functionality described in this document, but few other details. Section 2.2 presents the model in full detail. The model has explicilty been designed to conduct proof-of-concept experiments, not to compare the performance of two protocols in any realistic way.

Network simulator OPNET (version 12.0.A, see [28]) was used to implement the Ck-HDS and CBR system. However, as the simplified simulation model does not make use of any specific OPNET models, the designs could have been implemented and tested in any way, achieving similar results.

2.1 Verification

Both the Ck-HDS algorithm and the CBR system have been implemented at a conceptual level (see Section 2.3, A simplified simulation model). In both cases correct operation of the implementation was verified by means of custom made Java-based tool. This tool is able to visualize the behaviour that occurred at a simulation run, by analyzing a log file generated during the run. Figure 2.1 shows a screenshot of the application.



Node 8 transmitted a CBR message.

25.0143

Node 18
Dominator
Down= 2
Neighbours=4,19,24,39,48,50

a/-2,-2,1
b/-2,-2,1
c/-2,-2,2
d/-2,-2,1

Figure 2.1. The Java-based tool able to visualize both the Ck-HDS system and the CBR system.

2.2 A simplified simulation model

Nodes are in one of three states: inactive, active, or failed. An inactive node is a node that is waiting to become active; a failed node is a node that was previously active but has become inactive. Once a node has entered the failed state it will never leave that state. Only active nodes are able to transmit/receive packets, perform computations and move around.

Nodes are modeled as points in a two-dimensional plain: their position can be represented by an (x,y) coordinate. Mobility is achieved by altering this coordinate. Two nodes can communicate with each other if they are neighbours, i.e., if the distance between their relative positions is smaller than or equal to a node's *transmission range*, which differs per simulation. To give a somewhat realistic feel, both positions and distances presented in this document will usually be expressed in meters.

For mobility the Random Waypoint Model with uniform and stable speeds in [1] was used. In this model nodes move within a defined region. At startup every node chooses uniformly a point within this region as its destination, and draws uniformly a speed from the the range [V_{min},V_{max}]. Then whenever a node has reached its destination it will again uniformly choose a destination, but its speed is now drawn according to the distribution $F_V(x) = \frac{x^2 - V_{\min}^2}{V_{\max}^2 - V_{\min}^2}$. This ensures that the average node speed for a network is stable at $\frac{V_{\max} - V_{\min}}{2}$, refered to as $V_{average}$, from the start of the simulation and throughout the whole simulation. The only effect that one should still bear in mind is that if nodes are distributed uniformly over a mobility region, they tend to be 'drawn inside' at the start of a simulation. This is because of the fact that with the random mobility model nodes on average tend to spend more time in the middle of the mobility region.

System-specific packets are used to exchange information between nodes. Packets can either be addressed to a single neighbour (unicast) or to all neighbours at once (broadcast). Addressing nodes is done by means of unique identifiers, which are pre-assigned to each node and can be compared to a MAC-address. A node is only allowed to perform unicast transmissions with nodes whose identifiers it knows; a broadcast transmission uses no addressing but is delivered to each and every neighbour of the sender.

Nodes are not able to interfere each other's communication, so both contention and collisions are ignored. In stead a transmission probability is introduced, T_p, which determines the chance whether a packet reaches a receiver. Communication either succeeds or fails and does so instantaneously (there is no transmission propagation). In case of a unicast transmission the sending node is informed whether a transmission was succesful or not. In case of a broadcast transmission the sending node has no way of telling whether a transmission succeeded or not.

2.3 Creating network graphs for static simulations

To ensure that the networks used in static simulation runs have a random topology but are still connected, the Ad Hoc Network Graph Model was used, described in [12]. Setting *R* as the transmission range, a graph $G_n(a,b,g)$, consisting of *n* nodes and with parameters $0 < a,b,g \leq 1$, is generated with this model from a graph G_{n-1} via three steps (a graph G_1 is created by simply initializing a node with some given coordinates):

1. one node *u* in G_{n-1} is randomly chosen by means of the distribution $Pr(u) = g \cdot (1 - g)^{|N(u)|-1}$;
2. taking the coordinates of node *u* as a starting point a new node *v* is initialized at random polar coordinates *(r, t)*, with $r = a \cdot R$, $0 \leq t \leq 2 \cdot b \cdot \pi$;
3. for every node *w* already present in G_{n-1} the edge *vw* is drawn if the distance between the two nodes is at most *R*.

The density of a generated network increases if parameter a decreases or *g* increases: a smaller value for *a* leads to nodes being initialized more closely together, while a bigger value for *g* increases the chance in step 1 that a node *u* is chosen with a small set of neighbour. Parameter *b* influences the shape of the network: a smaller value gives a more elongated network.

PART II

The implementation & evaluation of a distributed connected k-hop dominating set algorithm for mobile ad-hoc networks

This part focuses on the implementation and evaluation of an algorithm that can create and maintain a Ck-HDS in a MANET. To the best of our knowledge no existing work on the implementation of such an alg exists. The only available work is a small set of theorethical algorithms that all make assumptions that are unrealistic in a MANET, such as perfect topological knowledge. Designing an algorithm from scratch would take too much time, however, so the algorithm that would require the least effort was eventually chosen to be implemented. This was the algorithm presented by Yang et al. in [7], mainly because it was the only available algorithm that addressed maintenance of a Ck-HDS during mobility.

This part is organized as follows: it starts with a short introduction explaining what a Ck-HDS is in Chapter 3, which also list the available work on Ck-HDSs in MANETs. Next, the chosen algorithm is briefly explained in Chapter 4 and the efforts needed to make it suitable for use in a MANET are discussed in Chapter 5. Chapter 6 describes the resulting specification. Experiments performed on the implementation and their results are described in Chapter 7, followed by some conclusions and suggestions for further work in Chapter 8.

Chapter 3
Creating and maintaining a connected k-hop dominating set in a mobile ad-hoc network

Creating and maintaining a connected *k*-hop dominating set (C*k*-HDS) in a static network with stable links is not very difficult. When it comes to a mobile ad-hoc network (MANET) however the volability of the network links becomes a significant problem. After introducing C*k*-HDSs in Section 3.1, Section 3.2 shows available work on C*k*-HDSs in MANETs. It turned out that, as far as could be verified, there exists no practical solution to this problem. Section 3.3 described how available work of Yang et al. [7] has been used to design such a system.

3.1 Connected *k*-hop dominating sets

Consider a connected and undirected graph *G(v,e)*, such as can be created from a MANET by taking the nodes as vertices and the communication links as edges (see also Figure 1.1). Furthermore assume the following definitions to be true:

Two nodes (or vertices) are said to be connected if they share an edge. (3.1)

A path *is defined as a set of nodes* $n_0 \cdots n_{i-1} n_i n_{i+1} \cdots n_n$ *wherein for 0 <* i < n *each node* n_i *is connected to nodes* n_{i-1} *and* n_{i+1}, *and nodes* n_0 *and* n_n *are respectively connected to nodes* n_1 *and* n_{n-1}. *The length of a path is equal to the number of nodes that make up the path: a path* $n_0n_1n_2$ *has length 3.* (3.2)

Using the above definitions we make the following incremental statements:

A dominating set (DS) D *in* G *is defined as a set of nodes (or vertices) for which holds that every node in* G *is either part of* D *or is connected to a node that is part of* D. *A node that is in a dominating set is called a dominator. A node that is outside the DS is called a member. If a dominator has any members connected to itself it is also called a dominating node.* (3.3)

A *k-hop dominating* set *(k-HDS)* D *is a set for which holds that every node in* G *is either part of the DS or has a path of member nodes of length k or less (excluding the node itself) to a dominator. This path is called a member path.* (3.4)

A connected dominating set (CDS) D *is a k-hop dominating set for which holds that for every dominator* d *in* D *there exists a path consisting solely of dominators to every other dominator in* D. (3.5)

A connected k-hop dominating set (Ck-HDS) D *is a k-hop dominating set for which holds that for every dominator* d *in* D *there exists a path consisting solely of dominators to every other dominator in* D*.* (3.6)

Figure 3.1 shows some examples of (connected) *k*-hop dominating sets.

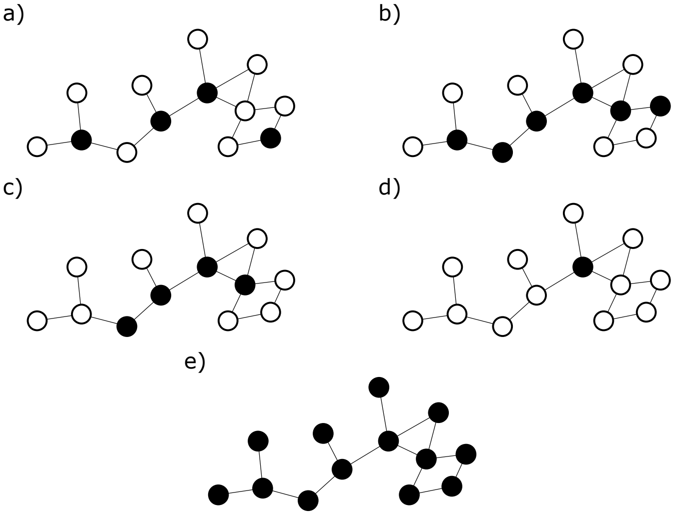


Figure 3.1. Figures a-e show the same graph but with differing dominating sets. The dark colored nodes are dominators. a) shows a 1-HDS, b) shows a C1-HDS, c) shows a C2-HDS, d) shows a C*k*-HDS wherein k can have any value higher than 3 and e) shows a C0-HDS. Because the maximum distance to the DS is 0 in e) all nodes are part of the DS.

As the examples in Figure 3.1 show a C*k*-HDS is a suitable means for creating a backbone in a network, with the resulting CDS as the backbone itself. By varying the size of *k* the size of the backbone can be varied: a small *k* will produce a large backbone with short member paths, while a large *k* will give a small backbone with long member paths. Such a backbone of flexible size is exactly what is needed in the content-based routing (CBR) system introduced in Part I and fully described in Part III.

3.2 Available work

The idea of creating a C1-HDS (or CDS) in a (mobile) ad-hoc network is far from new, so a number of solutions are available for this simplified version of a C*k*-HDS. In general these solutions use either one of the following two strategies:

1. First a DS is created: this is called clustering. Then the dominators making up the DS are connected by means of special connector nodes. Ideally the dominators are connected by means of a minimum spanning tree.
2. Initially, all nodes enter the DS. Then each node for which holds that after its removal from the DS the DS is still connected, is removed in a distributed fashion. The problem here lies in determining which nodes can leave the DS.

Examples of the first strategy can be found in [19], [20] and [21]; of the second in [22], [23] and [24]. Although crucial for correct operation in a MANET, maintaining a

CDS during reconfiguration of the network due to node movement is often considered as a different problem and therefore not always addressed (although in the cases above only [1] does *not* address maintenance).

When it comes to C*k*-HDSs available work is scarce. To the best of our knowledge there exists no actual implementation of a Ck-HDS algorithm specifically designed for MANETs. [7], [25], [26] and [27] each present an algorithm, but all four designs are purely theorethical and are based on assumptions that will often not hold in a MANET, such as perfect topological knowledge. Of these papers only [7] by Yang et al. addresses the issue of maintaining the C*k*-HDS during reconfiguration of the network (e.g., due to mobility). Being the most recent paper of the four the paper also shows, for different properties of the algorithm, a number of analyses of its performance compared to the other three algorithms mentioned above. In these Yang et al.'s algorithms it performs either equally good or better than the other algorithms.

3.3 Designing a protocol able to create and maintain a connected k-hop dominating set in a mobile ad-hoc network

As no protocol exists that is able to create and maintain a C*k*-HDS in a MANET (needed for the CBR system described in Part III) it was necessary to design such a protocol myself. This Part describes the design, implementation and analysis of the resulting protocol. The algorithm by Yang et al. has been used as a basis for the design: the design itself mainly focuses on creating a fully distributed version of the original algorithm, capable of operation in a MANET.

Henceforth in this thesis 'the (orignal) algorithm' refers to the original algorithm by Yang et al. and 'the (distributed) protocol' refers to the fully distributed protocol designed in this thesis.

Yang et al.'s algorithm has been selected as the basis for the new protocol because it was at the time the only available algorithm that addresses the maintenance of a C*k*-HDS. It is however far from suitable for use in a MANET. The two main lacunae with regard to its use in a MANET, and which have been the focus of the design efforts of the distributed protocol, are the following:

1. nodes know at all times who their neighbours are;
2. communication has been left unspecified.

In Chapter 4 the algorithm is described in detail. Chapter 5 then describes the design of the protocol based on the algorithm; its is fully specified in Chapter 6. Chapter 7 gives an analysis of the protocol by means of simulation and Chapter 8 gives conclusions and tips for future developments.

3.4 Some notes on backbone networks in mobile ad-hoc networks

Graph theory plays an important part in the formation of routing structures in computer networks, such as for instance a backbone, which is a quite common routing structure in fixed communication networks. Cellular networks use a wired backbone to connect access points, and most of your internet-traffic will eventually find its way along one or a few high-capacity backbone links. Backbones may aid in providing flow-support, multicasting and fault-tolerant routing. So far however, similar routing structures have scarcely been put to use in mobile ad-hoc networks.

This is mainly due to the volatile nature of links in a MANET, which considerably restricts the reliability of any multi-hop route and hampers the maintenance of the backbone. Another reason is that in a network where each node has only a limited lifetime (since, in the MANET scenario's considered, nodes are usually battery powered), it may not be such a good idea to concentrate traffic on a small set of nodes, as this may lead to failure of those nodes. Factors influencing the efficiency of a backbone differ widely and this (part of the) thesis neither promotes nor discourages the use of a backbone.

Chapter 4
A distributed algorithm for efficient construction and maintenance of connected k-hop dominating sets in mobile ad-hoc networks

In this chapter the behaviour of Yang et al.'s algorithm and the resulting C*k*-HDS structure that it imposes on the network is described. It should be noted that although the algorithm is distributed in its behaviour (nodes independently base their decisions on local knowledge) some parts of it are not distributed. The algorithm is not meant for direct application in a MANET but serves as a starting point for further research such as has been performed in this Part of the thesis.

4.1 The structure of the connected k-hop dominating set

Two nodes in a network are said to be neighbours if they are within each other's transmission range. Nodes know at all times who their neighbours are. The addition or loss of a neighbour is instantaneously noticed. The set of neighbours of a node *x* is referred to as the neighbour set of *x*, or N*(x)*. Each node has an attribute *neighbours* that lists all its neighbours. The set of neighbours of *x* that are dominators is refered to as D*(x)*.

The algorithm structures the network in such a way that a Ck-HDS is created in which the DS may have cycles and each member has a next hop (called the parent) towards the DS, which is the first hop of a path of length *k* or less. Paths are not necessarily shortest paths. Each node *x* (either dominator or member) that acts as a parent for another node *y* (which by definition is a member as dominators have no parents) refers to *y* as its child. Any children of children (and their children, etc.) are called descendants. So if *y* should have a node *p* as its child which in turn has a node *q* as its child then both *p* and *q* are descendants of *x*. Node also know their own distance to the DS (in hops), refered to with the attribute *up*. For a node x the path length of the descendant that has the longest path to *x* is refered to with the attribute *down*.

Each node has a chosen number and dominators also have a priority number; these numbers define the C*k*-HDS and are constantly revaluated. Member nodes that are situated closer to the DS generally have a higher chosen number; by definition dominators have ∞ as their chosen number. The parent set for a member *x* (*P(x)*) is the set of neighbours that all have a higher chosen number than *x* (dominators don't have a parent set). The priority numbers are used to impose a hierarchy on the DS when dominators wish to leave the DS.

Figure 4.1 shows an example C*k*-HDS structure with some of the attributes specified. It can be seen how the DS has a cycle and the chosen numbers decrease as the nodes are situated farther from the dominating set. Table 4.1 lists all notations and definitions used here and Table 4.2 lists all attributes a node can have.

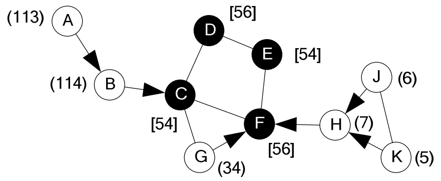


Figure 4.1. A C2-HDS. The characters in the nodes are the identifiers. The arrows denote the parent-child relations. The numbers in round brackets denote a node's chosen number and the numbers in the square brackets their priority numbers. In this example K.*up* = 2 and K.*down* = 0, H.*up* = 1 and H.*down* = 1, F.*up* = 0 and F.*down* = 2. The parent set of J, P(J), is made up by node H. The set D(G) is made up by nodes C and F.

Term	**Definition**
N(*x*)	The neighbour set of a node *x*. {*y* | "*x* and *y* are within each other's radio range"}
child	Node *x* is a child of node *y* if *y* is node x's parent (i.e., *x.parent* = *y*).
Ck-HDS structure	The structure imposed by the algorithm on the network, expressed in terms as dominators, members, etc..
D(*x*)	{*y* | *y* ∈ N(*x*) and *y*.num == ∞}
descendant	Node *x* is a descendant of node *y* if there exists a set of nodes $x_1,x_2,…,x_n$ with x_1=*x* and x_n=*y* such that x_i.*parent* = x_{i+1} for 1 ≤ *x* ≤ n-1.
P(*x*)	The parent set of *x*. {*y* | *y* є N(*x*) and (*y.num, y.id*) > (*x.num, x.id*)}

Table 4.1. Notations and definitions specific to the Ck-HDS algorithm.

Attribute	**Description**
id	The unique identifier of *x*.
num	The chosen number of *x*.
pri	The priority number of *x*.
up	The hop count from *x* to its dominator.
down	The maximum hop count from *x* to its descendants.
parent	A node *z* acts as a parent for a member *x* (*z* є P(*x*)) if it acts as *x*'s next hop towards the DS.
neighbours	The neighbour list of *x*.

Table 4.2. The attributes of a node x.

Any two nodes x and y can be ordered based on their identifers and chosen numbers. Let (*x.num*, *x.id*) > (*y.num*, *y.id*), said *x* has a larger chosen-identity number than *y*, if *x.num* > *y.num* or *x.num* == *y.num* and *x.id* > *y.id*. Finally let (*x.pri*, *x.id*) > (*y.pri, y.id*), said *x* has a larger priority than *y*, if *x.pri* > *y.pri* or *x.pri* == *y.pri* and *x.id* > *y.id*.

4.2 A behavioural specification of the algorithm

As theorethical algorithms for creating a (C)DS often tend to do, the algorithm starts at a certain time point at which every node in the network is put into the DS. After this point, nodes start a procedure for leaving the DS. Once a stable Ck-HDS has been constructed normal operation starts (described below). As it is impossible in a

MANET to demand that every node is present at some artificial point in time, so that construction of the Ck-HDS may commence, this first part of the algorithm is ignored. In stead we focus only on the normal operation.

At the heart of the algorithm lie two conditions which every node must adhere to:

1. For a node *x*, P*(x)* must induce a connected subgraph, else *x.num* = ∞ (*x* joins the DS).
2. For a node *x* there must be a node *y* in P*(x)* for which holds *x.down* + *y.up* < *k*, else *x.num* = ∞ (*x* joins the DS).

The first condition ensures that the backbone is at all times connected, proof of this can be found in [7]. The second condition ensures that each member has a path of length *k* or less to the DS. If any node in a network does not adhere to both these conditions, the Ck-HDS is considered invalid and the node will act so that it adheres to both conditions again. To ensure that the DS does not grow unnecessary large nodes may leave the DS whenever possible; this is explained in Section 4.2.1.

The decision for a node *x* whether P*(x)* forms a connected subgraph is based on *P(x)* itself and the neighbour list of each node *y* in P*(x)*. Note that the neighbour list does not indicate the role of a neighbour, so node D in Figure 4.1 has no way of knowing that it can simply leave the DS, because nodes C and E are connected by dominator F. In this way cycles will form in the DS whenever there exists a cycle in the network in which nodes are unable to leave the DS and form a connected parent set.

As nodes directly notice the addition or loss of a neighbour they will only act when the structure of the Ck-HDS is invalid. When they do act, they start by exchanging (a subset of) the attributes defined in Table 4.2 with each and every neighbour (the communication details of these exchanges are ignored). Nodes then update their attributes according to the rules that apply to the role the node fulfills, which are explained next.

In the following subsections the exact behaviour of a node is summed up in detail for each possible role that a node can be in: (i) dominator, (ii) member or (iii) a node that has just switched on.

4.2.1 The behaviour of a dominator

A dominator will leave the DS whenever this is possible without breaking the first condition and keeping every node within *k* hops of the DS. To prevent multiple dominant nodes from leaving the set at the same time – which could cause the DS to be disconnected – the priority property is introduced: only a dominator *x* that has a higher priority than any other node in D(*x*) may leave the set. If it leaves the set it sets its chosen number to a value higher than any other node in N*(x)*. If it cannot leave the set it sets its priority lower than any other node in D(*x*), so that another dominator may try to leave the set. The whole algorithm is as follows:

1. *x* echanges *x.id*, *x.num*, *x.parent*, *x.up*, *x.down* and {*y.id* | *y* ∈ *N(x)*} with every *y* ∈ *N(x)*, and additionally *x.pri* with every *y* ∈ *D(x)*.
2. *x* updates *x.down*.
3. If *x.pri* > *y.pri* for all *y* ∈ *D(x)* then *x* sets *x.num* such that *x.num* > *y.num* for all *y* ∈ *N(x)*. If *x.pri* ≤ *y.pri* for any *y* ∈ *D(x)* then exit.
4. If *P(x)* does not induce a connected subgraph or *x.down* + *z.up* ≥ *k* for all *z* ∈ P*(x)* then *x* resets *x.num* to ∞, *x.pri* is set such that *x.pri* < *y.pri* for all *y* ∈ D*(x)* and then the algorithm exits. Otherwise, *x* sets *x.parent* randomly to a node *y* ∈ P*(x)* such that *x.down* + *y.up* ≤ *k*.
5. *x.up* is updated.

It should be noted that with this algorithm the priority attribute of a node will always get smaller, never higher. Whenever some node's priority attribute reaches 0 the value of the priority attributes of *all* dominators are incremented[‡] (with some arbitrary number). This is of course not possible in a distributed network such as a MANET.

4.2.2 The behaviour of a member

A member x must join the dominating set if $P(x)$ does not induce a connected subgraph or if there is no neighbour $y \in N(x)$ that can act as a parent such that $x.down + y.up < k$. A member node stays faithful to its parent: it will not switch parent until either the parent z moves out of the node's range or $x.down + z.up \geq k$, even when there exist shorter routes to the dominating set (i.e., there exists at least one $y \in P(x)$ for which $y.up < z.up$ holds). The whole algorithm is as follows:

1. x echanges *x.id*, *x.num*, *x.parent*, *x.up* and *x.down* with every $y \in$ N*(x)*.
2. x updates x.up and x.down.
3. If P*(x)* does not induce a connected subgraph or $x.down + z.up \geq k$ for all $z \in$ P(*x*) then x resets *x.num* to ∞, *x.pri* is set such that $x.pri < y.pri$ for all $y \in$ D*(x)* and the algorithm proceeds to step 5.
4. Let $z = x.parent$. If $z \notin$ P*(x)* or $x.down + z.up \geq k$ then *x.parent* is randomly* set to a node $y \in$ P*(x)* such that $x.down + z.up < k$ and the algorithm exits.
5. *x.up* is updated.

4.2.3 The behaviour of a node that switches on

A node that switches on will preferably become a member, or if that isn't possible a dominator. The whole algorithm is as follows:

1. x receives *y.id, y.num, y.up* and N*(y)* from every neighbour y.
2. *x.num* is set such that $x.num < y.num$ for all $y \in$ N*(x)*.
3. If P*(x)* does not induce a connected subgraph or $z.up \geq k$ for all $z \in$ P*(x)* then *x.num* is set to ∞ and *x.pri* is set such that $x.pri < y.pri$ for all $y \in$ D*(x)*. Otherwise, *x.parent* is set to a node $y \in$ P*(x)* such that $y.up < k$.
4. *x.up* and *x.down* are updated.

4.3 Examples of operation

Now that the rules that the nodes follow have been laid out in the previous section, this section exemplifies their behaviour. Figure 4.2 shows a MANET with a valid C2-HDS structure that acts as the starting situation for each example. Figures 4.3, 4.4 and 4.5 respectively show what happens when:

1. a node that switches on can simply join the network as a member, without requiring reconfiguration of the C2-HDS;
2. a node that switches and joins the network requires some reconfiguration of the C2-HDS;
3. a dominator switches off.

[‡] Source: private communications with the author.

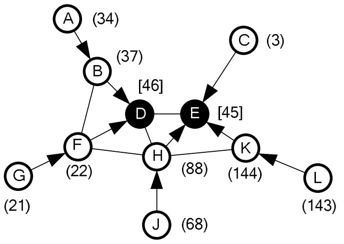


Figure 4.2. The reference MANET. The arrows represent the parent-child relations. Dark nodes represent dominators.

4.3.1 A simple join

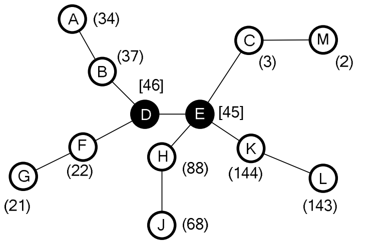
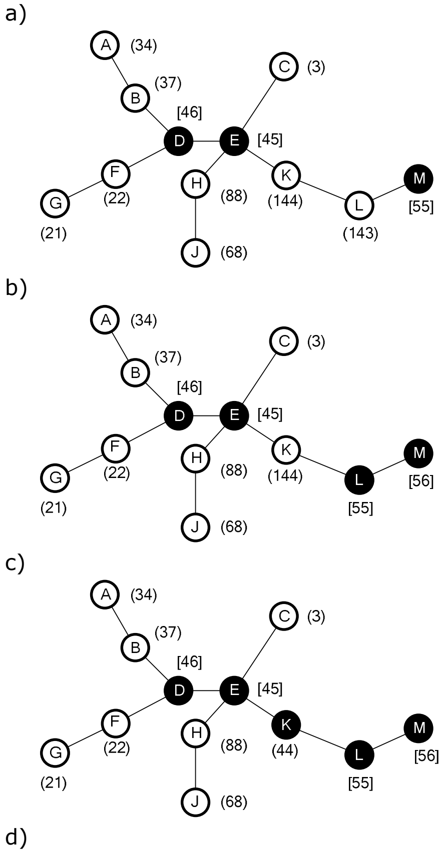


Figure 4.3. A similar MANET as the one in Figure 4.2. Only the Ck-HDS structure imposed on the network is shown. Node M is added to the C2-HDS structure without reconfiguration of the network.

Figure 4.3 shows how node M joins the network by simply choosing node C as its parent. The only effect this has is that node C will now have its down attribute set to 1 (node E's down attribute is unaffected because it already has two paths of length 2). Node M's chosen number is smaller than any of its neighbours' chosen numbers, so that P*(x)* remains connected for any neighbour *x* in N(M).

4.3.2 A join requiring a reconfiguration of the Ck-HDS



Node M again joins the reference network in Figure 4.4.a, now as a neighbour of node L. Because L.*up* < 2 does not hold (L.*up* is 2) node M has no alternative than to join the DS. Since it has no dominator as a neighbour it chooses a random priority number.

Since node L now no longer has a connected subgraph P(L) it also joins the DS, with a priority number lower than that of M. Node K acts likewise and joins the DS with a priority number lower than that of both E and L. When node K has joined the DS it is connected again.

After node L has joined the DS, node M will leave it again (which it can do because it has a higher priority number than node L). Node M assigns itself a random chosen number (since it has no member as a neighbour) and chooses L as its parent.

This process of leaving the DS is mimicked by L after K has joined it. It picks a chosen number higher than any of its neighbours that are not in the DS (i.e., node M) and node K as its parent. At this point the C2-

HDS has been fully reconfigured.

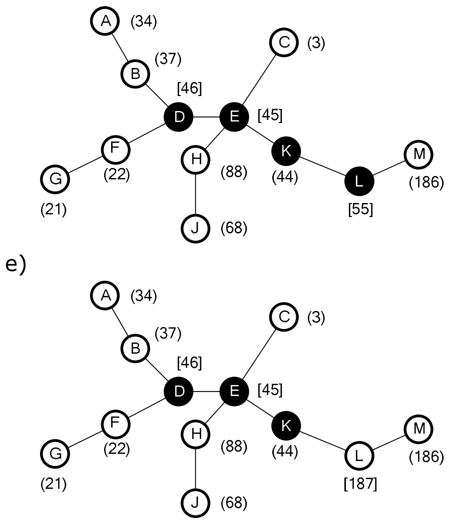


Figure 4.4. A similar MANET as the one in Figure 4.2. Only the C2-HDS structure of the MANET is shown. Node M’s appearance initiates a reconfiguration of the C2-HDS.

4.3.3 Loss of a dominator (recovery of the dominating set)

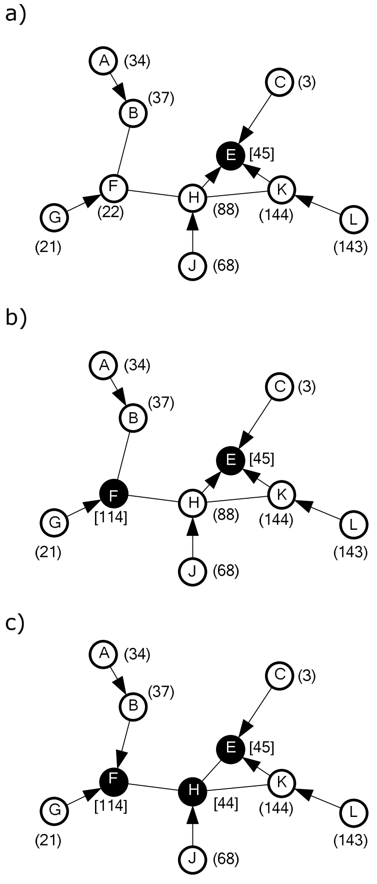


Figure 4.5. A similar MANET as the one in Figure 4.2. The arrows represent the parent-child relations. The loss of dominator D requires a reconfiguration of the C2-HDS.

Figure 4.5.a shows the state of the network right after dominator D has switched off. Node B no longer has a valid parent and node F neither has a valid parent nor a connected subgraph P(F). Either one of the two nodes may act first: in this example we start with node F.

F’s only option is to join the DS, which it does with a random priority number (as it has no neighbour in DS). Next, node B chooses F as its parent while node H, because it now no longer has a connected subgraph P(H), also joins the DS, with a priority number lower than either E and F. At this point the C2-HDS has been fully reconfigured.

Chapter 5
The design of a protocol for the creation and maintenance of a connected k-hop dominating set

This chapter presents a design that aims to lift the functionality of the Yang et al.'s algorithm to a fully distributed protocol. The standard behaviour of the protocol is as has been described in the previous chapter, the protocol only deviates from this behaviour in two places: the choice of a parent (Section 5.4) and the mechanism for leaving the DS. First however the communication details are discussed in Section 5.1 and the way a node constructs and maintains its own view of the network topology in Section 5.2 and 5.3.

5.1 Communication

Yang et al. use a rather abstract mode of communication in their paper, in which nodes are simply able to exchange attributes with their neighbours. They furthermore make use of symmetric radio links, in which nodes can always hear each other. In reality the issue of communication is of course rather more complex than that, and it may have a significant influence on the performance of the algorithm. The mode of communication is discussed below. The assumption of symmetric radio links has been maintained however, because time constraints prevent the redesign of the protocol that would be necessary to incorporate asymmetric links.

Two alternatives exist for communication: unicast and broadcast. Both have some issues when applied as the communication mode of choice. Broadcasting is, in terms of bandwidth efficiency, the most efficient form of communication as it enables a node to communicate its information to all of its neighbours at once, whether it knows those neighbours or not. Broadcasting lacks reliability however: as it does not make use of (negative) acknowledgements, a node has no guarantee that any node will receive its information and will not notice it when a neighbour has moved away. Alternatively, unicast does provide this reliability, making it a more reliable mode of communication ánd a source of accurate network knowledge. However, with only unicast transmissions two nodes that do not know each other never will, so clearly some use of broadcast cannot be avoided.

Because, especially Because it is the most efficient form of communication in a MANET where every node must receive every other node's information, and it is the only means to discover new communication partners, broadcast was chosen as the method for communication.

5.2 Beaconing

All nodes in [7] have perfect and instantaneous knowledge of their local topology. In this design nodes build and maintain their own view of the network by means of *beacons*: packets that contain topology information that are sent to any node within transmission range and that are soft state: the information contained in them is only valid for a certain amount of time. If after a certain period a beacon has not been updated the information contained in it is removed from the node's view. The use of beacons is no guarantee for a node that it will have a correct view of its network topology, nor is any other technique. The use of soft state beacons is however a good way to deal with the coming and going of neighbours without giving explicit notifications.

Below the steps are described that have been taken to construct and maintain an incrementally more accurate view of the network. This also involves some timing issues; these are specified in Section 5.6.

Each node regularly broadcasts a beacon containing all its state information. The (mean, as is shown later on) time between two successive beacons of a single node is a system parameter called the beacon interval (BI). The time between between the respective beacons of two neighbouring nodes is calle the inter beacon space (IBS). As nodes receive beacons from other nodes they can construct a view of their local topology. With every received beacon, the view of the network is updated. A beacon that has been received by a node is only valid at that node for a specified amount of time. After a period called the beacon timeout (BT) the beacon is no longer valid; any information contained in it that is not contained in beacons that a node has of other nodes is removed. *BT* is given as a ratio to *BI*. If two nodes move away from each other the beacons they have received from each other will eventually time out and the nodes will update their view without the other node in it. Table 5.1 shows the information contained in a beacon. It is almost fully based on the node attributes in the original algorithm. New attributes are introduced in their respective sections. Figure 5.1 shows an example of how nodes keep state of the network. Thanks to the beacons a node knows (with certain probability as new nodes may always come within transmission range and old nodes may always leave) its neighbours and for each neighbour its attributes: the identifier of the neighbour's neighbours, its identifier, its chosen number, its parent, its distance to the DS, its distance to its most distant descendant and any broken links (explained below) the neighbour has included.

Attribute name	**Type**	**Description**
id	integer	A unique identifier. Within this document nodes will often be identified by means of a letter in stead of an integer for readability purposes.
num	integer	The chosen number.
parent	integer	The identifier of the node's parent (-1 if the node is a dominator).
up	integer	The hop count from the member to its dominator (-1 if the node is a dominator).
down	integer	The maximum hop count from the node to its descendants (0 if the node has no descendants).
Nx	list of identifiers	The identifiers of all the node's neighbours.
broken links	list of tuples of identifiers	The broken links that the node knows of.

Table 5.1. The attributes that together make up a beacon. The priority attribute used in the original algorithm is no longer used (see Section 5.5).

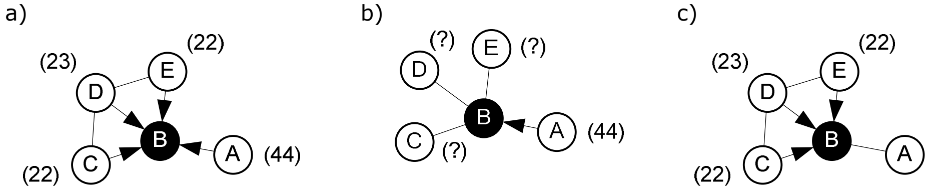


Figure 5.1. Black nodes are dominators, the arrows represent parent-child relations. a) represents a MANET, b) shows the view node A has of the network based on the beacons it received from its neighbours (in this case only node B) and c) shows the view node D has of the network. Node B has a fully accurate view of the network as it receives becaons from all nodes in the network. Table 5.2 shows the beacons of nodes A, B and C. Nodes are identified by means of a letter for readability purposes.

Attribute	**Node A**	**Node B**	**Node C**
id	A	B	C
num	44	∞	22
parent	B	-	B
up	1	0	1
down	0	1	0
Nx	{B}	{A,C,D,E}	{B,D}
broken links	{Ø}	{Ø}	{Ø}

Table 5.2. The beacons of nodes A, B and C in Figure 6.1. Nodes are identified by means of a letter for readability purposes.

The topology of the network continually changes through the (dis)appearance of communication links. There is some delay between the moment a link (dis)appears and the moment that the nodes involved notice its (dis)appearance. Furthermore, the nodes do not notice its (dis)appearance at the same time. Hence, a node may sometimes receive beacons that bear conflicting information. In such cases, newer information is trusted above older information.

Additionally, when nodes are aware that their neighbours have conflicting views of the network that may impair the Ck-HDS, they explicitly incorporate this information in their beacons. Two conflicting views of the network may impair the C*k*-HDS when for two previously connected nodes *x* and *y*, node *x* still believes that it is connected to *y*. The link between two such nodes is referred to as a broken link. If there exists a node *z* that has both *x* and *y* as neighbours, and *z* is aware of the conflicting views, then *z* will include the broken link between *x* and *y* in its beacon (in its *broken links* attribute, see Table 5.1).

The (dis)appearance of links in the network (e.g., due to mobility) may cause the C*k*-HDS structure to become invalid, for instance when two dominators become disconnected. Such events are here classified as either critical or non-critical. Critical events are events that disconnect either the dominating set or a members's path toward the dominating set. Because the design aims at keeping the Ck-HDS structure valid as much as possible the first responsible node to notice a critical event responds immediately with a *fast response*: a new beacon that is scheduled for transmission at a random moment within a time window called the Fast Response Window (FRW). Responsible nodes may either be nodes that are able to (i) recover the Ck-HDS, or if there is no such node present, (ii) further propagate the information that invalidates the Ck-HDS to the nodes that are able to recover the Ck-HDS. The time between the moment a responsible node notices that the Ck-HDS has been invalidated and the moment it transmits its beacon is called the response time. When there are multiple responsible nodes only the node with the shortest response time should respond fast; other nodes that also have a fast response scheduled cancel their fast respsonse if the first fast response has revalidated the C*k*-HDS structre. After node has scheduled a fast response it resumes its previous BI. Table 5.3 lists the critical events and the nodes that are responsible in such a case to respond fast.

Event	**Description**	**Responsible nodes**
A dominator loses a neighbouring dominator	A dominator notices that it has suddenly lost one of its neighbouring dominators, which indicates that the DS may be disconnected. It reacts by sending a fast response that no longer includes the lost node in its neighbour list, thus informing any receiver that the link with its previous neighbour is no longer valid. If the DS is truly disconnected then there must be at least one member *x* in the network that does not have a connected subgraph *P(x)*, which reacts by joining the DS by means of a fast response.	The two dominators d_1 and d_2, and any member *m* for which P*(m)* no longer forms a disconnected graph due to the disappearance of the link d_1d_2.
A parent loses a child	A parent *p* notices that it has lost a child *c* and reacts by sending a fast response, thus informing each node *x* that neighbours both parent and child (i.e., *p ϵ N(x)* and *x ϵ N(x)*) that the child no longer has a valid parent. Of this set of nodes the node with the highest chosen number reacts by also sending a fast response. When the child receives this response it will notice that it has no longer a valid parent, choose a new parent and send a fast response.	The parent, the child and any node *x* for which holds: *p* ϵ N*(x)* and *c* ϵ N*(x)*.
A child loses a parent	A child notices it has lost its parent; it reacts by choosing a new parent and sends a fast response.	The child.

Table 5.3. A number of critical events, the reaction of the Ck-HDS, and the nodes responsible for the reaction.

5.3 Beacon spacing

For the responsiveness of the system it is important that, for sets of neighbouring nodes, the moments that nodes beacon are spaced as evenly over time as possible: ideally each node in a set of *X* neighbouring nodes has an IBS of *BI*/*X* with respect to every other node in the set (see Figure 5.2).

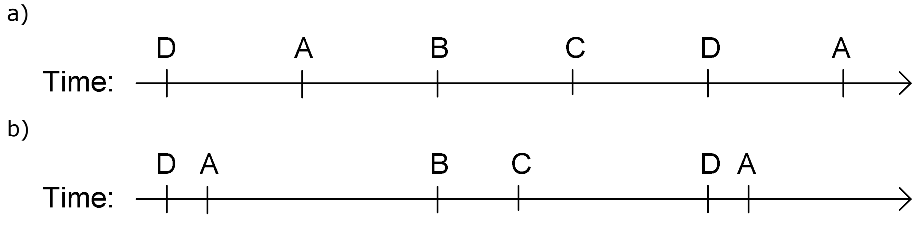


Figure 5.2. The beacon moments of a set of 4 nodes. In a) the beacons are perfectly spaced over time, in b) they aren't.

The farther that the IBS of two nodes lies from the ideal *BI*/2, the longer it will (on average) take for those nodes to respond to an event. Figure 5.3 shows how two neighbouring nodes that have a perfect IBS respond faster to an event to two nodes that haven't got a perfect IBS.

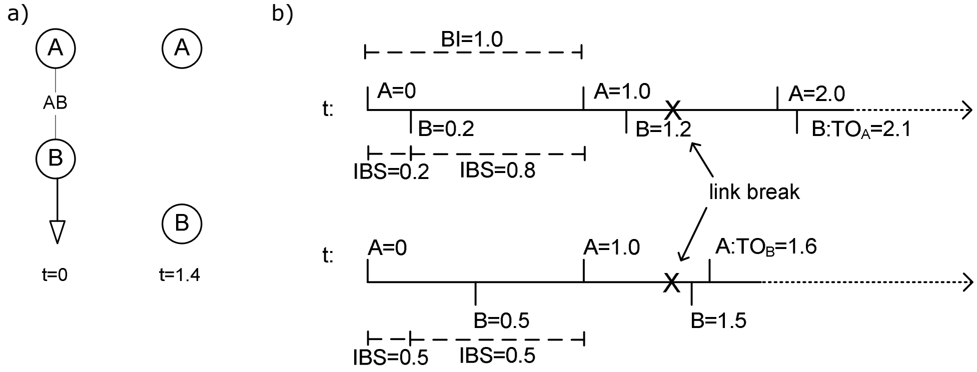


Figure 5.2. Two nodes move apart at time t=1.4. For the case where the inter beacon space is 0.2/0.8, the loss of link AB is noticed at t=2.1 because the beacon node B received at t=1.0 times out. For the case where the IBS is 0.5/0.5 the loss of the link is noticed at t=1.6 because the beacon node A received from B at t=0.5 timed out.

The worst case with regard to the IBS (and thus the average response time of the system) is the case when two nodes beacon close together, called in phase beaconing. Unfortunately this situation is a direct effect of the fast response machenism described in Section 5.2. Indeed, test runs of the implemented algorithm showed how after time large parts of the Ck-HDS were divided into sets of nodes that were beaconing in-phase due to their (often chained) reaction to an event. To counter this behaviour two timing windows are introduced in this section: the beacon window (BW) and the resume window (RW).

The BW is a window of time spaced evenly around a node's expected beacon moment, in which a node schedules its beacon. Thus, in stead of scheduling a beacon every BI seconds (disregarding fast responses), a node has some random variation in its BI. Introducing such randomness into the system will not prevent worst cases from occurring, and can even bring nodes that previously had their respective beacons perfectly spaced closer together. It does however succesfully ensure that the probability that neighbouring nodes stay in phase with each other for multiple BIs decreases as the size of the BW increases. BW is given as a ratio of BI. Note, however, that the maximum value of the BW should never exceed the BT (the exact timing constraints are listed in Section 5.6). The effect of RW is experimentally tested in Section 7.1.

The RW is the time window in which a node schedules the beacon following a fast response and is given as a ratio to BT. By choosing a suitably large value for RW the chances diminish that neighbouring nodes that responded fast to the same event will beacon in phase (this is experimentally tested in Section 7.3).

5.4 Choosing a parent

In the original algorithm a node *x* that becomes a member chooses a parent at random from the set of nodes in P*(x)* that have their attribute *up* < *k*, disregarding any further considerations. Once chosen the node stays faithful to its parent. As some of the underlying assumptions that have lead the authors to this choice are invalid in a practical situation (notably each node's instantaneous and perfect knowledge of the state of the network), we take a different approach and discuss it here.

Choosing a parent may have impact on the length and connectivity of member paths (the latter being dependent on the former). Solutions for choosing a parent range from the simple (choose a parent at random from the set of nodes in P*(x)* that have attribute *up* < *k*) to the more involved (base the decision on known parent-child relations in the neighbourhood). Furthermore, once a node has chosen a parent it can choose to keep that parent as long as possible ('stay faithful'), pick a better parent as soon as it finds one or revaluate its parent at a certain time in the future, or at the occurrence of a certain event.

Which strategy for choosing (and keeping) a parent performs best is hard to predict, not only because it differs per situation but also because the performance metric may differ. A strategy that ensures a stable C*k*-HDS may provide for very inefficient routing, and vice versa. Which of the two metrics is more important depends both on the environment in which the implemented algorithm will be deployed, and the way it will be used.

Ideally this document would describe a number of strategies for choosing a parent that give the best performance for a given performance metric (e.g. stability, length of member paths) and a given environment (e.g. a static network or a highly mobile one). As this is not possible due to lack of time a single strategy has been chosen, described below. A more in-depth solution such as described above is refered to as future work.

In the implemented algorithm a node *x* becoming a member chooses a parent at random from the set of nodes *Y* ($y_i \in Y$) for which hold:

1. $y_i \in$ P(*x*);
2. y_i*.up < (k - x.down)*;
3. there is no y_j ($y_j \in Y$, $y_j \in$ P(*x*), y_j*.up* < *(k – x.down*)) such that y_j*.up* < y_i*.up*.

A member will furthermore keep its parent until it becomes invalid. This design was chosen with the following (inconclusive, as was argued above) notions in mind:

1. Shorter paths are more stable.
2. Randomness of choice will generally prevent the creation of a single point of failure.
3. Dependency on state information of neighbouring nodes creates vulnerability, as there is a certain probability that the state information is outdated, so dependency should preferrably be kept low.
4. Stability of the C*k*-HDS depends on a per-node up to date view of the network. Whenever a member chooses a new parent, the view of its (former) neighbours is outdated: hence, continually choosing a new parent destabilizes the C*k*-HDS.

5.5 Dropping the priority number

In the original algorithm dominators use a priority number to determine which node is allowed to leave the DS. Its use is necessary as it prevents neighbouring dominators from leaving the DS at the same time, which could cause the DS to become disconnected. In this design however a dominator hasn't left the set until it has beaconed as such. Assuming that two neighbouring nodes in a MANET cannot transmit data at the same time this makes it impossible for two neighbouring dominators to leave the DS at the same time. As this makes the priority number obsolete it has not been implemented.

5.6 Timing constraints

Table 5.4 lists the timing constraints discussed in the previous sections, Figure 5.3 visualizes their boundaries.

Name	**Unit**	**Range**	**Description**
Beacon Interval (BI)	seconds	BI > 0	The mean time between any two consecutive beacons of a node.
Beacon Timeout (BT)	ratio to BI	BT > 1	The time after which the beacon is outdated.
Beacon Window (BW)	ratio to BI	0 ≤ BW < 2·(BT – 1)	The window in which the beacon will be transmitted; ranges over [BI – BW/2; BI + BW/2].
Fast Response Window (FRW)	ratio to BI	0 < FRW < 1	The window in which a fast response is scheduled; ranges over [0; FRW·BI].
Resume Window (RW)	ratio to BT	0 ≤ RW < 1	The window in which the regular beacon following a fast response is scheduled; ranges over [BT * (1 – RW); BT].

Table 5.4. The parameters that determine the timing constraints of beacons.

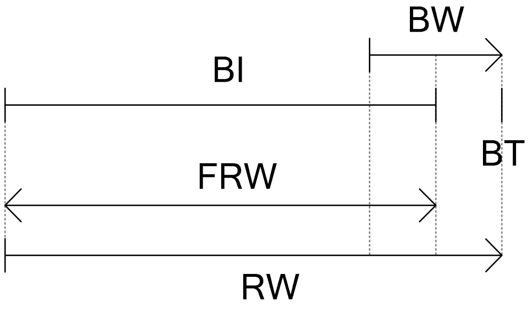


Figure 5.3. The timing windows of the C*k*-HDS protocol.

Chapter 6
Specification of the implemented algorithm

Below the all the rules governing the behaviour of the protocol are defined. Together with the specification of the beacon and the timing constraints in Chapter 5 this gives a complete definition of the protocol.

6.1 Generic behaviour

Figure 6.1 shows a role-independent, high-level state diagram of a node's behavioural lifecycle. After a node has switched on it immediately schedules a transmission of a beacon 1 BI later. Any beacons that are received in the interval between the node switching on and the first beacon are used to create a view of the network. When the time has come to transmit, the beacon is constructed based on the node's view and broadcasted to the node's neighbours; finally the next regular beacon is scheduled. With its first beacon a node proclaims its role and enters its active state, in which it will beacon regularly, keep its view of the network up to date by means of received or timed-out beacons up to date and (if necessary) respond fast to events that invalidate the C*k*-HDS. If an event occurs but a fast response is not necessary then beaconing continues as normal (i.e. with BIs ranging over the interval [*BI* – *BW*/2; *BI* + *BW*/2]).

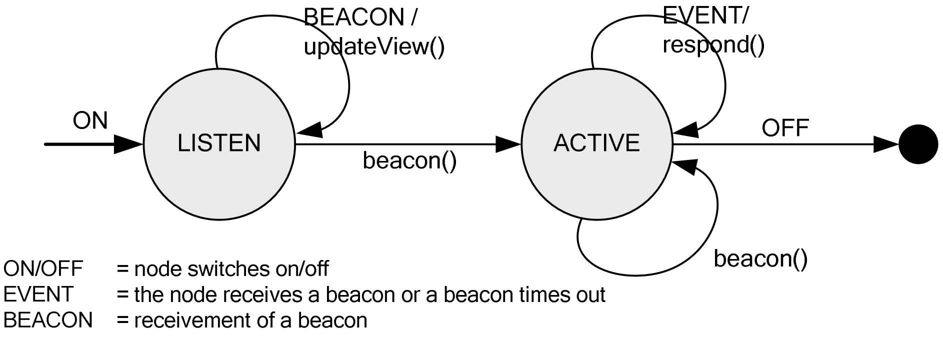

BEACON /
updateView()
EVENT/
respond()
ON
LISTEN
beacon()
ACTIVE
OFF
beacon()
ON/OFF = node switches on/off
EVENT = the node receives a beacon or a beacon times out
BEACON = receivement of a beacon

Figure 6.1. The generic cycle of behaviour for a node. Function updateView() updates the node's view of the network based on the newly received information. Function respond() first calls updateView() and if the view shows an invalid CkHDS structure it will respond by sending a new beacon.

Whether or not a fast response is necessary as well as constructing the beacon are role-dependent functions which are explained in the three sub-Sections 6.2, 6.3 and 6.4. First updating a node's view is explained in the next section.

6.1.1 Updating the view

A node's view is updated every time when (i) it receives a beacon or (ii) a beacon times out. As the two cases require quite distinct handling they are also treated as such below.

Whenever a node *x* receives a beacon from a node *y*, it will perform the following actions:

1. if *y* belongs to a new network (explained below) the newNetworkEncountered flag is set;
2. if not already present *y* is added to *x*'s view, else the information stored about *y* is updated;
3. any link *yr* (*r ϵ* N(*x*)) existing in *y's* view is added to *x's* view if it isn't already present;
4. any link *yz (z ϵ* N(*x*)) that exists in *x*'s view but does not exist in y's view is removed from *x*'s view and added to the set of broken links;
5. any broken link *qz* (*q, z* ϵ N(*x*)) in *y*'s view is removed from *x*'s view if present;
6. if *x* is a dominator the set of lost dominators is updated;
7. the set of lost children is updated;
8. if x is a member P(*x*) is recalculated;
9. *x.down* is updated.

A node *x* will conclude that a node *y* that it received a beacon of belongs to a new network when all of the following holds:

1. *x* has no previous beacon entry of *y*;
2. *x* is not listed as a neighbour of *y*;
3. none of x's neighbours are listed as neighbours by *y* or vice versa (N*(x)* and N*(y)* are disjunct).

Whenever the beacon that a node *x* received from a node *y* times out, the following actions are performed:

1. *y* is removed from *x*'s view;
2. if both *x* and *y* are (were) dominators then *y* is added to the set of lost dominators;
3. if *y* was a child of *x* then *y* is added to the set of lost children;
4. *x.down* is updated;
5. if *y* was the parent of *x* then *x*'s parent is updated.

6.2 The behaviour of a dominator

A dominator *x* deems a fast response necessary if:

1. the newNetworkEncountered flag is set;
2. the set of children that *x* has lost is non-empty;
3. all of the following hold:
    a. there exist in *x*'s view two nodes *y, z* (*y, z* ϵ N(*x*)) for which hold that *y.parent == z.id*;
    b. the link *yz* does not exist in *x*'s view;
    c. there exists no node *q (q ϵ* N(*x*), q ϵ N(*y*)) for which holds (*q.num, q.id*) > (*x.num, x.id*);
4. the set of lost dominators is non-empty.

The algorithm for creating a beacon as a dominator is as follows:

1. Set $x.up = 0$ and $x.parent = -1$
2. Let C be the set of x's children (i.e., $c_i.parent == x.id$) and c_j the child for which holds that $c_j.down \geq c_k.down$ for any k. Then set $x.down = c_j.down + 1$.
3. Try to set $x.num$ such that there is no member y ($y \in N(x)$) for which holds $y.num \geq x.num$. If this is not possible set $x.num = \infty$ and exit.
4. If $P(x)$ gives a connected graph choose a parent by picking a node z at random from the set of nodes Z for which holds: $z \in P(x)$, $z.up < k$ and there exists no node y ($y \in P(x)$) for which holds $y.up < z.up$. Then set $x.parent = z.id$ and $x.up = z.up + 1$. If $P(x)$ is not connected or if no such parent can be found than set $x.num = \infty$.

6.4 The behaviour of a member

A member x deems a fast response necessary if:

1. the newNetworkEncountered flag is set;
2. the set of children that x has lost is non-empty;
3. all of the following hold:
    a. there exist in x's view two nodes y, z ($y, z \in N(x)$) for which hold that $y.parent == z.id$;
    b. the link yz does not exist in x's view;
    c. there exists no node q ($q \in N(x)$, $q \in N(y)$) for which holds $(q.num, q.id) > (x.num, x.id)$;
4. $P(x)$ is disconnected;
5. $x.parent \notin P(x)$;
6. for x's parent p the following holds: $k - p.up \geq x.down$.

The algorithm for creating a beacon as a member is as follows:

1. Let C be the set of x's children (i.e., $c_i.parent == x.id$) and c_j the child for which holds that $c_j.down \leq c_k.down$ for any k. Then set $x.down = c_j.down + 1$.
2. If $P(x)$ gives a connected graph go to step 3, else go to step 5.
3. Let p be the parent of x. If $p \in P(x)$ and $k - p.up < x.down$, then set $x.up = p.up + 1$, $x.parent = p.id$ and exit. If $p \notin P(x)$ or $k - p.up \geq x.down$ go to step 4.
4. Choose a parent by picking a node z at random from the set of nodes Z for which holds: $z \in P(x)$, $z.up < k$ and there exists no node y ($y \in P(x)$) for which holds $y.up < z.up$. Then set $x.parent = z.id$, $x.up = z.up + 1$ and exit. If no such parent can be found go to step 5.
5. Set $x.up = 0$, $x.parent = -1$ and $x.num = \infty$.

6.5 The behaviour of a node at initialization

The algorithm for creating the first beacon of a node is as follows:

1. Set *x.down* = 0.
2. Try to set *x.num* one lower than the current minimum value of the chosen number in the neighbour set. If this is not possible because the current minimum already has the lowest possible value assign a random value to *x.num*.
3. If P(*x*) gives a connected graph choose a parent by picking a node *z* at random from the set of nodes *Z* for which holds: *z* ϵ P(*x*), *z.up* < *k* and there exists no node *y* (*y* ϵ P(*x*)) for which holds *y.up* < *z.up*. Then set *x.parent* = *z.id* and *x.up* = *z.up* + 1. If P(*x*) is not connected or if no such parent can be found than set *x.num* = ∞, *x.parent* = -1 and *x.up* = 0.

Chapter 7
Performance evaluation

The goal of this chapter is to show well the protocol performs for differing network conditions such as network mobility and network density. Performance is expressed in (i) the ability of the protocol to maintain a connected DS, (ii) to maintain for each member a valid path to the DS, (iii) the average size of the DS, (iv) the average length of a member path and (v) the average node beacon rate.

Testing has been done by means of simulation, using the simplified simulation model described in Section 2.3: a model that does not incorporate existing network-technologies but in stead gives a conceptual representation of a MANET. The results of these tests should therefore be interpreted as a proof of concept, a means to gauge the feasability of the design. In no way are the results directly comparable to the performance results of existing techniques. Static networks have been generated using the Ad Hoc Network Graph Model, ensuring connected and randomly generated networks; for node mobility the Random Waypoint Model with uniform and stable speeds has been used. Both techniques are also described in Section 2.3.

The simplified simulation model allows for complete control and track-keeping of the state and actions of the individual nodes and the network as a whole. Measurements have been made by checking the state of the C*k*-HDS structure at determined intervals. A network has a connected DS whenever every dominator has a path to every other dominator consisting solely of dominators. A member has a valid path to the DS when every assigned next hop of the path (i.e., a member's parent) is within transmission range. Path validity is averaged over all members. During mobile simulations nodes are often dispersed over several networks of nodes: in that case the connectivity of the DS is averaged over all the networks. To ensure that statistics are not skewed too much, single-node networks are not taken into account, as they already by definition have a connected DS. Figure 7.1 shows a number of networks and their combined connectivity. The average node beacon rate is calculated as the average amount of beacon transmissions per node per BI. If for example 5 nodes beacon 12 times during 2 seconds and BI = 1.0 seconds, the average node beacon rate is $\frac{12}{5 \cdot \frac{2}{1.0}} = 1.2$.

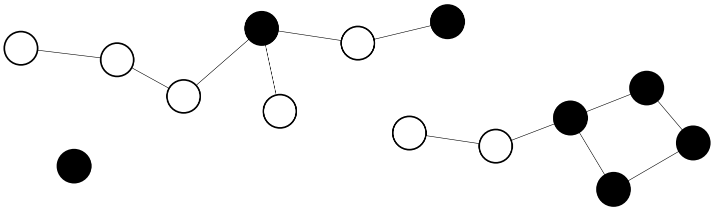


Figure 7.1. Fourteen nodes separated over three networks. The single node network is not taken into account when making measurements. The average connectivity of this set of nodes is 0.5: the 7-node network is disconnected (value 0.0) and the 6 node network is connected (value 1.0). The average DS size is 3 ((2+4)/2), the average path length is (1+1+1+2+3+1+2)/2=5.5.

This chapter starts with the analysis of a number of specific performance properties in Section 7.1: the effect of parameters *BW* and *RW* is given as well as the response time of a single node and of two nodes on a critical event. Section 7.2 shows the average backbone size and path length for the protocol in a static environment with differing network sizes and densities. These have been measured both for the original algorithm as for the protocol. In the original algorithm a different algorithm is used to construct the C*k*-HDS at time t=0, called the Restricted k-Dominant Pruning algorithm. This algorithm has not been described here as it is rather complex and it has not been used in the design of the protocol. It is fully described in [7]. It has been implemented in Java however, and the same static experiments have been performed on it as on the protocol, to compare the initial size of the DS. Finally Section 7.3 gives a full performance analysis on the protocol in a mobile environment with differing average node speeds. The effect of transmission loss is tested by statistically letting 5% of all beacon transmissions fail. Time constraints prevented testing of the protocol in a mobile environment with differing network sizes and densities.

7.1 Performance properties of the implemented algorithm in specific situations

7.1.1 The effect of the beacon window on the inter beacon space of nodes that beacon independently

The goal of this experiment is to show whether the IBS of two independently beaconing nodes is influenced by differing values for *BW*. This has been done by measuring for each beacon the offset of its IBS (with regard to the last beacon of the other node) to the ideal IBS. Before the experiment is described below the details of the ideal IBS and the expected IBS are explained.

Ignoring the randomness introduced by the BW, the expected average time between the consecutive beacons of two independently beaconing nodes (A and B) can be calculated as follows. Suppose that node A beacons every *BI* seconds as in Figure 7.2. Let *X* then be the amount of time between the moment B first beacons and A beaconed last. Because A beacons at deterministic intervals *X* is uniformly distributed over [0,*BI*> and the expected average E[*X*] = *BI*/2. This is equal to the ideal IBS of two nodes (which was previously defined as *BI* divided by size of the set of neighbouring nodes).

The average IBS of two nodes is not a very useful metric however, as it will always average to the ideal of *BI*/2: two nodes that have their beacon moments close together will have two sets of IBSs, one with very short times spaces and one with very long time spaces, that together average each other out. In stead we are interested in the average offset of the IBS to the ideal. This can be calculated in a similar way as was done with E[*X*], by taking the expected average value for *Y* in Figure 7.2. Since the ideal beacon moment is also deterministic *Y* is uniformly distributed over the interval *Z* in Figure 7.2, and the expected average offset of the IBS is given as E[*Y*] = *BI*/4.

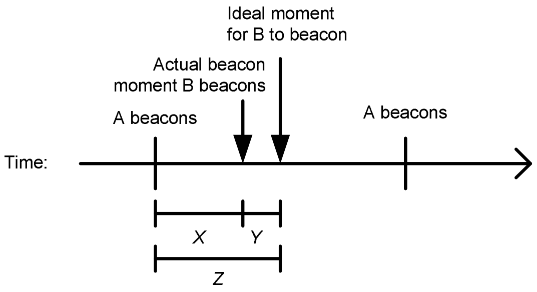


Figure 7.2. The time between the consecutive beacons of two nodes A and B. *X* is the time between the beacon moment of A and B, moment of A and B, *Y* is the time between the ideal beacon moment of B and the actual beacon moment of B. *Z* is the ideal IBS.

For the experiment two nodes are placed outside each others transmission range and are initialized at a randomly chosen moment using a uniform distribution in the time window [0,*BI*]. For 300 seconds the offset of each IBS to the ideal IBS is measured. The experiment was performed for differing values of *BW*. Parameters *BI, BT, FRW and RW* have been kept static at 1.0, 1.05, 0.01 and 1.0 respectively. For each different value the experiment was performed 100 times with differing seeds.

Figure 7.3 shows how the experimental results all stay within 1% of the expected average except for the case where *BW*=0.1, which is slightly lower. A possible cause for this is the fact that one would expect the worst case scenario, in which two nodes repeatedly beacon close together, to become less likely as *BW* increases. Although the difference is only about 5% we conclude that the larger *BW* here has a positive effect on the average offset of the IBS to the ideal of two independently beaconing nodes.

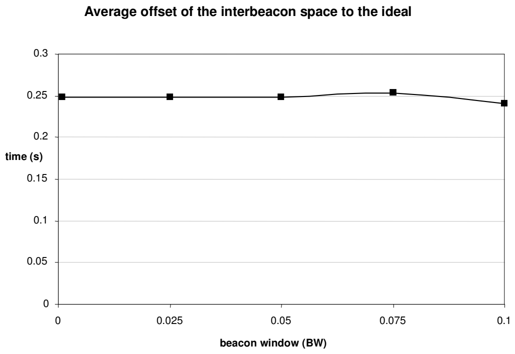


Figure 7.3. The average measured offset of the IBS to the ideal IBS. Ideally the measured offset would be nil. At 95% confidence the intervals stay within a margin of 1%.

7.1.2 The effect of the beacon window and resume window on the inter beacon space of nodes that beacon dependently

In this experiment the IBS offset is again measured (similar to the previous experiment), but now for two nodes that are not independent of each other. The goal of this experiment is to see whether the timing windows RW and BW are indeed able to regain the independece of beaconing between two nodes that have recently reacted on each other. Any averages and value-ranges of the IBS mentioned here are based on the explanation in the previous section.

At the start of the simulation two nodes are placed outside of each other's transmission range. After a period in which both nodes initialize plus a random delay uniformly distributed in the range [0,*BI*], the two nodes are placed within each other's range. When next a node (called node A in the rest of this section) beacons, the other (called B) will respond immediately (since it does not recognize the beaconing node, see Table 5.3). Upon receiving node B's fast response, node A also responds immediately. Taking the offset of the IBS to the ideal IBS of these two fast responses as the first value, the offset of the next 1000 consecutive beacons of A and B are measured.

Parameter *BW* has value 0.050 and 0.100; the RW is either used to its full extent (*RW*=1.0) or not at all, in which case a node schedules its beacon that follows the fast response as it normally does. Parameters *BI, BT and FRW* have been kept static at 1.0, 1.05 and 0.01 respectively. For each combination of the experiment's parameters, the experiment has been performed 100 times with differing seeds.

Figure 7.4.a shows the average offset of the IBS to the ideal when the RW is not used, for the two different values for *BW*, for the first 300 *measured* IBSs. It can be seen how after an initial value of almost 0.500 (the worst case when *BI*=1.0) it converges faster to its expected average of *BI*/4 as *BW* increases. This is to be expected: as *BW* increases more variation is introduced in the moments at which the two nodes beacon, making them more independent of each other.

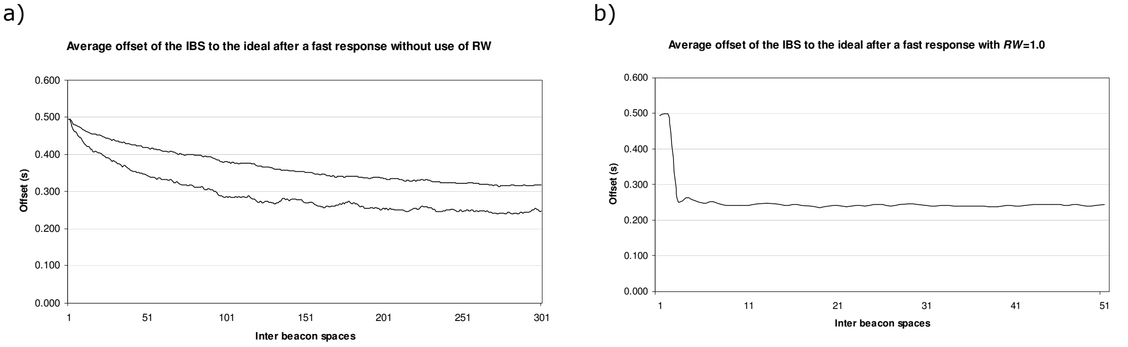


Figure 7.4. The effect of BW and RW on the offset of the IBS to the ideal IBS. In a) the RW is not used: the upper line represents the average offset when BW=0.050; the lower line when BW=0.010. In b) the RW is used and *BW*=0.100.

In Figure 7.4.b the effect of the RW is clearly visible: the IBS following B's fast response now lies in the range [0, *BI*/2], with an approximate average of BI/4. Using the RW thus ensures that any depencies in the beacon moments of nodes because of a fast response are immediately lost again.

7.1.3 The response time of a single node on the loss of a neighbour

The Ck-HDS structure is in most cases invalidated because of the loss of a neighbour, so the time it takes to respond on such a loss is an important measure. The goal of this experiment is to see whether the measured time it takes for a node to respond comes close to the expected time. The expected response is calculated below after the experiment has been described. The experiment is performed for differing values of *BW* to measure its influence.

Two nodes A and B are initialized within each other's transmission range at a random moment uniformly distributed over the range [0,*BI*]. Parameter *k* is set to 0 so both nodes will join the DS by means of a normal beacon, as nodes do not respond fast during initialization. The nodes beacon therefore independently of each other. After a random amount of time node A fails. This will eventually trigger a fast response from B. The time between the moment A fails and B responds is the measured response time.

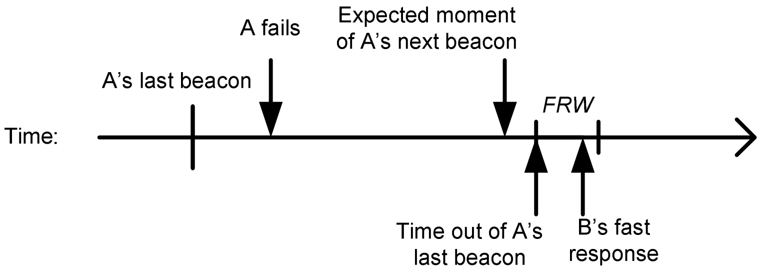

A fails
Expected moment of A's next beacon
A's last beacon
FRW
Time:
Time out of A's last beacon
B's fast response

Figure 7.5. The relevant moments for calculating a node's response.

Again ignoring the randomness introduced by the BW the expected response time can be calculated as follows (see also Figure 7.5). Node A beacons at deterministic moments and the amount of time between the moment A fails and the moment A's next beacon is expected is therefore uniformly distributed with an expected average of *BI*/2. Node B then waits for the period *BT-BI* before scheduling a response in the time window FRW. As *FRW* also has a uniform distribution the expected average delay of the FRW is *FRW*/2. Therefore the total expected average response time is *BI*/2 + (*BT-BI)* + *FRW/2* seconds. For the parameters used in this experiment (*BI*=1.0, *BT*=1.05 and *FRW*=0.01) the expected average then becomes 0.5 + 0.05 + 0.005 = 0.555 seconds.

Parameter *BW* ranges over values {0.001, 0.050, 0.100}. Parameters *BI, BT, FRW and RW* have been kept static at 1.0, 1.05, 0.01 and 1.0 respectively. For each combination of parameters the experiment was performed 50 times with differing seeds.

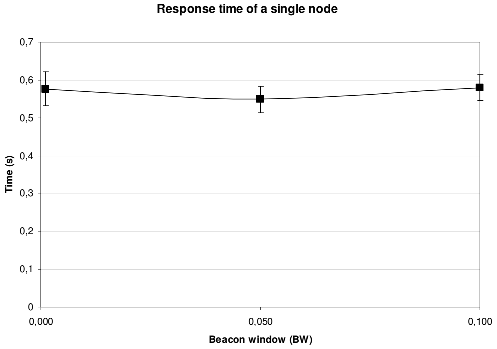


Figure 7.6. The response time of a single node on a critical event. *BI*=1.0, *BT*=1.05 and *FRW*=0.01. Confidence intervals are set at 95%.

As can be seen in Figure 7.6 the expected averages are all within the 95% confidence interval of the obtained results: it is concluded that the measured response time of a single node is expected. Although the average response time for

BW=0.05 is slightly lower than is the case for *BW*={0.001, 0.100}, the difference is not significant as all values lie within each other's confidence interval.

7.1.4 The response time of two nodes on the loss of each other

One would expect that two nodes would be quicker to repsond on the loss of a link than the one node in the previous experiment. The goal of this experiment is too see whether this is the case, and how much quicker two nodes are. Calculating the expected average time for two nodes has here been left out due to time constraints. Again the experiment is performed for differing values of *BW* to measure its influence.

Two nodes A and B are initialized within each other's transmission range at a random moment uniformly distributed over the range [0,*BI*]. Parameter *k* is set to 0 so both nodes will join the DS by means of a normal beacon, as nodes do not respond fast during initialization. The nodes beacon therefore independently of each other. After a random amount of time node the nodes are placed outside of each other's transmission range. This will eventually trigger a fast response from both nodes. The time between the moment the nodes are moved and the moment the first node responds is the measured response time.

Parameter *BW* ranges over values {0.001, 0.050, 0.100}. Parameters *BI, BT, FRW and RW* have been kept static at 1.0, 1.05, 0.01 and 1.0 respectively. For each combination of parameters the experiment was performed 50 times with differing seeds.

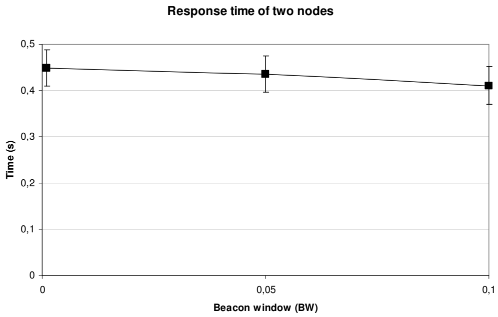


Figure 7.7. The response time of two nodes on a critical event. *BI*=1.0, *BT*=1.05 and *FRW*=0.01. Confidence intervals are at 95%.

Comparing the results in Figure 7.7 with those of Figure 7.6 it is clear that the response time of two nodes is indeed shorter than that of a single node, although only with ± 0.1 seconds for *BW*={0.001, 0.050} and ± 0.015 seconds for *BW*=0.100, which is about 18% and 27% of the total *BI*. The declining graph hints at a positive effect of an increasing *BW* as the number of nodes increases, but the difference is not significant as the average for *BW*=0.100 is still within the 95% confidence interval of the value for *BW*=0.001.

7.2 An overall performance analysis of the implemented algorithm

7.2.1 Static experiments

For this set of experiments graphs were created using four sets of Ad Hoc Network Graph Model parameters ((0.5,0.5,0.1), (0.5,0.5,0.5), (1.0,1.0,0.1) and (1.0,1.0,0.5) for parameter set (*a*,*b*,*g*)) for networks consisting of 50, 75, 100, 125 and 150 nodes. These are the same sets of parameters that Yang et al. used in their paper to create the networks for their experiments. Average densities of the resulting networks are given in the caption of Figure 7.8. For each combination of Ad Hoc Network Graph Model parameters and network sizes 100 different networks have been generated using as many different seeds.

For each generated network a stable C*k*-HDS structure has been constructed, with *k*={0,1,2,3}, using both the Restricted *k*-Dominant Pruning algorithm and the protocol. The Restricted k-Dominant Pruning algorithm has been implemented in Java. As it is a theorethical algorithm it was simply given each network as input. For the resulting C*k*-HDS the backbone size has been measured as explained in the beginning of the chapter. Results have been averaged over the 100 runs for each different set of paramters.

The experiments with the protocol have been performed using the simplified simulation model described in Section 2.3. For each generated network every node is initialized at a random moment uniformly distributed over the range [0,*BI*]. The simulation is then run for 10 seconds in which time it stabilizes. After the 10 seconds the backbone size and average path length of the resulting C*k*-HDS structure is measured. Results have been averaged over the 100 runs for each different set of paramters.

Figure 7.8 shows the resulting average backbone sizes, Figure 7.9 the resulting average member path lengths. The graphs of the protocol are denoted as 'k=*k*'. The graphs for the pruning algorithm are denoted as 'D*k*'.

Defining a smaller DS as better, the protocol outperforms the pruning algorithm as networks become more dense. In 7.8.a (where a node has on average more than 20 neighbours) the pruning algorithm is clearly better, while in 7.8.d (where a node on average has between 5 and 6 neighbours) the protocol performs best. In 7.8.b and 7.8.c, which have similar densities but are formed in a different way, the protocol performs either equally good (in 7.8.b) or better. The details of this difference in performance are not addressed as the protocol and the pruning algorithm are two techniques that operate at wildly differing levels (the former being a distributed protocol designed to operate in a MANET, the latter being a non-distributed theorethical algorithm).

The DS of the protocol grows linearly as the networks increase in size, which is to be expected. The average length of a member path, however, stays almost the same (if anything, decreases) for increasing network sizes, and depends almost fully on the value of k. This shows that the efficiency (the size of the DS with respect to the total network size) of the protocol does not depend on the size of the network

The effect of *k* diminishes for both the protocol as the pruning algorithm as *k* grows larger: in all cases the difference between 'k=1' and 'k=2' is larger than between 'k=2' and 'k=3'. With the protocol this effect is mainly caused by the algorithm's tendency to form loops in the DS which prevents dominators from leaving the DS.

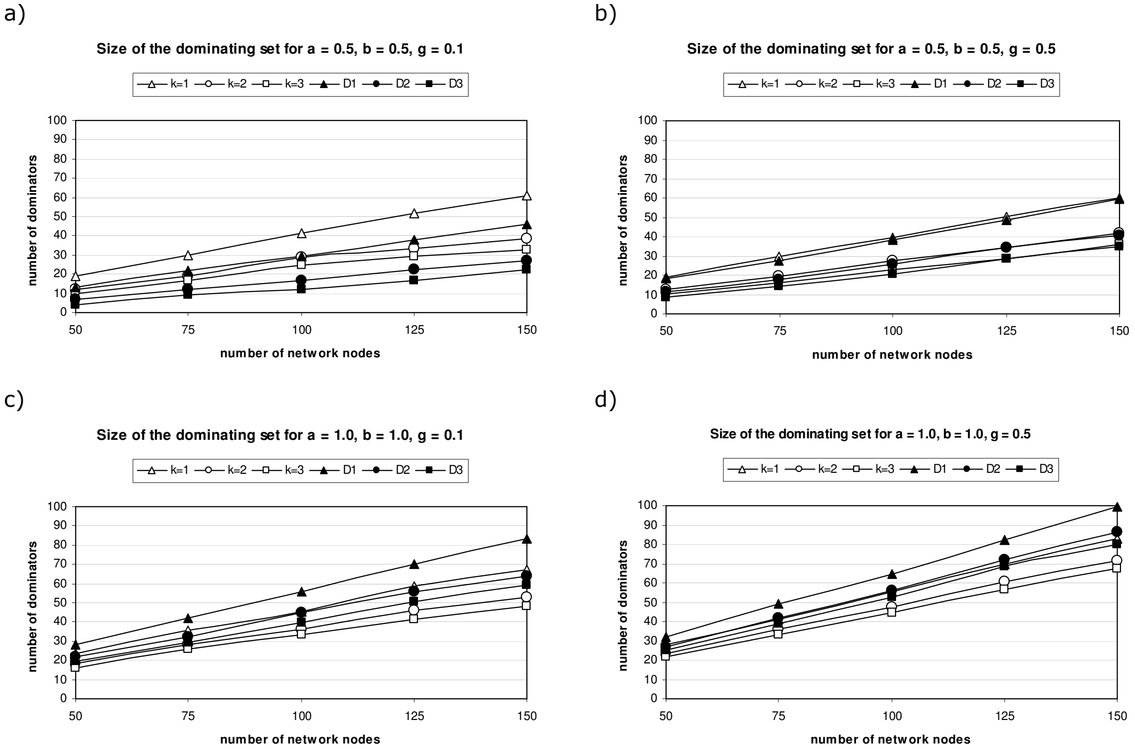


Figure 7.8. The resulting backbone sizes for the protocol and the Restricted k-Dominant Pruning algorithm for differing networks (the Ad Hoc Network Graph Model parameters are given) and differing values of *k*. The protocol is indicated with graphs 'k=1', 'k=2' and 'k=3'. The pruning algorithm is indicated with graphs 'D1', 'D2' and 'D3' for *k*={1,2,3}. At 95% confidence the intervals stay within a margin of 1%. In a) the average densities of the networks of 50, 75, 100, 125 and 150 nodes are 20.6, 23.4, 25.3, 26.6 and 27.8 neighbours per node respectively. In b) the average densities of the networks of 50, 75, 100, 125 and 150 nodes are 11.2, 11.9, 12.4, 12.7 and 12.9 neighbours per node respectively. In c) the average densities of the networks of 50, 75, 100, 125 and 150 nodes are 8.3, 10.0, 11.3, 12.3 and 13.2 neighbours per node respectively. In d) the average densities of the networks of 50, 75, 100, 125 and 150 nodes are 5.4, 5.9, 6.1, 6.3 and 6.5 neighbours per node respectively.

As is to be expected the DSs of the more denser networks are smaller than those of the more sparser networks: fewer dominators are needed in a denser network to cover all nodes. Although the results for the protocol are similar for networks generated with graph model-parameters (0.5,0.5,0.5) and (1.0,1.0,0.1) for 'k=1', the resulting DS of the former set of networks is smaller for 'k=2' and 'k=3' than the latter. A possible cause for this could be that the tendency to form loops in the DS is less in the networks generated with parameters (0.5,0.5,0.5), due to their expected more elongated form.

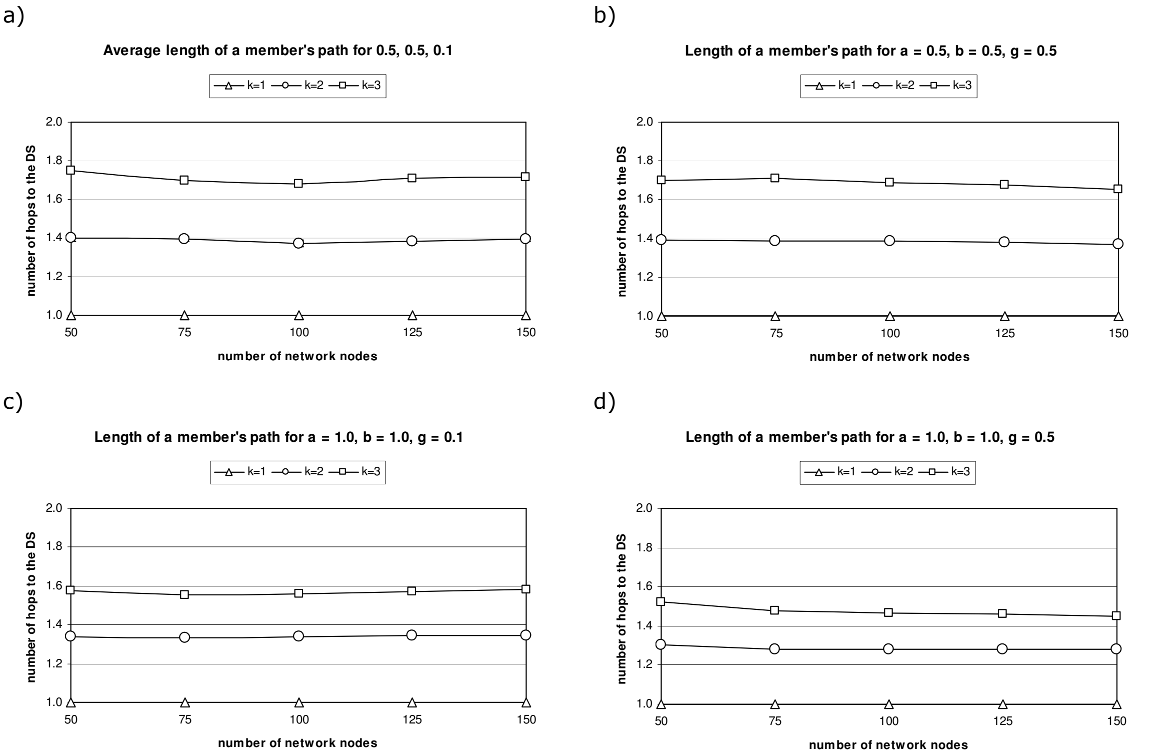


Figure 7.9. The resulting average path lengths for the protocol and the Restricted k-Dominant Pruning algorithm for differing networks (the Ad Hoc Network Graph Model parameters are given) and differing values of *k*. The protocol is indicated with graphs 'k=1', 'k=2' and 'k=3'. At 95% confidence the intervals stay within a margin of 1%.

7.2.2 Mobility experiments

For each simulation 50 nodes have been spread over a region of 1500x1500 metres using the same uniform distribution. Nodes have a transmission range of 250 nodes. Mobility is managed by the Random Waypoint Model described in Section 2.3. The average node speed has been varied between 5 km/h (1.39 m/s), 10 km/h (2.78 m/s), 20 km/h (5.56 m/s), 30 km/h (8.33 m/s) and 50 km/h (13.89 m/s). Node speeds vary per average speed between 0 km/h and twice the average speed. In the case of an average node speed of 5 km/h the network is first initalized for 580 seconds (to accommodate for the effect that nodes tend to move to the middle of the region) before the protocol starts; with the other runs the initialization period is 280 seconds. Measurements start at t=600 seconds for an average node speed of 5 km/h and at t=300 otherwise; measurements are then made for 10 minutes after which the simulation is ended.

When the protocol starts each node is initialized at a random moment uniformly distributed over 1 *BI*. After 20 seconds of operation the Ck-HDS structure is assumed to have reached a stable state and measurements are started. Measurements are performed every 0.2 seconds when the average node speed is 5 km/h and every 0.4 seconds otherwise. Measurements are made of the size of the backbone, the connectivity of the backbone, the validity of the member paths, the average beacon rate and the average length of the member paths.

To simulate the effect of packet loss on the protocol when *k*=1, one set of experiments has been performed with the transmission probability (*T_p)* set to 0.95. This has the effect that 5% of all transmissions will fail. In all other cases T_p is 1.

Parameters *BI, BT, FRW, BW and RW* have been kept static at 1.0, 1.05, 0.01, 0.1 and 1.0 respectively. For each combination of parameters the experiment was performed 50 times with differing seeds.

All obtained results are shown in Figure 7.10. The case for T_p=0.95 (and *k*=1*)* has been labeled 'k=1^*'.

Similar to the static simulations, *k* seems to be the main factor determining the size of the DS and the length of the member paths, together with T_p. Member paths only become slightly shorter as mobility goes up.

Connectivity of both the DS and the member paths drops in a linear fashion as average node speeds increase. Difference in connectivity of the DS between the cases 'k=1' and 'k=2' is almost non-existent, but significant between 'k={1,2}' and 'k=3'. Although it is unsurprising that a smaller DS is easier to maintain (as there are fewer points of failure), one would expect, based on the differences in the size of the DS for different values of *k*, a bigger difference between 'k=1' and 'k={2,3}'. The difference between connectivity of the member paths for different values of *k* is almost neglible, as is the difference in the average length of a member path.

As T_p goes below 1, a number of effects can be observed:

1. as more events are generated because nodes do not act as they should (e.g., because of beacon timeouts that are caused by transmission failures), the average beacon rate increases;
2. nodes will unnecessarily join the DS because they did not receive the beacon of a neighbouring node that would have created a connected parent set;
3. nodes will leave the DS because they did not receive the beacon of a neighbouring node and therefore mistakenly believe they have a connected parent set and (possibly) disconnect the DS.

The second effect causes the DS to increase in size. The third effect will initially make the DS smaller, but it will likely lead to some other node(s) joining the DS because it has no connected parent set. Together this explains why the DS of 'k=1^*' is larger than that of 'k=1'.

Surprisingly the connectivity of the DS for 'k=1^*' gains on 'k={1,2,3}' as mobility goes up, until it even outperforms 'k={1,2}' at 40 km/h and 'k=3' at 50 km/h: the increased beacon rate of 'k=1*' gives the protocol a lower response time, enabling it to better deal with the increased reconfiguration speed of the network. Connectivity of the member paths is also better for 'k=1*'. As more nodes join the DS it only becomes easier to create a path towards it.

Finally, the beacon rate is independent of k and only influenced by the average node speed and T_p. Since the average size of a beacon depends mostly on a network's average degree (the only other influence being the average amount of broken links that a node has to include in its beacon, which is ignored here), this means that the algorithm's load on the network is independent of k.

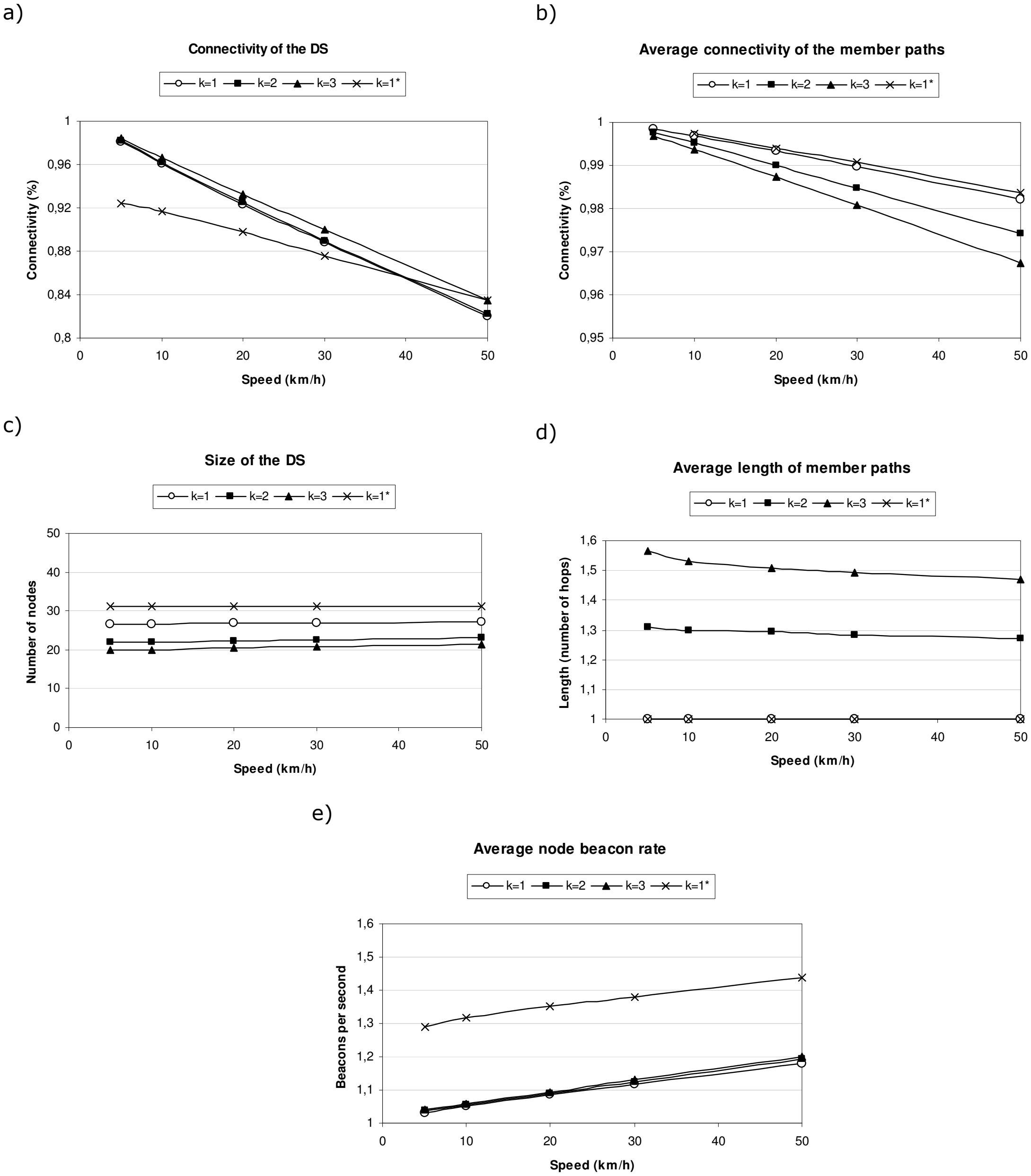


Figure 7.10. The results of the mobility experiments. At 95% confidence the intervals stay within a margin of 1%. For T_p=1.0 the labels are 'k=*k*', with T_p=0.95 and k=1 the label is 'k=1*'.

Chapter 8
Conclusions and future work

This part of the thesis presents a complete design and specification of a fully distributed protocol capable of creating and maintaining a Ck-HDS algorithm for MANETs, based on the theorethical algorithm by Yang et al. and using the assumption of symmetric radio links. The performance of the protocol has been tested by means of a simplified simulation model. As there was no existing work to be found on implementations of Ck-HDS algorithms in MANETs, it is hard to tell how good (or bad) the implemented algorithm performs. The obtained results serve as a proof of concept and as a reference point, which can be either used as a goal for, e.g., a prototype, or as a starting point for future research, here of course in particular the design of a CBR system in part III of the thesis.

The experimental results show how the validity of the Ck-HDS depends mostly on the mobility of the network. As mobility increases the validity decreases linearly: for the maximum measured mobility (an average node speed of 50 km/h with a transmission range of 250 meters) the connectivity of the DS is around ~83% and the connectivity of the member paths is around 97%. These results are however based on perfect communication. To give a more realistic view transmission errors have been introduced (5% of all transmissions fail): these results show that as communication becomes more unreliable the rate with which nodes beacon increases and connectivity of the DS decreases when mobility is low. As mobility increases however the increased beacon rate enables the protocol to respond more quickly on reconfiguration, and it even performs better than when perfect communication is used at 40 km/h and above. The connectivity of member paths (for k=1) is always better when transmission errors are introduced, due to the increase of the DS.

It has been shown for static networks that the relative size of the DS with regard to the size of the network stays the same. Although time constraints prevented doing mobility tests for networks of a size other than 50 nodes, the algorithm should be scalable: the size of a beacon is almost fully dependent on a network's average degree and it is not expected that the beacon rate will be influenced by the size of the network, as a node will only react on events that occur within its two-hop neighbourhood.

Some issues are still left open for future work such as the effect that the strategy for choosing a parent has on performance. The two most important topics for future work are however (i) the use of asymmetric radio links and (ii) the prevention of small loops in the DS. Both topics have been neglected in this thesis because the primary goal was to implement the existing algorithm; not to improve on it. Incorporation of asymmetric links would however greatly improve the implementation, as the assumption of symmetric links is highly unrealistic in a MANET. Preventing small loops in the DS only requires that a node indicates for each neighbour whether it is a dominator or not: in this way nodes can make a more informed decision on the connectivity of their parent set.

PART III

The design & analysis of a content-based routing system for a mobile ad-hoc network, using the connected k-hop dominating set algorithm as a backbone

This part presents a CBR system that makes use of a backbone of flexible size to route its content over, as described in Section 1.2. The CBR system aims to find a balance between the size of the backbone and the amount of content that must be routed. By performing experiments with backbones of different sizes and with different content loads the performance of the system is tested. To create a backbone of flexible size in a MANET the C*k*-HDS protocol discussed in the previous part is used.

This part starts by giving a short introduction to content-based rounting (CBR) in Chapter 9, after which an overview of the design is given in Chapter 10. The design is fully discussed in Chapter 11 and specified in Chapter 12. Chapter 13 describes a number of simulation experiments that have been performed to test the performance of the design. Conclusions on the results of those experiments are given in Chapter 14.

Chapter 9
An introduction to content-based routing in mobile ad-hoc networks

9.1 An introduction to the publish/subscribe paradigm

Publish/subscribe (henceforth referred to as P/S) is an asynchronous interaction paradigm oft-used in distributed systems. Most content-based routing (CBR) systems today are based on P/S; in this Section a short introduction is given to the model and its entities.

P/S is made up of three (types of) entities: publishers, subscribers and an event dispatcher. Publishers act as a source of information for a specified set of subscribers. In general one publisher has many subscribers; a subscriber may be subscribed to multiple publishers. Publishers publish information in the form of messages. Subscribers have the ability to subscribe to a specific publisher by issuing a subscription: during the lifetime of the subscription the subscriber will receive all messages published by the publisher. A subscriber may explicitly end a subscription by issuing an unsubscription. The delivery of subscriptions to publishers and messages to interested subscribers is done by the event dispatcher, which acts as a medium for publishers and subscribers alike. Publish/subscribe is described in some detail in [13]. Figure 9.1 below shows the full model.

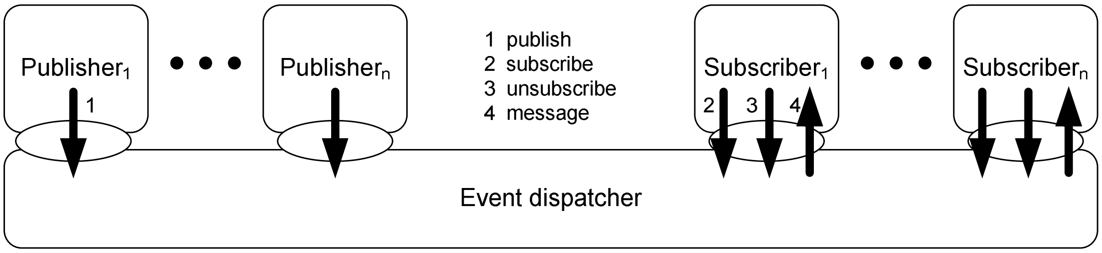


Figure 9.1. A high level view of the publish/subscribe paradigm showing the enitites involved and their actions.

The reason why P/S is so popular in distributed environments is the decoupling of the involved entities along three dimensions:

- Space; there is no need for direct contact between entities (or stated informally: the entities do not need to know each other).
- Synchronization; operations (the publishing of events or the issuing of a subscription) may be performed asynchronously, preventing processes from blocking or creating tight dependencies with other processes.
- Time; entities do not need to participate in an action at the same time.

The operations described in the second point are performed between entities that are able to communicate with each other at a certain point in time. Decoupling in time describes operations between entities that may never directly communicate with each other. This requires buffering of information, a requirement that is not easily met in a MANET and perhaps impossible to guarantee. We will therefore largely ignore this dimension in the context of P/S systems in MANETs.

The P/S paradigm provides an excellent means to implement the CBR model over and is the dominating alternative in current research. Solutions presented in [14], [15] and [16], amongst others, are all academic examples of P/S-based CBR. In this paper, unless specifically stated otherwise, CBR is also tackled as a special form of P/S.

9.2 The content-based dissemination model

The CBR model ([17], [18]) is based on the P/S model, as was stated in the previous section, and makes use of the same entities: publishers, subscribers and the event dispatcher. The main difference with P/S is the following: whereas subscribers in the P/S model subscribe themselves to a publisher (demanding to receive all the messages it publishes), in the case of CBR a subscriber subscribes itself to message content. Thus, from the moment a subscriber has issued a subscription each and every message that matches that subscription, irrespective of the message's source, must be delivered to that subscriber. Publishers may advertise the content that they have to offer. If this is the case subscribers may choose whether or not they wish to subscribe themselves at all, based on the content being advertised. Subscribers may however also blindly subscribe themselves to any content without knowledge of what content publishers actually have to offer.

Within the CBR model an entity may perform any and all of the three roles of publisher, subcriber or event dispatcher at any time. Roles may thus overlap.

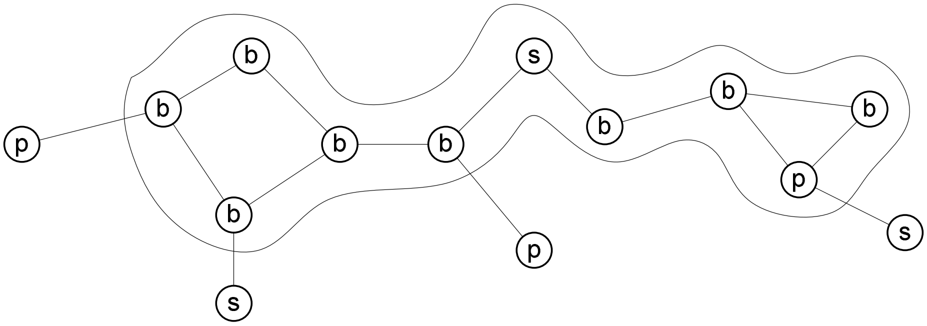


Figure 9.2. An example CBR system implemented over a MANET, modelled as in Figure 1.1. Nodes labeled p act as publishers, those labeled s act as subscribers and those labeled b act as brokers, The group of nodes that fall within the surrounding line all (amongst others) act as brokers and together form the event dispatcher. Some nodes within the event dispatcher perform multiple roles.

Figure 9.2 shows a model of an example CBR system over a MANET. The publishers within the MANET are labelled with a *p* and the subscribers with an *s*. Publishers and subscribers are connected by means of a dispatcher network. The nodes that make up the dispatcher network are usually called brokers and are here labeled with a *b*.

There are three fields that together cover the main problems associated with CBR (see for instance [13]): (i) the subscription language, (ii) the architecture of the event dispatcher and (iii) the routing of content. Each field is described in its own subsection below.

The performance of a CBR system is expressed by means of a number of properties, the most important of which are:

- Completeness: the ratio of the subscribers that receive the content they subscribed to versus the total number of subscribers that should have received the content.

- Precision: the ratio of the total number of received messages that are wanted. A message is wanted if the receiver is subscribed to it and the node has not yet received it before.
- Delay: the time between a message is published and that it is received by a subscriber.
- Overhead: the total number of system messages (i.e., advertisements, subscriptions, messages, etc.), including the routing of the published content.

Security is usually ignored by assuming that all routing will only be performed over a trusted network.

9.2.1 The subscription language

The subscription language refers to the syntactic model used to define content in messages, advertisements and subscriptions. It is usually considered an application-specific problem and often not addressed in academic CBR system designs, which focus more on the other two problem fiels. A syntactic model must be capable of defining the content being routed, but should do so in an efficient manner.

A message contains content. An advertisement defines the range of messages (or the range of content) a publisher may publish. A subscription defines content. A message is said to match a subscription (or vice versa) if the content defined in the description is contained in the message. A subscription is said to match an advertisement if the content defined by the subscription falls within the range defined by the advertisement.

The subscription language may take any form, which is exactly the reason why it is often not addressed in academic solutions. It may range from simple models (content is defined as a string of characters that can each be subscribed to) to the more complex solutions (content is in the shape of XML documents that can be subscribed to by means of XPath queries). The subscription language has also been left open in the design presented in this thesis, although a simple syntactic model was used to accommodate for simulations.

9.2.2 The architecture of the event dispatcher

The event dispatcher has the responsibility to deliver published messages to the subscribers that have subscribed to those messages. In doing so it may take any form suitable for the environment at hand. The event dispatcher is made up out of nodes in the network. Choosing which nodes should be part of the event dispatcher is an important step, as it must deliver all published content but do so with as little overhead as possible. If the event dispatcher is too big it may give too much overhead; if it is too small it may not be able to deliver the content to all subscribers.

Event dispatchers usually take the following form:

- the whole network is part of the event dispatcher;
- only CBR actors (publishers and subscribers) are part of the event dispatcher;
- a routing structure connecting all CBR actors;
- a permanent routing structure.

Whichever structure performs best first and foremost depends on the message routing employed, second on other more external influences such as the network conditions and the amount of content being published.

9.2.3 Message routing

Message routing encompasses both the routing and the forwarding of messages throughout the dispatcher network. It is the core mechanism behind any CBR system as it is responsible for the actual delivery of published messages to their respective subscribers. A number of general message routing techniques exist; their performance usually depends most heavily on the structure of the event dispatcher, the amount of content load and network conditions. The most important performance properties are the completeness of message delivery to subscribers, the precision of message delivery to the network as a whole (ideally nodes should only receive the messages they subscribed to and only once) and the load placed on the network. The main problems associated with message routing in a CBR system in a MANET are those of any routing model in a MANET: resource contention, scalability and the inconsistent topology.

The simplest message routing model is to flood any message that reaches the dispatcher. Flooding is simple and can be quite effective. Unless a large part of the network is interested in the content it is however quite inefficient. If the content load is high the whole dispatcher may also become congested as every message must be routed everywhere. Figure 9.3.a shows the resulting flow of messages for a given network when messages are floded.

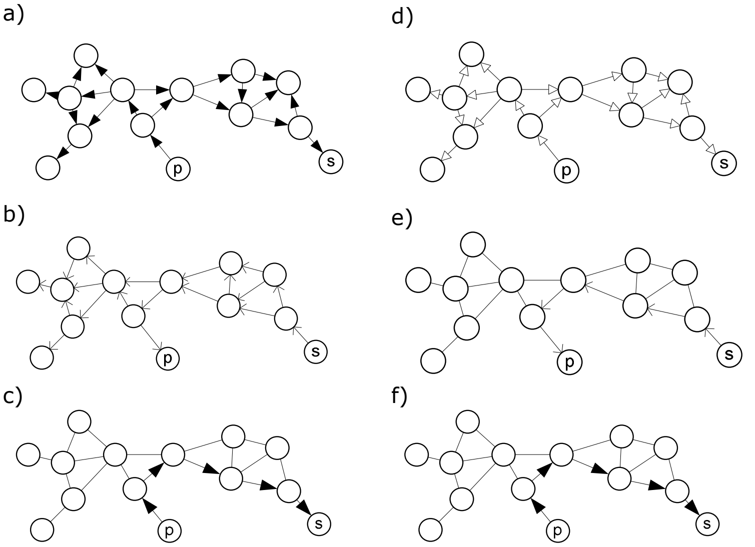


Figure 9.3. Different message routing solutions and their resulting flows of advertisements, subscriptions and messages for the same network with a publisher and a subscriber. Fat arrows are messages, thin arrows are subscriptions and open arrows are advertisements. Fig. a) shows the resulting flow when messages are flooded, fig. b) and c) show the resulting flows when subscriptions are used to create a message routing layer, fig. d), e) and f) shows the resulting flows when both advertisements and subscriptions are used to create routing layers.

By using advertisements or subscriptions to create routing layers the inefficiencies of message flooding can be avoided. Figure 9.3 shows an example of such a solution in which the subscriptions of every subscriber are flooded throughout the dispatcher (here made up by the entire network). By storing state information about the subscriptions at every node in the dispatcher, published messages may be reversely routed to the subscribers. Part of the dispatcher network where no subscribers are situated are ignored. In a similar way advertisements may build a routing layer for the subscriptions to be routed over, so that they only reach those parts of the network where publishers are present. An example of this is also shown in Figure 9.3.

Routing layers such as described here provide for very efficient routing, but maintaining accurate routes is extremely hard and is likely to give a relatively high amount of overhead. As the size of networks and the number of actors increase, scalability is often also a problem with these designs.

Gossiping is yet another routing technique and makes use of the mobility of a MANET: nodes in the dispatcher network will store any message they either receive or have published themselves for a defined period of time and may decide to forward it to any node they endounter during this period. This decision can be made selectively (messages are for instance only forwarded to nodes that explicitly want it) or at random (messages are forwarded to any other node with a certain probability). Gossiping gives very little reliability with regard to completenessof of message delivery but is extremely scalable.

Chapter 10
Balancing loads: an introduction to the content-based routing system

Before delving into all the details of the CBR system in Chapter 11 an overview of the system is given together with the thoughts that go behind it.

10.1 A high level view of the design

The design makes use of a permanent routing structure of flexible size that acts as the dispatcher network to route content over. The dispatcher network is henceforth refered to as the backbone. Each node that is not part of the backbone knows a next hop towards it. All communication goes over the backbone. Publishers that have content to offer send their advertisements to the backbone where they are flooded and routed to any subscribers outside the backbone (how the backbone nodes know whether there are subscribers outside the backbone is explained later). The advertisements set up a routing layer over which the subscriptions are routed. Subscribers only subscribe to content being advertised: their subscriptions are first sent to the backbone where they are flooded, and then sent on to any publishers (that issue have matching advertisements) outside the backbone via the routing layer set up by the advertisements. The subscriptions set up a routing layer for the messages to be routed over. How this layer looks and how the messages are routed differs:

- with a so-called dumb backbone little routing information is stored in the backbone nodes, and messages that reach it are flooded over the entire backbone, after which they are routed to any subscribers that are outside the backbone;
- with a so-called smart backbone more routing information is stored in the backbone nodes and messages are only routed to those parts of the backbone where the messages are wanted, before being routed to any subscribers outside the backbone.

Figure 10.2 shows how advertisements, subscriptions and messages are routed when the system uses either a dumb backbone or a smart backbone. It can be seen that the routing of messages over a smart backbone is more efficient than over a dumb backbone.

The backbone is created by means of the connected *k*-hop dominating set (C*k*-HSD) protocol presented in part II, which is capable of maintaining a backbone and give each node that is not part of the backbone (called a member) a path towards it (called a member path). These paths can have a maximum length (in hops) *k*, which implicitly determines the size of the backbone. By increasing *k* the average length of a member path is thus also increased and the backbone will decrease in size. The network in Figure 10.1.a is an example of a backbone with maximum path length 2, refered to as a C2-HDS. Nodes outside the backbone (called members) only know the next hop of their path to the backbone. The CBR system only routes over the structure imposed on the network by the Ck-HDS protocol (i.e., backbone and resulting paths). Section 10.2 shows how the CBR system interacts with the C*k*-HDS protocol as a lower routing layer.

10.2 Design goals

Performance in this part of the thesis means the completeness, precision and overhead of the system. Overhead can be split into the overhead of advertisements, subscriptions and messages. The goal of this part of the thesis is to see how performance of the CBR system is influenced by differing backbone sizes (i.e., differing alues for *k*) and behaviour (smart or dumb), for differing external conditions such as network mobility and content load. Or stated differently: for a given set of advertisement, subscription and message loads, which combination of backbone size and behaviour performs best?

A number of expectations can already be formulated based on the high level description in the previous section:

- completeness will suffer as mobility goes up, especially when the backbone is smart;
- as content load increases the overhead will be lower when the backbone is smart;
- precision is higher with a smart backbone;

With these in mind the following design questions are formulated:

Is the system, using either a smart or a dumb backbone, still able to deliver messages as network mobility and the number of actors increase? (10.1)

How is overhead influenced by the size and behaviour of the backbone, for given network conditions and the number of actors? (10.2)

How is precision influenced by the size and behaviour of the backbone, for given network conditions and the number of actors? (10.3)

Experiments described in Chapter 13 aim to find the answers to the formulated questions. Final conclusions based on the experimental results are given in Chapter 14.

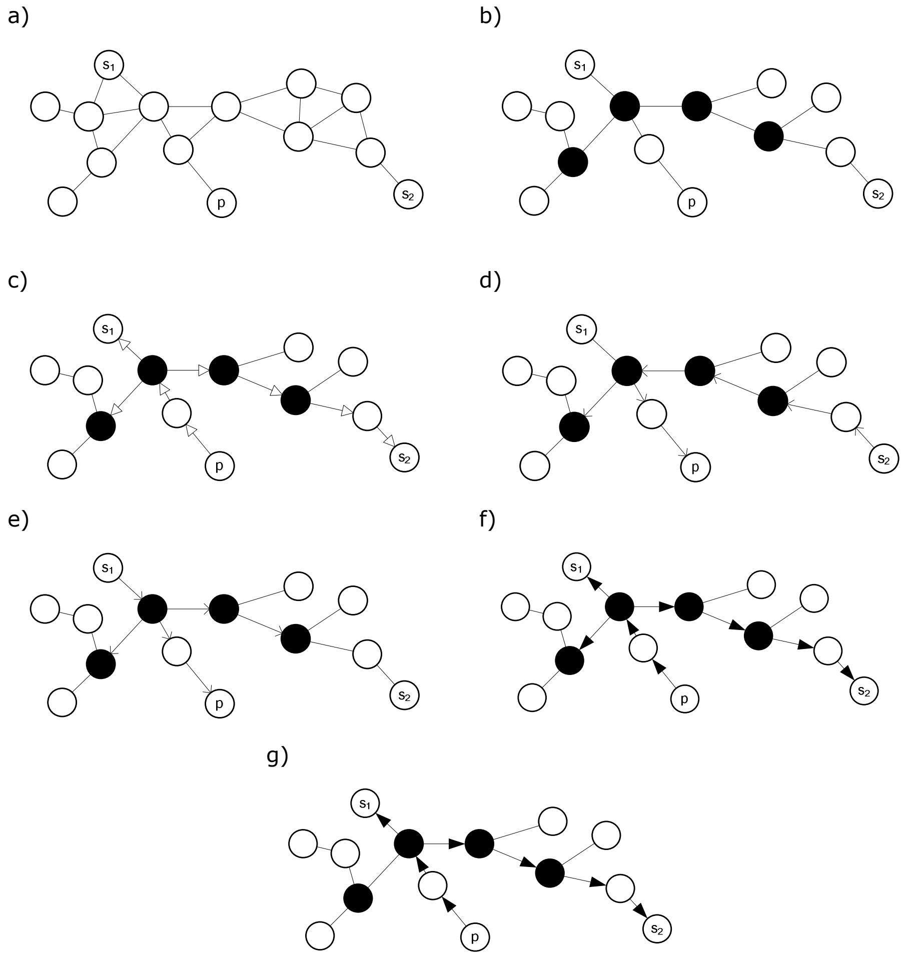


Figure 10.1. Fig. a) shows the reference network with three actors: one publisher, two subscribers. Fig. b) shows in black the nodes that form the backbone, created by the Ck-HDS algorithm in part II of the thesis. Fig. c) shows how all advertisements (open arrows) are routed from the publisher to the backbone, over the entire backbone and to the subscribers outside the backbone. Fig. d) and e) show the subscriptions (thin arrows) issued by respectively subscribers s_1 and s_2, and how they are routed to and over the backbone and to the publisher outside the backbone. Although subscriptions are routed in a similar way for the case of a dumb and a smart backbone, the state information stored at the backbone nodes differs. Fig. f) shows how a message that matches the subscriptions of both s_1 and s_2 is routed to the dumb backbone, flooded over it and routed to the subscribers outside the backbone. Fig. g) shows how a message that matches the subscriptions of s_1 and s_2 is routed to the smart backbone, routed to those parts of the backbone where it is needed and then routed to the subscribers outside the backbone.

Chapter 11
The design of the content-based routing system

This chapter presents the design of the CBR system in detail. While reading, the reader should keep in mind that the system intentionally has been designed to test whether or not it is possible to maintain routing layers over, to and from a backbone to route subscriptions and messages over. This involves a number of design choices that perhaps make the system less effective and more vulnerable than otherwise would have been the case: these are pointed out throughout the text. These design choices have been made because due to time constraints it was not possible to design a system that fully covers every weakness of the current design. In stead of designing some hybrid system that covers only part of the weaknesses of the current design the design has therefore been kept as pure as possible, the better to test the concept of creating routing layers over a backbone.

11.1 The C*k*-HDS algorithm as a lower layer

The C*k*-HDS layer presents at all times a simplified topological view of the network on which upper layers must do their routing. The view given consists solely of (the identifiers of) all the the relevant neighbours of a node: its parent, its siblings and its children. See Figure 11.1 for an example. The list of identifiers is updated every time the C*k*-HDS algorithm beacons, or when it's view is updated due to the receipt of a beacon from a neighbour, or when a neighbour's beacon times out (see Section 6.1.1 for how the C*k*-HDS algorithm updates its view).

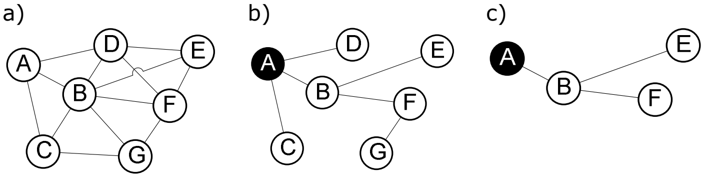


Figure 11.1. A MANET, the Ck-HDS structure imposed on it (*k* is 3 or larger) and the topological view that B has of the network. Node B ignores nodes C, D and G and will only route to and fro nodes A, E and F.

The presented view is not guaranteed to be correct: presented neighbours may in fact no longer be within transmission range of the node. If the upper layer notices the loss of a neighbour (for instance because it was unable to reach it via unicast transmission) it can increase the connectedness of the C*k*-HDS structure by reporting the loss of the neighbour to the C*k*-HDS layer. Due to time constraints such a feature has however not been added to the design.

11.2 The subscription syntax

The system presented here focuses on forwarding, not on the subscription syntax. The employed syntax is therefore the simplest possible. Publishers publish messages consisting of strings of characters (of arbitrary length), which they advertise by including in their advertisement the range from which the characters are taken.

Subscribers subscribe themselves to single characters and wish to receive any message containing that character. A subscriber will only subscribe itself to characters that fall within the range of at least one advertisement.

As an example, consider a network consisting of nodes A, B, and C. Node A has messages to publish in which characters will fall in the range [b,g], and advertises this by issuing an advertisement specifying this range. Node B and C thereupon issue subscriptions subscribing themselves to any message containing the characters [e] and [g], respectively. When node A publishes the message "eeb" it will only be delivered to node B. The next message "bbdgfffe" will however be delivered to both nodes.

11.3 Creating routing layers with flows

11.3.1 Flows

The routing layers used for routing subscriptions and messages are created by the dissemination of advertisements and subscriptions respectively. As advertisements and subscriptions are disseminated from hop to hop through a network they form flows. By storing state information about the advertisement or subscription at each hop of the flow, the flows form routing layers. How these flows are created and how they are used as routing layers is discussed in detail in Section 11.5. This section focuses on how flows are used to create routing layers.

Flows may be directed or undirected (or not directed). If a flow hop doesn't know its previous hop it is said to be undirected. In a directed flow each hop of the flow knows the previous hop in the flow by storing some extra state information at the hop. Directed flows therefore give more overhead. They also allow for reverse routing along the *shortest* path of the flow up to its source; to reach the source of an undirected flow *every* path must be followed.

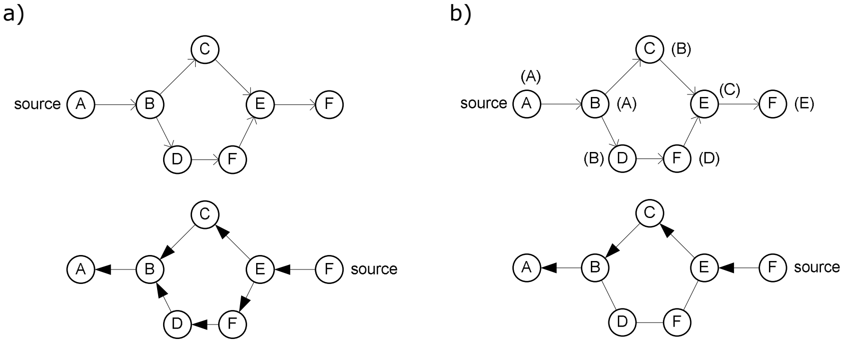


Figure 11.2. Routing over flows. Fig. a) shows how the first undirected flow sets up a routing layer over which the second flow is flooded. Fig. b) shows how the first directed flow sets up a routing layer and that each hop in the flow stores the identifier of the previous hop. The second flow is then routed reversely along the shortest path.

If advertisements (or subscriptions, the following holds for both) are forwarded from a node A to a node B and from B to a node C, the advertisements and the advertisement flow are both said to be forwarded to (or to go to) B and C from the perspective of A. From the perspective of node B the flow comes from A and goes to B; from the perspective of node C the flow comes from A and B. Node A is not necessarily aware of the fact that the flow it forwards to B also reaches C however, nor is C necessarily aware of the fact that the flow also comes from A. Nodes only know with certainty their previous and their next hop.

Advertisement flows are created by publishers and can be identified by the content they advertise. Two publishers advertising the same content thus have identical advertisement flows. Subscription flows are created by subscribers. There exist two types of subscription flows: smart flows and dumb flows. Smart flows are identified by their source and the content they subscribe to. Dumb flows are only identified by the content they subscribe to. When two identical flows meet each other they join into one flow, see Figure 11.3.

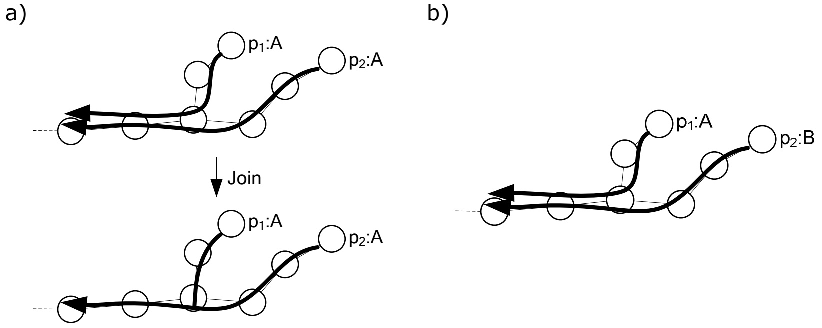


Figure 11.3. The joining (or not) of two advertisement flows. In fig. a) publishers p_1 and p_2 both publish content advertised as 'A'. In stead of forwarding both flows the flows are joined at the node where they meet, and only a single flow is forwarded. In fig. b) this is not possible as both publishers advertise different content.

11.3.2 Forwarding flows

Subscribers that are situated outside the backbone and that do not lie on, or are the final destination of, an advertisement flow coming from the backbone, do not know what content is being advertised in the network. They therefore act as the source of an empty subscription flow that is forwarded to the backbone or until it is able to join a subscription flow that also goes to the backbone. A so-called empty subscription is a subscription used to indicate that there is a subscriber that wishes to receive advertisements, but does not want to subscribe itself to any content. It is the mechanism for nodes to become aware of the content being advertised. It has been chosen because of its simplicity and consistency with regard to the rest of the design.

Publishers by definition advertise their content. When a publisher is outside the backbone it acts as a source of a directed advertisement flow towards the backbone. If the flow encounters any subscription flows on its way to the backbone that are also going to the backbone, then it will follow these subscription flows reversely. When the advertisement flow reaches the backbone it will become an undirected flow and be flooded around the backbone. Finally it will reversely follow any subscription path it encounters that comes from outside the backbone.

A subscriber situated outside the backbone that lies on, or is the final destination of, an advertisement flow coming from the backbone, can subscribe itself to any of the content being advertised. It does so by acting as a source for a matching smart directed subscription flow, that is forwarded reversely along any matching advertisement path. This means that a subscription path will always reach the backbone and will also reversely follow any matching advertisement paths going to the backbone. When the subscription flow reaches the backbone behaviour differs.

When an advertisement flow reaches a subscriber that has an empty subscription flow the subscriber can either subscribe itself to the advertised content or not. If the subscriber doesn't want to subscribe itself it will keep acting as a source of empty subscriptions. If it does want to subscribe itself the empty subscription flow is destroyed and a new subscription flow is created.

If the backbone is smart, each and every smart subscription flow that reaches the backbone is flooded throughout the entire backbone and routed over any matching advertisement flows coming from outside the backbone. When being routed outside the backbone the flow may branch out to follow multiple advertisements flows.

If the backbone is dumb the smart subscription flow that reaches the backbone is converted into a dumb flow. Subscription flows that match the same content are thus identical, even if they come from different sources. This has the effect that if a subscriber's flow reaches the backbone where a subscription flow matching the same content is already present, the second flow will merely join the first flow, in stead of being flooded throughout the backbone as well. After a subscription flow has been flooded around the backbone it is routed reversely over any matching advertisement flows coming from outside the backbone.

Figure 11.4 shows an example in which most of the behaviour described above is visualised.

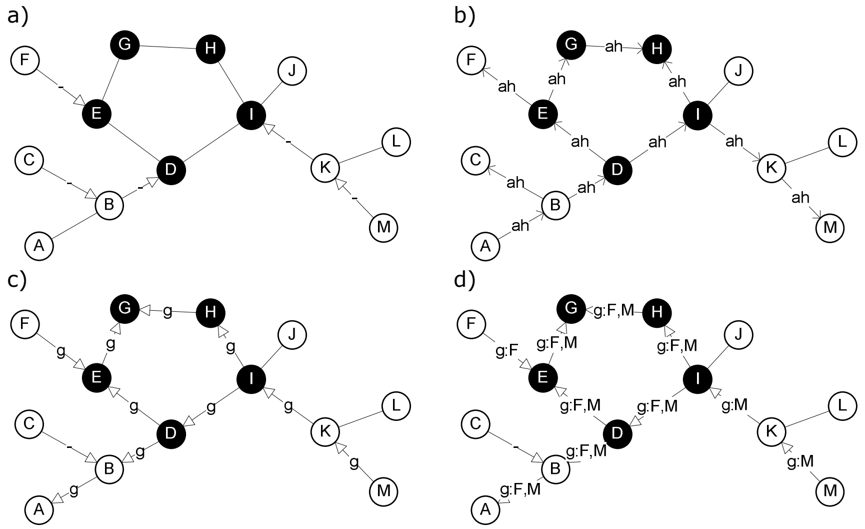


Figure 11.4. The advertisement and subscription flows in a network through time. Fig. a) shows the empty subscription flows when no content has been advertised yt. Fig. b) shows the advertisement flow (advertising content range 'az') routed to the backbone and over any subscription flows that go towards the backbone. Upon receiving these advertisements the nodes that wish to subscribe themselves to any content that is present (i.e., nodes F and M for content 'g') destroy their empty subscription flow and create a new subscription flow subscribing them to the content of choice. In the dumb backbone of fig. c) the identical subscriptions flows of nodes F and M are joined. In fig. d) when the subscriptions flows reach the smart backbone they are not considered identical and are both flooded around the backbone.

If a publisher wishes to stop advertising content it does so by simply stopping to act as a source for advertisements. This will eventually remove the advertisement flow from the network if there is no other publisher present. It takes some time because the advertisement flows are undirected. In the meantime needless advertisement and subscription flows may be maintained. This is however not deemed a problem as the overhead associated with such unnecessary advertisements and subscriptions is deemed acceptable.

Because the overhead associated with the unnecessary routing of unwanted messages is *not* deemed acceptable, a subscriber that wishes to stop subscribing will actively destroy its subscription flow (destroying a flow is discussed in Section 11.5). This is possible for any directed flow and is done without additional delays. When a subscriber destroys its flow it will at least be desroyed *until the backbone*. What happens next depends on the backbone. If it is a smart backbone destruction of the flow will continue, also outside the backbone, until it no longer exists. If it is a dumb backbone it will take some time, similar to the advertisement flow, before it is removed from the backbone and outside it. The flow is only removed if there are no other subscribers with matching subscription flows.

11.4 Message routing

Messages are routed reversely along any matching subscription flow (messages are only published when the publisher lies on or is the final destination of such a flow). If the flow is directed a message will (try to) follow the shortest reverse path, if the flow is undirected the message will be forwarded along every path.

A message published outside the backbone is *always* routed reversely over the shortest path of any matching subscription flow. This means that in the case of a dumb backbone a message is always routed to the backbone *and* reversely along any shotest paths it encounters before reaching the backbone. When it reaches the dumb backbone it is flooded around the entire backbone because (i) the subscription flow present in the backbone is undirected and all of its paths must therefore be followerd and (ii) because all equal dumb subscription flows are joined when they reach the backbone and the message does not know whether or not there are any more matching subscribers outside the backbone. Finally a message is routed reversely along the shortest path of any subscription flows that come from outside the backbone. Figure 11.5.a shows an example of a message being routed over a dumb backbone.

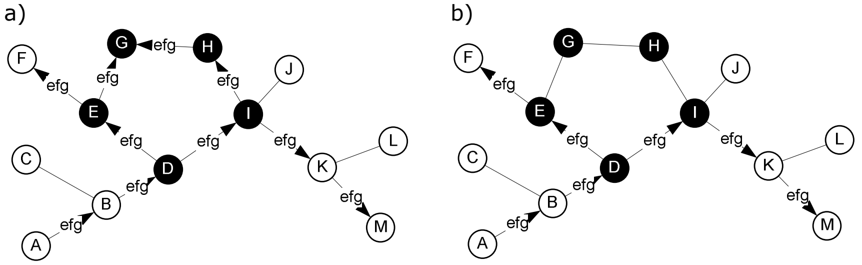


Figure 11.5. Message routing over the network of fig. 11.4. In a) the message containing content “efg” is routed over the dumb backbone of Figure 11.4.c. In b) the message is routed over the smart backbone of fi. 11.4.d.

With a smart backbone a message published outside the backbone is not necessarily routed to the backbone, but only when there is a subscription path coming from the backbone. This is only the case when there is at least one subscriber present in the network outside the publisher’s dominating group[§]. If a message reaches the backbone it is routed reversely along a single path for every subscription flow. This path does not need to be the shortest path: in stead, each backbone node will try to make a minimal selection of neighbours to forward the message to that will ensure that every subscription flow is covered but with the least number of neighbours and the smallest total number of message transmissions needed. Finally a message is routed reversely along the shortest path of any subscription flows that come from

[§] A dominating group is a group of members whose paths to the backbone all end at the same dominator.

outside the backbone. Figure 11.4.b shows an example of a message being routed over a smart backbone.

The previous section showed how a routing layer is created to route the messages over. This routing layer provides a means to perform efficient routing with high precision, especially outside the backbone or inside a smart backbone. To utilize this messages are forwarded by means of unicast, as broadcast would quickly kill the gained precision. This does however necessitate that messages must usually be transmitted more often than would be the case with broadcast. In some cases, especially when a large dumb backbone is applied, broadcast would perhaps be the more efficient and effective option. For the sake of consistentcy and the simplicity of design (mentioned at the beginning of the chapter) messages are always unicasted.

11.5 Creating, maintaining and destroying advertisement and subscription flows with beacons

In the previous section the dissemination of advertisements and subscriptions was discussed at flow level. Here the details of how such flows are created, maintained and destroyed are discussed. First however the structure of advertisements and subscriptions are presented.

11.5.1 The structure of advertisements and subscriptions

Both an advertisement and a so-called dumb subscription are made up out of the attributes *content description* (defining the content that is either advertised or subscribed), *previous hop* (used to create directed flows, explained below) and *hop count* (used to create both directed and undirected flows, also explained further below). A so-called smart subscription has attributes *content description*, *source identifier, previous hop* and *hop count*. The attribute source identifier is used to distinguish subscription flows that have identical content descritions but come from different sources. The structures are all listed in Table 11.1.

What	Attribute	Type	Description
The structure of an advertisement	content description	application-specific	Defines the content being advertised.
	previous hop	identifier	Identifies the previous hop in the flow.
	hop count	counter	The distance from the flows source.
The structure of a smart subscription	content description	application-specific	Defines the content being subscribed.
	source identifier	identifier	Identifies the source node of the flow.
	previous hop	identifier	Defines the content being advertised.
	hop count	counter	The distance from the flows source.
The structure of a dumb subscription	content description	application-specific	Defines the content being advertised.
	previous hop	identifier	Identifies the previous hop in the flow.
	hop count	counter	The distance from the flows source.

Table 11.1. The structure of an advertisement, a dumb subscription and a smart subscription.

Advertisements and subscriptions are indentifed first by their content description and second, only if it is a smart subscription, by their source identifier. Two advertisements (or subscriptions) that have a similar content description (and source identifier in the case of a smart subscription) are said to be equal or of the same type.

Note that although two smart subscriptions with equal content description but different source identifers are presented here as two completely separate subscriptions, this strict separation is not necessary when implementing the design. In the simulation model created for the benefit of the experiments described in Chapter 13, two smart subscriptions that have equal content description but different source identifiers are not stored seperately by a node. In stead only a single subscription is stored but with two [source identifier, previous hop, hop count] tuples, one for each included subscription. In a similar way each advertisements or dumb subscription is constructed with a single [previous hop, hop count] tuple. This also affords two views: one that looks at all subscriptions distinctly and one that combines them. Which of the two views is used within this thesis depends on what is practical.

11.5.2 Creating flows

Advertisements and subscriptions are disseminated through the network by means of beacons. Beacons are packets in which a node includes all the advertisements and subscriptions for which it has some state information stored. The view that a node has of another node is determined by the last beacon it received from that node.

Flows are created by means of beacons as follows: a node that acts as a source of, e.g., an advertisement flow, inludes the advertisement in its beacon and transmits the beacon, informing its neighbours that it has created an advertisement flow. From that point forward any neighbour that receives a beacon from a node (in this case from the source of the flow) and knows that it is designated as a next hop in the flow will also include the advertisement in its beacon and transmit it (how a node determines which flows it should include in a beacon is specified in the next chapter). Figure 11.6 shows how this is done.

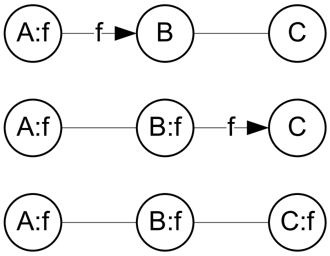


Figure 11.6. Creation of a flow. First node A sends a beacon containing the flow info to B, and stores the flow info locally. Then B also stores the flow info locally and send a beacon with it to C. When C has also stored the flow info the flow goes from A to B to C.

In this thesis a flow is defined as a chain of nodes that all have included an advertisement or subscription of a certain type into their last transmitted beacon. Flows are identified in a similar way to the advertisements and subscriptions of which they consist. Flows may be undirected or directed, smart or dumb (if left unspecified the flow is dumb): each is described below.

To create an undirected flow only the *hop count* attribute of an advertisement or subscription is needed. Each time a node x has to beacon it will include each flow that it is the source of with a hop count of 0. For other flows it first chooses, for each flow that it wants to include in its beacon, the neighbour that acts as the previous hop for that flow. This is the neighbour that has the lowest hop count for that flow. Node x then includes the flow into its own beacon with a hop count 1 higher than that hop count. Figure 11.7.a shows the resulting hop counts stored by nodes for two flow sources in a network.

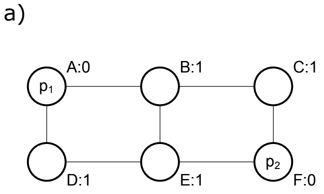


Figure 11.7.a. Two identical undirected flows in a network. The labele p_1 and p_2 indicate the sources of both flows. The nodes are labeled with their identifier and the hop count of the flows. As the flows are identical they are joined. Supposing that the flows are advertisement flows, each node will include an advertisement in its beacon with identical content description (left unspecified) and with the hop count set to the values next to the nodes. The *previous hop* attribute is not used.

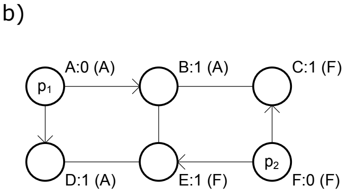


Figure 11.7.b. Two identical directed flows in a network. The labele p_1 and p_2 indicate the sources of both flows. The nodes are labeled with their identifier, the hop count of the included flow and the previous hop of the flow. The flows are identical and thus joined. The arrow denotes the direction of the flows. Supposing that the flows are advertisement flows, each node will include an advertisement in its beacon with identical content description (left unspecified), and the *previous hop* and *hop count* attributes set to the values next to the nodes.

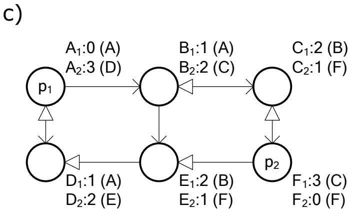


Figure 11.7.c. Two directed smart flows in a network. The nodes labeled p_1 and p_2 have source identifiers 1 and 2. The labels next to the nodes show for each flow the identifier of the node, the flow source identifier and the previous hop. The thin arrows show the flow of p_1, the open arrows the flow of p_2. Supposing that the flows are subscription flows, each node will include a subscription in its beacon with a content description (left unspecified) and the *previous hop, source identifier* and *hop count* attributes set to the values next to the nodes.

To create a directed flow the *hop count* attribute and the *previous hop* attribute are needed. Each time a node *x* has to beacon it will include each flow that it is the source of with a *hop count* of 0 and its *previous* hop set to its own identifier. For other flows it first chooses, for each flow that it wants to include in its beacon, the neighbour that acts as the previous hop for that flow. This is the neighbour *y*, taken from the set of neighbours that do not have the *previous hop* attribute of their flow set to x’s identifier, that has the lowest hop count for that flow. Node x then includes the flow into its own beacon with *previous hop* set to y’s identifier and *hop count* set 1 higher than y’s hop count. Figure 11.7.b shows the resulting hop counts and previous hops stored by nodes for two flow sources in a network.

Creating a smart directed flow is similar to creating a dumb directed flow. The difference lies in how flows are identified, which in the case of a smart flow is also done based on the source identifier. Each flow source will therefore set the flow’s source identifier to its own identifier, and every next hop in the flow will copy this source identifier. Figure 11.7.c shows how two smart directed flows both are spread throughout the entire network, as opposed to the way the dumb directed flows in 11.7.b only spread halfway.

When the single source of an undirected flow stops acting as the source it will simply not include the flow in its next beacon. To prevent the other nodes in the network (and also itself) from counting-to-infinity (a situation in which the source of a flow is removed from the network without explicitly voiding its flow and the remaining nodes that together make up the flow will increase their hop counts iteratively and unboundedly, see Figure 11.8.a) the hop count of a flow can not exceed a predefined maximum. This maximum should be set high enough to allow a flow to be forwarded throughout the entire network. If a flow has reached its maximum it will no longer be included in beaon. In this way, when a source stops, the other nodes will iteratively increase their hop counts until they reach the defined maximum. Figure 11.8.b shows the case for three nodes and a maximum hop count of 4. If there are multiple sources for a given flow and only one stops acting as a source the nodes will not count to infinity: in stead the flows hop counts kept at the nodes will, after some beacon rounds, reflect the distance to the sources that have not been removed. Figure 11.8.c shows the stable state of the network in Figure 11.7.a when the top-left source is removed.

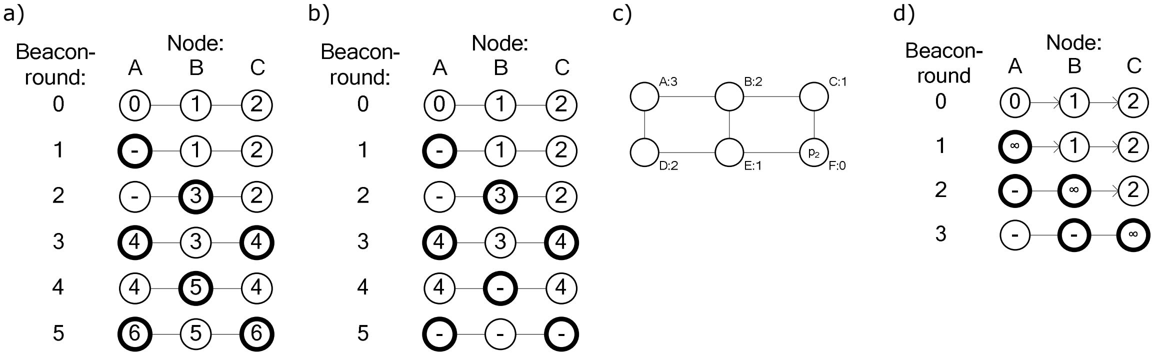


Figure 11.8. Destroying a flow. The numbers in the nodes represent the hop counts of the flows. The fat nodes are the nodes that have most recently beaconed. In a) the nodes count to infinity after node A did not explicitly destroy the flow. Each round a node will pick a neighbour as its previous hop and set its own hop count one higher. In b) this is prevented by limiting the max hop count of a flow to 4. Fig. c) shows how a the flow from the bottom right node in fig. 11.7.1 takes over the whole network when the only other flow is removed. Finally in d) the nodes do not count to infinity or any predefined maximum because the flow is explicitly destroyed by setting its hop count to infinity.

When the source of a smart directed flow stops acting as the source it will set the hop count of its flow to infinity the next time it beacons, indicating that it has stopped issuing the flow. The next beacon after that will no longer contain the flow. When the neighbours detect that there is flow with its hop count set to infinity they will also include the flow with its hop count set to infinity in their next beacon, and refrain from including it in the beacon after that. A flow that has its hop count set to infinity is said to be poisoned. Figure 11.8.d shows how this is done for the same network as in fig. 11.8.a. The case of a dumb directed flow is ignored, as the design does not make use of such flows.

11.6 Beaconing as a means for disseminating advertisements and subscriptions

The previous section already explained that the view that the network has of a node is determined by the node's last beacon. Table 11.2 shows the structure of a CBR beacon. Creating a beacon is a task that for a large part is determined by the inclusion of advertisements and subscriptions which was also described in the previous section. In this section the timing and the means of transmission of beacons are discussed. The complete specification of how a beacon is created in differing cases can be found in Chapter 12.

Attribute name	Description
source	Identifier of the node that created the beacon.
id	A counter that increases with each beacon that is transmitted.
beacon identifiers	Maps the identifier of each relevant neighbour to the *id* of the last beacon received from that node.
advertisements	The included advertisements. For the structure of an advertisement see Table 11.1.
subscriptions	The included subscriptions. For the structure of a subscription see Table 11.1.

Table 11.2. The attributes that together make up a beacon.

A node will create a new beacon every time it receives a beacon from a relevant neighbour and every time its set of relevant neighbours is updated by the Ck-HDS layer. At the least it will create a beacon a parameterized amount of time after the previously created beacon. This time window is called the CBR Beacon Interval (CBI). A node will however only transmit its beacon when the beacon has changed with respect to the last beacon that it transmitted, or when there are one or more relevant neighbours that did not receive the last beacon the node transmitted. A node knows whether or not all its neighbours have received its last beacon by inspecting the beacon identifiers of their most recent beacons. When a node decides to beacon it will itself include the identifier of each relevant node together with the identifier of the most recent beacon it has of that node. After also having included the advertisements and subscriptions it will transmit its beacon randomly in a time window called the CBR Beacon Window (CBW).

Transmitting beacons every time something has changed has the advantage that information is spread quickly throughout the network, enabling adequeate routing. It has the disadvantage however that slight changes in the network (e.g., its topology) may result in a flood of beacons being transmitted, emanating from the point of change. This can be considered a positive effect when a change prevents adequeate routing from taking place, but a negative one when changes have little effect on message routing. Another option would be to simply transmit beacons in predefined intervals, which is however less responsive to changes. Because this design focuses on maintaining accurate routing layers to route messages over even during node mobility beacons are sent for every change. To prevent floods of beacons to be created for every trivial change in the network topology however a delay has been included in the timing of beacon transmissions: when a node's beacon only differs from the last beacon that was transmitted because either an advertisement's or a subscription's hop count has been changed, the node will wait a period of time before transmitting the beacon called the Dampening Period (DP).

Beacons are transmitted by means of broadcast, although there is an argument for using the Ck-HDS structure as a topology for unicast transmission. With unicast only relevant neighbours will receive the beacons and communication is reliable. Broadcast is more efficient with bandwidth however, as it only needs to transmit a beacon once. Furthermore in a mobile network nodes can benefit from beacons received from (previously) unrelevant neighbours when the topology changes. Finally, the beacon identifiers have been added to counter at least some of the unreliability of broadcast transmission: if a node notices that its neighbour has an old beacon identifier included in its beacon it will transmit beacons until it has received a beacon from its neighbour with an up to date beacon identifier.

11.7 Smart versus dumb

It is easily observed that the smart backbone provides for more efficient routing. This comes at the price of keeping more state information at the backbone nodes and, more importantly, of keeping this information up to date. The previous section showed that every time the network changes this may trigger a wave of beaconing rounds. As the network grows larger, as it becomes mobile or as the number of subscribers increase, the effort needed to keep all smart subscription flows up to date rises dramatically (as will be shown in the analysis of Chapter 13).

As mobility increases subscription flows will sometimes break, causing messages not to be delivered to all subscribers. Smart flows are more vulnerable to breaking as they require throughout the entire backbone a valid directed shortest path, whereas with dumb flows the flows only need to be present in the backbone, not necessarily valid. Outside the backbone paths are more vulnerable to breaking as they increase in length.

The behavioural differences of both types of backbones grow larger as the backbone increases in size. As a smart backbone becomes larger, message routing becomes increasingly efficient. A large smart backbone also means an increased amount of effort to keep subscription flows up to date however. For a small backbone the difference in message routing becomes smaller, because message routing to and from the backbone is always efficient.

11.8 Scalability

No CBR system that has a central point where all the advertisements and subscriptions are present is truly is scalable, and neither is this system. Here the backbone acts as a central meeting point: everyone who has anything to offer will tell the backbone, and everyone who wants anything will first ask the backbone what's available and then tell it what it wants. As the number of publishers and subscribers increase therefore the overhead of advertisements and subscriptions will also keep on increasing.

Scalability is influenced by the number of actors and by the type of the backbone. A smart backbone is less scalable than a dumb backbone as it requires more information to be stored for each subscription flows. The impact of this difference is relative to the total size of a subscription and the total number of subscribers however and is thus application specific. Section 13.1 goes deeper into the subject of calulating overhead.

Chapter 12
Specification

The design described in the previous chapters is here fully specified.

12.1 Creating, transmitting and receiving beacons

Section 10.5.2 already describes the creation of flows. Here the inclusion of advertisements and subscriptions into beacons is fully specified. Every time a node creates a beacon it does so based on the advertisements and subscriptions that are held by the node itself and that have been included in the last beacon the node received from each of its relevant neighbours. The subsections below explain in detail how nodes with different roles create different beacons. First however a shorthand notation is defined (used later on in various examples) and some definitions are given that are used throughout the chapter.

The shorthand notation of content begin advertised is a two-character string defining the start and end of the range of the content (explained in Section 11.2). The shorthand notation of content being subscribed to is a single character.

The shorthand notation to list a node's advertisements is as follows:

<content_description>/<previous_hop>,<hop_count>*[/<previous_hop>,<hop_count>][*||<content_description>/<previous_hop>,<hop_count>*[/<previous_hop>,<hop_count>]]

The shorthand notation for a node's subscriptions is:

<content_description>/<source_identifier>,<previous_hop>,<hop_count>*[/<previous_hop>,<hop_count>]*[||<content_description>/<source_identifier>,<previous_hop>,<hop_count>*[/<previous_hop>,<hop_count>]]

The shorthand for a node's beacon is (only advertisements and subscriptions are included):

$B_{\text{<node\_identifier>}}$: {<advertisements>,-}_{<subscriptions>,-}

Figure 12.1 shows a node A that advertises content in the range [a,a] and subscribes to content [c] and has a neighbour B that advertises contant in the range [b,g] and is also subscribed to [c], and the resulting shorthand notations.

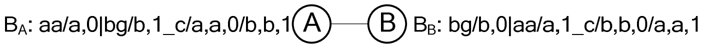

B_A: aa/a,0|bg/b,1_c/a,a,0/b,b,1
A
B
B_B: bg/b,0|aa/a,1_c/b,b,0/a,a,1

Figure 12.1. An example of the shorthand notation used in this chapter.

Advertisements or subscriptions come from another node if that node is the previous hop in the advertisement/subscription flow: if a node X has a neighbour Y which has an advertisement A with its previous hop set to X, then A comes from X for Y. In Figure 12.1 advertisement b/b,0 comes from node B for node A. When an advertisement/subscription comes from anywhere it means that it comes from either the node itself, from a sibling, from the parent or from a child. When it comes from elsewhere it means coming from any node except itself. An advertisement/subscription coming from below means that it comes from a child.

An advertisement/subscription has already been added to the beacon if an advertisement/subscription of the same type is already present in the beacon. An advertisement/subscription has been added ‘by the node itself' if the node is a source of an advertisement/subscription of similar type and this advertisement/subscription has already been included in the beacon. An advertisement/subscription coming from the node itself has hop count 0 and the previous hop is set to the node's own identifier.

If an advertisement/subscription coming from a neighbour is included in a node’s beacon, the advertisement/subscription will always have a hop count that is 1 higher than the advertisement/subscription at the neighbour. The previous hop will also always be set to the neighbour’s identifier, unless both nodes are dumb dominators: in that case the previous hop in the included advertisement/subscription will be set to the generic dumb dominator identifier (DDI).

A [subscription source, previous hop, hop count] tuple is refered to as a subscription-tuple. A [previous hop, hop count] tuple is refered to as an advertisement-tuple.

A member or smart dominator may add multiple subscriptions of the same type to the beacon: each time a subscription-tuple is added to the subscription in the beacon. A member or smart dominator can only list each neighbour once as a previous hop for a subscription of a certain type however. In Figure 12.2 node B has multiple subscription-tuples for subscription ‘x’ as it has added a tuple for the subscription coming from node C and the subscription coming from node D. Node A however can only list node B once as its previous hop and has to choose a tuple to add to its own subscription. It chooses, from the set of ‘x's tuples that do not come from A, the tuple with the lowest hop count (in this case tuple [D,D,1]).

An empty subscription is a subscription whose attribute *content description* has a special value. An empty subscription flow indicates that there is a subscriber who wishes to receive advertisements, but doesn't want to subscribe itself to any of the content being advertised.

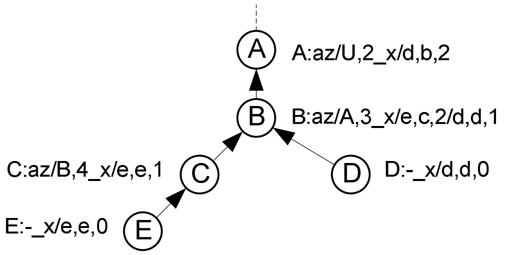


Figure 12.2. Two subscription flows outside the backbone. Node A is connected to the backbone via a node U that has been left out of the figure. The arrows show the subscription flows. Node D and E both acts as sources for subscription ‘x’. It shows how node A must choose which subscription-tuple of B to include in its beacon. It chooses d,d,1 because it has a lower hop count.

A node *x* is said to know a subscription *s* if *x* has at least one beacon, received from a relevant neighbour *y*, in which *s* is included. Neighbour *y* is said to have a subscription path for a subscription *s* (*s* being included in the last beacon that *x* received from *y*) if *y.s* has a previous hop other than *x*. The length of a path is equal to the hop count of *y.s*. Neighbour *y* is said to have the shortest path for *s* if there exists no neighbour *z* (*z* in N*(x)*) that has a shorter path for *s.* If this holds multiple neighbours then, from that set of neighbours, one is randomly chosen as the node that has the shortest path for *s*.

12.1.1 When to create beacons

Beacons are created:

- whenever a beacon has been received;
- whenever the Ck-HDS layer has updated the topology, or
- CBI seconds after the last time a beacon was created.

Whether or not to transmit a beacon is evaluated according to the rules laid down in the next section.

12.1.2 When to transmit beacons

A node x will only schedule a newly created beacon for transmission in the following cases:

1. If the most recently received beacon for a relevant neighbour does not contain a beacon identifier for x.
2. If the most recently received beacon for a relevant neighbour contains an obsolete beacon identifier for x.
3. When *DP* > 0 only if the beacon has changed in the non-strict sense (defined below).
4. When *DP* = 0 if the beacon has changed in the strict sense.

A beacon B_X is equal to another beacon B_Y in the strict sense if and only if:

- if B_X.source = B.source;
- if for every advertisement in B_X an equal advertisement exists in B_Y with exactly the same set of advertisement-tuples;
- if for every advertisement in B_Y an equal advertisement exists in B_X with exactly the same set of advertisement-tuples;
- if for every subscription in B_X an equal subscription exists in B_Y with exactly the same set of subscription –tuples, and
- if for every subscription in B_Y an equal subscription exists in B_X with exactly the same set of subscription –tuples.

Two beacons are non-strictly equal if they are equal in everyhting except for the hop counts of the advertisements and subscriptions. Thus a beacon B_X is equal to another beacon B_Y in the non-strict sense if and only if:

- if B_X.source = B_Y.source;
- if for every advertisement in B_X an equal advertisement is also present in B_Y;
- if for every advertisement in B_Y an equal advertisement is also present in B_X;
- if for two equal advertisement A_X in B_X and A_Y in B_Y:
  - every previous hop in A_X is also present in A_Y;
  - every previous hop in A_X is also present in A_Y;
- if for every subscription in B_X an equal subscription is also present in B_Y;
- if for every subscription in B_Y an equal subscription is also present in B_X;
- if for two equal subscriptions A_X in B_X and A_Y in B_Y:
  - every previous hop in A_X is also present in A_Y;
  - every previous hop in A_X is also present in A_Y.

Beacon B_X:aa/h,3/t,2|bb/x,0_f/x,x,0/u,a,2 is thus strictly equal to
B_X:aa/t,2/h,3|bb/x,0_f/u,a,2/x,x,0 and non-strictly equal to
B_X:aa/t,6/h,3|bb/x,0_f/u,a,3/x,x,0 because advertisements aa/t,2 and aa/t,6 and

subscriptions f/u,a,2 and f/u,a,3 only differ in their hop counts. Beacon B_X:aa/s,2/h,3|bb/x,0_f/u,g,2/x,x,0 is not equal in any way because advertisement aa/s,2 and subscription f/u,g,2 now have a different previous hop.

A transmission is scheduled, if at all, randomly in the window CBW that starts immediately or, if the last transmission was within *CBW* seconds, *CBW* seconds after the last transmission.

Whenever a beacon has been transmitted any poisoned subscriptions (subscriptions with their hop count set to infinity) present are removed.

12.1.3 Receiving beacons

When a beacon has been received by a node x from a node y:

- x’s view of y is updated;
- any poisoned subscriptions in y’s beacon are added to the set of poisoned subscriptions that should be included in the next beacon;
- for each poisoned subscription included in y’s beacon any equal subscriptions in beacons previously received from other nodes are removed;

12.1.4 Creating beacons as a member

When a member creates a beacon it:

1. Adds it source identifier.
2. Adds the identifiers of the beacons it last received from its parent and children.
3. Adds its own advertisements.
4. Adds any advertisement coming from its children that has not already been added by itself. If multiple children have the same advertisement the advertisement with the lowest hop count is eventually added.
5. Adds any advertisement that comes from the parent if at least one child has a subscription that comes from elsewhere. If an equal advertisement has already been added replace it iff the parent's advertisement has a lower hop count.
6. Adds any of its own subscriptions that match any advertisements coming from elsewhere.
7. Adds an empty subscription, with the node itself as the subscription's source and its previous hop and with a hop count of 0, iff this node is a subscriber but none of its subscriptions have been added.
8. Adds any subscription coming from the parent if it matches any advertisement coming from below and if an equal subscription hasn't already been added by the node itself.
9. Adds any subscription coming from its children if that subscription matches any advertisement coming from elsewhere and if an equal subscription hasn't already been added by the node itself.
10. Adds an empty subscription if its own set of subscriptions is non-empty but it has not yet added any subscriptions yet.
11. Adds any poisoned subscription.

12.1.5 Creating beacons as a smart dominator

When a smart dominator creates a beacon it:

1. Adds it source identifier.
2. Adds the identifiers of the beacons it last received from its parent and children.
3. Adds its own advertisements.
4. Adds any advertisement coming from its children that hasn't already been added by itself. If multiple children have the same advertisement the advertisement with the lowest hop count is eventually added.
5. Adds any advertisement coming from its siblings that hasn't already been added by itself. If multiple siblings have the same advertisement the advertisement with the lowest hop count is eventually added.
6. Adds any of its own subscriptions that match any advertisements coming from anywhere.
7. Adds any subscription coming from its children if that subscription matches any advertisement coming from anywhere and if an equal subscription hasn't already been added by the node itself.
8. Adds any subscription coming from its siblings if that subscription matches any advertisement coming from anywhere.
9. Adds any poisoned subscriptions.

Figure 12.3 shows an example network with a smart backbone.

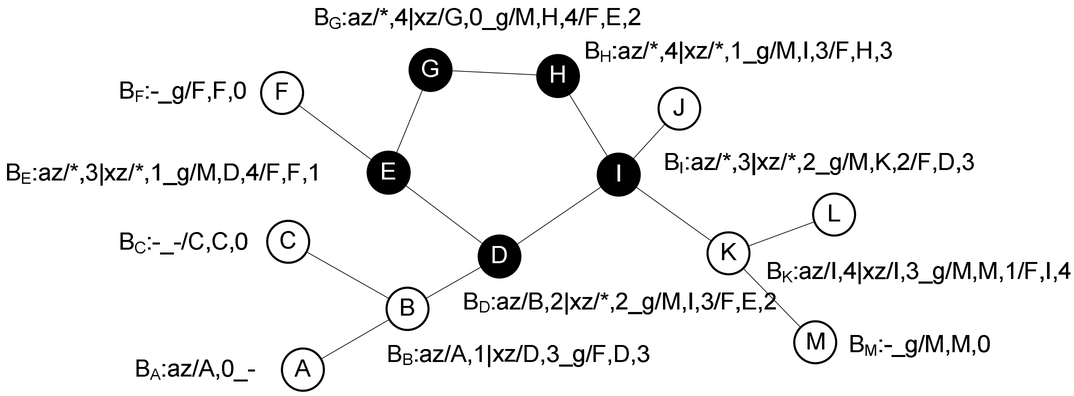

B_G:az/*,4|xz/G,0_g/M,H,4/F,E,2
B_H:az/*,4|xz/*,1_g/M,I,3/F,H,3
B_F:-_g/F,F,0
F
G
H
J
B_E:az/*,3|xz/*,1_g/M,D,4/F,F,1
E
I
B_I:az/*,3|xz/*,2_g/M,K,2/F,D,3
L
B_C:-_-/C,C,0
C
D
K
B_K:az/I,4|xz/I,3_g/M,M,1/F,I,4
B
B_D:az/B,2|xz/*,2_g/M,I,3/F,E,2
M
B_M:-_g/M,M,0
B_B:az/A,1|xz/D,3_g/F,D,3
B_A:az/A,0_-
A

Figure 12.3. An example of a stable network with a smart backbone in which the last transmitted beacon of each node is shown using the shorthand notation, A star ('*') has been used to represent DDIs. Node A is an advertiser, nodes F and M are both subscribed to the content A publishes. Node B is also a subscriber but not to any content currently being advertised on the network.

12.1.6 Creating beacons as a dumb dominator

When a dumb dominator creates a beacon it:

1. Adds it source identifier.
2. Adds the identifiers of the beacons it last received from its siblings and children.
3. Adds its own advertisements.
4. Adds any advertisement coming from its children that hasn't already been added by itself. If multiple children have the same advertisement the advertisement with the lowest hop count is eventually added.
5. Adds any advertisement coming from its siblings that hasn't already been added by itself or by its children.
6. Adds any of its own subscriptions that match any advertisements coming from elsewhere.
7. Adds any subscription coming from its siblings that matches any advertisements coming from anywhere and that wasn't already added to the beacon by the node itself. If multiple siblings have the same subscription the subscription that has the lowest hop count is eventually added.
8. Adds any subscription coming from its children that matches any advertisements coming from anywhere and that wasn't already added to the beacon by the node itself. If multiple children have the same subscription the subscription that has the lowest hop count is eventually added.
9. Adds any poisoned subscription.

Figure 12.4 shows an example network with a dumb backbone.

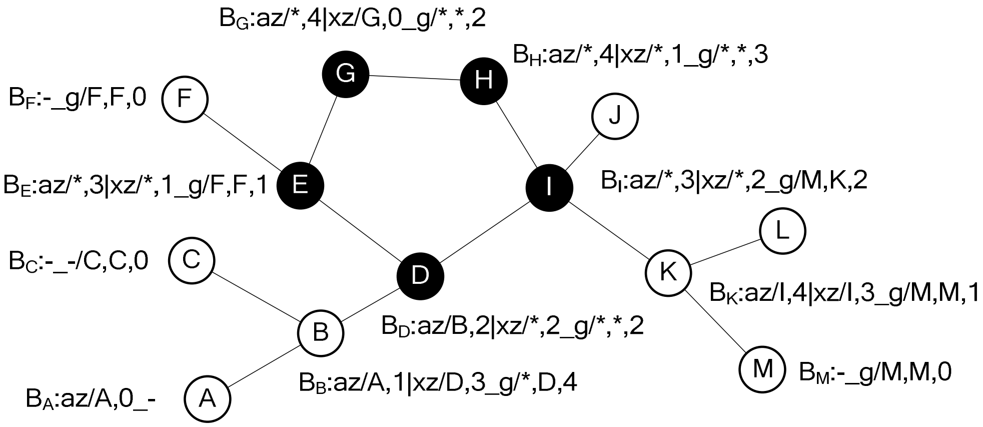

B_G:az/*,4|xz/G,0_g/*,*,2
G
H
B_H:az/*,4|xz/*,1_g/*,*,3
B_F:-_g/F,F,0
F
J
B_E:az/*,3|xz/*,1_g/F,F,1
E
I
B_I:az/*,3|xz/*,2_g/M,K,2
L
B_C:-_-/C,C,0
C
D
K
B_K:az/I,4|xz/I,3_g/M,M,1
B
B_D:az/B,2|xz/*,2_g/*,*,2
M
B_M:-_g/M,M,0
B_B:az/A,1|xz/D,3_g/*,D,4
B_A:az/A,0_-
A

Figure 12.4. An example of a stable network with a dumb backbone in which the last transmitted beacon of each node is shown using the shorthand notation,A star ('*') has been used to represent DDIs. Node A is an advertiser, nodes F and M are both subscribed to the content A publishes. Node B is also a subscriber but not to any content currently being advertised on the network.

12.2 Message routing

Every message has an attached lists of identifiers, called the don't-list, to whom the message shouldn't be routed. For a message *m* this list is refered to as *m.dont*.

When a node receives or publishes a message it schedules it for transmission uniformly in the message window (MW).

12.2.1 Forwarding messages as a member

When a member *x* has a message *m* to send it will first build a list of next hops (identifiers of neighbours to whom the message should be forwarded) and for each next hop a list of sources to cover:

1. If *x.parent* is not in *m.dont* go over all subscriptions that match m. For every subscription-tuple of each subscription:
    a. If the subscription-tuple comes from *x.parent* and the source identifier is not in m.dont add the source identifier to the list of sources for *x.parent*. Add *x.parent* to the list of next hops iff its list of sources is not empty.
2. For every child *c* that is not in *m.dont*:
    a. For every subscription that comes from *c* and that has at least one subscription source in its set of subscription-tuples that is not in *m.dont*, add for every subscription-tuple that comes from *c* and has a source-identifier that is not in *m.dont* the source identifier to the list of sources for *c*. Add *c* to the list of next hops iff its list of sources is not empty.

Next, for each identifier *y* in the list of next hops:

1. Create a copy m_c of the message *m*.
2. Add every identifier in *m.dont* to m_c*.dont*.
3. Add every source that is covered by a node other than *y* to m_c*.dont*.
4. If *x* is a subscriber add *x*'s identifier to m_c*.dont*.
5. Send m_c to y.

12.2.2 Forwarding messages as a dumb dominator

Dumb dominators forward a message to anyone interested. A dumb dominator *x* that has a message *m* to send first builds a list of next hops (identifiers of neighbours to whom the message should be forwarded):

1. Add every sibling to the list of next hops that has at least one matching subscription that does not have *x* as its previous hop.
2. Add every child to the list of next hops that has at least one matching subscription that does not have *x* as its previous hop.

Next, for each identifier *y* in the list of next hops:

1. Create a copy m_c of message m.
2. Add every next hop other than *y* to m_c*.dont*.
3. If *x* is a subscriber add *x*'s identifier to m_c*.dont*.
4. Send m_c to *y*.

Figure 12.5 shows an example of message routing for the network with the dumb backbone in Figure 12.4.

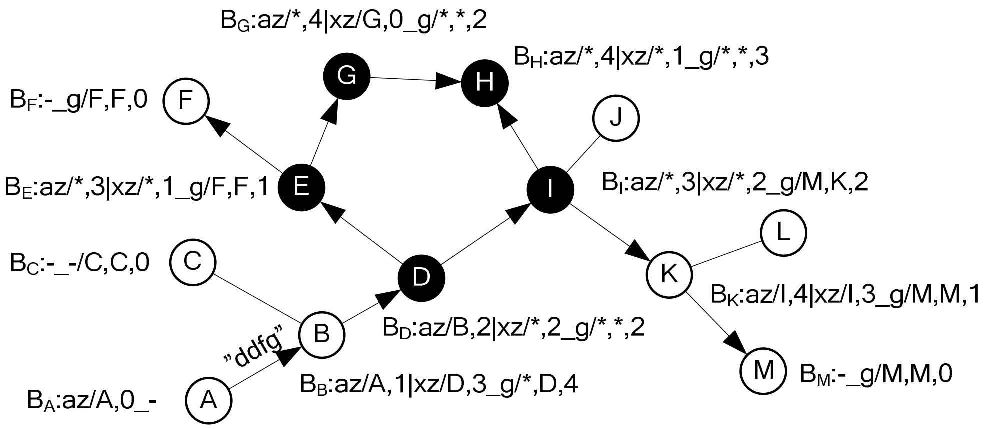

B_G:az/*,4|xz/G,0_g/*,*,2
B_H:az/*,4|xz/*,1_g/*,*,3
B_F:-_g/F,F,0
F
G
H
J
B_E:az/*,3|xz/*,1_g/F,F,1
E
I
B_I:az/*,3|xz/*,2_g/M,K,2
L
B_C:-_-/C,C,0
C
D
K
B_K:az/I,4|xz/I,3_g/M,M,1
B
B_D:az/B,2|xz/*,2_g/*,*,2
"ddfg"
M
B_M:-_g/M,M,0
B_B:az/A,1|xz/D,3_g/*,D,4
B_A:az/A,0_-
A

Figure 12.5. Routing of the message "ddfg" in the network with the dumb backbone of Figure 12.4. The message is flooded around the entire backbone and is received twice by node H.

12.2.3 Forwarding messages as a smart dominator

Smart dominators try to to forward a message to as few next hops as possible, while still covering every source. Based on the hop counts of their neighbours' subscriptions they also try, for a minimal set of next hops, to keep the total hop count for every source as low as possible.

A smart dominator *x* builds a minimal list of next hops (identifiers of neighbours to whom the message should be forwarded) in a number of steps. The set of next hops, here refered to as N_{NH}, is said to be covering for a message *m* if for every known subscription *s* coming from elsewhere there is at least one hop in the set of next hops that has a path to *s*.

For a a given set of next hops (that each have a set of subscriptions) and a given set of subscription sources, assuming that the set of next hops is covering, the total minimum hop count T_{HC} is calculated by adding for every subscription source s_s the path length of the shortest path for s_s.

In the first step, for every known subscription *s* that comes from elsewhere and whose source identifier is not included in *m.dont*:

1. if there exists only one neighbour that is not included in *m.dont* and that has a path for *s* add that neighbour to N_{NH};
2. add every neighbour *y* that is a subscriber of *s* and that is not included in *m.dont* to N_{NH}.

If the resulting set of next hops is a covering set, go on to the third step. Else a set of additional next hops must be created in the second step, called N_{add}, that cover any subscriptions left uncovered by N_{NH}.

Let T_{HC} represent the total minimum hop count for the set of all known subscriptions and the combined sets of next hops N_{add} and N_{NH}. For each neighbour y of x that has a shortest path for an as yet uncovered subscription create a list of additional next hops called N_y:

1. Add y to N_y.
2. If N_{NH} and N_y together are not covering then, until N_{NH} and N_y are covering, keep adding additional neighbours to *Ny* that have the shortest path for *any* as yet uncovered subscription[**].
3. Let T_y be the total minimum hop count for all subscriptions and the combined sets N_{NH} and N_y. If $T_y < N_{NH}$ set N_{add} to N_y.

In the third step, for a minimal covering set, the node x decides which next hop is responsible for which subscription source. For every neighbour y in the set of next hops:

1. Create a copy m_c of message m.
2. Add every known subscription source that is not in *m.dont* and to which y does not have the shortest path to m_c*.dont*.
3. If x is a subscriber add x's identifier to m_c*.dont*.
4. Send m_c to y.

Figure 12.6 shows an example of message routing for the network with the dumb backbone in Figure 12.3.

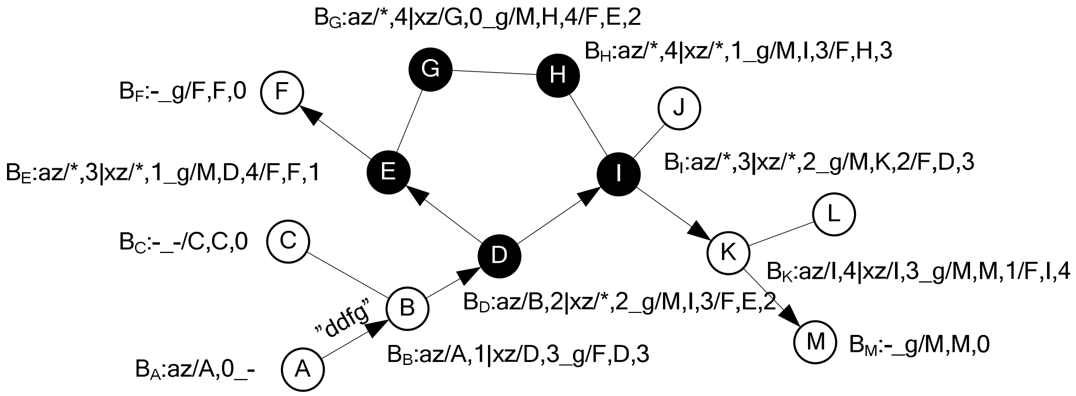

B_G:az/*,4|xz/G,0_g/M,H,4/F,E,2
B_H:az/*,4|xz/*,1_g/M,I,3/F,H,3
B_F:-_g/F,F,0
F
G
H
J
B_E:az/*,3|xz/*,1_g/M,D,4/F,F,1
E
I
B_I:az/*,3|xz/*,2_g/M,K,2/F,D,3
L
B_C:-_-/C,C,0
C
D
K
B_K:az/I,4|xz/I,3_g/M,M,1/F,I,4
"ddfg"
B
B_D:az/B,2|xz/*,2_g/M,I,3/F,E,2
M
B_M:-_g/M,M,0
B_B:az/A,1|xz/D,3_g/F,D,3
B_A:az/A,0_-
A

Figure 12.6. Routing of the message "ddfg" in the network with the smart backbone of Figure 12.3. Node D chooses E as its next hop to reach subscription source F, and node I as a next hop to reach subscription source M.

[**] For a set of neighbours, each with their own set of subscription paths, the neighbour with the shortest path for *any* subscription is the neighbour that has a path for which holds that there is no other neighbour with a shorter path. The type of the compared paths (i.e., the type of the subscription that makes up the path) is irrelevant.

Chapter 13
Performance evaluation of the content-based routing system

The goal of this chapter is to compare the performance of different parameter settings (a backbone that is either smart or dumb, small or large) for differing network conditions (sparse or dense networks, static or mobile) and different CBR loads (light loads, heavy loads). Performance is expressed in completeness (what fraction of the subscribers eventually receives a published message), precision (what fraction of the messages that a node receives was the node subscribed to and was received for the first time) and by means of a cost function that determines the load placed on the network by the system.

Testing has been done by means of simulation, using the simplified simulation model described in Section 2.3: a model that does not incorporate existing network-technologies but in stead gives a conceptual representation of a MANET. The results of these tests should therefore be interpreted as a proof of concept, a means to gauge the feasability of the design. In no way are the results directly comparable to the performance results of existing techniques. Static networks have been generated using the Ad Hoc Network Graph Model, ensuring connected and randomly generated networks; for node mobility the Random Waypoint Model with uniform and stable speeds has been used. Both techniques are also described in Section 2.3.

The simplified simulation model allows for complete control and track-keeping of the state and actions of the individual nodes and the network as a whole. Measurements have been made by keeping track of every transmitted beacon and message. For every transmitted beacon the number of beacon identifiers, advertisements and subscriptions has been recorded, as well as the size of the latter two (more on this in the next section). Taken together this allows for an expression of the load placed on the network by transmitted beacons that is independent of any application-specific properties. How this is done is described in Section 13.1. For messages the set of subscribers present in the publisher’s network at the time of publication is recorded, as well as the number of times a copy of a published message is forwarded and to whom. Together this allows for the calculation of, for every published message, its completeness, precision and the load it places on the network (the total number of times it is forwarded). To ensure that statistics are not skewed in case of mobility, where it is rarely the case that all nodes belong to the same network but in stead are dispersed over a set of subnetworks, messages are only published when a publisher is in a subnetwork that consists of at least half of all the active nodes.

What is *not* measured is delay, e.g., the delay between the moment of publication of a message and the moment it reaches a subscriber. This has not been included in this performance evaluation due to the fact that the delays of the simulation model used do not bear any relationship to real-world delays.

The cost functions for the different parameter settings only have meaning for mobile networks, as the cost of beaconing in a static network becomes zero as soon as the flows of advertisements and subscriptions have stabilized. The goal of the static simulations therefore is to show how well the various systems (i.e., the systems resulting from the various parameter settings) perform in a static environment in terms of precision and message cost, as completeness in such a system is by definition 100%.

All simulations have been performed with the transmission probability (a simulation parameter that models the chance that a transmission succeeds) set to 1, so communication can be considered perfect. This is not a very realistic assumption, so the analysis here must be considered a starting point for further analysis; an analysis which time constraints prevent from giving here. In a similar vain there was no time for static simulations for different network sizes or mobility simulations for differing network densities, so their influences are not considered.

For all simulations, both static and mobile, Ck-HDS parameters *BI, BT, BW, FRW* and *RW* have been held constant at 1.0, 1.05, 0.1, 0.01 and 1.0 respectively. CBR parameters *DP*, *MW*, *CBW* and the maximum hop count for a flow have been held constant for all simulations at 1.0, 0.01, 0.01 and 16 respectively.

A simple flooding protocol has been used as a reference point to compare the obtained results of the CBR system to. In the simple flooding protocol a publisher publishes at a given rate regardless whether it knows any subscribers and broadcasts each published message to its neighbours. Each time a node receives a message that is has never received (or published) before it will also broadcast it. Each published message is thus forwarded once by every node in the network.

13.1 Expressing load

Essential to determining which configuration of parameters performs 'best' is the ability to express the load that the system places on the network. Load is divided into message load and beacon load. The load for a single message is made up by the total number of times it is forwarded, times its size. The load of a single beacon is determined solely by its size, which in turn is determined by the number of beacon identifier tuples, advertisements and subscriptions it holds (see Section 11.6 for more details on the structrue of a beacon). The size of an advertisement is made up by the size of the content description (which is application specific) plus the size of the 2-tuple [previous hop, hop count] (which depends on the identifiers used and the expected size of the network). The size of a subscription is made up by the size of the content description and by the size and number of 3-tuples [source identifier, previous hop, hop count] (again, all are application specific). Within a dumb backbone a subscription will only have a single 3-tuple, otherwise the subscription may have as much of such tuples as there are nodes issuing that type of subscription.

Below two comparable and parameterized cost functions for the load of a beacon and a message are created by means of a number of incremental statements that are based on the above descriptions.

Let u *be the standard unit of size.* (13.1)

Let the size of a beacon identifier (S_{BI}) and the size of the tuple-values 'source identifier' (S_{SI}), 'previous hop' (S_{PH}) and 'hop count' (S_{HC}) be expressed as multiples of u:

$$S_{BI} = j\ u. \tag{13.2}$$

$$S_{SI} = f\ u. \tag{13.3}$$

$$S_{PH} = g\ u. \tag{13.4}$$

$$S_{HC} = h\ u. \tag{13.5}$$

This defines the value of a 2-tuple [previous hop, hop count] as *(f + h)* u and the value of a 3-tuple [source identifier, previous hop, hop count] as (*f* + *g* + *h*) u. Let the average size of a content description of an advertisement (S_{CA}) or of a subscription (S_{CS}), be a multiple of u:

$$S_{CA} = n\ u. \tag{13.6}$$

$$S_{CS} = q\ u. \tag{13.7}$$

Then, defining S_B as the size of a beacon, *d* as the number of beacon identifiers in the beacon, *a* as the number of advertisements in the beacon, *s* as the number of subscriptions in the beacon and *t* as the number of 3-tuples that an average subscription in the beacon has, S_B can be expressed as follows:

$$S_B = d\cdot j+a\cdot(n+(f+h))+s\cdot(q+t\cdot(f+g+h)). \tag{13.8}$$

Define *B* as the number of beacons transmitted per second, *D* as the average number of beacon identifiers in a beacon, *A* as the average number of advertisements in a beacon, *S* as the average number of subscriptions in a beacon and *T* as the average number of 3-tuples for any subscription. As parameters *B*, *D*, *A*, *S* and *T* can all be measured during simulation the measured beacon load per second (L_B) can be expressed as the following function in u:

$$L_B(j, f, g, h, n, q) = B\cdot(D\cdot j+A\cdot(n+(f+h))+S\cdot(q+T\cdot(f+g+ h))). \tag{13.9}$$

Next define *M* as the average number of times a published message is forwarded and *p* as the combined publication rate (expressed in u per second) of all the publishers in the simulation. The message load per second then becomes:

$$L_M(p) = M\cdot p, \tag{13.10}$$

and the total load per second (*L*) becomes:

$$L(j,f,g,h,n,q,p) = B\cdot(D\cdot j+A\cdot(n+f+h)+S\cdot(q+T\cdot(f+g+h)))+M\cdot p. \tag{13.11}$$

To make (13.11) more suitable to use for the comparison of different experimental results the following assumptions are made for the remainder of this chapter:

$$j = 7. \tag{13.12}$$

$$f = 6. \tag{13.13}$$

$$g = h = 1. \tag{13.14}$$

$$n = q = 1024. \tag{13.15}$$

Assuming (but not defining) u to represent 1 byte, the values of f is roughly based on the use of a MAC-address and j on mapping of a MAC-address to an identifier of 1 byte. Previous hop value g is only given 1 unit because it only needs to refer to a neighbour identifiers already included in the neighbour/beacon identifier map. The value for h is actually a bit large (if u is equal to 1B) as it now accomodates networks with a diameter up to 255 hops. Although highly specific to the application at hand, it does not matter which value is assigned to *n* and q when the results of different parameter settings are compared against each other, as they all suffer equally from the amount of overhead. Substituting (13.12), (13.13), (13.14) and (13.15) into (13.16) gives the following cost function which will be used when comparing different parameter settings in Section 13.3:

$$L(p) = B\cdot(7\cdot D+1031\cdot A+S\cdot(1024+8\cdot T))+M\cdot p. \qquad (13.16)$$

13.2 Static simulations

Because the beacon overhead for a static network is zero when the network has stabilized, this section shows how message overhead and precision are influenced by network density and content load. Simulations have been performed for the CBR system and the simple flooding protocol described in the beginning of this chapter.

Two sets of static networks have been generated: for each set 100 networks of 100 nodes each. The Ad Hoc Network Graph Model parameters used give a relative node density of 6.1 (sparse network) and 25.3 (dense network) neighbours per node. The CBR-loads differ between 'light' (1 publisher with 10 subscribers), 'heavy' (4 publishers with 10 subscribers each), 'flooding' (1 publishers, all other nodes are subscribers) and 'draining' (49 publishers, 1 subscriber). Actors (publishers or subscribers) are chosen randomly from the set of nodes, although for every actor per set of publisher/subscribers a distinct node must be chosen. Either the backbone is made up by the entire network (k=0), part of it (k=1) or is as small as possible (k=10). It is furthermore either smart or dumb. All varying CBR parameters are listed in Table 13.1.

Network	
Size	50 nodes
Sparse	Network has been generated with Ad Hoc Network Graph Model parameters (1.0,1.0,0.5).
Dense	Network has been generated with Ad Hoc Network Graph Model parameters (0.5,0.5,0.1).
CBR loads	
Light	1 publisher has 4 subscribers
Heavy	4 publishers that each have 4 subscribers
Flooding	1 publisher has 49 subscribers
Draining	49 publishers, 1 subscriber
Backbone size	
Ck-HDS paramater k ranges over {0,1,10}.	
Backbone behaviour	
The backbone is either smart or dumb.	

Table 13.1. The parameters that are being varied w the set of static simulations.

The testing of the CBR system was done as follows: for each simulation the network is generated and the Ck-HDS structure is constructed by initializing each node at a random moment in the uniformly distributed time range [0,*BI*], and letting the protocol run for 10 seconds to stabilize it. Then the nodes that have been appointed as publishers and subscribers will create their advertisement and subscription flows. When they in turn have stabilized each publisher publishes a single message: for this message overhead and precision is measured (completeness is always 1 in a static network for stable flows). For each combination of the experiment's parameters, the experiment was performed 100 times with differing seeds.

Testing of the simple flooding protocol was done as follows: per simulation the network is generated. Each publisher publishes a single message: for this message overhead and precision is measured (completeness is always 1 in a static network for stable flows). For each combination of the experiment's parameters, the experiment was performed 100 times with differing seeds.

Figure 13.1. shows the resulting average overhead per published message for the different CBR systems (considered different because they have differing parameters) and the simple flooding algorithm. Figure 13.2 shows the resulting average precisions. Looking at the figures one should keep in mind that *k*, the parameter that determines the size of the backbone, does not increase linearly but has values 0, 1 and 10.

The simple flooding protocol sets a baseline: each published message is by definition transmitted 50 times. Except for the case where the content load is 'flooding' (where every node except the publisher is a subscriber) the precision is therefore in all cases almost 0. For the same content load a sparse network always has slightly better precision because a node suffers less from the inefficient forwarding of its neighbours (since it has less neighbours).

It is easy to see that it is never a good idea to construct a dumb backbone consisting of every node in the network (i.e., *k*=0). Messages are transmitted much more often than in any other case due to the inefficient routing of the dumb backbone. Precision is almost as low as with the simple flooding algorithm. As *k* becomes higher however (and the backbone smaller) the dumb backbone outperforms the flooding protocol with respect to the number of message transmissions and the precision in all cases but for the case where the content load is set to 'flooding', where the flooding protocol needs less message transmissions. When *k* is set to 0 or 1 the dumb backbone performs better in a sparse network than in a dense network for the same reason as the simple protocol does. As *k* becomes higher however the effect of the inefficient routing inside the backbone becomes smaller, and another effect starts to dominate: the increased effectivity of directed routing (in this case only done outside the backbone) as the network becomes denser. Directed routing becomes more effective for increased density because it takes less steps to traverse the network.

Not suprisingly a smart backbone outperforms both a dumb backbone and the flooding protocol in all cases for the number of message transmission and the precision. When content load is set to 'flooding' the number of message transmissions differs little with the flooding protocol and the dumb backbone, but the precision is still better. The smart backbone performs better when the network is denser because of the increased efficiency of directed routing explained above. As *k* becomes larger the margin with with a smart backbone outperforms a dumb backbone becomes smaller (both for the message transmissions and the precision) because the effect of the inefficient routing in the dumb backbone becomes less and because the routing over the smart backbone becomes less efficient as the paths to the backbone increase in length. For the case of a 'flooding' content load, or a 'heavy' content load for a dense network, the performance is almost equal.

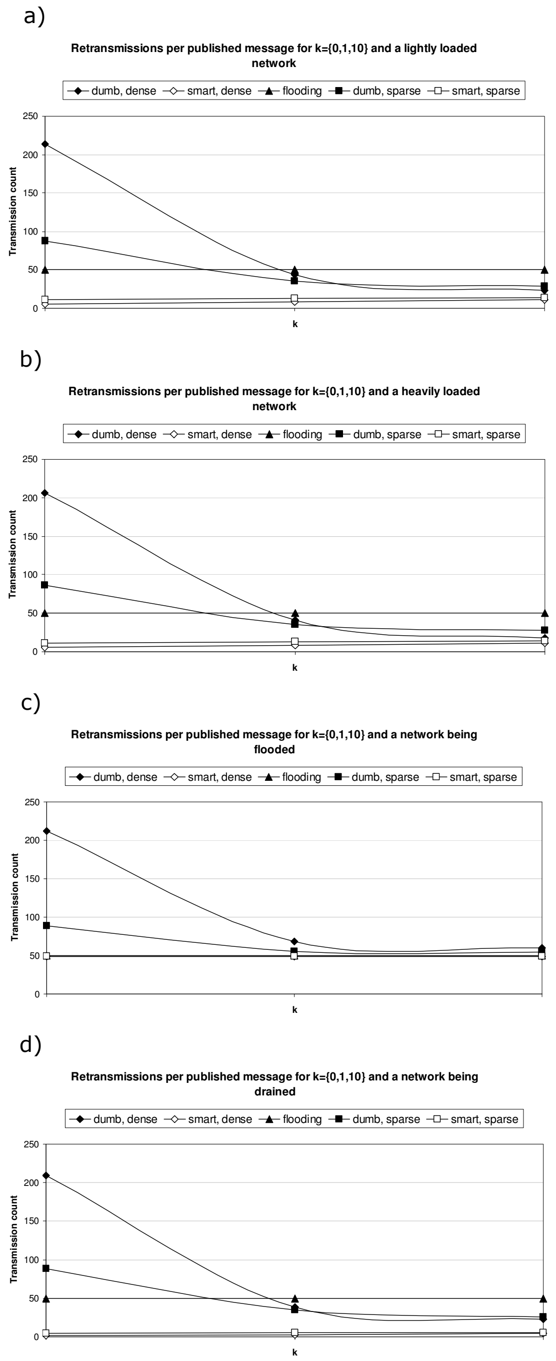


Figure 13.1. The average number of retransmissions for a published message in a network of 50 nodes for differing network conditions, values of *k* and the protocol applied. The graph labeled 'flooding' refers to the flooding algorithm, the other graphs to either the use of a smart or a dumb backbone for the given network density (dense vs. sparse) and value of *k.* The average density of a dense network is 25.3 neighbours per node, for a sparse network 6.1 neighbours per node. The 95% confidence intervals for fig. a) range between 2% and 5%, except for the cases k=10 and a dumb backbone, where the confidence intervals are 15% (≈3.5 hops), 14% (≈2.5 hops), 6% (≈4 hops) and 22% (≈5 hops) when the content load is respectively light, heavy, flooding and draining. The 95% confidence intervals for b) range from 0% to 4%.

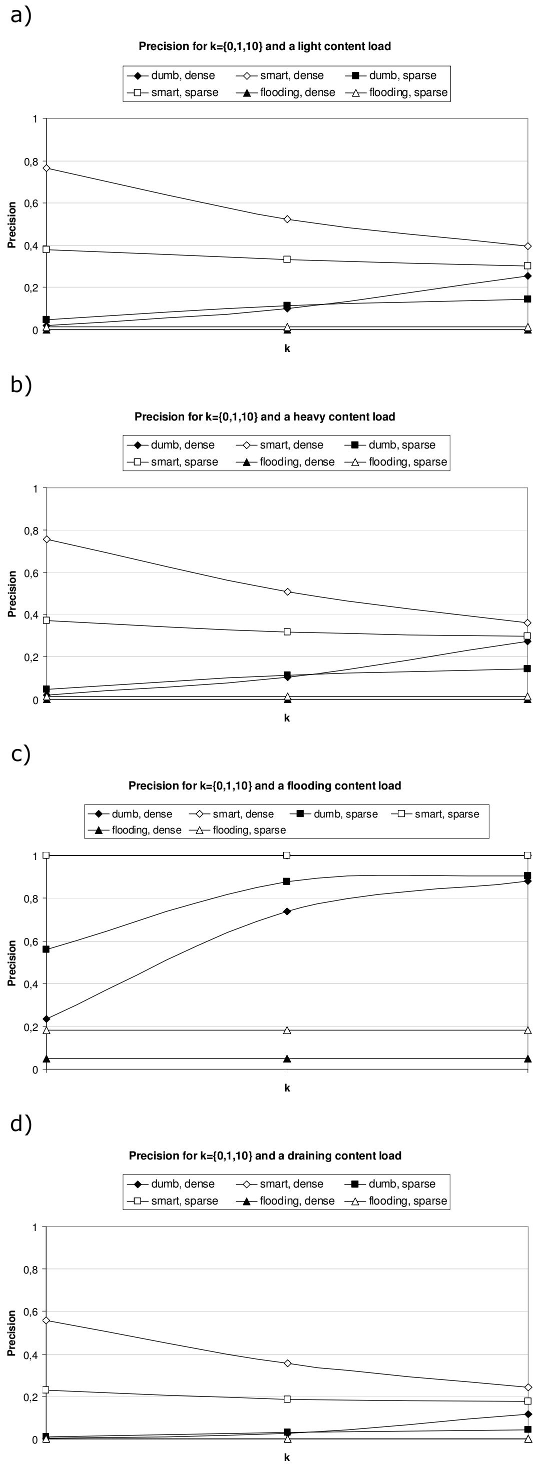


Figure 13.2. The average precision for a published message that is routed over a static network of 50 nodes by means of the CBR system and a simple flooding algorithm. A dense network has an average density of 25.3 neighbours per node, a sparse network 6.1 neighbours per node. All 95% confidence intervals range from 2% to 7%.

13.3 Mobility simulations

The performance (completeness, precision and overhead) of the CBR system is tested for mobile environments. Simulations have been performed for the CBR system and the simple flooding protocol described in the beginning of this chapter.

For each simulation 50 nodes have been spread over a region of 1500x1500 metres using the same uniform distribution. Mobility is managed by the Random Waypoint Model described in Section 2.3. The average node speed has been varied between 5 km/h (1.39 m/s) and 50 km/h, so node speeds vary from 0 to 10 km/h and 0 to 100 km/h. Content load is either 'light' (1 publisher, 10 subscribers) or 'flooding' (1 publisher, 49 subscribers). Parameter *k* has been varied over the range {0,1,10} and the backbone is either smart or dumb. During the simulations the average node density is 5.5 neighbours per node.

For the experiments with the CBR system the network is first initalized for 550 seconds in the case of an average node speed of 5 km/h (to accommodate for the effect that nodes tend to move to the middle of the region) before the C*k*-HDS is constructed: each node is initialized at a random moment in the uniformly distributed time range [550,550+*BI*]. After another 20 seconds (at t=570) the CBR system becomes active: publishers begin to advertise and to publish and subscribers begin to subscribe. After another 30 seconds (t=600) measurements are started, which are performed for 600 seconds. At t=1200 the simulation run is finished. In the case of an average node speed of 50 km/h the network does not need so long to initialize and C*k*-HDS construction is started at t=250, the CBR system becomes active at t=270 and measurements are started at t=300. The simulation then runs for 600 seconds before it is finished at t=900. Publishers publish with a rate of 1 message per seconds if they know any matching subscription path.

For the experiments with the simple flooding protocol the network is first initialized for 600 seconds (in the case of an average node speed of 5 km/h) or 300 seconds (in the case of 50 km/h). The publisher then starts to publish messages with a rate of 1 message per second. This is done for 600 seconds, after which the simulation is finished. As the publishers start publishing the measurements are also started.

Measurements are made during the CBR simulations of the parameters needed for the cost function (13.16). All values are averaged over each entire run: the number of beacons per second, the number of beacon identifiers per beacon, the number of advertisements per beacon, the number of subscriptions per beacon and the number of subscription tuples per subscription. For the simulations of the CBR system and of the simple flooding protocol the completeness and the precision are measured of each published message as described in the beginning of this chapter.

Table 13.1. shows the completeness and precision for all simulations that have been performed. All results have been listed in Table A.1. The resulting cost functions (their parameters are also included in Table A.1) are shown in Figure 13.3 as a function of the publication rate (in units per second). These cost functions have been created by substituting the results of the simulations into (13.16). Only systems that have a completeness of 80% or higher are considered: systems that do not reach this mark are not considered a valid solution for the given combination of mobility & content load. The graphs shows for an increasing publication rate which system gives the least total overhead, where the total overhead consists of the message overhead and the beacon overhead. The value for each graph when the publication rate is 0 is the beacon overhead. This is by definition 0 for the flooding protocol. Note that the range for which the cost functions have been drawn differs per figure.

For a given mobility and a given content load, the simple flooding protocol has in all cases has a better completeness than the CBR system has for any value of *k*. This is not very surprising: the completeness of any routing protocol that attempts directed routing will suffer under mobility. Since the CBR system does not contain any additional mechnisms to increase completeness (such as buffering messages) the completeness of the CBR system decreases as mobility increases. The directed routing has effect however: the CBR system's precision is for every case orders or magnitude larger than the flooding protocol's precision.

As can be seen in Figure 13.3 the flooding protocol always starts with the lowest overhead since it has no beacon overhead. This effect is stronger when there are more subscribers in the network. As the publication rate increases however the message overhead becomes the dominant factor and the CBR system is almost always more efficient (this is only *not* the case when mobility is high, the backbone is dumb and *k* is set to 0).

Mobility	Load	k	Backbone	Completeness	Precision
LOW	light	0	dumb	**0.99** (±0.00)	0.20 (±0.01)
	light	0	smart	**0.96** (±0.00)	0.52 (±0.01)
	light	1	dumb	**0.99** (±0.00)	0.28 (±0.01)
	light	1	smart	**0.92** (±0.01)	0.45 (±0.01)
	light	10	dumb	**0.96** (±0.00)	0.32 (±0.01)
	light	10	smart	**0.88** (±0.01)	0.42 (±0.01)
	light	flooding		**1.00** (±0.00)	0.03 (±0.00)
	flooding	0	dumb	**0.99** (±0.01)	0.57 (±0.01)
	flooding	0	smart	N/A	N/A
	flooding	1	dumb	**0.99** (±0.01)	0.90 (±0.00)
	flooding	1	smart	**0.82** (±0.01)	1.00 (±0.00)
	flooding	10	dumb	**0.98** (±0.01)	0.93 (±0.01)
	flooding	10	smart	0.73 (±0.01)	0.99 (±0.00)
	flooding	flooding		**1.00** (±1.67)	0.17
HIGH	light	0	dumb	**0.98** (±0.01)	0.11 (±0.00)
	light	0	smart	**0.81** (±0.01)	0.44 (±0.01)
	light	1	dumb	**0.90** (±0.00)	0.27 (±0.00)
	light	1	smart	0.70 (±0.01)	0.40 (±0.00)
	light	10	dumb	**0.84** (±0.01)	0.29 (±0.00)
	light	10	smart	0.65 (±0.01)	0.38 (±0.00)
	light	flooding		**1.00** (±0.00)	0.03 (±0.00)
	flooding	0	dumb	**0.98** (±0.00)	0.56 (±0.00)
	flooding	0	smart	N/A	N/A
	flooding	1	dumb	**0.91 (**±0.00)	0.89 (0.00)
	flooding	1	smart	0.60 (±0.00)	0.99 (0.00)
	flooding	10	dumb	**0.86** (±0.00)	0.90 (±0.00)
	flooding	10	smart	0.57 (±0.00)	0.99 (±0.00)
	flooding	flooding		**1.00** (±0.00)	0.17 (±0.00)

Table 13.1. The completeness and precision of all performed simulations. All results are listed in Table A.1. 95% Confidence intervals are given in parentheses. The completeness results that are in bold are all above 80% and have been included in the Figure 13.3.

For all cases the dumb backbone has a higher completeness, but also a lower precision, than the smart backbone. This difference can also fully be contributed to the fact that the smart backbone tries to perform more directed routing. This difference grows larger as mobility increases.

Completeness of the smart backbone is negatively influenced by mobility, an increasing *k and* the number of subscribers in the network. The beacon overhead is significantly larger for k=0 than when *k* is 1 or 10, because it is harder to keep all subscription paths valid in a backbone that increases in size. When the backbone is made up by the whole network *and* the content load is 'flooding' a broadcasting storm of beacons is the result, congesting the entire network. The two cases in which this happened here have not been included because the simulations took too long. Clearly using a smart backbone is in those cases *not* a good idea. For the average node speed of 50 km/h the completeness of the smart backbone drops to 60 and 70 percent, except for the 'lightly' loaded network when *k*=0. This case does however also suffer from a high beacon rate.

For low mobility the dumb backbone suffers less from a drop in completeness as *k* increases, although it still does. The completeness of the dumb backbone is also less susceptible to mobility than the smart backbone. The beacon rate does however increase substantially when mobility increases: with high mobility and *k*=0 the beacon overhead is even larger than when a smart backbone is used (the dumb backbone has a significantly higher completeness however).

Apart for the case of high mobility and load set to 'light' Figure 13.3 clearly shows how the smart backbone almost always starts with a higher overhead than the dumb backbone, because it has to make more effort to maintain its routing paths. As the publication rate increases the smart backbone gains on its dumb sibling however because it delivers content with fewer message retransmissions. As was already pointed out above however the completeness of the smart backbone can drop substantially as for instance mobility increases, and when mobility is high and load is set to 'flooding' no value of *k* gives a completeness for the smart backbone that reach 80%.

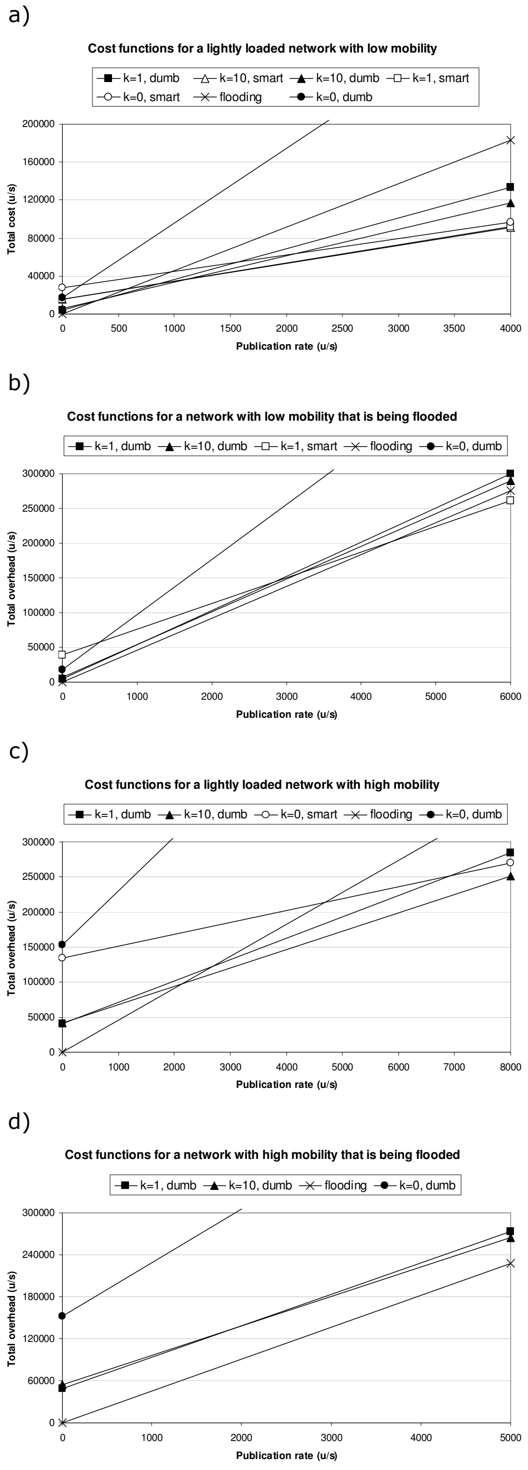


Figure 13.3. The resulting overhead cost functions of the mobility experiments. The cost functions have been created by substituting the results of Table 13.2 into (13.16). Only the cases where completeness was 80% or higher are included however: other cases are considered invalid soltuions to the problem at hand (i.e., they are not able to cope well enough with mobility, content load, etc.). The 95% confidence intervals all lie within 5% of the average except for the cases ‘k=0, dumb’ in c) en d), where the intervals are at 15%.

Chapter 14
Conclusions and future work on the content-based routing system

This part of the thesis present a design, specification and analysis of a CBR system that is capable of routing content in a directed wat over a backbone in a MANET. The system has intentionally been designed to test how well content can be delivered by forwarding messages over routing paths that go to, from and over the backbone. Such a design comes with a number of weaknesses, such as the relative ease with which such routing paths can (temporarily) be disconnected. These weaknesses can again be covered using other techniques (local flooding of messages for instance). This has however not been done here. With time constraints being tight designing a system that covers every possible weakness would be impossible. Therefore the design has been left intentionally 'pure', with only the created routing paths as a means for messages to be forwarded over. This makes analysis of the design all the more easy, as there are no additional influences to be tested. For future work on CBR in MANETs this design can act as a basis, to be improved by additional techniques that do cover its weaknesses.

Analysis was done by means of simulation and was also somewhat hampered by time constraints. No real conclusions can therefore be given on the influence of transmission errors, the size of a network or the density of the network in case of mobility, although some thoughts exist. Initial runs of the system with transmission errors introduced showed the beacon rate to increase, as it requires more effort to update the routing paths when beacons become lost. Simulations performed with static networks of different densities show how a smart backbone becomes more efficient as the density goes up. It is unknown however how the beacon rate will be influenced by density in case of mobility. Although unproven it is expected that as the size of the network increases the beacon rate also increases, because changes in the backbone in one part of the network will result in changes in the backbone in another part of the (same) network. The system is thus not expected to be scalable with respect to the size of the network.

Based on the simulation results that were obtained the design questions that were posed in Chapter 10 are reposted below and each is answered in turn.

> *Is the system, using either a smart or a dumb backbone, still able to deliver messages as network mobility and the number of actors increase?* (10.1)

First and foremost: in the case of a high number of actors one should *not* employ a smart backbone, and unless mobility is extremely low and the number of actors is limited one should – based on the current results – always opt for the use of a dumb backbone. The simulations results described in Section 13.3 show how the smart backbone is not able to keep routing layers intact as mobility goes up, even for a small number of actors. When the number of actors increases the smart backbone is in danger of creating a broadcasting storm (of beacons), congesting the network. The dumb backbone however seems capable of delivering content even at higher node speeds and when the number of actors increases: even at average node speeds of 50 km/h completeness lies around 90%.

How is overhead influenced by the size and behaviour of the backbone, for given network conditions and the number of actors? (10.2)

Another reason for *not* using a smart backbone for a mobile network is the fact the the smart backbone suffers from a high beacon overhead as it tries to keep its routing paths intact. This overhead decreases as the backbone decreases in size. The dumb backbone has substantially less beacon overhead, except when there are a lot of actors and the backbone covers the entire network (*k is* set to 0): in that case the system suffers from a high beacon rate.

How is precision influenced by the size and behaviour of the backbone, for given network conditions and the number of actors? (10.3)

Precision is always better when the backbone is smart. For a smart backbone precision decreases as the backbone increases in size. For a dumb backbone precision *in*creases as the backbone decreases in size. Precision also increases as the number of actors increases.

Compared to a simple flooding protocol the system performs better when mobility is low. The system is always more precise than the flooding protocol, especially when the backbone is smart. As mobility increases and one should, as stated above, use a dumb backbone, the precision is still orders of magnitude larger than with the flooding protocol. The flooding protocol does however outperform the system for completeness, which was around 100% for all simulations. The overhead of the simple flooding protocol is less when the publication rate is very low, as flooding then has little impact on the network. As the publication rate increases however the system’s increased routing gives it a better efficiency.

Based on the conclusions above the system using a dumb backbone (that does not consist of the entire network) can be considered a valid starting point to base any future work on for routing content over MANETs. A good point to focus this work on would then be the decreasing of the beacon rate, to make the system more scalable. The prefered size of the backbone mainly depends on the mobility of the network and whether completeness or precision is deemed more important: as mobility increases a larger backbone gives more completeness but also a higher overhead.

Although a smart backbone seems invalid in all cases, it may still be a good solution for networks in which mobility is very low and routing paths remain stable, such as sensor networks. The increased complexity of the smart backbone may however prevent this, as nodes in such networks often have very few resource.

PART IV

Appendices

Appendix A
Results of the content-based routing mobility simulations

Average node speed (km/h)	**Load**	**k**	**Backbone**	**Completeness**	**Precision**	**D**	**B**	**A**	**S**	**T**	**M**
5	light	0	dumb	**0.99** (±0.00)	0.20 (±0.01)	6.37 (±0.11)	8.03 (±1.17)	0.99 (±0.00)	0.99 (±0.00)	0.99 (±0.00)	78.74 (±1.17)
5	light	0	smart	**0.96** (±0.00)	0.52 (±0.01)	6.43 (±0.12)	12.64 (±0.31)	0.99 (±0.00)	0.99 (±0.00)	9.48 (±0.07)	17.13 (±0.26)
5	light	1	dumb	**0.99** (±0.00)	0.28 (±0.01)	3.82 (±0.00)	2.31 (±0.11)	0.84 (±0.01)	0.87 (±0.01)	0.99 (±0.00)	32.21 (±0.33)
5	light	1	smart	**0.92** (±0.01)	0.45 (±0.01)	3.82 (±0.05)	7.55 (±0.22)	0.94 (±0.01)	0.95 (±0.01)	8.15 (±0.14)	19.05 (±0.37)
5	light	10	dumb	**0.96** (±0.00)	0.32 (±0.01)	3.44 (±0.05)	3.29 (±0.09)	0.73 (±0.01)	0.83 (0.01)	1.06 (±0.01)	27.98 (±0.44)
5	light	10	smart	**0.88** (±0.01)	0.42 (±0.01)	3.44 (±0.05)	7.83 (±0.20)	0.90 (±0.01)	0.93 (±0.01)	6.15 (±0.15)	18.95 (±0.32)
5	light	flooding		**1.00** (±0.00)	0.03 (±0.00)	N/A	N/A	N/A	N/A	N/A	45.93 (±0.33)
5	flooding	0	dumb	**0.99** (±0.01)	0.57 (±0.01)	6.37 (±0.11)	8.45 (±1.24)	0.99 (±0.00)	0.99 (±0.00)	0.99 (±0.00)	79.21 (0.96)
5	flooding	0	smart	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
5	flooding	1	dumb	**0.99** (±0.01)	0.90 (±0.00)	3.82 (±0.05)	2.58 (±0.22)	0.82 (0.00)	0.97 (0.01)	0.99 (±0.00)	49.16 (±0.37)
5	flooding	1	smart	**0.82** (±0.01)	1	3.78 (±0.06)	16.43 (±0.50)	0.96 (±0.00)	0.99 (±0.00)	25.92 (±0.35)	37.00 (±0.35)
5	flooding	10	dumb	**0.98** (±0.01)	0.93 (±0.01)	3.43 (±0.05)	3.72 (±0.09)	0.67 (±0.00)	0.94 (±0.00)	0.96 (±0.01)	47.3 (±0.32)
5	flooding	10	smart	0.73 (±0.01)	0.99 (±0.00)	3.44 (±0.05)	13.49 (±0.30)	0.94 (±0.00)	0.99 (±0.00)	17.36 (±0.37)	32.94 (±0.43)
5	flooding	flooding		**1.00** (±1.67)	0.17	N/A	N/A	N/A	N/A	N/A	45.90 (±0.24)
50	light	0	dumb	**0.98** (±0.01)	0.11 (±0.00)	7.29 (±0.09)	72.43 (±2.05)	1.00 (±0.00)	1.00 (±0.00)	1.00 (±0.00)	77.13 (±0.66)
50	light	0	smart	**0.81** (±0.01)	0.44 (±0.01)	6.41 (±0.09)	60.83 (±0.49)	0.99 (±0.00)	0.99 (±0.00)	9.81 (±0.01)	16.86 (±0.11)
50	light	1	dumb	**0.90** (±0.00)	0.27 (±0.00)	3.81 (±0.04)	22.67 (±0.34)	0.85 (±0.00)	0.88 (±0.00)	0.99 (±0.00)	30.43 (±0.26)
50	light	1	smart	0.70 (±0.01)	0.40 (±0.00)	3.84 (0.04)	34.74 (±0.28)	0.91 (±0.00)	0.93 (±0.00)	8.42 (±0.04)	15.90 (±0.13)
50	light	10	dumb	**0.84** (±0.01)	0.29 (±0.00)	3.43 (0.04)	24.76 (±0.51)	0.79 (±0.00)	0.84 (±0.00)	1.00 (±0.00)	26.15 (±0.36)
50	light	10	smart	0.65 (±0.01)	0.38 (±0.00)	3.57 (±0.03)	31.80 (±0.26)	0.87 (±0.00)	0.89 (±0.00)	7.08 (±0.07)	15.51 (±0.14)

50	light	flooding		**1.00** (±0.00)	0.03 (±0.00)	N/A	N/A	N/A	N/A	N/A	45.70 (±0.11)
50	flooding	0	dumb	**0.98** (±0.00)	0.56 (±0.00)	7.35 (±0.10)	72.16 (±2.38)	1.00 (±0.00)	1.00 (±0.00)	1.00 (±0.00)	76.56 (±0.70)
50	flooding	0	smart	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
50	flooding	1	dumb	**0.91** (±0.00)	0.89 (0.00)	3.82 (±0.06)	25.77 (±0.57)	0.84 (±0.00)	0.97 (±0.00)	0.99 (±0.00)	44.86 (±0.33)
50	flooding	1	smart	0.60 (±0.00)	0.99 (0.00)	3.75 (±0.03)	39.09 (±0.26)	0.91 (±0.00)	0.98 (±0.00)	33.20 (±0.31)	26.60 (±0.15)
50	flooding	10	dumb	**0.86** (±0.00)	0.90 (±0.00)	3.27 (±0.05)	30.03 (±0.48)	0.77 (±0.00)	0.96 (±0.00)	0.98 (±0.00)	42.11 (±0.41)
50	flooding	10	smart	0.57 (±0.00)	0.99 (±0.00)	3.45 (±0.04)	36.30 (±0.32)	0.86 (±0.00)	0.98 (±0.00)	23.23 (±0.38)	25.14 (0.21)
50	flooding	flooding		**1.00** (±0.00)	0.17 (±0.00)	N/A	N/A	N/A	N/A	N/A	45.65 (±0.10)

Table A.1. Results of the mobility experiment. The letters *D, B, A, S, T, M* refer to the parameters in (13.16). 95% Confidence intervals are given in parentheses. The completeness results that are in bold are all above 80% and have been included in the Figure 13.3.

Glossary

Abbreviations are listed in the order as they are encountered in the text.

MANET	mobile ad-hoc networks
CBR	content-based routing
DS	dominating set
k-HDS	*k*-Hop dominating set
CDS	connected dominating set
C*k*-HDS	connected *k*-hop dominating set
N(x)	the neighbour set of a node x
D(x)	the dominator set of a node x
P(x)	the parent set for a node x
BI	beacon interval
IBS	inter beacon space
BT	beacon timeout
FRW	fast response window
BW	beacon window
RW	resume window
CBI	CBR beacon interval
CBW	CBR beacon window
DP	dampening period
DDI	dumb dominator identifier
MW	message window

Bibliography

[1] G. Lin, G. Noubir, R. Rajamaran. *Mobility Models for Ad hoc Network Simulation*. In Proceedings of the IEEE INFOCOM 2004, April 2004.

[2] J. H. Schiller. *Mobile Communications*. 2nd edition, Addison-Wesley, 2004, ISBN 0-321-12381-6.

[3] S. Corson, J. Macker. *Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations*. RFC 2501, IETF, 1999.

[4] P.T.H Goering, G.J. Heijenk. *Service discovery using Bloom filters*. In Proc. of the twelfth annual conference of the Advanced School for Computing and Imaging, June 14-16, 2006, Lommel, Belgium. pp. 219-227.

[5] W. Klein Wolterink. *A Study on Bloom Filters and Content-Based Routing in Ad-Hoc Networks*. July 2007, University of Twente.

[6] X Liu, Q Huang, Y Zhang. *Combs, needles, haystacks: balancing push and pull for discovery in large-scale sensor networks*. In Proceedings of the 2nd international conference on Embedded networked sensor systems, 2004, pp. Baltimore, MD, USA, pp. 122-133.

[7] H.-Y. Yang, C.-H. Lin, M.-J. Tsai. *Distributed Algorithm for Efficient Construction and Maintenance of Connected k-Hop Dominating Sets in Mobile Ad Hoc Networks*. In IEEE Transactions on Mobile Computing, vol. 7, no. 4, April 2008, pp. 444-457.

[8] S. Kurkowski, T. Camp, M. Colagrosso. *MANET Simulation Studies: The Incredibles*. In Mobile Computing and Communications Review, vol. 9, no. 4.

[9] T.R. Andel, A. Yasinsac. *On the Credibility of Manet Simulations*. Computer, vol. 39, no. 7, pp. 48-54, July, 2006.

[10] D. Kotz et al.. *Experimental Evaluation of Wireless Simulation Assumptions*. In Proc. of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems, Venice, Italy, 2004, pp. 78-82.

[11] D. Cavin, Y. Sasson, and A. Schiper. *On the Accuracy of Manet Simulators*. In Proc. of the second ACM International Workshop Principles of Mobile Computing, ACM Press, 2002, pp. 38-43.

[12] S.K.S. Gupta, P.K. Srimani. *Adaptive Core Selection and Migration Method for Multicast Routing in Mobile Ad Hoc Networks*. In IEEE Transactions on Parallel and Distributed Systems, vol. 14, no. 1, January 2003.

[13] T. Eugster, A. Felber, R. Guerraoui, A-M. Kermarrec. *The Many Faces of Publish/Subscribe*. In ACM Computing Surveys, vol. 35, no. 2, pp. 114-131, June 2003.

[14] E. Yoneki, J. Bacon. *Content-Based Routing with On-Demand Multicast*. In Proceedings of the 24th International Conference on Distributed Computing Systems Workshops (ICDCSW), 2004.

[15] G. Banavar et al.. *An efficient multicast protocol for content-based publish-subscribe systems*. In Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, pp. 262-272, 1999.

[16] L. Opyrchal et al.. *Exploiting IP Multicast in Content-Based Publish-Subscribe Systems*. In IFIP/ACM International Conference on Distributed Systems Platforms, pp. 185-207, April 2000.

[17] A. Carzaniga, A. L. Wolf. *Content-Based Networking: A New Communication Infrastructure*. In NSF Workshop on an Infrastructure for Mobile and Wireless Systems, October 2001.

[18] R. Baldoni, A. Virgillito. *Distributed Event Routing in Publish/Subscribe Communication Systems: a Survey* (revised version). Technical report, MIDLAB 1/2006, Dipartimento di Informatica e Sistemistica "A.Ruberti", Università di Roma la Sapienza, 2006.

[19] K.M. Alzoubi, P.J. Wan, O. Frieder. *Distributed Heuristics for Connected Dominating Sets in Wireless Ad Hoc Networks*. IEEE ComSoc/KICS Journal on Communications and Networks, vol. 4, no. 1, pp. 22-29, 2002.

[20] K.M. Alzoubi, P.J. Wan, and O. Frieder. *Message-Optimal Connected Dominating Sets in Mobile Ad Hoc Networks*. In Proc. of ACM MOBIHOC, pp. 157-164, 2002.

[21] L. Bao and J. Garcia-Luns-Aceves. *Topology Management in Ad Hoc Networks*. In Proc.of ACM MOBIHOC, pp. 129-140, 2003.

[22] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris. *Span: An Energy-Efficient Coordination Algorithm for Topology Maintenance in Ad Hoc Wireless Networks*. ACM Wireless Networks J., vol. 8, no. 5, pp. 481-494, 2002.

[23] F. Dai and J. Wu. *An Extended Localized Algorithm for Connected Dominating Set Formation in Ad Hoc Wireless Networks*. In IEEE Transactions on Parallel and Distributed Systems, vol. 53, no. 10, pp. 908-920, 2004.

[24] J. Wu and H. Li. *A Dominating-Set-Based Routing Scheme in Ad Hoc Wireless Networks*. Telecommunication Systems, vol. 18, no. 1-3, pp. 13-36, 2001.

[25] S. Kutten and D. Peleg. *Fast Distributed Construction of k-Dominating Sets and Applications*. In Journal of Algorithms, vol. 28, pp. 40-66, 1998.

[26] D. Penso and C. Barbosa. *A Distributed Algorithm to Find k-Dominating Sets*. Discrete Applied Mathematics, vol. 141, pp. 243-253, 2004.

[27] S. Yang, J. Wu, and J. Cao. *Connected k-Hop Clustering in Ad Hoc Networks*. In Proc. of ICPP, pp. 373-380, 2005.

[28] Opnet modeler software, http://www.opnet.com/products/modeler.