Implementing Opportunistic Routing Algorithms in OMNeT++ 4.1

Bachelorarbeit
der Philosophisch-naturwissenschaftlichen Fakultät der
Universität Bern

vorgelegt von

Björn Mosler
Februar 2012

Leiter der Arbeit:
Professor Dr. Torsten Braun
Institut für Informatik und angewandte Mathematik

Contents

Abstract

The primary challenge in building a MANET is equipping each device to continuously maintain the information required to properly route traffic. MANET routing protocols are traditionally using a fixed route and a pre-defined next-hop. Opportunistic routing offers a new approach by neither enforcing a fixed route nor a particular next hop. Instead, the routing decision is delegated to the receivers.

In this thesis we implement two opportunistic routing protocols MORE and ExOR. MORE randomly mixes packets before forwarding them. This randomness ensures that routers that hear the same transmission do not forward the same packets. Thus, MORE needs no special scheduler to coordinate routers and can run directly on top of 802.11.

In ExOR the source node includes a list of candidate forwarders in each packet, prioritized by closeness to the destination. The highest priority forwarder then broadcasts the packets in its buffer and remaining forwarders then transmit in order, sending only packets which were not acknowledged by higher priority nodes. The forwarders continue to cycle through the priority list until the destination has received enough packets.

We then evaluate MORE, ExOR and OLSR and compare their performance based on throughput, round-trip and transmission delay. Finally we discuss the results and try to give an explanation as to why the results are not as expected.

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Background

1.1.1 Mobile Ad Hoc Networks

A mobile ad-hoc network (MANET) is a self-configuring infrastructure-less network of mobile devices connected by wireless links. The moving devices have the possibility to connect over a wireless medium and form an arbitrary and dynamic network with wireless links. Each device in a MANET is free to move independently in any direction, and will therefore change its links to other devices frequently. This means that links between the nodes can change during time, new nodes can join/leave the network. A MANET is expected to be of larger size than the radio range of the wireless antennas, because of this fact it could be necessary to route the traffic through a multi-hop path to give two nodes the ability to communicate. The primary challenge in building a MANET is equipping each device to continuously maintain the information required to properly route traffic. Such networks may operate by themselves or may be connected to the larger Internet[1].

1.1.2 MANET routing protocols

Routing is one of the most important issues for MANETs. Routing protocols for MANETs can be divided into proactive and reactive routing protocols.

Proactive routing

This type of protocols maintains fresh lists of destinations and their routes by periodically distributing routing tables throughout the network. The main disadvantages of such algorithms are high overhead because of constant route maintenance and slow reaction on restructuring and failures. On the other hand there is little delay when sending a packet.

Reactive routing

This type of protocols finds a route on demand by flooding the network with Route Request packets. The main disadvantages of such algorithms are a high latency time in route finding and possible network clogging due to excessive flooding. On the other hand there is no constant maintenance traffic.

[1] http://en.wikipedia.org/wiki/Mobile_ad_hoc_network

1.1.3 Opportunistic routing

Opportunistic routing protocols are a rather recently devised class of routing protocols for wireless multi-hop networks. What separates opportunistic routing protocols from fixed-route routing protocols is the fact that a packet's route is not pre-determined by its source before the packet transmission. Because of its broadcast nature, a wireless transmission's recipients are not necessarily known at the moment of transmission. This is due to the ever-changing physical conditions and in some cases also changes in the network's topology. Enforcing a pre-determined, possibly optimal route might produce the following outcomes:

(a) The transmission succeeds and the chosen route is still the optimal route.

(b) The transmission succeeds but some of the conditions have changed such that there is now a shorter route or that the transmission reached further than expected. Both result in unnecessary hops in order to reach the destination.

(c) The network topology or the physical conditions have changed in such a way that the chosen route is rendered invalid, necessitating a re-transmission.

Since power consumption is usually a concern in mobile and sensors networks, cases (b) and (c) are to be avoided. Opportunistic routing protocols leave it to a transmission's recipients to decide on who will be the forwarder of the received packets. This scheme has two advantages:

(a) The decision on who is going to be the next hop is made using the latest possible information.

(b) The sender does not have to make — possibly faulty — assumptions on radio range and other physical conditions, thereby maximizing the progress made towards the destination.

But there are also some disadvantages. For instance the protocols have to make sure that no two nodes might forward the same packet at the same time. As of now there have been two ways of dealing with that:

(a) Strict coordination of medium access: At any given time, only one node in the entire network may transmit. The first node to transmit would ideally be the one that is closest to the destination. The other nodes listen to what is transmitted, so that they can discard already sent packets. This is roughly the approach taken by ExOR.

(b) Mixing packets via network coding: Upon receiving a new packet a node "mixes"[2] it with already received packets and then transmits the mixed packet. Depending on the mixing procedure, duplicate transmissions by independent nodes are unlikely. This is roughly the approach taken by MORE.

MORE

MORE [3] is a MAC-independent opportunistic routing protocol. MORE randomly mixes packets before forwarding them. This randomness ensures that routers that hear the same transmission do not forward the same packets. Thus, MORE needs no special scheduler to coordinate routers and can run directly on top of 802.11.

[2] Using XOR, linear combinations etc.

ExOR

ExOR [2] is an integrated routing and MAC protocol for bulk transfers in multi-hop wireless networks. ExOR exploits the broadcast nature of radios by making forwarding decisions based on which nodes receive each transmission. The spatial diversity among receivers provides each transmission multiple opportunities to make progress in the face of packet losses. As a result ExOR can use long links with high loss rates, which would be avoided by unicast routing.

ExOR operates on batches of packets. The source node includes a list of candidate forwarders in each packet, prioritized by closeness to the destination. Receiving nodes buffer successfully received packets and await the end of the batch. The highest priority forwarder then broadcasts the packets in its buffer, including its copy of the "batch map" in each packet. The batch map contains the sender's best guess of the highest priority node to have received each packet. The remaining forwarders then transmit in order, sending only packets which were not acknowledged in the batch maps of higher priority nodes. The forwarders continue to cycle through the priority list until the destination has enough packets to recover the original data using forward error correction.

1.2 Task/Problem Formulation

The main goal of this thesis is to implement the MORE and ExOR protocol in OMNeT++ using the OR framework [4]. In a next step we compare them with OLSR[3].

1.3 Motivation

There already are simulator-based implementations of opportunistic routings protocols, in particular for ONE. [5] Unfortunately ONE employs an idealized and static model of the physical layer. This makes it unsuitable for meaningful evaluations in wireless networks, where physical conditions are subject to constant change. [4, p. 1]

OMNeT++ [6] combination with the INETMANET framework (see section 2.2) take into account the dynamic conditions in wireless networks and are therefore better suited for evaluating opportunistic routing protocols.

1.4 Contribution

The contributions of this thesis are some extensions to the OR framework [4] (see section 4.1), and implementations of MORE and ExOR in OMNeT++ 4.1.

1.5 Outline

In chapter 2, we look at projects that are related to or used in this thesis. Chapter 3 delivers more conceptual details about MORE and ExOR, while chapter 4 expands on the details regarding the two protocol's implementation. Chapter 5 is about evaluating MORE's and ExOR's performance and how they compare to OLSR. Finally a conclusion is reached in chapter 6.

[3] `http://tools.ietf.org/html/rfc3626`

Chapter 2

Related Work

2.1 OMNet++ Simulator

OMNeT++ is an extensible, modular, component-based C++ simulation library and framework, primarily for building network simulators. "Network" is meant in a broader sense that includes wired and wireless communication networks, on-chip networks, queueing networks, and so on. Domain-specific functionality such as support for sensor networks, wireless ad-hoc networks, Internet protocols, performance modeling, photonic networks, etc, is provided by model frameworks, developed as independent projects. OMNeT++ offers an Eclipse-based IDE, a graphical runtime environment, and a host of other tools. There are extensions for real-time simulation, network emulation, alternative programming languages (Java, C#), database integration, SystemC integration, and several other functions. [1]

2.2 INETMANET

INETMANET is based on INET Framework[2] and continuously being developed. Generally it provides the same functionality as the INET Framework, but contains additional protocols and components that are especially useful while modeling wireless communication like propagation models, link layer and routing protocols and mobility models.[3]

2.3 Opponet

The Opponet framework offers mechanisms for simulating opportunistic and delay-tolerant networks in the OMNeT++ discrete event simulator. The mechanisms allow for simulating open systems of wireless mobile nodes where mobility or contact traces are used to drive the simulations. This way mobility generation is separated from the core OMNeT++ protocol simulations which facilitates importing synthetic or real data from external mobility generators, real mobility tracking data or real contact traces. [7]

[1] Verbatim from `http://omnetpp.org/`, 10.08.2011

[2] `http://inet.omnetpp.org/`

[3] INETMANET README on `https://github.com/inetmanet/inetmanet/blob/master/README.INETMANET`, 18.01.2012

2.4 OR framework

A framework [4], which is based on the Opponet project (see section 2.3), has been developed by the CDS research group at the University of Bern for simulating opportunistic routing protocols in OMNeT++. It provides common functionality like ETX computation, neighbour management etc as well as an abstract protocol base class which allows for easy implementation of further opportunistic routing protocols.

The class hierarchy is built on the assumption that all opportunistic routing protocols share functionality and that the way they work can roughly be split up into four steps. In order to implement a new protocol, one needs to subclass the abstract base class and implement the four virtual functions, which correspond to the four common steps in opportunistic routing protocols. [4] Those four steps are:

Forwarder Candidates Selection Build a set of potential forwarders.

Forwarder Selection Determine the forwarder.

Forwarder Role Change Notification Notify other nodes about who will be forwarding the packet.

Collision Avoidance Make sure that no two nodes transmit at the same time.

See Figure 3.3 and Figure 3.1 for how ExOR and MORE map onto those four steps.

2.5 ETX

The expected transmission count metric (ETX) [8] of a link is calculated using the forward and reverse delivery ratios (FDR and RDR resp.) of the link. The forward delivery ratio, d_f, is the measured probability that a data packet successfully arrives at the recipient. The reverse delivery ratio, d_r, is the probability that the ACK packet is successfully received. These delivery ratios can be measured by sending ping probes. The expected probability that a transmission is successfully received and acknowledged is $d_f \cdot d_r$. A sender will retransmit a packet that is not successfully acknowledged. Because each attempt to transmit a packet can be considered a Bernoulli trial, the expected number of transmissions is:

$$\text{ETX} = \frac{1}{d_f \cdot d_r}$$

Chapter 3

Opportunistic routing protocols

3.1 MORE

3.1.1 Terminology

Native packet	Original packet, before encoding
Coded packet	Packet created from a random linear combination of native packets
Code vector of a coded packet	Vector containing the coefficients used in the linear combination
Innovative packet	Packet whose code vector is linearly independent from a set of other code vectors
Closer to destination	A node that has a lower ETX to the destination
Upstream	Further from the destination than the current node
Downstream	Closer to the destination than the current node

Table 3.1: MORE terminology

3.1.2 Node roles

Source

The source creates coded packets from native packets and broadcasts them until it receives an acknowledgement (ACK) from the destination. It compiles a forwarder list which is attached to every packet.

Forwarder

Forwarders store innovative packets and forward a packet if the traffic control heuristic (see subsection 3.1.5) allows them. If a forwarder overhears an ACK, it deletes the current batch and waits for the next batch.

Destination

The destination also only stores innovative packets. As soon as it has received enough packets for decoding, it sends an ACK towards the batch source using a traditional, route-based protocol.

It then decodes the batch, processes the native packets and finally deletes all data associated with the batch.

3.1.3 Network coding

MORE combines packets by creating a per-byte linear combination of the native packets' payload.

At the source n native packets are combined into a single packet by generating a random vector c of size n and then computing the following sum for each byte:

$$v_i = c_0 \cdot b_i^0 + \ldots + c_n \cdot b_i^n$$

where b_i^k is the i-th byte of the k-th native packet's payload and v_i is the coded packet's i-th byte. The destination can recover the original payload if it has received n innovative packets. At that point the destination has n equations (the code vectors) to solve for n variables (the original data bytes). So the decoding boils down to solving a system of n linear equations. There are multiple ways to do this, two of them will be discussed in subsection 4.2.3

Furthermore packets are recombined at each forwarder. Each forwarder has to keep ready a pre-encoded packet for transmission. If the forwarder has only received one packet so far, the pre-encoded packet is just that packet. Otherwise the two packets are combined by multiplying both the newly arrived packet's code vector and each individual payload byte by the same random number and adding it to the pre-encoded packet. It is easy to show that the so constructed packet p still constitutes a linear combination of the native packets:

$$\begin{aligned} p' &= c \cdot p_{new} + p_{pre} \\ p' &= ca_0 \cdot p_0 + \ldots + ca_n \cdot p_n + m_0 \cdot p_0 + \ldots + m_n \cdot p_n \\ p' &= (ca_0 + m_0)p_0 + \ldots + (ca_n + m_n)p_n \end{aligned}$$

In the above equations, p_i are the native packets, a is p_{new}'s code vector and m is p_{pre}'s code vector.

3.1.4 Forwarder list

MORE employs the ETX metric as a measure of distance between nodes. The forwarder list contains all nodes that have a lower ETX to the destination than the source. It is sorted by ETX to the destination in ascending order and stays the same during the batch's lifetime.

3.1.5 Traffic control

In order to keep nodes from transmitting all the time and thus occupying the medium, MORE employs a traffic control heuristic that is based on delivery probabilities.

The heuristic is computed using three formulas. In the following equations, $i > j$ denotes that node i is closer to the destination than node j, ϵ_{ij} denotes the loss probability from node i to j and z_i is the expected number of transmissions that forwarder i must make to route one packet from the source to the destination. The first formula computes L_j, the number of packets that node j must forward. It computes for each node i the number of packets j received from i, taking into account the probability that that particular packet is not heard by any other node downstream from j.

$$L_j = \sum_{i>j} (z_i(1 - \epsilon_{ij}) \prod_{k<j} \epsilon_{ik})$$

The next equation describes how z_j is computed:

$$z_j = \frac{L_j}{1 - \prod_{k<j} \epsilon_{jk}}$$

The denominator describes the probability that some node downstream from j receives the packet. In the implementation, the z_i values are computed at each node using an incremental algorithm, which is also described in [3]. Finally, the node computes its `tx_credit`, that is, the number of transmissions that a node should make for every packet it receives from a node upstream:

$$\text{tx\_credit}_i = \frac{z_i}{\sum_{j>i} z_j(1 - \epsilon_{ji})}$$

The actual traffic control works as follows: Every node keeps a `credit_counter` variable. If a forwarder receives an innovative packet from a node upstream, it increases its `credit_counter` by the corresponding `tx_credit`. If the `credit_counter` is positive, the forwarder transmits a coded packet and then decrements its `credit_counter`. Otherwise it does not transmit at all. For a more detailed description refer to [3, p. 26-28].

3.2 ExOR

3.2.1 Terminology

Fragment	The set of packets a node forwards, i.e. the set of packets not yet received by higher priority nodes
Batch map	An array indexed by the packet number which contains the highest priority node's priority known to have received the packet designated by the index
Forwarder id	A node's position in the forwarder list (starts at zero)
Node priority	A node's forwarder id + 1 (source is not included in forwarder list)
Null-Packet	A packet only containing a batch map and no payload

Table 3.2: ExOR terminology

3.2.2 Batch map

The batch map plays a central role in ExOR. First of all a node uses it as an acknowledgement mechanism, that is, to find out, which packets it has yet to forward. This is easily achieved by picking the packets whose entry in the batch map is smaller than the current node's priority.

On the other hand the number of packets to forward — the fragment size — is used to schedule medium access (see subsection 3.2.3).

Example

Figure 3.2 graphically describes a batch map update, highlighting indirect updates due to gossiping. The received map shown on the right is received in a packet from forwarder 3, and differs in 13 locations which are highlighted in shades of gray. For ten of the entries, which are shown in dark gray, the map is updated to show forwarder 4 as the highest priority forwarder responsible

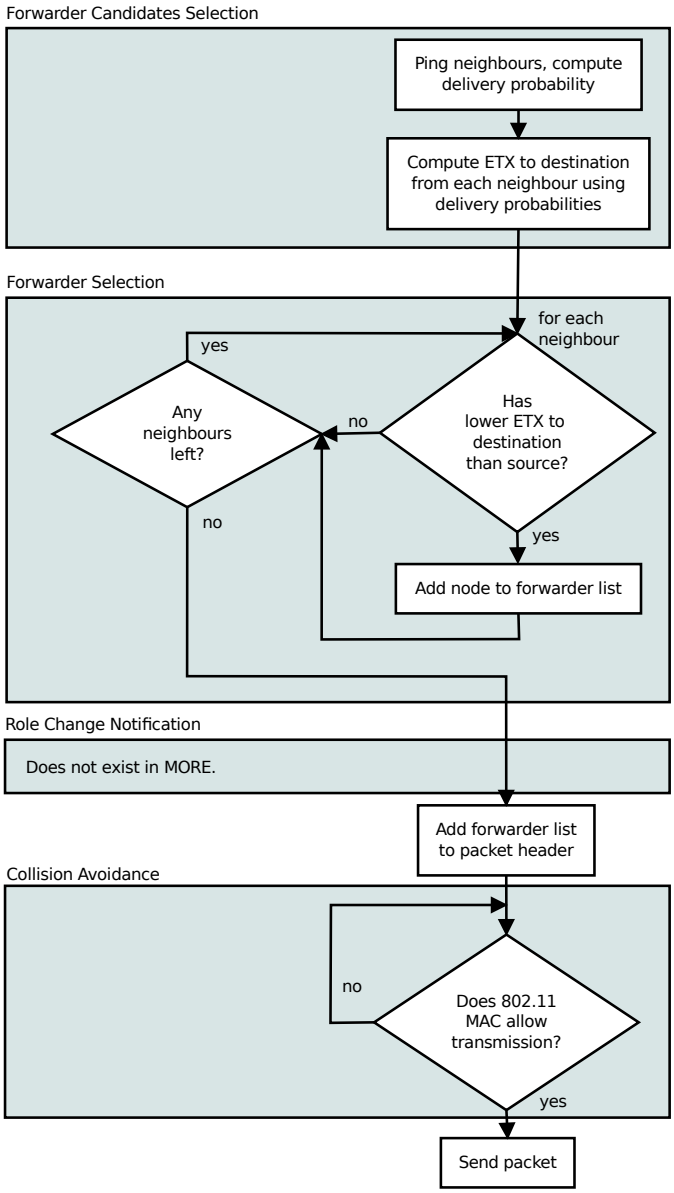

Figure 3.1: How MORE maps onto the framework's methods.

for the packet, even though information came via forwarder 3. For the other three entries, shown in light gray, the map is updated to list forwarder 3. This gossiping mechanism helps ExOR recover from batch map losses.

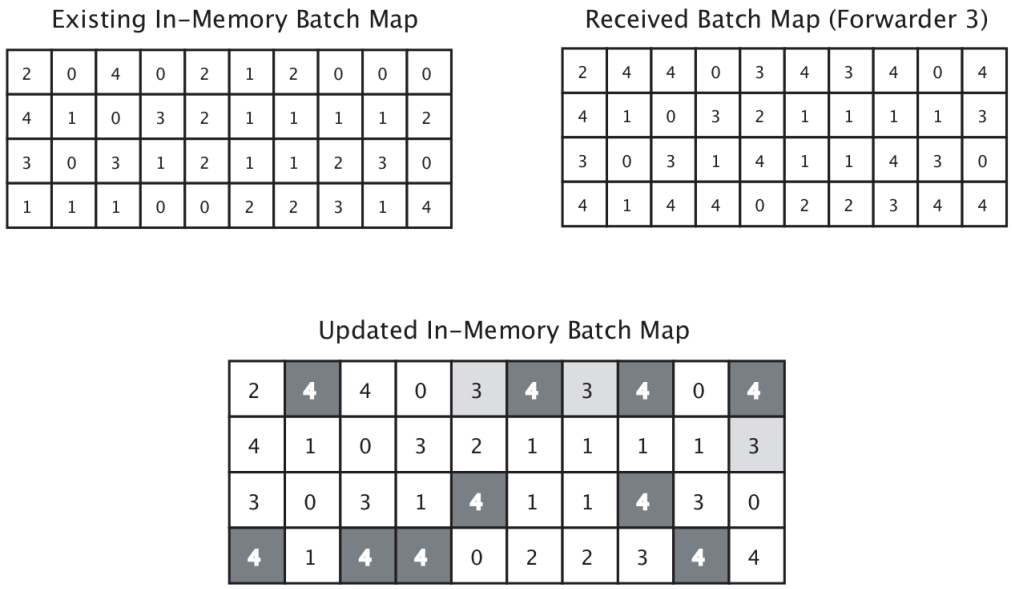

Figure 3.2: Example of a batch map update. Each entry in the batch map represents a packet. The number is the highest priority forwarder known to have a copy of the packet. In this example, the received batch map from node 3 contains updates about node 3's packets and node 4's packets. The direct updates from three are shown in light gray, the indirect updates from four are highlighted in dark gray.

3.2.3 Medium access

ExOR nodes schedule their transmissions using an exponential weighted moving average (EWMA) estimation with a smoothing factor of 0.9. After each received packet, a node updates its data rate estimation for that particular forwarder. It then uses the estimated data rate and the fragment information included in the packet to estimate the end of the current fragment. If the current forwarder is the next higher priority node, the node will schedule its transmission just after the current fragment's end. Otherwise it will add five packet transmission times for each higher priority node to the back off time.

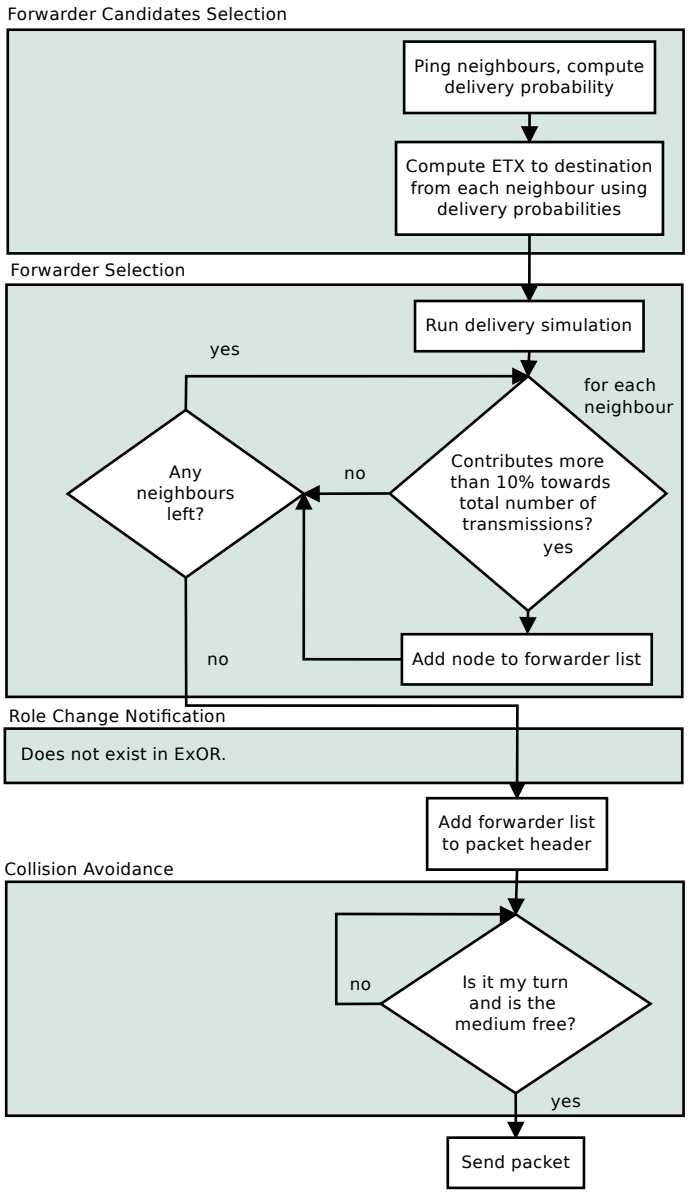

Figure 3.3: How ExOR maps onto the framework's methods.

Chapter 4

Design and Implementation

4.1 OR Framework

Over the course of this project, several modifications to the current OR framework have been made. Some of them will be described in the following sub sections.

4.1.1 Neighbor management

Originally the network nodes in the framework were represented by a rather heavy-weight data structure, which included a lot of internal information that are unnecessary for developing routing protocols. In the case of MORE and ExOR, a node is represented by an IP address and its ETX, [8], forward and reverse delivery ratio with respect to current node.

Listing 4.1: How nodes are represented within the framework

```
1  struct Node {
2      IPAddress address;
3
4      double etx;
5      double fdr;
6      double rdr;
7
8      double tx_credit; // MORE-specific
9
10     [...]
11 };
```

4.1.2 Network metrics

Both MORE and ExOR heavily rely on the ETX network metric. ExOR uses it to select and prioritize its forwarder set, which in turn determines how transmissions are scheduled. MORE uses it to select its forwarder set and to compute its traffic-control heuristic.

It is likely that future opportunistic routing protocols also rely on network metrics like ETX. Since a framework is supposed to relieve the protocol designer of burdens that are not protocol-specific, it makes sense to make network metrics a part the framework.

The base class `ORBase` includes a method **double** `getMetric(Metric metric, IPAddress ip)` which is supposed to be called by the protocols. The first parameter specifies which metric should be returned — at the moment only ETX is supported. The second parameter specifies the

node for which the metric should be returned with respect to the node which invokes the method. In the future, the framework should support the computation of a full weighted network graph using shortest-path algorithms and make them available through the method **double** `getMetric (Metric metric, IPAddress source, IPAddress destination)`. This would be helpful as both MORE and ExOR — and probably future protocols — rely on knowing the whole network graph.

4.1.3 ETX reader

INETMANET offers ways to compute ETX values during the simulation using ping probes. But sometimes it might be desirable to use static ETX values which stay the same for the duration of the simulation. For that purpose a helper class was implemented, which reads ETX values from a CSV file and makes them available through the framework methods discussed in subsection 4.1.2. Thus it is possible to switch between dynamic and static ETX values just by changing the simulation configuration, but without changes to the source code.

The CSV file is assumed to have the following format: If there are n nodes in the network, then the file is supposed to have n lines. Each line (row) contains n comma-separated strings (columns). The row and column index then corresponds to the source's and the destination's IP's host number respectively. For example: The ETX of 192.168.0.12 to 192.168.0.23 is in row 12, column 23.

In order to support more than one metric per host-pair, each string can hold multiple space-separated numbers. The current implementation assumes that each string contains three values in this order: FDR, RDR, ETX. At the moment the reader only supports flat networks up to 254 hosts.

4.1.4 Problems

Currently the protocols have too little control over the radio/MAC layer. MORE and ExOR solely rely on broadcast transmissions. But broadcast packets are not re-transmitted in case of a collision and generally not as reliable as unicast transmissions[1]. Furthermore the protocols never learn about a collision and so just have to assume that the transmission was successful. It is also not possible for a protocol to cancel a scheduled transmission. Some simulation results will be given in subsection 5.2.2 to illustrate this problem.

4.2 MORE

4.2.1 Basics

The basic architecture is pre-defined by the OR-Framework, which means that there is a class `MORE`, which inherits from `ORBase` and is meant to contain most of the protocol's logic. Additionally there are a few helper classes for specialised tasks, in order to keep the main class as uncluttered as possible.

[1] IEEE Std 802.11™-2007, section 9.2.7

In the following sections a few key components of the implementation and major design decisions will be discussed. The MORE implementation mainly includes the following methods:

Listing 4.2: Snippet of MORE.h

```
1  virtual std::vector<Node> candidateSelection(Node source, Node destination
       );
2
3  virtual Node forwarderSelection(std::vector<Node>& candidatesRanked);
4
5  void handleCodedMessage(MorePacket* packet);
6
7  void handleAckMessage(MorePacket* packet);
8
9  void computeNeededTransmissions();
10
11 void computeTxCredit(std::valarray<double>& z, const std::vector<Node>&
       nodes, std::vector<Node>::const_iterator it);
```

We spend the rest of the section with a more detailed discussion of the above methods.

- **candidateSelection()**
  This method retrieves the list of nodes in the network from `ORBase` and store them together with their ETX to the batch's destination.
- **candidateRanking()**
  This method sorts the forwarder list by ETX to the destination in ascending order.
- **handleCodedMessage()**

Listing 4.3: How coded packets are handled by MORE

```
1  void MORE::handleCodedMessage(MorePacket* p) {
2     CodedPacket* cp = check_and_cast<CodedPacket*>(p);
3
4      if ( (!cp->inForwarderList(ip_address_) && !(cp->getDst() ==
           ip_address_)) || cp->getSrc() == ip_address_ ) {
5          delete cp;
6          return;
7      }
8
9      if (packet_buffer_ == 0) {
10         if ( cp->getBatch_id() <= last_batch_id_ ) {
11             delete cp;
12             return;
13         }
14         packet_buffer_ = new more::PacketBuffer(cp, gf_, ip_address_,
               getNeighborEtx(cp->getDst()));
15
16         forwarders_ = cp->getForwarderList();
17         batch_start_ = cp->getTimestamp();
18         last_batch_id_ = cp->getBatch_id();
19
20         if ( !IAmDestination() ) {
21             computeNeededTransmissions();
22         }
```

```
23          scheduleAt(simTime(), batch_timeout_check_);
24      } else if (cp->getBatch_id() > packet_buffer_->batch_id()) {
25          dropBatch("out␣of␣date.");
26
27          packet_buffer_ = new more::PacketBuffer(cp, gf_, ip_address_,
                getNeighborEtx(cp->getDst()));
28
29          if ( !IAmDestination() ) {
30              computeNeededTransmissions();
31          }
32
33          last_batch_id_ = cp->getBatch_id();
34          batch_start_ = cp->getTimestamp();
35      }
36
37      if ( cp->getForwarderEtx() >= getNeighborEtx(cp->getDst()) && !
            IAmDestination() ) {
38          credit_counter_ += cp->getForwarderTxCredit();
39      }
40
41      bool was_innovative = packet_buffer_->addOrDiscard(cp);
42
43      if (packet_buffer_->ready() && IAmDestination() ) {
44          AckPacket* ack = new AckPacket("MORE␣ACK", more::ACK);
45          // ...
46
47          sendToIp(ack->dup(), packet_buffer_->source(), 30, 0,
                IP_PROT_MANET);
48
49          uint8_t* payload = packet_buffer_->decode();
50
51          dropBatch("decoded.");
52
53          delete[] payload;
54          delete packet_buffer_;
55          packet_buffer_ = 0;
56      } else if ( credit_counter_ > 0 && !IAmDestination() &&
            was_innovative) {
57          broadcast(packet_buffer_->getPreEncodedPacket(), 30,  par("
                packetForwardDelay").doubleValue());
58          credit_counter_--;
59          packets_sent_++;
60      }
61  }
```

First the cases are covered when the node should ignore the packet (line 4–7), that is, when the receiver is neither in the forwarder list nor the destination, and also if it is the batch's source.

Then, if there is no current buffer, but the packet belongs to an outdated batch, the packet is ignored as well (line 10–12). Otherwise a new buffer is created and computations for the traffic control heuristic are made (line 14–23).

If there is a current buffer, but its batch id is lower than the packet's, the current buffer is

dropped and a new buffer is created.

In line 37–38 the node's `credit_counter` is updated depending on whether the packet was sent by a node from up- or downstream. It follows the packet's inspection by the packet buffer in line 41, which will yield whether the packet is innovative or not.

Afterwards, if the packet buffer is ready for decoding and if the current node is also the destination, it sends the ACK towards the batch's destination, decodes and subsequently discards the batch (line 43–55). Otherwise, given that its `credit_counter` is positive, it is not the destination and that the received packet was innovative, the node transmits a packet (line 56–59).

- **handleAckMessage()**
  This method handles received ACKs. When a node receives an ACK which belongs to an outdated batch, it will simply ignore it. If the ACK belongs to a currently active batch, the node will discard the current buffer. Finally if the ACK belongs to a batch of which the current node is the source, the node records its throughput and the round-trip time before stopping the simulation.

- **computeNeededTransmissions() & computeTxCredit()**
  In those two methods MORE's traffic control heuristic is computed. The detailed formulas and algorithms are specified in [3, p. 26–29]. Although we implemented the algorithms as specified, they did not perform very well. For instance, the algorithm computes that the source would have to do less than one transmission in order to get a packet to the destination. As a consequence, the computed `tx_credit` values are wrong as well and therefore the traffic control is ineffective.

4.2.2 Network coding

Just as in [3], network coding related calculations are done over the Galois field $GF(2^8)$[2]. Its arithmetic operations can be implemented in a computationally efficient fashion and its elements can be expressed in one byte.

The class `Gf256` offers methods for performing arithmetic operations in $GF(2^8)$. In $GF(2^8)$ subtraction and addition are both equivalent to the XOR operation, which is part of virtually every CPU instruction set. Multiplication is more complex and there are multiple ways to implement it. Here are the two most efficient:

1. Create two look-up tables $\{exp(a) : a \in GF(2^8)\}$ and $\{log(a) : a \in GF(2^8)\}$. Because of $log(a \cdot b) = log(a) + log(b)$, multiplication can be reduced to $a \cdot b = exp(log(a) + log(b))$. The look-up tables require $2 \cdot 256 = 512$ bytes of memory.

2. Create a look-up table $\{a \cdot b : (a, b) \in GF(2^8) \times GF(2^8)\}$ and design a hash function which maps every pair of elements in $GF(2^8)$ to the corresponding result. This look-up table requires $256 \cdot 256 = 65536$ bytes.

Currently method 1 is implemented because it is a good compromise between memory usage and computational complexity.[3]

[2] A Galois field (GF) with an order of 2^8. It contains the elements $0 \ldots 255$.

[3] There are algorithms which compute every result on the fly, but since they would be run (possibly tens of) thousands of times, while just producing 255 different values, it would not be economical to use them.

4.2.3 Packet buffer

Packet management is a more delicate matter for MORE than for other protocols. In most cases it would be enough to have a queue, which accepts newly received packets and returns packets which are due for transmission. A simple queue would not add much clutter to a class.

But MORE requires various calculations to be done on a packet's header and payload, such as check for innovation, decoding and keeping a pre-encoded packet ready. Having the corresponding code in the main class would significantly decrease its readability while offering no visible advantage.

The class `PacketBuffer` has the following responsibilities:

1. Inspect incoming packets and check whether they are innovative. See Listing 4.4.
2. Store innovative packets. See Listing 4.4.
3. Maintain a pre-encoded packet which is ready for transmission and includes the latest information, i.e. all packets received until then. See Listing 4.6.
4. Decode a batch of packets when there are enough innovative packets. See Listing 4.5.

It follows a more detailed description of important methods in `PacketBuffer`.

- **addOrDiscard()**

Listing 4.4: Decide whether to store or discard a packet

```
1  bool PacketBuffer::addOrDiscard(CodedPacket* packet) {
2      if (ready() && !isForUs()) {
3          clear();
4          cvm_.zero();
5          code_vectors_.zero();
6          payloads_ = new uint8_t*[batch_size_];
7          std::fill_n(payloads_, batch_size_, static_cast<uint8_t*>(0));
8      }
9
10     if (isInnovative(packet)) {
11         code_vectors_.setCol(buffer_.size(), packet->getCodeVector(),
12                 batch_size_);
13         payloads_[buffer_.size()] = packet->getPayload();
14         buffer_.push_back(packet);
15
16         if (!isForUs()) {
17             addToPreEncodedPacket(packet);
18         }
19
20         std::time(&last_activity_);
21         return true;
22     } else {
23         delete packet;
24         return false;
25     }
26 }
```

The method starts with resetting the buffer if it is full and if the current node is not the destination (lines 2–7). This is necessary because otherwise the forwarders might stop transmitting[4] prematurely, that is, before the destination has received enough innovative packets.

Then if the packet is innovative, it is saved in the buffer and important data is copied from its header (line 10–14). If the node is a forwarder, then the packet is combined with the already pre-encoded packet (line 16–18). Otherwise the packet is discarded in line 23.

- **decode()**

Listing 4.5: Decode payload

```
1  uint8_t* PacketBuffer::decode() {
2      assert(buffer_.size() == batch_size_);
3
4      Matrix A(batch_size_, gf_);
5      A.transpose(code_vectors_);
6
7      uint8_t* x = new uint8_t[batch_size_ * payload_size_];
8      std::fill_n(x, batch_size_ * payload_size_, 0);
9  #ifdef INVERT_MATRIX
10     A.invert();
11     A.mappedProduct(payloads_, batch_size_, payload_size_, x);
12 #else
13     Vector y(batch_size_, gf_);
14     uint8_t x_i = 0;
15     Matrix L(batch_size_, gf_);
16     Matrix U(batch_size_, gf_);
17
18     A.LUDecomposition(L, U);
19
20     for ( int l = 0; l < payload_size_; l++ ) {
21         for (int i = 0; i < batch_size_; i++) {
22             y(i) = payloads_[i][l];
23
24             for (int j = 0; j < i; j++) {
25                 y(i) = gf_.sub(y(i), gf_.mul(L(i, j), y(j)));
26             }
27
28             y(i) = gf_.mul(y(i), gf_.inv(L(i, i)));
29         }
30
31         for (int i = batch_size_ - 1; i >= 0; i--) {
32              x_i = y(i);
33
34             for (int j = i + 1; j < batch_size_; j++) {
35                 uint8_t x_j = x[j *  payload_size_ + l];
36                 x_i = gf_.sub(x_i, gf_.mul(U(i, j), x_j));
37             }
```

[4]Remember that forwarders only transmit after they have received an innovative packet. Let batch size be n. Every packet's code vector is actually a basis vector. If the buffer contains n packets, their code vectors are by definition independent. n independent code vectors form a basis for the vector space $GF(2^8)^n$. It follows that every other code vector lies in the span of said basis, making it linearly dependent and the associated packet non-innovative.

```
38
39              uint8_t inv = gf_.inv(U(i,i));
40              x_i = gf_.mul(x_i, inv);
41              x[i * payload_size_ + l] = x_i;
42          }
43          y.zero();
44          x_i = 0;
45      }
46 #endif
47      return x;
48 }
```

This is where the original payload is retrieved from the coded packets. As mentioned before, the code vectors form the coefficient matrix (denoted A in the listing) of a system of linear equations. So the system to solve is $Ax = b$, where x is the original payload and b is the coded payload. In the above method, two ways of solving this system are implemented.

The first way (lines 11 and 12) consists of inverting the matrix and then retrieving the original payload by repeatedly computing $A^{-1}b$ with an appropriate[5] b.

The second way (line 14–49) uses LU-Decomposition[6], which decomposes A into the lower-triangular matrix L and the upper-triangular matrix U with $A = LU$. Solving the system $Ax = b$ can now be reduced to first solving $Ly = b$ for y and then $Ux = y$ for x. Since L and U are both triangular, this can be quickly done by using forward (line 23–31) and backward substitution (line 33–45) respectively. Similar to the first method, those two substitution steps need to be performed for all appropriate b, as explained in footnote 5.

- **addToPreEncodedPacket()**

Listing 4.6: Add newly received packet to pre-encoded packet

```
1  void PacketBuffer::addToPreEncodedPacket(CodedPacket* new_packet) {
2      if ( pre_encoded_ == 0 ) {
3          pre_encoded_ = new_packet->dup();
4      } else {
5          Vector new_packet_code_vector(batch_size_, gf_);
6          new_packet_code_vector.copy(new_packet->getCodeVector(),
               batch_size_);
7
8          uint8_t random = more::randomInt();
9          Vector new_code_vector(batch_size_, gf_);
10         new_code_vector.copy(pre_encoded_->getCodeVector(),
               batch_size_);
11
12         new_packet_code_vector.mul(random);
13         new_code_vector.add(new_packet_code_vector);
14
15         uint8_t* new_packet_payload = new_packet->getPayload();
16         uint8_t* payload = pre_encoded_->getPayload();
17         uint8_t* new_payload = new uint8_t[payload_size_];
18
19         for ( uint i = 0; i < payload_size_; i++ ) {
```

[5]The product $A^{-1}b$ has to be computed as many times as there are bytes in the payload. For iteration i, b is equal to $(b_i^1, \ldots, b_i^n)$, where b_i^k is the i-th byte of the packet associated with the vector in A's k-th row.

[6]http://mathworld.wolfram.com/LUDecomposition.html

```
20              new_payload[i] = gf_.add(payload[i], gf_.mul(random,
                    new_packet_payload[i]));
21          }
22
23          delete pre_encoded_;
24          pre_encoded_ = 0;
25
26          pre_encoded_ = new CodedPacket("CodedPacket (PB)", more::CODED
                );
27          // ...
28          // set variables
29      }
30  }
```

If there is no pre-encoded packet, a duplicate of the newly received packet is used (line 2–3). Then the new code vector v_{new} is computed in line 8–13. Let $r \in GF(2^8)$ be the random number, p the pre-encoded packet's code vector and q the newly received packet's code vectors. Then $v_{new} = r \cdot q + p$.

In line 15–21, the newly received packet's payload is added to the pre-encoded packet's payload, just as with the code vectors. So for each payload byte the following sum is calculated: $b_i = r \cdot u_i + v_i$, where b_i, u_i, v_i are the new payload's, the newly received packet's and the pre-encoded packet's payload's i-th byte respectively.

Finally the old pre-encoded packet is deleted and the new pre-encoded packet is created.

4.2.4 Batch transmission

Before the source can start to transmit a batch of size n, it has to create coded packets from the native packets. This is done by generating a random vector c for each coded packet and then compute $v_i = c_0 \cdot b_i^0 + \ldots + c_n \cdot b_i^n$ where b_i^k is the i-th byte of the k-th native packet's payload and v_i is the coded packet's i-th byte.

Since this kind of computation only happens at the source, we decided to implement it in a separate class and not in `PacketBuffer`.

4.2.5 Matrix and Vector

At first we used simple `uint8_t`[7] arrays for vectors and two-dimensional arrays for matrices. But this approach produced code that was hard to read. Because no matrix and vector classes operating over $GF(2^8)$ were available, we chose to implement them on our own.

Those two classes provide abstractions of vectors and matrices over $GF(2^8)$. Implemented functionality includes Gaussian elimination, LU decomposition, matrix inversion, forward and backward substitution, check for linear independence, matrix product, dot product, scalar multiplication, vector addition and various row/column swapping/copying methods.

4.3 ExOR

4.3.1 Basics

As with MORE, there is a class `ExOR` which inherits from `ORBase` and holds most of the protocol's logic. But apart from `PacketBuffer` ExOR does not require any helper classes —

[7] a typedef for **unsigned char**

mostly because it does not any employ network coding techniques.

In the following sections a few key components of the implementation and major design decisions will be discussed. The ExOR implementation mainly includes the following methods:

Listing 4.7: Snippet of ExOR.h

```
1  double estimateDataRate(Node forwarder);
2
3  simtime_t estimateWhenToForward(int forwarderNum,  double numPacketsLeft);
4
5  void handleExORPacket(ExORPacket* packet);
6
7  void cullForwarderSet(std::vector<Node>& forwarders);
8
9  virtual std::vector<Node> candidateSelection(Node source,
10     Node destination);
11
12 virtual std::vector<Node> candidateRanking(std::vector<Node>&
       candidateVector);
```

It follows a discussion of ExOR's most important methods.

- **candidateSelection()**
  This method gets all nodes in the network from `ORBase`'s neighbor management and stores them in a list along with their ETX to the current destination.
- **candidateRanking()**
  This methods sorts the list compiled by `candidateSelection()` by ETX to the destination in descending order. The sorted list is the forwarder list that will be included in every packet of this batch.
- **Fragment forwarding**

  Listing 4.8: Forwarding fragment

```
1  if ( currentBatch_->getCurrentFragmentSize() - 1 > currentBatch_->
       getLastReceivedFragNum() ) {
2      scheduleAt(simTime() + 5 * currentBatch->ONE_PKT_TIME,
           forward_fragment_);
3      return;
4  }
5
6  if ( !currentBatch_->isForUs() && currentBatch_->getFragmentSize() > 0
       ) {
7      for ( int i = 0; i < currentBatch_->getFragmentSize(); i++ ) {
8          ExORPacket* p = currentBatch_->getNextInQueue();
9          broadcast(p, 30, 0);
10     }
11 } else {
12     ExORPacket* ack = currentBatch_->getNullPacket();
13     ExORPacket* dup = 0;
14
15     for ( int i = 0; i < 10; i++ ) {
16         dup = ack->dup();
17         dup->setFragmentNum(i);
18         broadcast(dup, 30, 0);
```

```
19      }
20      delete ack;
21  }
22
23  if ( currentBatch_->forwardedEnough() ) {
24      recordScalar("buffer_size", currentBatch_->getBufferSize());
25      batchBuffer.erase(currentBatch_->getBatchId());
26      delete currentBatch_;
27      currentBatch_ = 0;
28      cancelEvent(forward_fragment_);
29  }
```

This piece of code handles the forwarding of a fragment. It starts with postponing the forwarding if another node's fragment is still in progress (lines 1–4). This behaviour is not mentioned in [2] but seemed necessary as nodes would frequently interrupt each other during a fragment's transmission.

It then goes on to the actual forwarding: The if clause in line 6 is only true if the current node is a forwarder or the source and if there are packets to forward. In that case the node will broadcast the fragment (lines 7–10). Otherwise it will broadcast a fragment of 10 Null-Packets to relay its possibly more up-to-date batch map to the surrounding nodes (lines 12–20).

Finally, from line 23 to 28, the node deletes the current batch if more than 90% of the batch's packet were received by higher priority nodes.

- **handleExORPacket()**

Listing 4.9: How received packets are handled by ExOR

```
1  void ExOR::handleExORPacket(ExORPacket* packet) {
2      if ( !packet->isInForwarderList(ip_address_) ) {
3          if ( packet->getSource() == localNode.address &&
4                  batchWasCompleted(packet) ) {
5              delete packet;
6              endSimulation();
7          }
8          delete packet;
9          return;
10     }
11
12     if ( lastBatchId_ >= packet->getBatchId() && currentBatch_ == 0 )
           {
13         delete packet;
14         return;
15     }
16
17     if ( batchBuffer.count(packet->getBatchId()) == 0 && packet->
           getBatchId() > lastBatchId_ && !packet->isNull() ) {
18         currentBatch_ = new exor::PacketBuffer(packet, localNode);
19         batchBuffer[packet->getBatchId()] = currentBatch;
20         lastBatchId_ = packet->getBatchId();
21     } else {
22         int fragNum = packet->getFragmentNum();
23         int fragSize = packet->getFragmentSize();
```

```
24          int forwarderNum = packet->getForwarderNum();
25          Node forwarder = packet->getForwarder();
26
27          if ( packet->isNull() || currentBatch_->hasPacket(packet) ) {
28              currentBatch_->updateBatchMap(packet);
29              delete packet;
30          } else {
31              currentBatch_->add(packet);
32          }
33
34          estimateDataRate(forwarder, fragSize, fragNum);
35          simtime_t forwardTime = estimateWhenToForward(forwarderNum,
                fragSize, fragNum);
36
37          cancelEvent(forward_fragment_);
38          scheduleAt(forwardTime, forward_fragment_);
39      }
40  }
```

Packet handling starts with checking whether the current node is in the forwarder list (line 1). If it is not, then it will check whether it is the packet's source[8] and whether the packet's batch map indicates that the batch was successfully received by the destination (line 3 and 4). In that case the node will end the simulation. Otherwise it will just return from the method.

If the current node is in the forwarder list, it will first check whether the packet belongs to an outdated batch. In that case the method exits (line 12–15).

If the node does not yet have a buffer for this batch, the batch is not outdated and the packet is not a Null-Packet, then the node creates a new packet buffer for the packet's batch (lines 17–21). Line 27 to 29: If the packet has already been received or if it is just a Null-Packet, then just its batch map will be looked at and the packet discarded. Otherwise the packet will be stored in the buffer in line 31.

After taking care of the packet, the node will use the information in the packet header to estimate the current forwarder's data rate and then schedule its next transmission (lines 34–38).

- **estimateDataRate()**

Listing 4.10: Estimate data rate

```
1  double ExOR::estimateDataRate(Node forwarder, int fragmentSize, int
       fragmentNum) {
2      if (currentForwarder != forwarder) {
3          currentForwarder = forwarder;
4          currentRate = -1.0;
5          return -1.0; // not ready to give estimation
6      }
7
8      if ((simTime() - currentBatch->getLastReception()) < 0.000001) {
9          return currentRate;
10     }
```

[8]The source is not included in the forwarder list and the source of a batch packet stays the same throughout a batch's lifetime. Only the forwarder number in the header changes.

```
11
12      double rate = 1 / (simTime() - currentBatch->getLastReception());
13
14      if (currentRate > 0) {
15          currentRate = ALPHA * rate + (1 - ALPHA) * currentRate;
16      } else {
17          currentRate = rate; // first rate estimation
18      }
19
20      return currentRate;
21  }
```

If this is the first packet in a fragment, there can be no estimation of a forwarder's data rate. Therefore the method exits and sets the current rate to -1.0 (lines 2–5).

It is also not possible to give a new estimation if the interval between packet reception is below a certain threshold. In that case the current rate stays untouched (lines 8–10).

In line 12 the data rate between the current and the last reception is computed. If the current rate is larger than zero, that is, if the last reception was not the first reception, then the EWMA[9] formula is used to estimate the current rate. Otherwise the rate computed in line 12 is taken as the current rate.

- **estimateWhenToForward()**

Listing 4.11: Estimate when to forward fragment

```
1  simtime_t ExOR::estimateWhenToForward(int forwarderNum, double
       numPacketsLeft) {
2      double backoffTime = 0;
3      double DEFAULT_BACKOFF_TIME = 5.0 * currentBatch_->ONE_PKT_TIME;
4
5      if (currentRate_ < RATE_TH) {
6          backoffTime += DEFAULT_BACKOFF_TIME;
7      } else {
8          backoffTime += numPacketsLeft / currentRate_;
9      }
10
11     if ( forwarderNum - 2 != currentBatch_->getLocalPriority() ) {
12         int numHigherPriorityNodes = currentBatch_->getNumForwarders()
               - currentBatch_->getLocalPriority() - 1;
13
14         backoffTime += DEFAULT_BACKOFF_TIME * numHigherPriorityNodes;
15     }
16     return simTime() + backoffTime;
17 }
```

First the method adds a delay for the current forwarder. If the current rate is invalid, the default back off time is used. Otherwise the fragment information from the packet's header and the current rate are used to estimate the amount of time it takes the current forwarder to finish its fragment (lines 5–9).

Then if the current forwarder is not the next higher priority node in the forwarder list, the node adds the default back off time for each higher priority node (lines 11–15).

[9]Exponential weighted moving average

Finally it returns the simulation time when the node should start transmitting its fragment.

Although only one node is supposed to transmit at any time, the rather low default back off time of five packet transmission times often leads to simultaneous transmissions and nodes being interrupted in the middle of their fragment.

- **cullForwarderSet()**
  Having too many forwarders degrades overall throughput because a lot of nodes just transmit Null-packets without actually contributing to the forwarding process. To counter this the authors of ExOR say in [2, p. 22] that "the source runs a rough simulation of delivery using a complete prioritized forwarder set and selects the nodes which transmit at least 10% of the total transmissions in a batch".

  For lack of a more detailed description, we implemented the simulation as follows. Assuming a batch size of n, the source starts by broadcasting n packets. Exploiting a node's knowledge about the delivery probability p_i to its one-hop neighbours, the number of packets received by each of its one hop neighbors is $p_i \cdot n$. Then the next node in the forwarder list broadcasts as many packets n' as it has received during the last reception. The number of packets this node's neighbours receive is $p_i \cdot n'$. Then follows the next node in the forwarder list and so on. The simulation ends after 10 iterations over the forwarder list or if the destination received all the packets. In the end the total number of transmissions is computed and nodes that contributed less than 10% towards the total number of transmissions are removed from the forwarder set.

  The removal of some of the forwarders using the above simulation resulted in a higher batch transmission delay than if the forwarder set was left untouched. This suggests that nodes vital for the forwarding were removed and that the above modelling should be improved.

  But modelling, for instance, exactly which packet of a batch is lost during a transmission would add a lot of complexity which, depending on the deployment scenario, might require too much computing power considering its benefits.

4.3.2 PacketBuffer

The class `PacketBuffer` has the following responsibilities:

1. Inspect incoming packets and check whether they have already been received. Since packets are stored in a hash table with the packet number as the key, this task is trivial.
2. Store packets. As mentioned in the previous point, packets are stored in a hash table, making storage trivial as well.
3. Update the batch map and keep track of what packets have to be forwarded.

Listing 4.12: Update batch map and compute fragment size

```
1 void PacketBuffer::updateBatchMap(ExORPacket* pkt) {
2     std::valarray<int> batchMap = pkt->getBatchMap();
3
4     transmissionQueue_ = std::queue<ExORPacket*>();
5     numReceivedByHigherPriorityNode_ = 0;
6
7     for ( unsigned int i = 0; i < batchSize_; i++ ) {
8         if (batchMap[i] > batchMap_[i]) {
9             batchMap_[i] = batchMap[i];
```

```
10          }
11
12          if ( hasPacket(i) && needTransmit(i) && !isForUs() ) {
13              scheduleForTransmission(i);
14          } else if ( hasBeenReceivedByHigherPriorityNode(i) ) {
15              numReceivedByHigherPriorityNode_++;
16          }
17      }
18
19      fragmentSize_ = transmissionQueue_.size();
20  }
```

This method updates the local batch map by comparing it to the one included in the received packet. If the forwarder id in the received batch map is higher, it is copied to the local batch map (line 8–10). That way the node only forwards packets that were not received by higher priority nodes. In line 12 to 14 a packet is scheduled for forwarding if the node actually has it, if it has not been received by a higher priority node and if the local node is not the batch's destination.

Because the node needs to know the percentage of packets that have already received by higher priority nodes, those packets are counted in line 14 to 16. Since the fragment size is equal to the number of packets that were not received by a higher priority node and thus due for transmission, the fragment size is equal to the length of the transmission queue.

4. To provide main class with means to access packets which need to be forwarded whenever it is the node's turn to transmit. The main class can access packets due for transmission using the `ExORPacket* getNextInQueue()` method, and the fragment size with **int** `getFragmentSize()`.

Chapter 5

Simulation and Evaluation

5.1 Simulation Setup

All protocols were evaluated on the same static topology that was used in [3], see Figure 5.1 below. The network consists of 12 fixed nodes. Simulations were run for batch sizes 16 and 32, as well as for packet sizes of 128, 256, 512, 1024, 2048, 4096 and 8192 bytes. For each combination of those two parameters 20 runs were evaluated. Node 11 acted as the source and node 5 was the destination for all transmissions.

Size	120x80m (5m per grid square)
Radio range	$\sim$ 25m
Propagation model	Path-loss reception
Path-loss alpha	4.5
Number of nodes	12
Bit-rate	11 Mbps
Radio sensitivity	-90dBm
Transmission power	59mW

Table 5.1: Simulation parameters

5.2 Metric & Results

5.2.1 Metrics

The following metrics are used to compare the different routing protocols.

- **Transmission delay** was measured at the destination as follows:

  $$\text{transmission delay} = \text{arrival time of last packet} - \text{arrival time of first packet} \quad \text{[seconds]}$$

- **Round-trip delay** was measured at the source as follows:

  $$\text{round-trip delay} = \text{ACK arrival time} - \text{batch start time} \quad \text{[seconds]}$$

- **Throughput** was measured at the source as follows:

  $$\text{throughput} = \frac{\text{batch size}}{\text{ACK arrival time} - \text{batch start time}} \quad \text{[packets/sec]}$$

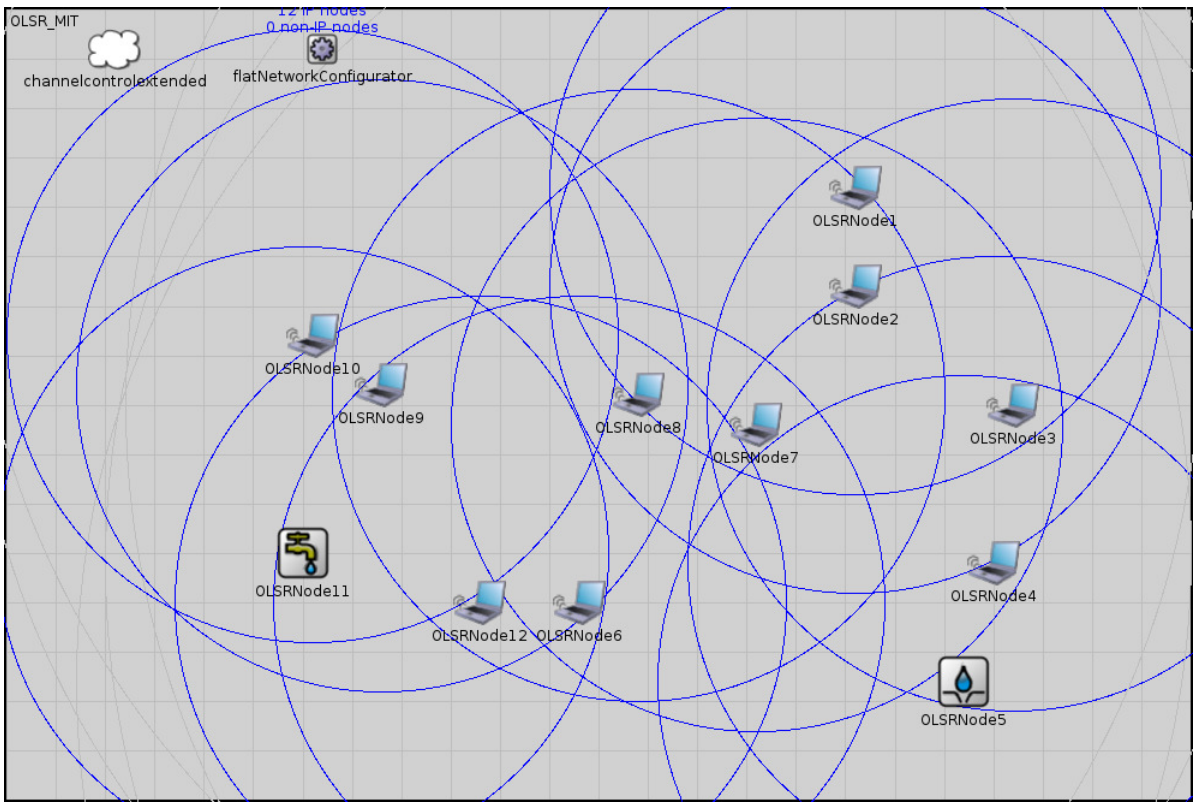

Figure 5.1: Simulation network topology: the blue circles mark the transmission ranges.

5.2.2 Results

The number beside a boxplot indicates how many samples it is based on and will be referred to as "sample size". Figure 5.2 and Figure 5.3 show that OLSR is outperformed by ExOR, but not by MORE. There are probably three reasons why MORE performs so poorly.

First, there is the problem of a crowded medium. This is in part caused by the faulty (see 4.2.1) traffic control heuristic, which has to be circumvented by setting the `credit_counter` to almost the same value as the batch size in order to get the whole batch through to the destination. With that adjustment, every node that receives an innovative packet now also forwards a packet. Together with OLSR keeping its routing tables up-to-date as well, this results in a very congested medium with a lot of collisions.

This leads straight on to the second reason, which is the lack of control over the radio/MAC layer. In its current form routing protocols in the OR framework are not aware of collisions. This is particularly important for opportunistic routing protocols since all packets are broadcasted and broadcast transmissions are not retransmitted by the MAC layer.[1] In a crowded medium with lots of packets lost due to collisions this might seriously impair a batch's progress towards the destination. See Figure 5.8 and Figure 5.9 for more details on collisions during the simulation. Those two figures show that for both MORE and ExOR a lot of collisions happen across all nodes, in particular for those nodes located at the center of the topology.

The third reason has to do with how MORE's forwarders work. When a forwarder receives

[1] IEEE Std 802.11^{TM}-2007, section 9.2.7

an innovative packet and its `credit_counter` is positive, then it will immediately transmit a pre-encoded packet. Unfortunately nodes with the same distance to the sender receive the packet almost at the same time and therefore will try to forward it at almost the same time, which leads to collisions. In order to avoid synchronisation among forwarders, forwarding a packet is delayed by a random amount of time. This helps to reduce collisions, but on the other hand also increases the transmission delay.

ExOR's performance gain on OLSR might be because its ability to exploit multiple forwarders, the ACK redundancy introduced by the batch map and because it does not need per-packet acknowledgements.

The ranking changes quite a bit in Figure 5.4 and Figure 5.5. OLSR performs best, followed by MORE and then ExOR. ExOR's rather erratic performance is easily explained: It is only meant to deliver 90% of packets in a batch [2, p. 29], so our metric is not really applicable. ExOR relies on the source receiving a batch map which indicates that the destination has received all packets. The problem is that a forwarder discards a batch and ignores subsequently received packets of that batch, when more than 90% of packets have been received by a higher priority node. Since this will happen earlier in a batch's lifetime, the closer we get to the batch's source, it is very likely that in a large enough network a lot of forwarders between the source and the destination already discarded the batch when the destination finally received all the packets and announces this by transmitting a batch of Null-Packets. Thus this information never reaches the source. Judging by ExOR's sample sizes in Figure 5.4 and Figure 5.5, the evaluation network qualifies as "large enough". It is not clear where ExOR's large variance for smaller packet sizes comes from.

Compared to ExOR, MORE's way of handling ACKs seems to be more efficient. The round-trip time is only slightly higher than the transmission delay, there is only a minor increase in variance and the overall shape of the curve is preserved. It has to be considered that contrary to the specification in [3], forwarders cannot "overhear" an ACK and then discard their batch[2]. Consequently there is more traffic while the ACK is on its way, which is likely to have some negative effect on the ACK's travel time.

There are no surprises in Figure 5.6 and Figure 5.7 because of the way the throughput is computed, which is batch size (constant) divided by the round-trip delay, which was discussed in the last section.

In general, batch size does not seem to have any significant effect on performance when the protocols are evaluated in a simulation environment.

[2]This is due to the commonly used OMNeT++ node architecture. Protocols are placed above the IP layer, which only forwards unicast packets to upper layers if they are meant for that node. Since the ACK is addressed to the batch's source, forwarders never learn about it.

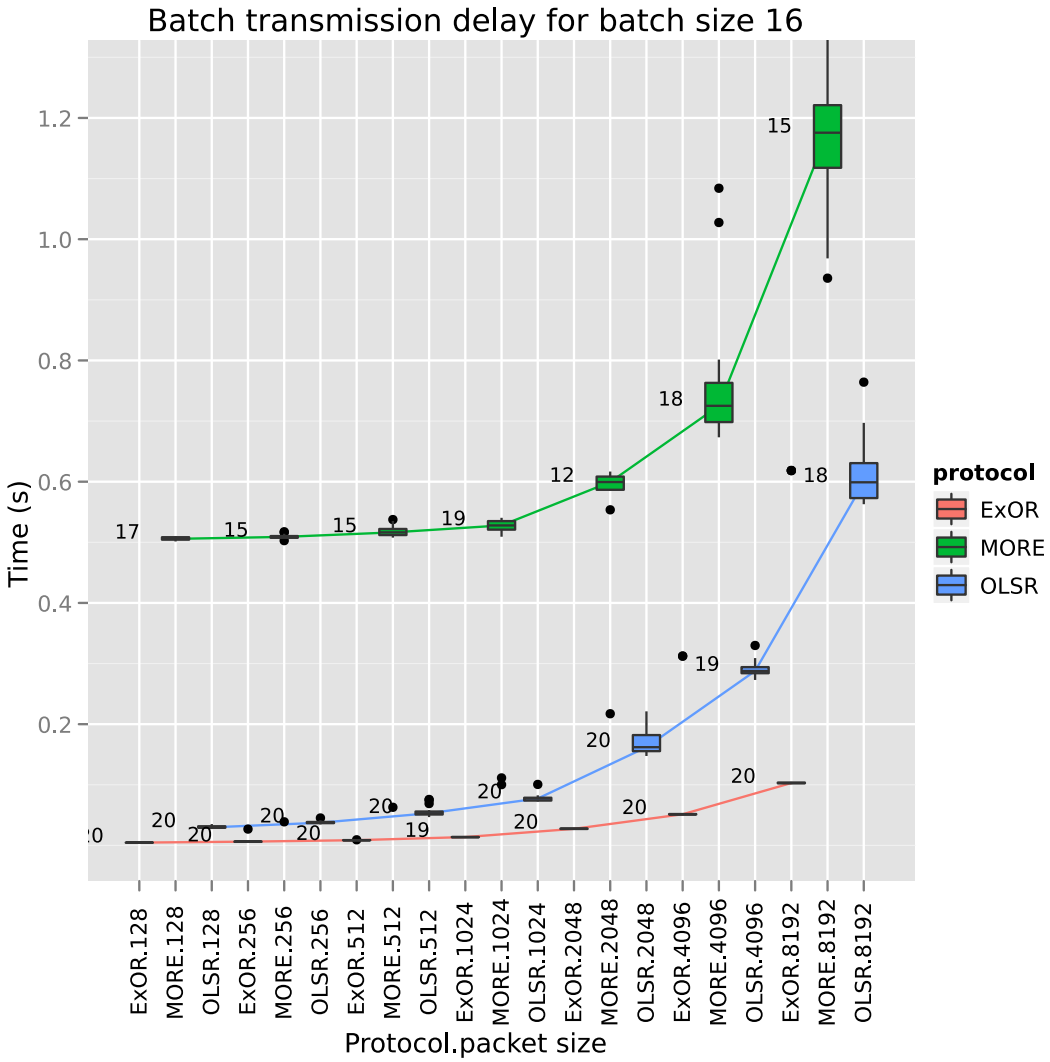

Figure 5.2: Plot comparing transmission delays of MORE, ExOR and OLSR for batch size 16

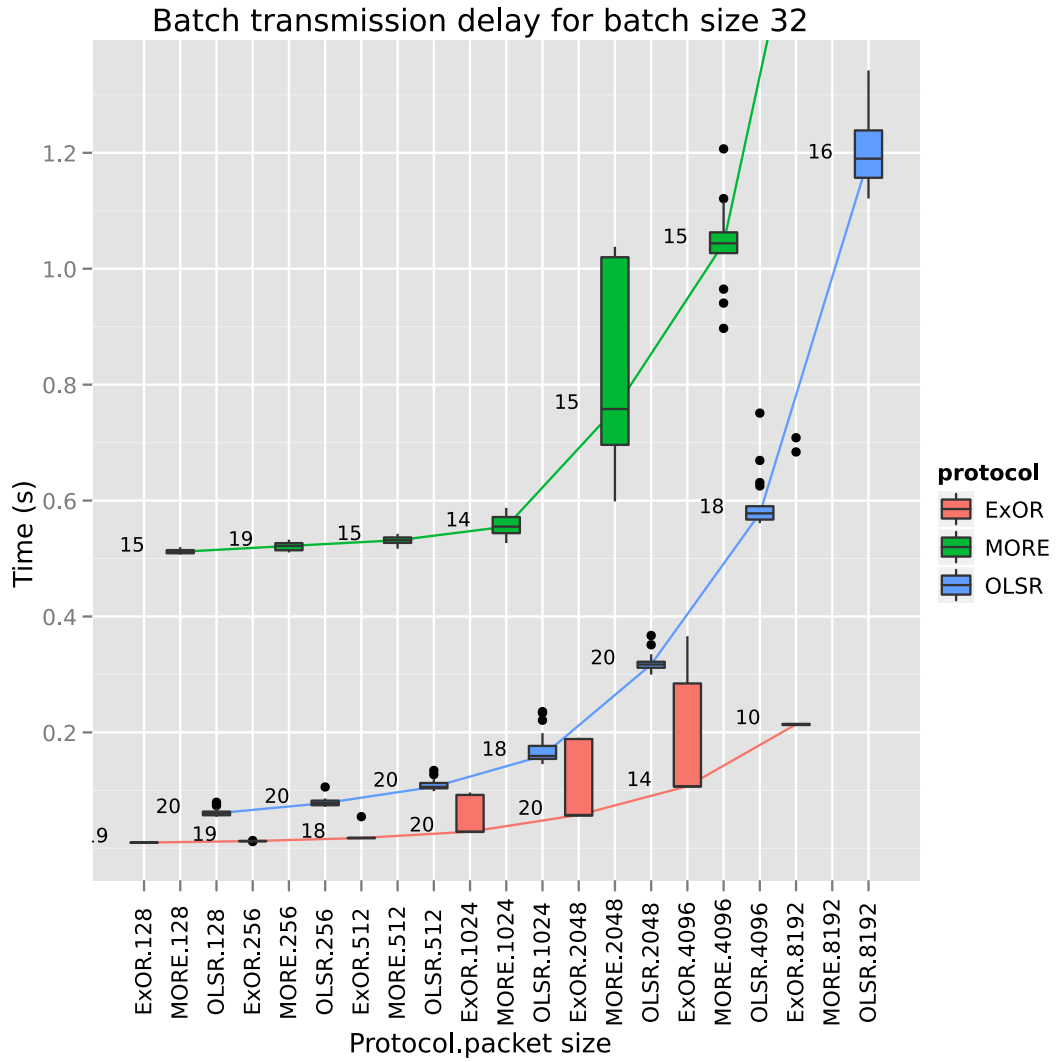

Figure 5.3: Plot comparing transmission delays of MORE, ExOR and OLSR for batch size 32

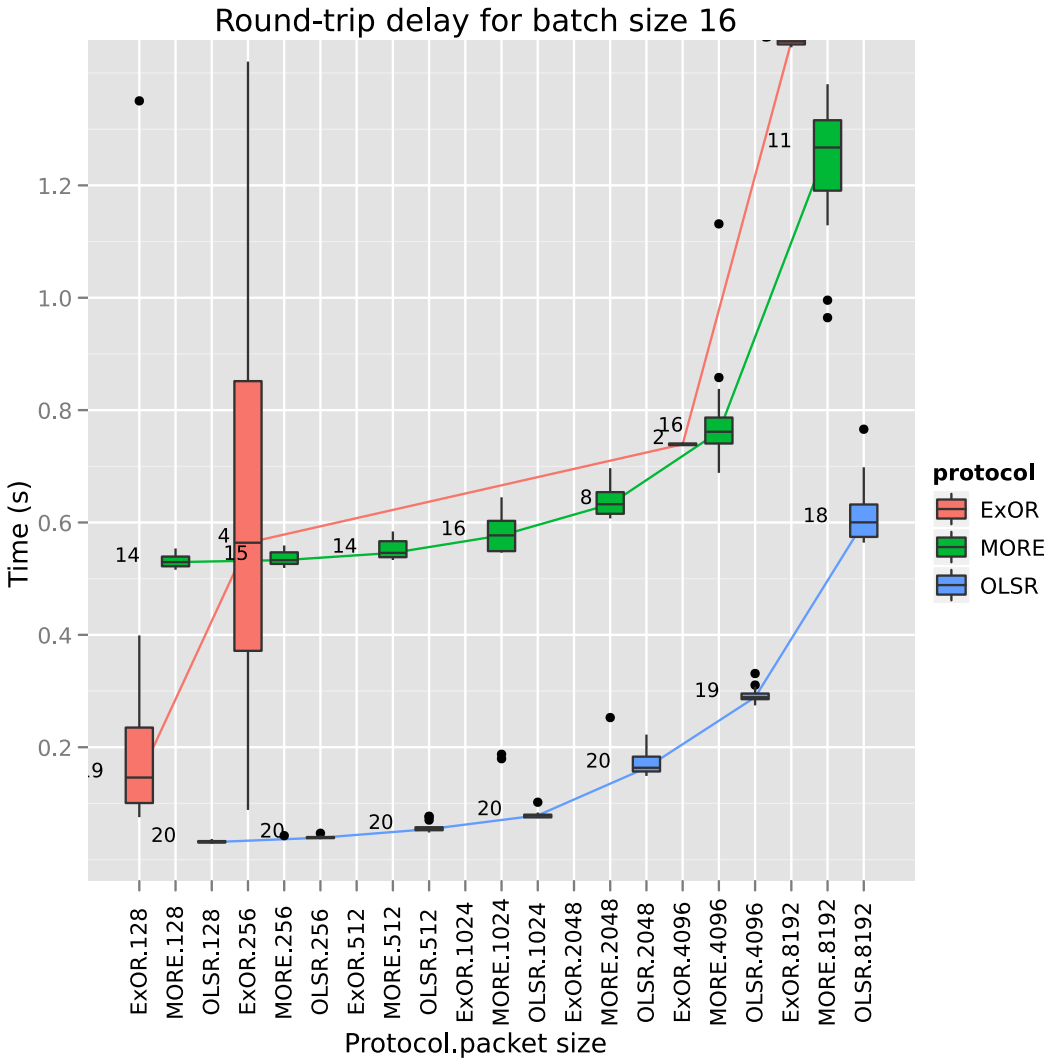

Figure 5.4: Plot comparing round-trip delays of MORE, ExOR and OLSR for batch size 16

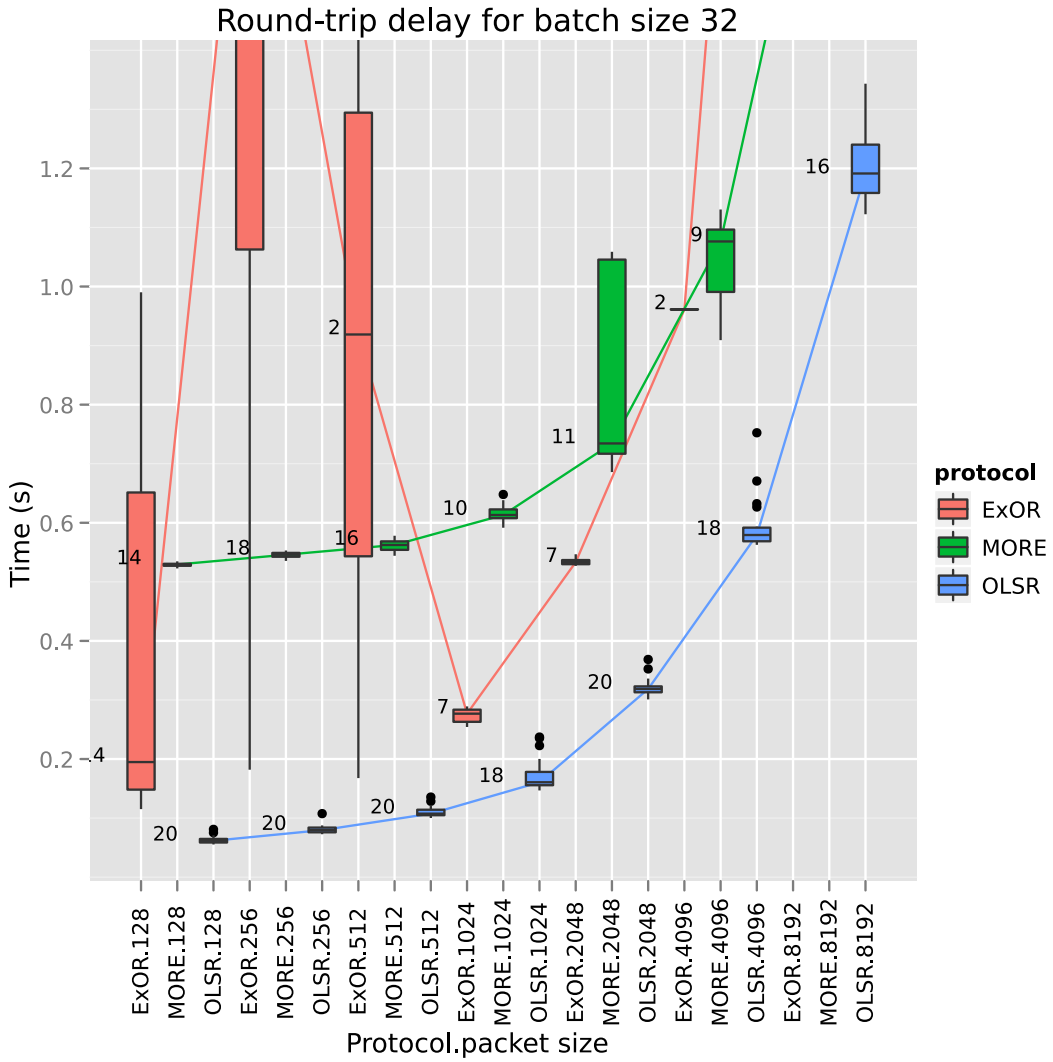

Figure 5.5: Plot comparing round-trip delays of MORE, ExOR and OLSR for batch size 32

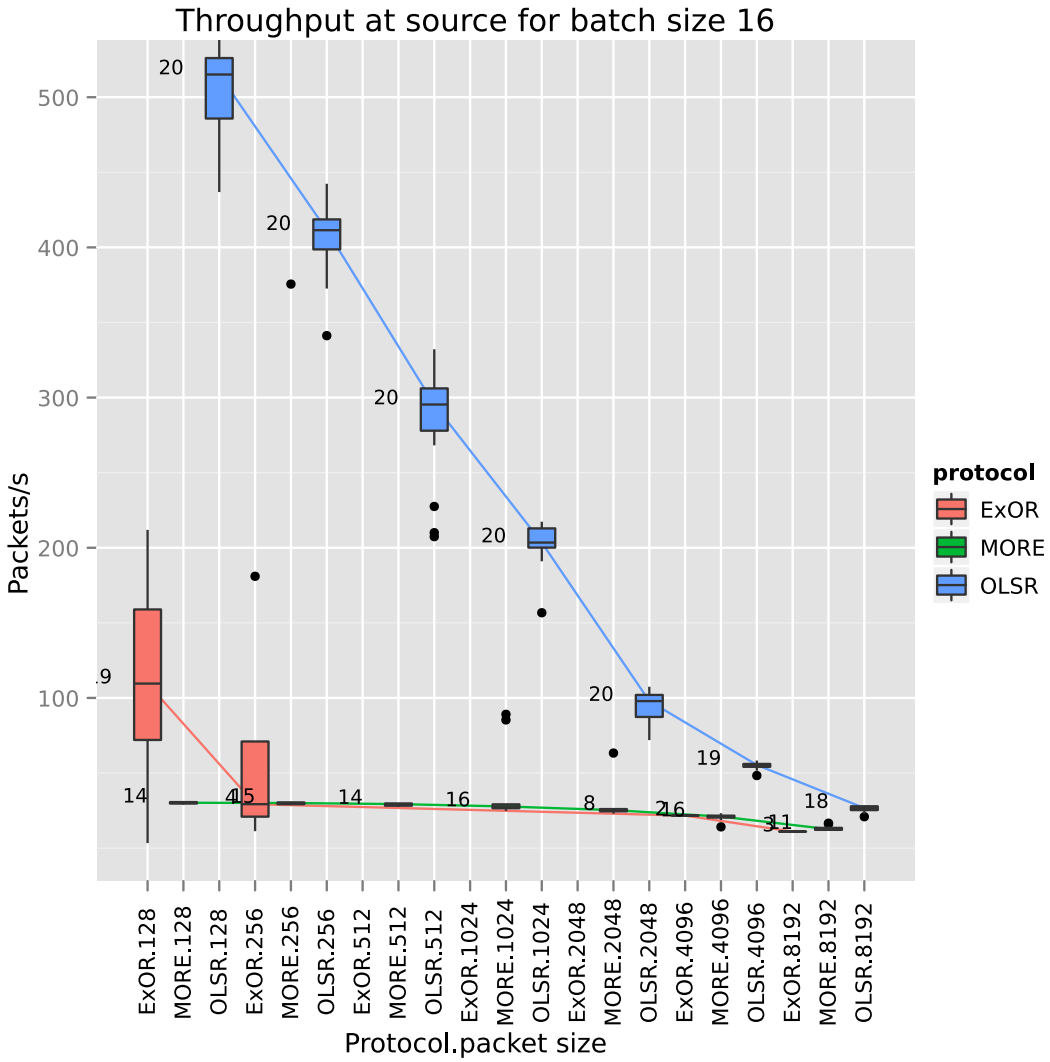

Figure 5.6: Throughput measurements for MORE, ExOR and OLSR for batch size 16

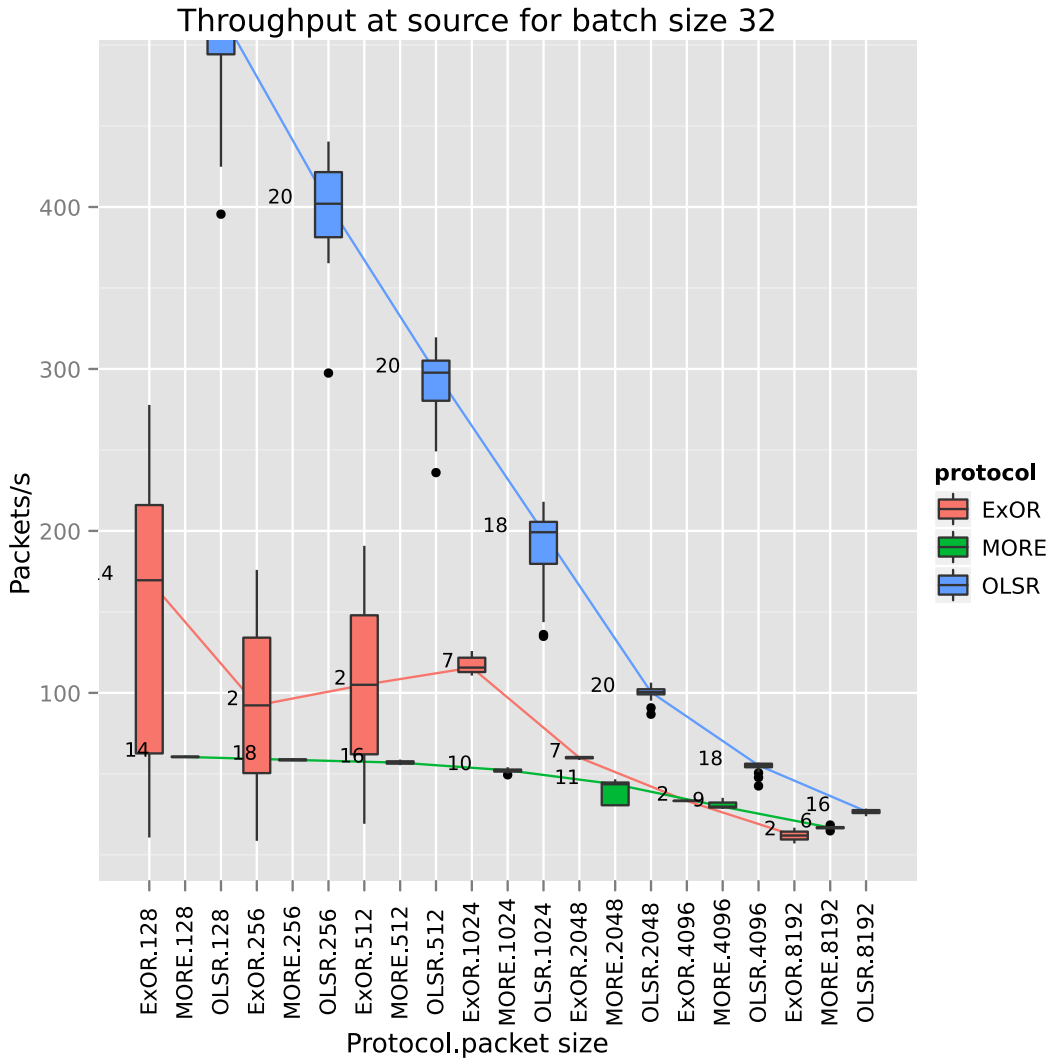

Figure 5.7: Throughput measurements for MORE, ExOR and OLSR for batch size 32

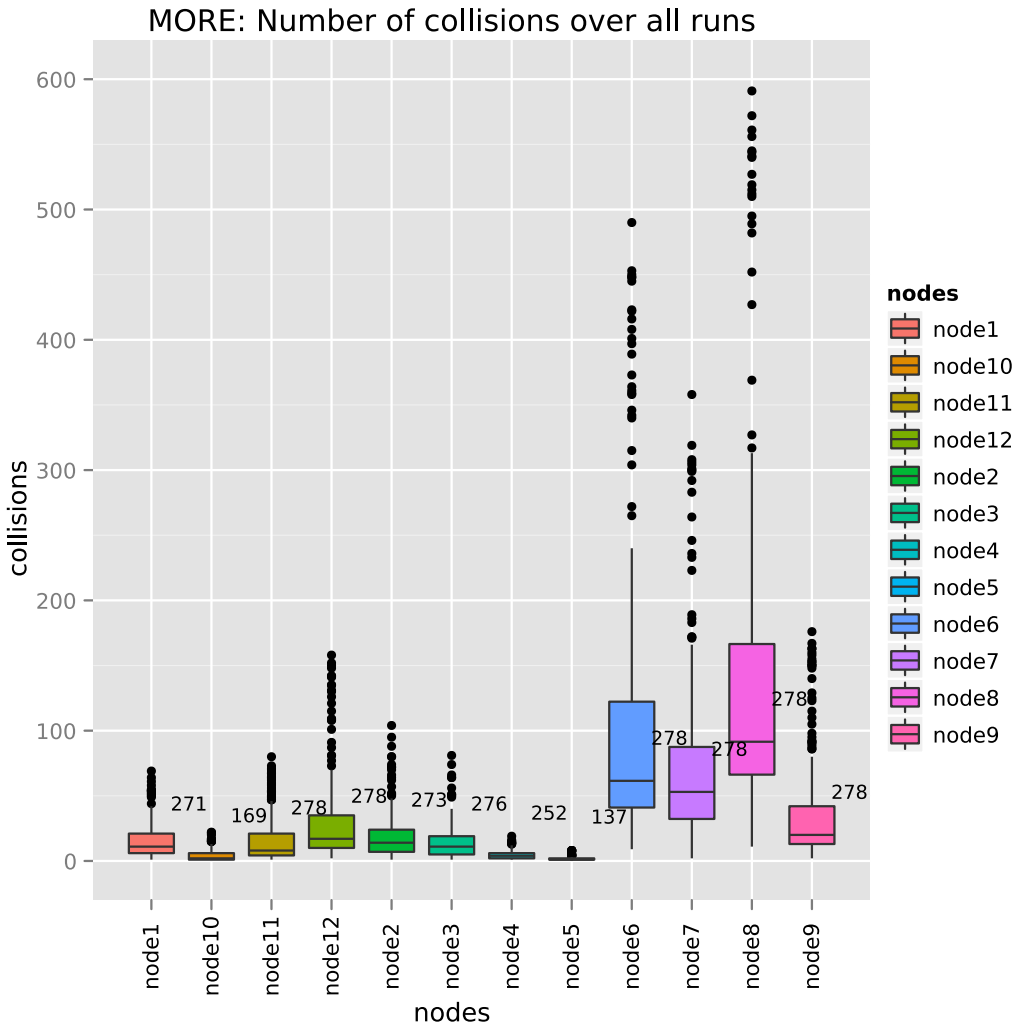

Figure 5.8: Number of collisions per node over 558 runs for MORE

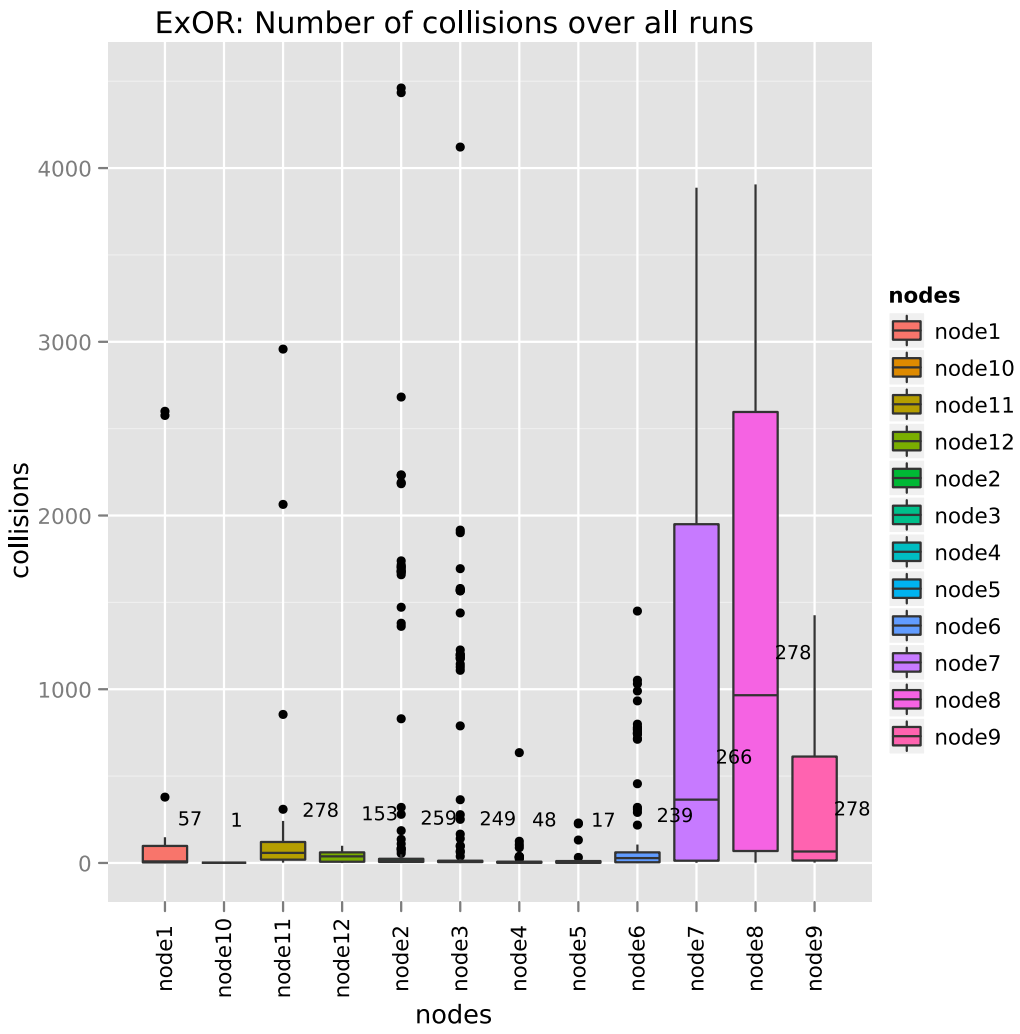

Figure 5.9: Number of collisions per node over 558 runs for ExOR

Chapter 6

Conclusion

6.1 Summary & Conclusion

In this thesis we implemented two opportunistic routing protocols MORE and ExOR in OMNeT++, evaluated their performance and compared them to OLSR. Unfortunately our results are not as promising as those in [2] and [3], where significant throughput gains were achieved over traditional routing protocols.

There might be multiple reasons for this outcome. First of all, the descriptions in [2] and [3] do not include all details of their implementations. Although for MORE there is even pseudocode for two algorithms, there are quite important things left unexplained, such as the "rough simulation" for reducing the forwarder set in ExOR (see 4.3.1). For MORE, some things are unclear, such as what a forwarder should do when its buffer is full or exactly which `tx_credit` should be added to a node's `credit_counter` upon receiving a packet. Furthermore, in the case of MORE's traffic control heuristic, the provided algorithm did not work as intended.

The second reason has to do with software architecture. For the usual implementation of OMNeT++ nodes, the routing protocols sit above the IP layer, not above the MAC layer, as in [3]. In MORE this keeps forwarders from overhearing ACKs, which would cause them to discard the current batch and thus decrease the medium's load. Furthermore, in our simulation environment, the protocols cannot react to collisions or find out if the medium is free, an ability which is mentioned in both [2] and [3].

Finally, we have to consider the fact that simulation-based evaluations do not necessarily align with real-world evaluations. After all, a simulation is always based on a model of reality and thus only an approximation. For example, in discrete event simulators, such as OMNeT++, spatial reuse is not possible because events can only happen one after another, not in parallel. Another problem is a simulation's sensitivity to even minor changes to its parameters (i.e. the start and end point or the range of a random delay), which can make all the difference between a failed and a successful run.

6.2 Future work

The more nodes try to access the medium, the more likely collisions happen. Figure 5.8 and Figure 5.9 clearly show that collisions occur very frequently during the simulation. An obvious way to tackle this issue would be to reduce number of nodes involved in forwarding packets, a claim that is supported by [2, p. 21–23]'s findings. Therefore future work should investigate methods to reduce the forwarder set in an attempt to reduce collisions. Future work should also investigate a cross-layer approach, which enables communication between the routing and the MAC layer. This would allow the routing protocol to learn about collisions and react accordingly.

Bibliography

[1] A. Tanenbaum and D. Wetherall, *Computer Networks*, 5th ed. Prentice Hall, 2011.

[2] S. Biswas, "ExOR: Opportunistic routing in multi-hop wireless networks." in *Proceedings of ACM SIGCOMM*, 2005. [Online]. Available: http://pdos.csail.mit.edu/papers/roofnet:biswas-ms/roofnet_biswas-ms.pdf

[3] S. Chachulski, "Trading structure for randomness in wireless opportunistic routing," Master's thesis, Massachusetts Institute of Technology, 2005. [Online]. Available: people.csail.mit.edu/szym/sm.pdf

[4] T. B. Zhongliang Zhao, "OMNeT++ based opportunistic routing protocols simulation: A framework," in *10th Scandinavian Workshop on Wireless Ad-hoc Networks (ADHOC '11)*, 2011.

[5] A. Keränen, J. Ott, and T. Kärkkäinen, "The ONE simulator for DTN protocol evaluation," in *SIMUTools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, 2009.

[6] A. Varga, "The ONMeT++ discrete event simulation system." in *Proceeding of European Simulation Multiconference*, 2001.

[7] O. Helgason and K. Jonsson, "Opportunistic networking in OMNeT++," in *SIMUTools '08: Proceedings of the 1st International Conference on Simulation Tools and Techniques*, 2008.

[8] D. S. J. De Couto, D. Aguayo, J. Bicket, and R. Morris, "A high-throughput path metric for multi-hop wireless routing," in *Proceedings of the 9th annual international conference on Mobile computing and networking*, ser. MobiCom '03. New York, NY, USA: ACM, 2003, pp. 134–146. [Online]. Available: http://doi.acm.org/10.1145/938985.939000