Technische Universität Berlin
Fakultät IV

Diplomarbeit in Informatik

Implementation and Evaluation of an Opportunistic Mesh Routing Protocol

Nadi Sarrar

Aufgabenstellerin:	Prof. Anja Feldmann, Ph.D.
Betreuer:	Dipl.-Inf. Harald Schiöberg
	Ruben Merz, Ph.D.
	Prof. Anja Feldmann, Ph.D.
Abgabedatum:	2. Juni 2009

Technische Universität Berlin
Deutsche Telekom Laboratories, INET
Research Group Prof. Anja Feldmann, Ph.D.

Die selbständige und eigenhändige Ausfertigung versichert an Eides statt
Berlin, den 2. Juni 2009

_____

Unterschrift

Abstract

Today's wireless networks usually use a single-path routing protocol, derived from wired networks. Opportunistic routing makes use of the multicast nature of wireless networks. The goal of this thesis is to implement and evaluate an opportunistic routing protocol for wireless mesh networks. Since TCP is greatly important for a major fraction of the expected traffic, this work will also consider the question about the extent, to which TCP can be supported.

Zusammenfassung

Derzeit wird bei Drahtlosnetzwerken üblicherweise ein single-path Routingprotokoll eingesetzt, ähnlich zu denen der kabelgebundenen Netzwerke. Die Idee von opportunistischem Routing ist es, Eigenheiten der drahtlosen Übertragungsvariante gewinnbringend auszunutzen. Im Rahmen dieser Diplomarbeit wird ein solches opportunistisches Routingprotokoll für drahtlose Mesh Netzwerke implementiert und evaluiert. Zusätzlich werden Probleme beim Einsatz von TCP besprochen, denn ein großer Anteil an dem zu erwartenden Datenaufkommen setzt auf TCP.

Contents

Chapter 1

Introduction

In recent years, Wireless Mesh Networks (WMNs, Section 1.3) have gained popularity. They facilitate a cost-effective and rapid way to supply large areas with network connectivity. Besides the commercial attractiveness, there exist community projects (Berlin Freifunk Network[1]) and research projects (MIT roofnet, [2]) exploring the challenges and benefits of WMNs.

With the Bowl Project (Berlin Open Wireles Lab[2]), we are currently building a three tier WMN testbed. The smaller two are indoor networks meant for testing and debugging routing protocols in early stages of development. The main tier will cover the whole campus of the TU Berlin and provide Internet connectivity to all members of the university. With that, we are constructing a WMN that enables researchers to do experiments under real conditions and with real user traffic.

A primary difficulty with WMNs is routing (Section 1.4). Nowadays, WMNs typically use a single-path routing protocol, derived from wired networks. Opportunistic routing is a novel approach for routing in WMNs. It takes advantage of the multicast nature of the wireless transport medium, rather than ignoring it. The goal of this thesis is to implement and evaluate an opportunistic mesh routing protocol. Experiments at the indoor testbed will be conducted to compare the performance and behaviour of the implemented routing protocol to that of simple shortest path routing. For that, a crucial part is a legitimate measurement of the given topology. Finally, since TCP [20] is greatly important for a major fraction of the expected traffic, this work will also name the problems facing with TCP (Section 5.4).

With this diploma thesis, I was effectively the first user working with the indoor testbeds, which are still considered experimental. Numerous challenging problems were expected and arose, on both the software and hardware side. I will mention the actual

[1] http://berlin.freifunk.net/
[2] http://bowl.net.t-labs.tu-berlin.de/

difficulties at the respective parts of this thesis. In the end, the work on this thesis was challenging, but also rewarding and a lot of fun.

The following introductory sections are meant to provide a brief overview on WMNs and their terminology. It should help to provide an idea about their architecture, their special characteristics and their common classification. In-depth knowledge about most of the lower level functionality of wireless networks is not necessary to understand this writing and hence lies out of scope. Please consider reading the IEEE 802.11 standards document [1] for studying the underlaying technology of this work.

Wireless networks can either have an infrastructure, as in IEEE 802.11 networks or cellular networks like GSM, or they can be infrastructureless, which is the case for WMNs. Before discussing the architecture of WMNs, IEEE 802.11 networks are examined with a description of their core elements.

1.1 IEEE 802.11 radio networks

IEEE 802.11 radio networks are managed infrastructure networks consisting of a number of stations (STAs). A STA is a device with a 802.11 compliant radio interface. There are two types of STAs, access points (APs) and clients. APs form the infrastructure of the network. The coverage of the APs is the Basic Service Area (BSA). APs are usually connected to other networks, e.g. to an enterprise network or to the Internet, via arbitrary data links like IEEE 802.3 Ethernet. They provide gateway functionality and manage the communication among clients, given that they are part of the same network. All STAs that are part of the same network belong to the same Basic Service Set (BSS), which is identified by a BSS Identifier (BSSID). APs can be configured to announce a BSS by transmitting beacons that include its BSSID at regular intervals. In multi access point networks the Extended Service Set (ESS) is used. Here, all the APs must share the same ESS Identifier (ESSID), but they can have individual BSSIDs. The APs are either connected to each other via Ethernet or via a Wireless Distribution System (WDS), which operates either in bridge or repeater mode. In every case, the network provides roaming capabilities for its clients.

Figure 1.1 shows a simple example of a single access point IEEE 802.11 network. The filled circle represents the cloud of the network, which in that case is the radio coverage of the AP. Let the AP be connected to the Internet via Ethernet and provide gateway functionality. Nodes S1 to S4 and the access point all are associated with the same BSS. Station S0 is not inside of the cloud and so cannot communicate with the AP and hence cannot use its gateway functionality, although S0 might be able to communicate well with S3, but since S3 will not forward any data packets, S0 stays offline.

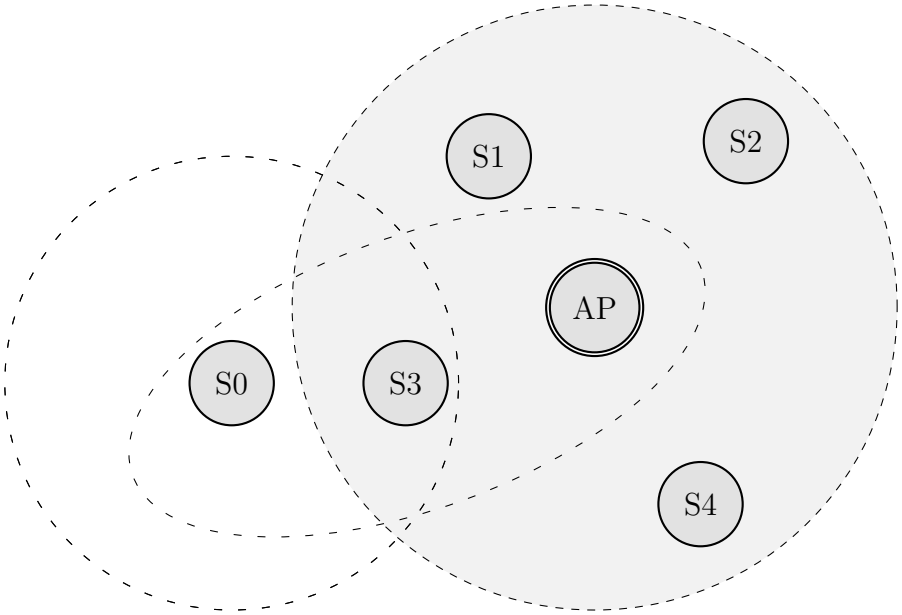


Figure 1.1: Managed wireless LAN network consisting of 6 stations. The filled circle around the access point (AP) represents its radio range. Client stations S1 to S4 are in range, S0 is not. The circle around S0 and the ellipse around S3 represent their radio ranges, respectively.

1.2 Mobile Ad Hoc Networks (MANETs)

In contrast to IEEE 802.11 managed networks, Mobile Ad Hoc Networks (MANETs) lack infrastructure. They are built spontaneously and they operate in a peer-2-peer fashion in that they have no central instances. All participators are at the same time the building blocks and the users of the network. An example of a trivial MANET is Bluetooth. The pairing functionality of the Bluetooth system allows the creation of a small one hop MANET with very limited coverage.

MANETs can also be built using the IEEE 802.11 radio technology. Here, all members of a MANET associate with the same Independent Basic Service Set (IBSS). They forward packets as a service for each other which makes it a multi-hop network. This enables a node to communicate with a destination, which is outside of its own radio range, provided that it can be reached by passing through other nodes of the same MANET. To accomplish that, a routing protocol is needed (Section 1.4). Since the overall quality of the network in terms of capacity, bandwidth, radio coverage and availability depends entirely on its participating nodes, it is a problem to maintain a certain quality of service. Besides that, a MANET does not provide a standard way to connect with other networks. These problems lead to Wireless Mesh Networks.

1.3 Wireless Mesh Networks (WMNs)

Wireless Mesh Networks (WMNs, [3]) are a subtype of MANETs. WMNs are infrastructureless as well, but they do not follow the straight peer-2-peer alike design in which all participating nodes are equal. In WMNs exist two types of nodes: Mesh routers and mesh clients. Mesh routers are stations dedicated to routing and other network management tasks. They often have multiple radio interfaces, some of which are used for mesh routing and others which are used for building a conventional IEEE 802.11 network with which mesh clients can associate themselves. Besides that, mesh routers may also allow clients to connect to the network via Ethernet or any other desired technology. Unlike clients, mesh routers are minimally mobile, they usually are always on-line and have no power consumption constraints. There are three types of WMNs: Infrastructure / Backbone WMNs, Client WMNs and Hybrid WMNs. The role of mesh clients varies with the type of WMNs.

Infrastructure / Backbone WMNs

In Infrastructure / Backbone WMNs, a set of mesh routers are deployed to build a WMN. The common radio technology is IEEE 802.11 but others may be used as well. Mesh routers are the only nodes that are taking active part in the WMN. Their positioning, radio capacity, and performance is crucial for the overall quality of the WMN. The resulting WMN forms a backbone network used to route user traffic. Users connect to the mesh nodes in wireless or wired ways and use its gateway and bridge functionality.

Client WMNs

In Client WMNs, the mesh clients take active part in packet forwarding and routing. This type of WMNs like MANETs, lacks infrastructure. They furthermore share the same characteristics, benefits and problems, see Section 1.2. The major difference is that Client WMNs are by definition multi-hop networks.

Hybrid WMNs

Hybrid WMNs are a combination of Infrastructure / Backbone WMNs and Client WMNs. Here, mesh clients can decide whether they want to take actively part in the WMN like mesh clients in Client WMNs, or whether they prefer to use it as a mesh client of an Infrastructure / Backbone WMN.

1.4 Problem: Routing in Wireless Mesh Networks

Routing in WMNs is a non trivial task. Common routing protocols that are used in today's wired networks, like OSPF [16], are not adequate for WMNs for several reasons. In wired networks, links usually are reliable and they remain available for long periods, which is not true for WMNs. OSPF would be overstressed by the frequent topology changes of a WMN, because that would lead to a high amount of management traffic like link state updates, which are flooded through the network. Besides that, wired networks also have a negligible rate of packet loss and they have a constant capacity. In WMNs, routing protocols must be designed to work well with a high amount of packet loss, time varying capacity and frequent topology changes. Furthermore, the characteristics of WMNs, especially the broadcast nature of its transport medium, can be exploited by routing protocols to gain overall performance.

Presently, the routing and packet forwarding in wireless mesh networks is a very active research field. There are a lot of publications inventing novel protocols or improving already published ones. Some different approaches have emerged and can be classified in various different manners[3]. Wireless mesh routing protocols can be proactive, reactive or a combination of both.

Proactive routing protocols constantly measure the topology and update the forwarding tables of all routers. The amount of traffic overhead that is needed for updating the forwarding tables as a consequence of topology changes is appropriate for networks with infrequent changes on its topology. The Optimized Link State routing protocol (OLSR, Section 2.1) is a proactive routing protocol.

Reactive routing protocols discover routes on demand. This fits well for scenarios where the nodes are highly mobile and the communication among them is initiated occasionally. Given that, reactive routing may have less overhead compared to its proactive companion. Routes are usually discovered by flooding route request messages and waiting for a route reply. The Ad Hoc On Demand Distance Vector protocol (AODV, Section 2.2) is a reactive routing protocol.

Hybrid routing protocols use proactive and reactive routing at the same time. They try to benefit from both approaches by selecting one best suited for each situation. The Zone Routing Protocol (ZRP, [12]) follows the proactive approach for routes within a local region ("zone") while determining routes to farther nodes reactively.

[3] http://en.wikipedia.org/wiki/Ad_hoc_routing_protocol_list

There exist, furthermore, an increasing number of novel routing protocols which differ from the conventional ones in other ways. So it is possible to further classify routing protocols by looking at whether a single or multi-path forwarding scheme is used.

Single-path routing protocols are closely related to protocols used in wired networks in terms of forwarding techniques. They specify a method to compute a next forwarding node and transmit packets in a unicast manner. Therefore, the path is determined before the actual transmission takes place. A good example of a single-path routing protocol is OLSR (Section 2.1).

Multi-path routing protocols try to exploit the characteristics of wireless mesh networks to gain performance and throughput. They make use of the broadcast nature of the wireless transport medium. Instead of plain unicast transmissions to a single predetermined next forwarding node, multiple nodes are being involved in the forwarding process at the same time. This is done by opportunistic routing protocols like ExOR (Section 2.3) and SOAR (Chapter 3). MAC-independent Opportunistic Routing (MORE, [5]) is also an opportunistic routing protocol, but with a different approach, it randomly reorders packets and makes use of network coding to increase throughput.

Chapter 2

Related work

This chapter provides a brief overview of currently available routing protocols which are designed to work well with WMNs. First, a prominent representative of proactive routing protocols, the Optimized link state routing protocol (OLSR, Section 2.1) is examined. Second, the reactive Ad Hoc On Demand Distance Vector protocol (AODV, Section 2.2) is presumed. After that, a fundamentally different approach is focused, the ExOR opportunistic multihop routing protocol (Section 2.3) which implements the multi-path routing approach.

2.1 Optimized link state routing (OLSR)

OLSR is a proactive table-driven link state routing protocol for WMNs, described in RFC 3626 [6]. As it is a proactive routing protocol, it needs to maintain full topology information on each mesh node. To accomplish that, it introduces a link state algorithm optimized for WMNs. In classical link state routing protocols like Open Shortest Path First (OSPF, [16]), flooding is used to spread topology information throughout the network. To optimize the overhead needed for message flooding by minimizing redundant retransmissions in the same radio range, OLSR nodes select Multi Point Relays (MPRs), a subset of its 1-hop neighbor nodes, that are exclusively retransmitting control messages. These control messages include:

HELLO messages are used for link sensing, neighbor detection and for selecting MPRs. They are broadcast periodically are are never forwarded.

Topology Control (TC) messages are broadcast and forwarded by MPRs to reach all nodes in the WMN. They contain link-state information sufficient for route calculation.

Multiple Interface Declaration (MID) messages are needed if nodes have more than one mesh network interface. They contain a list of all interface addresses used by a node to create a mapping of interface addresses and nodes.

OLSR maintains the forwarding tables of the mesh nodes and adds entries for every possible destination. Thus all routes are available at all times and there is no initial delay to construct newly established flows. OLSR fits well into scenarios where you have random and sporadic traffic with a larger number of nodes involved.

An implementation of OLSR is available as free software[1] and is currently run by various community mesh projects including the Athens wireless network[2] (approx. 2000 nodes) and the Berlin Freifunk network[3] (approx. 600 nodes).

2.2 Ad Hoc On Demand Distance Vector (AODV)

AODV is a reactive routing protocol for WMNs, described in RFC 3561 [17]. It determines unicast routes on demand, triggered upon encountering traffic bound to a destination node whose route is not yet known. Consequently, newly established connections incur an initial delay when utilising unknown routes.

To discover a route, a node transmits a route-request (`RREQ`) message, which needs to be flooded. AODV uses an expanding-ring algorithm to optimize message flooding. This algorithm utilizes the IP-header `TTL` (time to live) field by setting it to `TTL_START` for every `RREQ`, which limits the number of forwarding steps. If no route-reply (`RREP`) is received after a preconfigured timeout has elapsed, the `RREQ` is retransmitted with an incremented `TTL` value. This is done until the `TTL` value reaches a predefined threshold, from which point on it is set to `NET_DIAMETER`, which again is a configuration parameter, meant to provide the diameter of the WMN in hops.

Furthermore, AODV needs a way to avoid the Bellman-Ford counting-to-infinity problem to prevent routing loops. This problem is also known from traditional distance vector routing protocols like the Routing Information Protocol (RIP) and is explained in [15]. This kind of loop-freedom is guaranteed by maintaining sequence numbers for every route table entry. With that, every `RREQ` can be uniquely identified and the actuality of information can be determined.

[1] http://www.olsr.org
[2] http://wind.awmn.net/?page=nodes
[3] http://berlin.freifunk.net

AODV defines four control message types, which are exchanged using UDP:

Route Request (`RREQ`) messages are broadcast by a node to discover a route to a destination. This is either done if the destination is unknown, or if the route table entry for that destination is invalid or has expired.

Route Reply (`RREP`) messages are unicast by a node receiving a `RREQ` given that it is the requested destination or it knows about a valid route to it. Unicast is possible because every forwarding node caches a route back to the originator of the `RREQ`.

Route Reply Acknowledgement (`RREP-ACK`) messages are expected from nodes after receiving a `RREP` in case the latter has the acknowledgement request bit set. `RREP-ACK`s are used to test if a link currently is bidirectional.

Route Error (`RERR`) messages are sent whenever broken links of active routes are detected. `RERR`s are broadcast to the affected neighbors, if only a single neighbor is affected or if broadcast is inappropriate, the `RERR` is unicast. The neighbors in question are maintained in precursor lists, one for each route table entry. The precursors are nodes possibly forwarding packets on this route.

The AODV protocol is optimized in regard to processing and memory requirements. It adopts fast to changes in the topology of the WMN. The overhead in network utilization is kept low, but it has an initial delay for flows to destinations to which no route is known yet.

2.3 Opportunistic multihop routing (ExOR)

ExOR [4] can be considered the foundation for SOAR (Chapter 3), which also is an opportunistic multihop routing protocol. ExOR attempts to exploit the broadcast nature of WMNs to gain throughput. In contrast to the previously mentioned traditional single-path routing protocols OLSR and AODV, this multi-path approach does not compute a unicast path through the WMN before the transmission. Instead, it broadcasts packets and selects the next forwarding nodes opportunistically, among those which successfully received the packet. The problems that ExOR needs to solve include developing an agreement protocol to choose a forwarder. This must be robust in the sense that unnecessary forwardings are rare, the chosen forwarder should always be the one with the lowest remaining path cost to the destination and it should impose only low overhead. To achieve these goals, the agreement protocol of ExOR aggregates packets into batches and operates batch-wise.

The source node of a flow buffers packets bound to the same destination until a batch is complete. Before broadcasting the packets (all ExOR transmissions are broadcast), an ExOR header is added to the packet, right in between the MAC and the network header. Besides some version, size, fragment handling and checksum fields, the ExOR header contains the following metadata:

Batch ID Each batch is assigned a unique batch identifier. This batch identifier is included in the ExOR header of every packet to indicate the batch to which the packet belongs.

Packet Number The packet number is the offset of the packet in its respective batch.

Forwarder List The forwarder list is a list of possible forwarding nodes, ordered by priority. The highest priority node is the one metrically closest to the destination. All nodes need to have full knowledge of the topology and the link metrics for calculating this list.

Batch Map The batch map indicates per packet of a batch, which highest priority node successfully received the packet. Initially, the source node is the only receiver of each packet.

Upon packet reception, the node first checks whether it is included in the forwarder list, otherwise it drops the packet. It buffers packets of a batch until the batch's transmission cycle has ended. The highest priority node then forwards its received packets with an updated batch map. The lower priority nodes update their batch maps when overhearing transmissions. In descending order of priority, they then start to transmit packets that have not yet been acknowledged by a higher priority node. This cycle stops as soon as at least 90% of the packets of a batch have been received by the ultimate destination. For the remaining set of packets, traditional single-path routing is used.

Some necessary components required to implement and use ExOR in a real environment are left out in [4]. These include the process of link metric measurement and the traditional routing protocol used for the 10% fraction of a batch.

The evaluation section of [4] concludes that ExOR achieves a throughput gain of around 50% compared to traditional routing for most flows. The testbed was the MIT Roofnet [2], a scientific outdoor WMN consisting of 38 nodes with 802.11b radio interfaces distributed over an area of approximately six square kilometers.

Chapter 3

Simple opportunistic adaptive routing (SOAR)

Simple Opportunistic Adaptive Routing (SOAR, [21]) was published in September 2006 by Eric Rozner, Jayesh Seshadri, Yogita Mehta and Lili Qiu of the University of Texas at Austin. Much like ExOR, SOAR aims to exploit the characteristics of the wireless transport medium to gain overall performance over traditional single-path routing. SOAR advances ExOR, which is the seminal opportunistic routing protocol in order to achieve practical usefulness. ExOR outperforms traditional single-path routing significantly in terms of throughput, which is its primary design goal, however it does not support multiple simultaneous flows, which is an important requirement for the purpose of using it in real networks.

We select SOAR for this thesis, because it does not rely on special hardware or driver features. Furthermore, it supports multiple flows and thus seems to be well suited for experiments regarding practical usefulness. Being an opportunistic routing protocol, SOAR also is a good candidate for evaluating the possible benefits that the opportunistic approach promises.

In this chapter, the theoretical benefits of opportunistic routing are explained (Section 3.1), followed by a detailed description of the SOAR protocol (Section 3.2). To describe the protocol, the metric is explained first (Section 3.2.1), followed by a description of the proposed packet forwarding and acknowledgement systems and the role of the various different timers (Sections 3.2.2 to 3.2.4). Finally, to further exemplify how the components of SOAR cooperate, a routing example is given (Section 3.2.5).

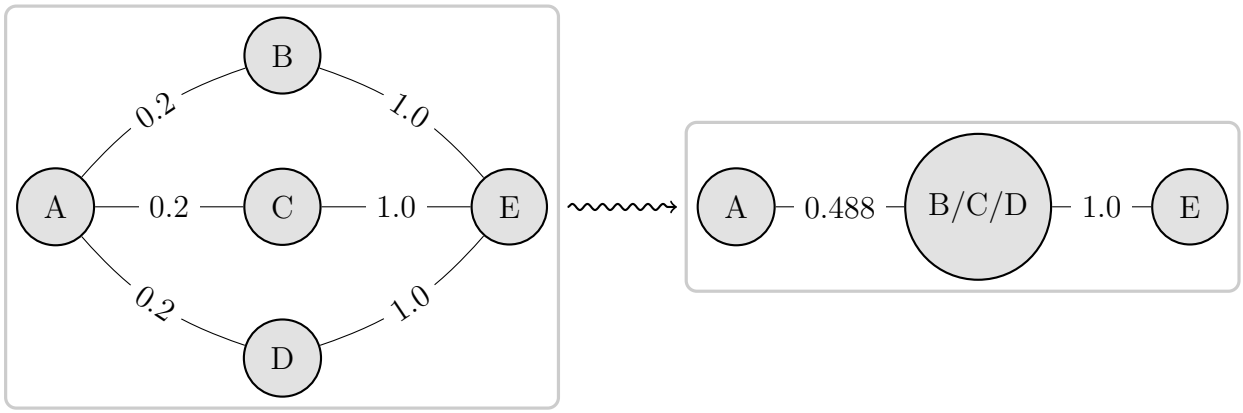


Figure 3.1: Example network topology graph with delivery probabilities shown along the edges to visualize the benefit of strong virtual links.

3.1 Benefits of opportunistic routing

SOAR aims to improve routing in WMNs by implementing the opportunistic approach (Section 1.4). Routing paths are not calculated in advance. Instead, SOAR defers the selection of the next hop to after the actual data transmission completed. With that, the following two promising benefits over traditional single-path routing protocols are expected:

Strong virtual links Opportunistic routing protocols can utilize multiple weak links simultaneously to achieve a behaviour similar to that of one strong link. Figure 3.1 gives an example of such a situation. The expected number of transmissions for a packet from A to E using a traditional routing protocol is 6, regardless of which intermediate node is selected as a relay. With the opportunistic approach, we can take advantage of the success rate, with which at least one of the intermediates receive the packet, which is $1-(1-0.2)^3 = 0.488$. With that, only $1/0.488 \approx 2.049$ transmissions are necessary on average for the packet to be received by one of the intermediates, and $3,049$ for the end-to-end route. That gives opportunistic routing a theoretical benefit of twice the throughput of traditional routing, given the example topology of Figure 3.1.

Lucky long transmissions Opportunistic routing can utilize links, that are weak enough so that traditional routing would have to ignore them, but at the same time have a great amount of progress regarding the route to the destination. Consider a linear topology as shown on Figure 3.2. Here, traditional routing would most probably compromise in favor of the two hop route from A to C with B relaying the packets, although there is a very weak but direct link to the destination. Opportunistic routing utilizes the two hop route and the direct link simultaneously, resulting in a

lower average count of transmissions. Traditional routing needs $1/(0.25+0.25) = 8$ transmissions on average. To calculate the expected total number of transmissions with opportunistic routing, let's first determine how many transmissions node A is going to perform. Let $NX(n)$ denote the expected number of transmissions by node n, and let $n \rightarrow m$ entail that m received a transmission from n. Given that the probabilities of $A \rightarrow B$ and $A \rightarrow C$ are independent, which already is questionable with wireless networks, the expected number of transmissions by node A then is:

$$\begin{aligned} NX(A) &= \frac{1}{P(A \rightarrow C) + P((A \rightarrow B) \wedge \neg(A \rightarrow C))} \\ &= \frac{1}{P(A \rightarrow C) + P(A \rightarrow B) * (1 - P(A \rightarrow C))} \\ &= \frac{1}{0.1 + 0.25 * 0.9} \approx 3.08 \end{aligned}$$

With that, we scale the probability of the path ABC in order to obtain the portion of packets that follow that path:

$$\begin{aligned} P(pathABC) &= NX(A) * P((A \rightarrow B) \wedge \neg(A \rightarrow C)) \\ &\approx 3.08 * 0.25 * 0.9 \approx 0.69 \end{aligned}$$

Further, let's calculate the expected number of transmissions by node B:

$$\begin{aligned} NX(B|pathABC) &= \frac{1}{P(B \rightarrow C)} * P(pathABC) \\ &\approx \frac{1}{0.25} * 0.69 \approx 2.77 \end{aligned}$$

Now, we can calculate the expected total number of transmissions with opportunistic routing:

$$NX(A) + NX(B|pathABC) \approx 3.08 + 2.77 \approx 5.85$$

This thesis is meant to evaluate, whether these benefits hold in practical uses, i.e. real networks. To the best of our knowledge, evaluations of opportunistic mesh routing

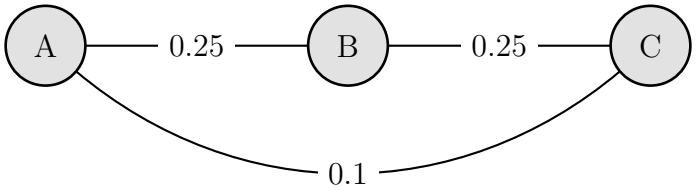


Figure 3.2: Example linear network topology graph with delivery probabilities shown along the edges to visualize the benefit of lucky long transmissions.

protocols in real networks have rarely been done. Experiments in that area were mostly done by simulating networks. Considering the large amount of factors that may influence wireless data transmission, one cannot rely completely on the outcome of a simulation. There are most likely a lot of factors that are ignored by simulations. Therefore it is an interesting experiment to evaluate how well opportunistic routing performs in reality.

3.2 Protocol description

SOAR is a straight forward representative of opportunistic routing protocols for WMNs. Route selection, i.e. the selection of the next forwarding node is deferred to after the actual data transmission. SOAR adds various meta information to the routed data packets for realizing its priority based agreement protocol necessary for selecting forwarding nodes. Beyond that, it is designed to provide best-effort reliability by adding packet based acknowledgements and retransmissions. The amount of reliability can be increased or decreased by increasing or decreasing the maximum number of retransmissions, respectively. If full end-to-end reliability is required, upper-layer protocols that provide a reliability guarantee like TCP [20] should be used on top of SOAR (Section 5.4).

While ExOR (Section 2.3) provides a foundation, SOAR differs significantly regarding the proposed forwarding technique. On the one hand, SOAR does not combine a number of packets into batches before initiating the opportunistic routing procedure, nor does it require traditional routing at any point. Instead, SOAR follows the idea of opportunistic routing at anytime, even for just one single data packet. Both protocols share the idea of priority based forwarding by adding a list of candidate nodes in order of priority as meta information to the routed packets, while the actual methods are different. Full topology information is needed for both, so scalability issues apply. As the aim of this thesis is to evaluate the idea of opportunistic routing in general, and not to produce a scalable solution ready for wide-spread deployment, this concern is considered out of scope.

Routing with SOAR works as follows. Consider a packet appearing at one of a set of mesh nodes running the SOAR protocol. If the destination of the packet is not the node itself, the shortest path is calculated with Dijkstra's shortest path algorithm [9]. Then, a

list of forwarding nodes ordered by priority is calculated and added to the packet. The highest priority node is the one with the lowest remaining path cost to the ultimate destination. Receivers drop packets unless they are part of the included forwarding list. The highest priority node immediately forwards the packet, while the others start forwarding timers based on their priority. If a node overhears the forwarding of a packet by a higher priority node for which it has a forwarding timer running, it cancels its timer and drops its copy of the packet. Or else, as soon as the forwarding timer elapses, it forwards the packet by itself.

Prior to each packet forwarding, the forwarding list is recalculated. As opposed to that, the shortest path is only determined once by the originating mesh node, and then remains unchanged during the whole routing process. This prevents routes from diverging.

For a complete understanding on how SOAR works, one must learn about the metric that is used, how forwarding lists are calculated, how the respective timers are set, and how packet acknowledgements work (Sections 3.2.1 to 3.2.3). After that, a complete walkthrough of a SOAR routing procedure for a single packet is given.

3.2.1 Metric

The metric that is being used in the original SOAR paper is the Expected Transmission Count (ETX) [7, 10], however several other metrics should work as well. ETX predicts the number of (re)transmissions needed to successfully deliver a packet over a link in either direction. Let d_f and d_r be the delivery ratios of both the forward and the return direction of a link, respectively. The ETX of a link then is:

$$ETX = \frac{1}{d_f * d_r}$$

The ETX of a path through a network is the sum the ETX of all passed links.

3.2.2 Forwarding list calculation

The algorithm for obtaining the forwarding list is different for nodes that are part of the shortest path, and nodes that reside nearby. Consider the calculation of the forwarding list from the current node `cur` to the ultimate destination `dest`. Let `ETX(a,b)` be the ETX of the link `a-b`, and let `pETX(a,b)` be the total ETX of the shortest path from `a` to `b`.

At first, let's examine the algorithm that is used if `cur` is part of the shortest path, because besides being more simple, it is essential for the subsequent algorithm.

```
1 forwarders = ()
2 for each node i in topology
3 do
4   if pETX(i,dest) < pETX(cur,dest) and ETX(curr,i) < threshold
5     add i to forwarders
6   fi
7   prune(forwarders)
8 done
```

With that, a node of the topology is added to the forwarding list, if and only if two conditions match (line 4). First, the path ETX to the destination from the candidate node has to be less than that from the current node. Second, the ETX of the link from the candidate to the current node must be lower than a given threshold. This threshold should be made configurable. At last, the forwarding list is pruned (line 7), meaning it is ensured that the link ETX of all node pairs are also within the before mentioned threshold. The prune method itself is not covered in the original SOAR protocol specification.

Now, the following algorithm is used whenever `cur` is not on the shortest path.

```
1 forwarders = ()
2 closest = none
3 for each node i in shortest path
4 do
5   if ETX(cur,i) < ETX(closest,i)
6     closest = i
7   fi
8 done
9 if pETX(closest,dest) < pETX(cur,dest)
10  add closest to forwarders
11 fi
12 for each node i in forwarding list of closest
13 do
14   if pETX(i,dest) < pETX(cur,dest) and ETX(curr,i) < threshold
15     add i to forwarders
16   fi
17 done
```

Here, lines 3 to 8 seek for the node on the shortest path, that is metrically closest to `cur`. If `closest` is at the same time closer to the ultimate destination, then it gets added to the list of forwarders (lines 9 to 11). Finally, in lines 12 to 17, the forwarding list of `closest` is obtained using the previously examined algorithm. From that list, all nodes that are both closer to the destination and within threshold to `cur` are added to the new list.

Section 3.2.5 goes through a complete SOAR routing procedure, which includes examples of forwarding list calculations. But before that, SOAR's packet acknowledgement system and the employed timers are introduced.

3.2.3 Hop-by-hop packet acknowledgements

SOAR uses hop-by-hop acknowledgements to provide best-effort reliability. For every forwarding of a packet, an acknowledgement is expected. If no acknowledgement is received within a timeout, the packet is retransmitted. This is repeated until a maximum number of retransmissions is reached.

Selective acknowledgements are being used to protect against negative effects due to loss of acknowledgements. With that, all recently received packets are acknowledged with a single acknowledgement, which in turn contains a bit field to flag received packets. Therefore, if an acknowledgement of a packet goes lost, the subsequent acknowledgement repeats that information and prevents from needless retransmissions.

There are multiple ways in which acknowledgements can be sent. Note, that all SOAR packets are broadcast, hence any outgoing packet can carry acknowledgements for any other packet. Further note, that forwarded packets can acknowledge themselves. Right before transmitting a packet, a node checks if there are acknowledgements waiting to be sent. If so, they are piggybacked to the packet, if there is enough room. If there are no regular packet forwardings occurring within an acknowledgement timeout (Section 3.2.4), stand-alone packets are created and transmitted. These carry nothing but acknowledgement information. Last but not least, there are implicit acknowledgements, that is, if a node overhears the forwarding of a packet for which it expects an acknowledgement.

3.2.4 Timers

SOAR uses three kinds of timers for various different purposes.

Forwarding timer The forwarding timer delays a packet to be forwarded, if the current node is not on highest priority. Let the priorities start at 0 (highest priority) ascending, then the forwarding timer will be set to expire after $t_{fw} = p * \delta$, where p is the priority and δ is the time it takes for queueing and transmitting a packet.

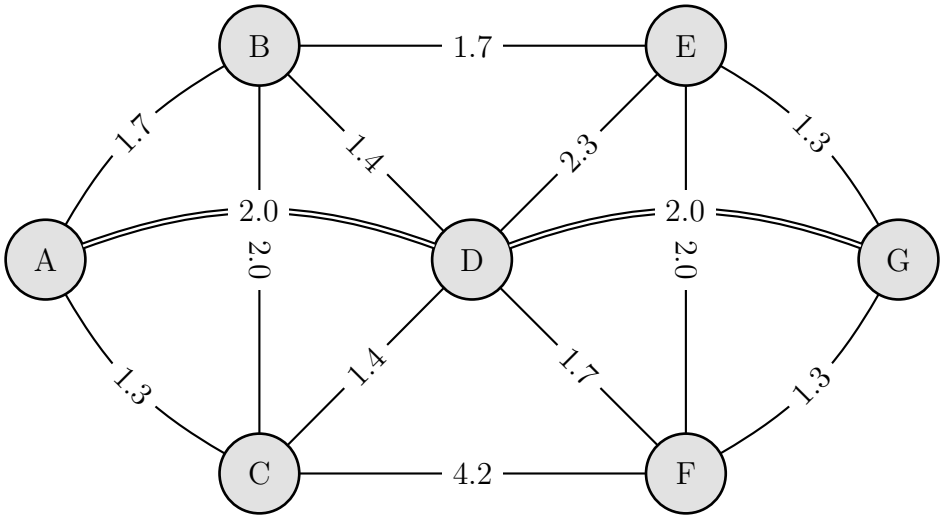


Figure 3.3: Example network topology graph with ETX shown along the edges, the double lined edges visualize the shortest path between A and G.

Retransmission timer The retransmission timer represents the due time for acknowledgements to arrive. If no acknowledgement is received before this timer elapses, a retransmission is initiated. This timer is set to expire after $t_{rt} = length(forwarders) * \delta$, where $forwarders$ is the current forwarding list and δ again is the time it takes for queueing and transmitting a packet.

Acknowledgement timer As an optimization, packet acknowledgements are not sent immediately after transmission of the respective packet, because this would make a lot of small stand-alone acknowledgement packets necessary, with additional protocol overhead. Instead, acknowledgements are delayed, hoping for the possibility of piggybacking them to a regular packet that is transmitted before the timeout elapses. If that fails, acknowledgement compression is being used, i.e. all delayed acknowledgements are efficiently accumulated into a single stand-alone packet.

3.2.5 Routing example

In this section we will proceed through a complete SOAR routing procedure. Figure 3.3 illustrates the topology including all available links with their respective ETX values. Remember that every node has full topology knowledge. Let the example traffic be a single packet originating from A with the destination G.

At first, node A calculates the shortest path, which is emphasized in the figure: A-D-G. After that, it needs to determine a list of forwarders. Let the ETX threshold be set to 2.0. Hence, the considered nodes are B, C and D, because they are within the ETX threshold to A, and because they also pass the check for being closer to the destination

than the current node A, path ETX wise. So they are all added to the forwarding list. The prune operation does not change the list, because the link ETX of all pairs of nodes in the forwarding list are within threshold. The forwarding list is complete: D-B-C, in descending order of priority. After adding the shortest path and forwarding list meta information to the data packet, the actual transmission can take place. At the same time a retransmission timer is started with a timeout value of $length(forwarders) * \delta = 3 * \delta$.

Assuming that only B and C receive the packet, both start a forwarding timer according to their priorities. The forwarding timer of B will elapse first, so B continues forwarding the packet. A and C receive that transmission, take it as an implicit acknowledgement and stop their efforts regarding that packet.

Before B actually forwards the packet, it needs to build a new forwarding list. Since B is not on the shortest path, it first finds the node on the shortest path which is metrically closest to itself, which happens to be D. Because D passes the test for being closer to the ultimate destination G, it is the first node added to the list. From the forwarding list of D, which includes F and G, none are taken because these are not reachable from B and thus not within threshold.

Finally, assuming that D receives the packet, D forwards the packet immediately with a forwarding list containing F and G. D's transmission will once more implicitly acknowledge the packet at B. Further assumed is that G receives the packet from D, it sends a stand-alone acknowledgement which is expected from D and F.

Based on this example, it seems legitimate to argue that the complexity of SOAR is high and the amount of protocol overhead is probably non negligible, hence possibly undoing the promising theoretical benefit of opportunistic routing. Therefore, realistic experiments are of interest. The remainder of this thesis covers a prototype implementation of SOAR and the evaluation in a real network.

Chapter 4

Protocol implementation

This thesis includes a prototype implementation of the SOAR protocol (Chapter 3). This implementation differs from the original specification of SOAR [21] in some parts. Section 4.1 describes the differences and the reasons for them. As a software framework, we decided to employ the Click modular router (Section 4.2). After an introduction into Click, the design of the SOAR element is examined.

4.1 Modifications and extensions

The original specification of SOAR [21] in some parts is not sufficiently explicit and thus can not be implemented right away. There are open questions concerning the handling of sequence numbers, how packet acknowledgements are supposed to proceed in detail and how parts of the forwarding list calculation algorithm are meant to be implemented. Furthermore, support for a wider range of metrics is added by providing support for directed link costs.

4.1.1 Metric

The implementation of SOAR is largely metric independent. Any metric based on unsigned integers should work fine without source code manipulations or the need for recompilation, as long as the smaller value out of two is the one with the lesser cost, and the sum of two values is the cost for the respective path. The ETX metric is based on real numbers, but can still easily be applied by converting it using a plain multiplication and a type cast operation. The reason for not using float values at any part of the implementation is that it may be compiled as a part of the Linux kernel, where float is not available as a data type. Furthermore, the implemented version of SOAR can work with directed weights, i.g. costs of the forward and the backward direction of a link may differ.

For the evaluation, we will use a directional metric (Section 5.2.1).

4.1.2 Sequence numbers

A proper sequence number scheme for SOAR should provide support for multiple flows. By our definition packets belong to the same flow, if and only if they share the same source and destination addresses. To provide support for multiple flows, sequence numbers need to be network-wide unique. Additionally, SOAR sequence numbers should occur in ascending order, separately for each flow. The latter is necessary for maintaining sequence number windows attached to flows.

This can be achieved by using a combination of flow identifiers and packet identifiers as sequence numbers. The flow identifier contains the identifiers of the source node and that of the ultimate destination. Because of the inclusion of the node self-identifier, every node has a network-wide unique space of sequence numbers. This is legitimate, because node identifiers in this case are simply IP addresses and therefore already unique for all reasonable network configurations. The implementation uses the last two octets of IPv4 addresses as node identifiers, hence supporting slash 16 networks with a maximum of $65,536$ nodes (not including the special broadcast and network addresses of IP networks). The packet identifier is a two byte unsigned integer value as well, hence allowing a maximum of $\frac{65,536}{2} = 32,768$ packets in flight for each flow.

4.1.3 Acknowledgement handling

Besides omitting selective acknowledgements, the implementation complies mostly with the original protocol specification. Thus, packet forwardings themselves are implicit acknowledgements, acknowledgements can be piggybacked to packets and they can be sent stand-alone (Section 3.2.3). Furthermore, there were three important questions regarding acknowledgement transmission:

Who sends acknowledgements? All nodes that are part of the packets forwarding list acknowledge packet reception.

Should acknowledgements be forwarded? This is not proposed by the original protocol specification hence the answer is no.

Acknowledge packets after forwarding? A node that is on highest priority, thus meant to forward the packet right away, could omit transmitting an acknowledgement, because the forwarding already is an implicit acknowledgement. In the implementation this is configurable, see Appendix A for a complete table of configuration options.

Open questions regarding handling of incoming acknowledgements arose as well:

Which nodes must act on which acknowledgements? An acknowledgement must meet certain conditions before a node is allowed to honor it. First of all, the node must have a copy of the packet, that is referenced by the acknowledgement, in one of its queues. These include the queue for retransmissions, i.e. the packet has been recently transmitted by the node, and the forwarding queue, i.e. the node is not on highest priority of the forwarding list.

What to do with incoming acknowledgements? The node first checks if the originator of the acknowledgement is closer to the destination than itself, path cost-wise. If this is true, all efforts regarding this packet are terminated. Else, if it is an implicit acknowledgement, another acknowledgement is sent to let the farther node know that it should cease its efforts.

4.1.4 Forwarding list calculation

The calculation of the forwarding list is implemented according to the pseudo code snippets given in Section 3.2.2. Two parts are not specified in detail. First, the algorithm of the prune method is left open. Second, a size limitation of the forwarding list is proposed, in order to limit the maximum forwarding delay. Recall, that the retransmission timer is based on the length of the forwarding list (Section 3.2.4). How to realize this limitation is left unclear as well.

Prune For each ordered pair of nodes from the forwarding list, the direct link cost is obtained via the `Cost()` function (line 3). If it is greater than the threshold (= `FWLIST_THRES`, Appendix A), the node with the lower priority is removed from the list (lines 5 to 9):

```
1  for each node i in forwarders ; do
2    for each node j != i in forwarders ; do
3      if Cost(i,j) < threshold
4        continue ; fi
5      if Priority(i) < Priority(j)
6            remove i from forwarders
7      else
8            remove j from forwarders
9      fi
10   done
11 done
```

Size limitation To limit the size of the forwarding list, the lowest priority nodes are removed until the size of the forwarding list is within limit. This limit in turn is a configuration parameter (`FWLIST_LIMIT`, Appendix A).

When it came to the evaluation (Chapter 5), problems concerning high CPU utilization arose. To determine the parts of the code where optimization would make sense, the CPU profiler of Google perftools[1] was consulted. This made clear that a high amount of CPU time is spent for forwarding list calculations which is reasonable because of the complexity of the algorithm as it must be executed once for every single packet. To get around this problem, caching of forwarding lists is added. This also reduces memory consumption because every forwarding list is allocated never more than once. A use counter helps with deallocations. With every topology change, this cache is cleared. This cache made the forwarding list calculation disappear as a performance critical part from the Google perftools analysis.

4.2 Click modular router

The Click modular router [14] is a software project by Eddie Kohler et al. and was originally developed at the Laboratory for Computer Science at MIT. The first public release was in October 1999. Click implements a flexible system for building software routers. The core blocks of Click routers are elements, packet processing modules with any number of inputs and outputs. These elements usually take care of simple tasks, e.g. decrementing the IP headers `TTL` field. A Click router consists of many of these simple elements, plugged together as defined in a Click configuration file. With that, one can build complex routers in a very convenient modular manner. For example, a standards-compliant IP router can be build using not more than sixteen Click elements.

We chose Click as a software framework for implementing the SOAR protocol, because it offers a fully functional packet processing system, which makes it unnecessary to care about things like how to get or put packets from or to a device, respectively. Besides that, Click is easily extensible by offering a well-engineered C++ programming interface for implementing custom elements.

The following sections explain the functionality of Click elements (Section 4.2.1), describe the design of the SOAR Click element (Section 4.2.2), and list implementational difficulties (Section 4.2.3).

[1] http://code.google.com/p/google-perftools/

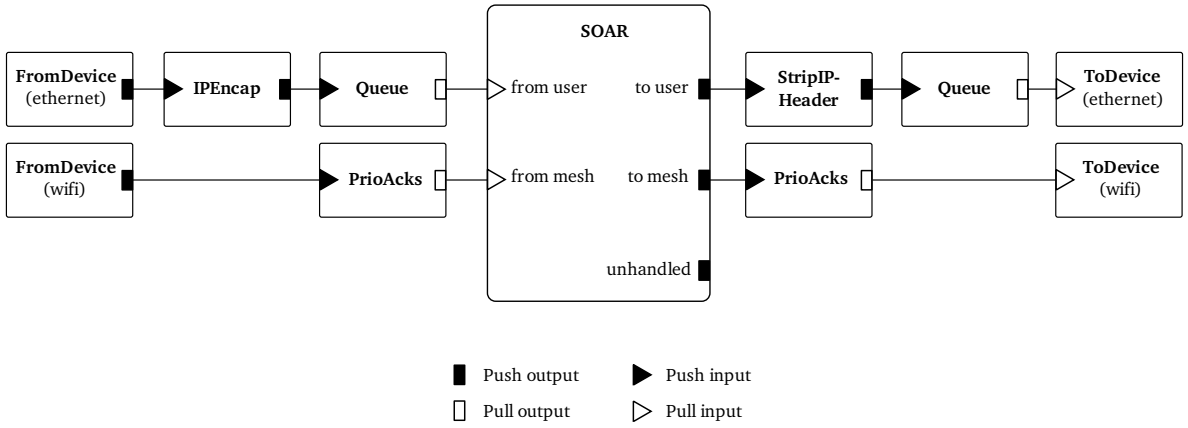


Figure 4.1: Schematic of the SOAR Click element.

4.2.1 Click elements

Click elements are used to construct packet forwarding paths and are meant to serve only simple tasks, each. A well populated list of such elements is distributed along with Click[2].

Parameters are used to configure elements. For example, the element that encapsulates packets in an IP header (`IPEncap`) uses element parameters for the source and destination address, amongst others.

Besides that, Click elements offer read and write handlers. Element handlers are accessible from other elements within a Click configuration. They also can be made accessible through TCP or Unix domain sockets. Handlers can serve arbitrary purposes, e.g. live reconfiguration of element parameters, obtaining statistics, halting and resuming packet processing and much more.

Input and output ports of Click elements can be of either push, pull or agnostic port varieties. With push connections, packet transfer is initiated by the source element, while with pull connections, the destination element triggers packet transfer. Agnostic ports can be both, push or pull. Which connection type is being used for a particular agnostic port is determined during router initialization. Packets coming from a network device are usually pushed into the system by the `FromDevice` element, while packets going to be transmitted by a network device are usually pulled by the `ToDevice` element. Therefore an element that converts the connection kind from push to pull is needed somewhere along the forwarding path, this can simply be a `Queue` element.

4.2.2 SOAR as a Click element

The schematic of the SOAR Click element is shown on Figure 4.1. SOAR mainly has two pairs of inputs and outputs, one for user and one for mesh packets. The output port

[2]http://www.read.cs.ucla.edu/click/elements

`unhandled` is used to eject all packets, that cannot be routed for whatever reason, e.g. the size of the packet prohibits from adding SOAR meta information or no path to the destination is available. If left unconnected, the default policy for the `unhandled` port is to silently drop the packets.

Figure 4.1 also shows an example of how the SOAR element may be used in a Click configuration. For clarity, elements that are needed to really run this configuration, but which are not important for understanding the routing part of this setup, are left out. These include elements for handling IP checksums, ARP [18] and Ethernet packet encapsulation.

At first, let's have a look at how the user packets are pushed into the router and how they are delivered to the ultimate destination. The respective `FromDevice` and `ToDevice` elements are bound to an Ethernet device, which in turn is connected to a load generator used for experiments (not shown on the figure). The load generator simulates the client side. This configuration realizes a backbone WMN (Section 1.3), by having disjunct IP address ranges for clients and mesh nodes. With that, we need to encapsulate packets coming from clients with an IP header with addresses from the mesh address space. The source address is the IP address of the mesh node that accepts the packet from the client. The destination address is the IP address of the mesh node, to which the destination client is connected. Therefore, a dynamic client location lookup service becomes necessary, as clients come and go and move between mesh nodes. For this thesis, such a lookup service is not relevant as the experiments do not include any aspects of client mobility. A statically parameterized `IPEncap` element as illustrated on Figure 4.1 serves the need. If a packet reaches its final mesh node, i.e. the one to which the destination client is connected to, it is pushed out of the `to user` output of the SOAR element. Prior the final delivery, the additional IP header is removed by the `StipIPHeader` element.

The mesh ports of the configuration (`wifi` device), are basically directly connected to the SOAR element, but some prioritization is done in between. `PrioAcks` is a Click compound element, which is a preconfigured chain of elements that can be used multiple times in the configuration, just like a normal element. The circuit of `PrioAcks` is shown on Figure 4.2. It basically prioritizes stand-alone acknowledgement packets as desired by the SOAR protocol specification. It is used on both the incoming and the outgoing side of the SOAR routing element.

The SOAR Click element provides various configuration parameters for environmental setup like packet size limit (`MTU`, Appendix A) and for controlling the behaviour of the SOAR protocol (e.g. timeout values like `ACK_TIMEOUT`). Besides that, there are various SOAR element handlers (Appendix B). These element handlers are used for handing over topology information to the SOAR element, to issue queries concerning the topology for

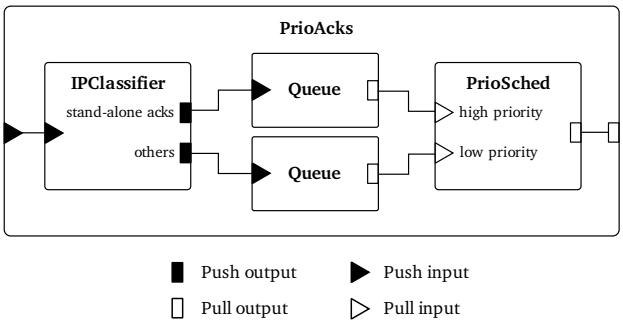


Figure 4.2: Schematic of the `PrioAcks` compound element.

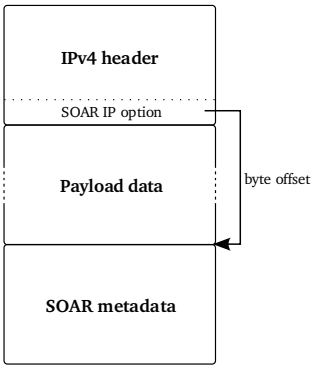


Figure 4.3: How SOAR meta data is appended to the packet.

maintenance and debugging (e.g. shortest path and link cost), and furthermore to obtain statistics like the number of packets seen on each input and output port.

Regarding the SOAR meta data embedding, we decided to append all necessary information to the end of a packet (Figure 4.3). To mark the beginning of SOAR meta data, we add an IP option to the IP header [19] providing the offset to the SOAR meta data.

4.2.3 Implementational difficulties

While implementing the SOAR protocol as a Click element, I faced two bugs in the timer implementation of the Click core. One of which caused the Click process to utilize 100% of CPU time, the other one triggered a rather infrequent segmentation fault. I proposed fixes to the Click mailing list for both problems, and thanks to Eddie Kohler the bugs were quickly fixed in the upstream version of Click.

100% CPU utilization This reproducible bug caused Click to spin, rather than block, when no timers were scheduled. The trigger of this problem is a faulty calculation of the timeout value, that is being passed on to one of `poll()`, `select()` or `kevent()`, depending on the target system. [3]

[3] http://pdos.csail.mit.edu/pipermail/click/2009-April/007927.html

Infrequent segmentation faults This bug was harder to reproduce, since it only occurs under high load and extreme usage of Click timers. Timer handler functions are allowed to delete (or unschedule) other timers, thus destroying timer objects. Under unhappy circumstances, the memory areas of previously destroyed timer objects are still dereferenced, which lead to attempted access of invalid memory, forcing a segmentation fault. [4]

[4] http://pdos.csail.mit.edu/pipermail/click/2009-April/007926.html

Chapter 5

Evaluation

At T-Labs and as a part of the Bowl Project (Berlin Open Wireles Lab[1]), we are currently working towards a three tier WMN testbed. One of the three is a mesh network consisting of nine nodes, all located side by side in the same room, thus all nodes are in radio range to each other. This testbed, we call it Smoketest, is designed for software debugging in early stages of development. Another testbed, we call it Indoor, consists of nine nodes, that are located at different places over two floors. This is designed for more sophisticated debugging and evaluation tasks, since not all pairs of nodes can reach each other. Finally, the Outdoor mesh network will cover the whole campus of the TU Berlin and provide Internet connectivity to all members of the university, enabling researchers to do experiments under real conditions and with real user traffic. For the evaluation part of this thesis, the Indoor testbed is used, since the Outdoor mesh is not ready yet.

After describing the experimentation environment itself (Section 5.1), a detailed description of how we measure the topology follows, which is a crucial task necessary prior to each of the actual experiments. After that, experimentation results are presented (Section 5.3). Finally, we examine problems which apply when it comes to TCP (Section 5.4).

5.1 Experimentation environment

The Indoor WMN testbed serves as the experimentation environment for the measurements within the scope of this thesis. It consists of nine mesh nodes distributed over the 16th (five nodes) and 17th floor (four nodes) of the Telefunken building, the home of T-Labs. Figure 5.1 pictures the floor plans with markers at the locations of the nodes along with their node identifiers.

[1] http://bowl.net.t-labs.tu-berlin.de/

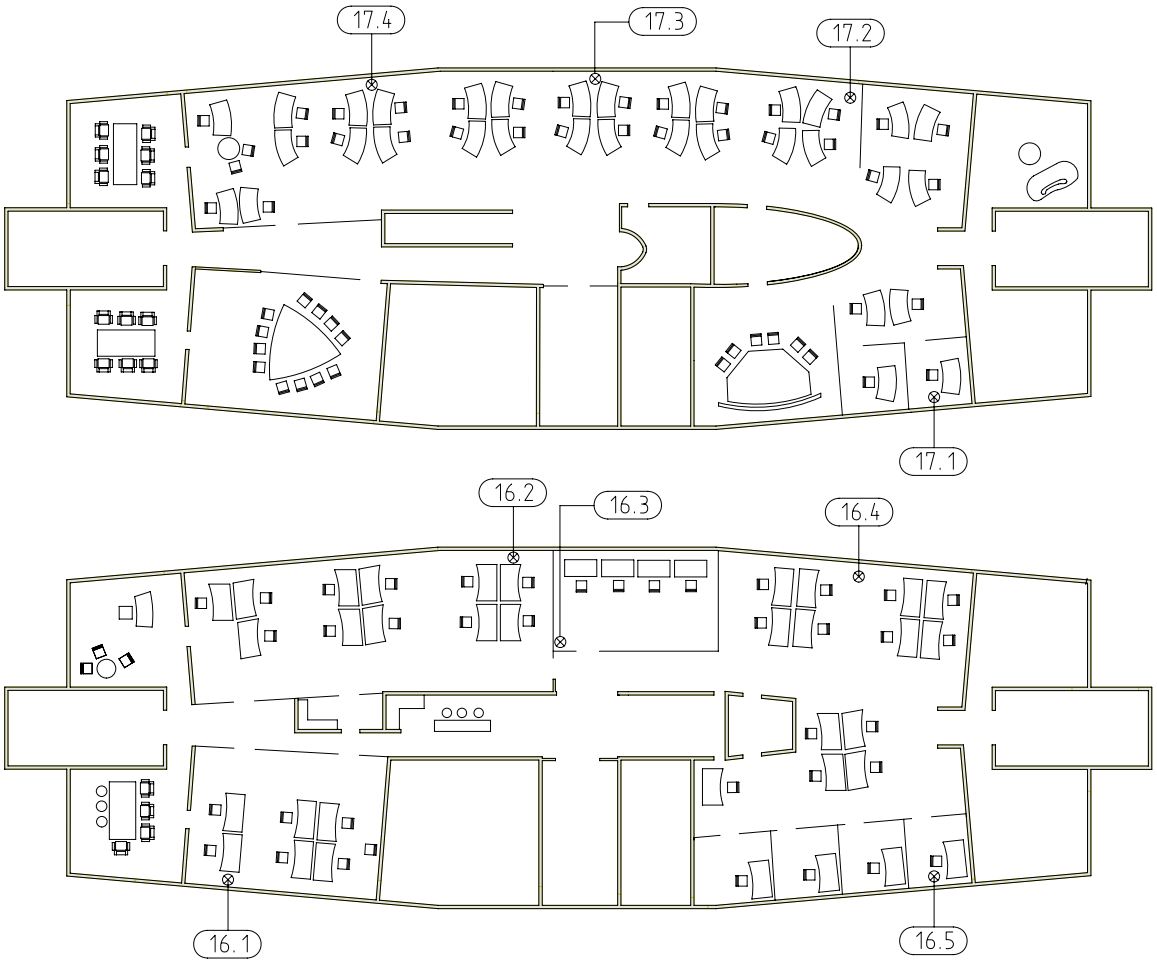


Figure 5.1: 17th (top) and 16th (bottom) floor of the Telefunken building, the home of T-Labs. Node locations are marked.

The nodes themselves are ARM-based embedded systems running a Linux operating system. The embedded platform we use is the Avila GW2348-4[2], which is a single board system built around a 533MHz Intel IXP425 XScale processor. They include 256 MBytes of SDRAM, but due to a driver bug, we can only use 64 MBytes. Besides two 10/100 Base-TX Ethernet ports, there are four type III mini-PCI slots, two of which we equipped with the common Winstron CM9-GP 802.11 a/b/g mini-PCI module[3]. For setup and configuration, monitoring and load generation the nodes are connected via Ethernet (100 MBit/s, full duplex) to our research network.

The Linux operating system we use is a customized version of the OpenWRT kamikaze release[4]. Our customization efforts primarily include various patches to support our platform and to include our version of the Click modular router including the SOAR protocol implementation.

[2]http://www.gateworks.com/products/avila/gw2348-4.php
[3]http://www.wneweb.com/Datacom/more/CM9_GP.htm
[4]http://www.openwrt.org

5.2 Topology Measurement

SOAR does not include measurement of the network topology, although it needs to have a full weighted graph of the present topology of the WMN. So what becomes necessary is a method for obtaining weights for the various links with time varying capacity different for both directions. We decide to do the measurement offline, i.e. during the measurement, the nodes are exclusively clogged by the measurement process. Our decision in favour of offline measurement is based on the fact that our nodes are stationary, i.e. the links are expected to have rather limited capacity variation over shorter periods of time. Also, we want to avoid extra management traffic that could have an impact on the results of the performance evaluation. Furthermore, we want to avoid a possible impact of the workload traffic on the measurement process hence its results. For example, good but congested links caused by a heavy workload should not be rated bad just because of their congestion state.

So what we ultimately strive for are link weights that provide a most stable view on the present topology. Sudden link changes should be covered by opportunistic routing anyway. The desired method for measuring the topology therefore has two major requirements:

1. The measurement results must be solid enough so that evaluation results are most likely not affected by an inapplicable view of the topology.

2. The measurement process must be fully automated so it can be redone at any time with only little effort.

We perform the whole topology measurement right before every single experiment to make sure that the topology information used are always up-to-date. That leads to the second requirement, which is realized easily by implementing a set of scripts for measurement automation.

In the following sections, the metric we have chosen to use is described first (Section 5.2.1). After that, the respective measurement process is explained (Section 5.2.2). Finally, to get an idea about the accuracy of the measurements, the respective confidence intervals are determined.

5.2.1 Metric

A suitable metric must be found. We chose to use a metric based on Expected Transmission Count (ETX) [7, 10]. Recall that ETX predicts the number of (re)transmissions needed to successfully deliver a packet over a link in either direction (Section 3.2.1):

$$ETX = \frac{1}{d_f * d_r}$$

With that, ETX handles extremely asymmetric links by simply avoiding them. Since the implementation of the SOAR protocol can work with separate weights for each link direction, it makes sense to prefer using a directional link cost metric, let's name it Directional Expected Transmission Count (DTX):

$$DTX = \frac{1}{d}$$

In this writing, DTX denotes the expected number of (re)transmissions for a directed link. The DTX of a route through the network again is the sum of all single-hop DTX that are part of the route.

5.2.2 Measurement process

The measurement is done by periodically broadcasting probe packets at a fixed rate from one selected master node while counting correctly received packets on all slave nodes. This is repeated so that every node becomes master once. It is important that the probe packets are broadcast, because there are no link-layer retransmissions for broadcast packets in IEEE 802.11. Note, that unicast packets in IEEE 802.11 demand acknowledgements and incur retransmissions if needed. For the purpose of measuring the topology, we use the following configuration:

Probes The probe packets are UDP packets each with a total IP length of 1 KByte, because most of the experimental flows use data packets of the same size.

Packet rate Probe packets are broadcast at a rate of 100 packets per second.

Physical rate The broadcast rate of the wireless interfaces is set to 24 MBit/s. The reason for that is given in Section 5.3.1.

Duration The duration for each iteration is five minutes. Considering the size of the confidence intervals for the medians, this is appropriate.

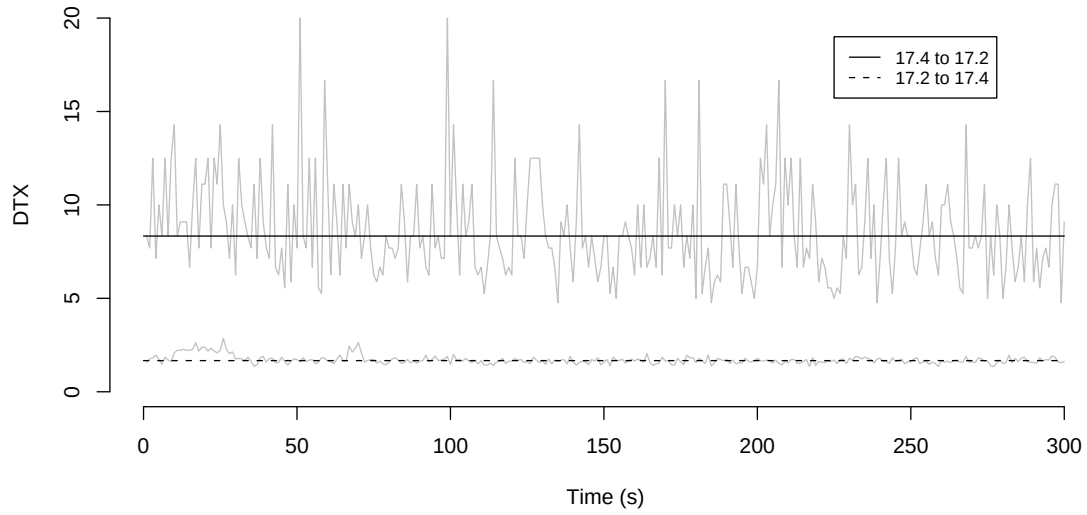


Figure 5.2: DTX values collected by measuring two directions of the same link. The straight horizontal lines are the medians of the respective DTX values, which in turn are shown in light grey.

On all slave nodes, the arrival times of the correctly received packets are remembered. For each time slice of one second, DTX is computed:

$$DTX = \frac{1}{d} = \frac{probes\_sent}{probes\_received}$$

The final DTX we use for a directional link is the median of all 300 DTX values we retrieve from the one second time slices. We prefer to use the median and not the average, because we want to avoid short term peaks to have a greater impact. Figure 5.2 shows the DTX values collected from two directions of the same link. Combining all the information retrieved out of each iteration, the result we get is a weighted directed graph well suited for SOAR.

5.2.3 Validation

To add some veracity to our topology measurement results, i.e. to validate that our medians are close to the true value with high probability, we will now determine the respective confidence intervals. A confidence interval is the uncertainty margin of summarized data due to the randomness of the measurements. It is meant to give some statement about the accuracy of the summarized data.

First, we assume that our collected data is independent and identically distributed (iid), which greatly simplifies the process of obtaining the respective confidence intervals. That assumption is valid, because our measurement experiments are designed to deliver iid values in the way, that there is little to no dependence between two different points in the measurement. That is, because one broadcast of a probe packet does not affect the delivery probability of its subsequent or preceding one due to the selection of the packet rate and the absence of retransmissions or acknowledgements. Furthermore, we are not aware of any hidden factors that could affect our measurements in a misleading or destructive way. This is by design of our measurement process, in which all probe packets are siblings and broadcast at a constant rate, all nodes are built using the exact same hardware, the software versions and configuration options are the same for all nodes as well. Note, that because this is wireless, we can not be sure that we are aware of all relevant hidden factors, but we tried to do everything that was possible. For example, we let all experiments run during nighttime, where the interference with regular wireless activities assumably is lower than at daytime.

The confidence interval can now be obtained as follows. Let $X_1, \ldots, X_n$ be the set of our collected DTX values. Moreover, let $X_{(1)} \leq X_{(2)} \leq \cdots \leq X_{(n)}$ be the same values sorted in increasing order. Now we determine the indices j and k, so that our confidence interval is $[X_{(j)}, X_{(k)}]$. For the median and a confidence level of $\gamma = 99\%$, j and k can be approximated:

$$j \approx \lfloor 0.50n - 1.288\sqrt{n} \rfloor$$
$$k \approx \lceil 0.50n + 1 + 1.288\sqrt{n} \rceil$$

In our case, n equals 300, so we get:

$$j \approx 127$$
$$k \approx 174$$

With that, we gain confidence that the true value resides within that interval, with a probability of 99%. Accordingly, Figure 5.3 shows the medians and the respective confidence intervals calculated from the measurement data already shown on Figure 5.2.

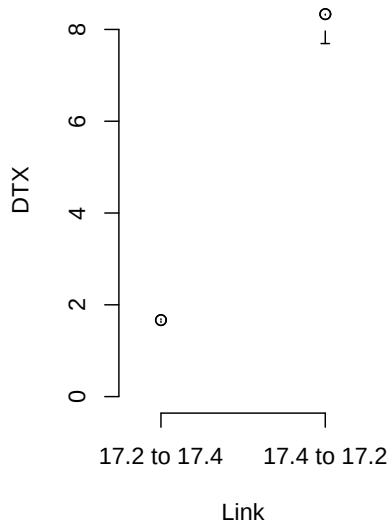
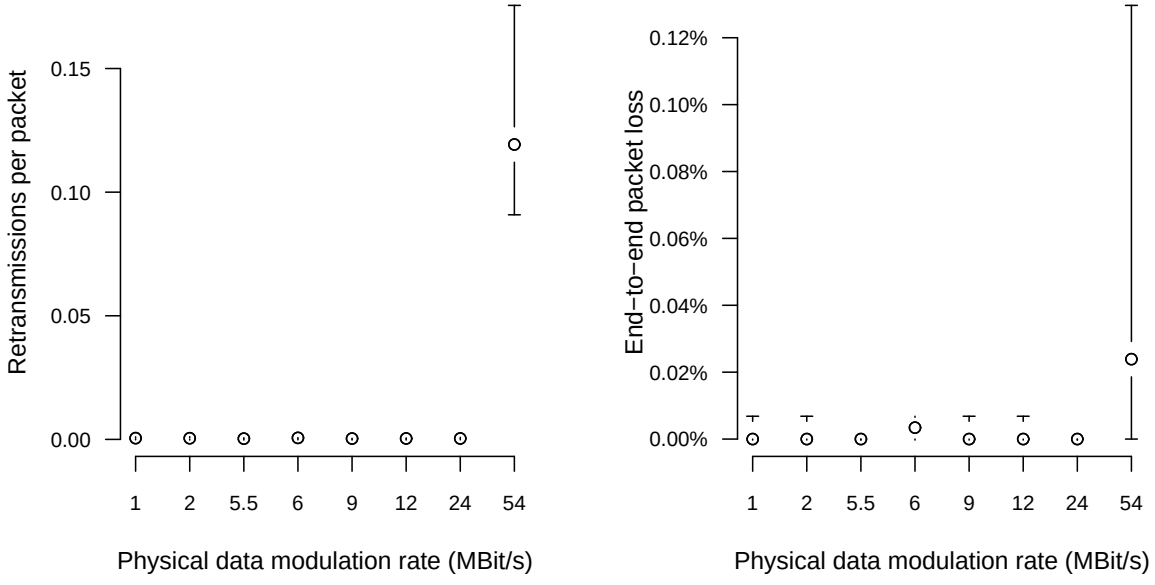


Figure 5.3: Median of DTX samples and confidence intervals ($\gamma = 99\%$) for two directions of the same link. The measurement data is the same as used in Figure 5.2.

5.3 Experimentation results

Prior to presenting the results of the experiments, the actual numbers obtained are described. Measuring pure data throughput may seem obvious, but is not feasible here, as the limitation due to the experienced CPU constraints currently prohibits us to fully utilize the mesh network by injecting data at a high rate. Instead, we count the total number of packet transmissions, including retransmissions and acknowledgements, for each routed packet and calculate its median. This ought to be a reasonable "work done" approximation, although ignoring air times of packet transmissions. It still is an open question, if there is a correlation between "work done" and throughput in the case of IEEE 802.11 networks.

All experiments will follow a similar procedure. In general, after initializing the whole system with the desired configuration, the experiment was run ten times, if not mentioned differently. Each cycle lasts 150 seconds. Before evaluating the results, 15 seconds are cut off from the beginning and from the end, to have the data coming from a system in steady state. To summarize the results, the median of the results of all cycles along with the confidence interval of the median is calculated.



(a) Transmissions

(b) Loss

Figure 5.4: (a) Number of transmissions including retransmissions needed for delivering a packet over one hop and (b) end-to-end packet loss. Both at different physical data modulation rates.

5.3.1 Preliminary experiments

There are mainly three important settings that have to be made before actually running experiments. First, the wireless interfaces must be set to a physical data modulation rate and a transmission power. Second, the timeout values of SOAR must be properly configured, especially the acknowledgement timeout. Finally, a reasonable data rate for the traffic to be injected into the WMN must be found. Following scientific principles, we find reasonable settings by experimentation and observation.

Physical data modulation rate and transmission power

Rate selection for wireless networks is a non trivial task. What we aim to achieve here, is a setup with a fair amount of packet loss, without breaking mesh connectivity. To determine a reasonable setting of the physical data modulation rate, we run a set of experiments on two nearby nodes that have a very good connectivity to each other (17.1 and 17.2). On both nodes, the SOAR protocol is run with the default configuration (Appendix A). Data at a rate of 1 MBit/s is injected into the system through node 17.1. The destination node of the data is 17.2. This experiment is rerun at different physical data modulation

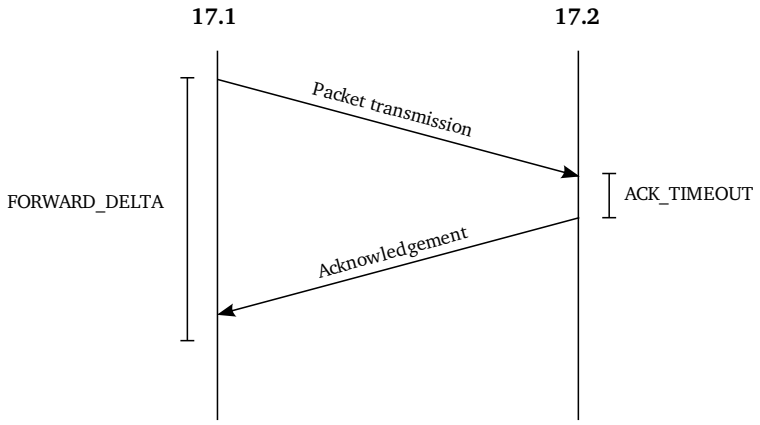


Figure 5.5: The role of the `FORWARD_DELTA` and the `ACK_TIMEOUT` settings with a single packet transmission at a two node (17.1 and 17.2) SOAR enabled WMN.

rates (Figure 5.4). The transmission power is set to 4 dB from a range of powers ranging 1 dB to 17 dB.

While looking at Figure 5.4, it may seem obvious to chose a rate of 54 MBit/s with regards to the requirement of lossy links. But as it turned out, it was impossible to find a good set of transmission power settings for the indoor nodes to remain certain connectivity while having at least some lossy links. Either the connectivity was perfect without packet loss, or not working at all, for most of the links.

Finally, we decided in favor of a physical data modulation rate of 24 MBit/s, which is the rate for all following experiments. The transmission powers are set to different values, nodes 16.1 and 16.2 are set to 8 dB, while all others are set to 4 dB. With that, some of the working links are lossy and most of the nodes have connectivity to not more than two other nodes, forcing multi hop routing for selected flows. As the topology was varying for most of the experiments which we ran at different nights, a weighted topology graph is provided along with each experiments results.

SOAR timeouts

The SOAR protocol must be configured with properly selected timeout values in order to operate well. These include the settings of the acknowledgement timeout (`ACK_TIMEOUT`, Appendix A) and the time it takes for queueing and transmitting a packet (`FORWARD_DELTA`), which are interdependent. To understand this interdependency, let's have a look at a small WMN consisting of only two nearby nodes 17.1 and 17.2 that have connectivity close to lossless. Furthermore, let node 17.1 transmit a packet destined for 17.2. The forwarding list of 17.1 then contains 17.2 as its only entry. After transmitting the packet, node 17.1 starts a retransmission timer set to expire after `FORWARD_DELTA` (Appendix A) and waits for an acknowledgement from 17.2 (Figure 5.5). In turn, 17.2 sends an acknowledgement

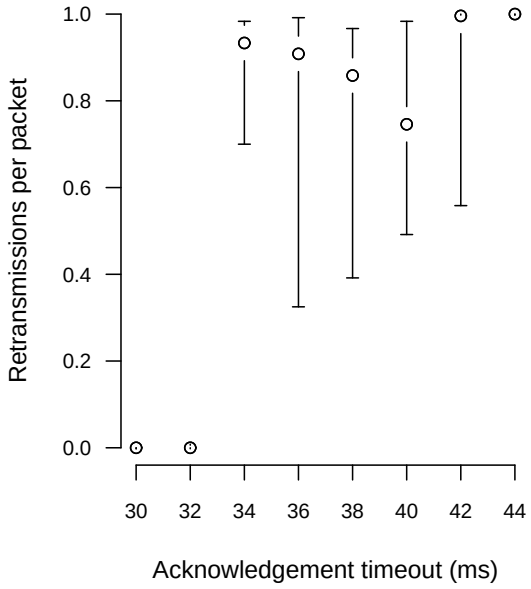


Figure 5.6: Number of retransmissions needed for delivering a packet over one hop with different acknowledgement timeouts. Each dot displays the median of ten experiment runs along with the confidence interval at a confidence level of 95%.

after its acknowledgement timer elapsed. At 17.1, if no acknowledgement is received before its retransmission timer elapsed, a retransmission follows. A retransmission should only happen with the loss of either the packet itself or the acknowledgement, and not because of misconfigured timeout values. Figure 5.5 visualizes the desired behaviour. Therefore, the difference between `FORWARD_DELTA` and `ACK_TIMEOUT` is important.

To find a proper setting, experiments are run with different acknowledgement timeouts, while keeping `FORWARD_DELTA` at its default value of $45ms$, which is the value also used in the original SOAR publication. The default value of `ACK_TIMEOUT` is $30ms$. With these experiments, the value of `ACK_TIMEOUT` is increased from $30ms$ up to $44ms$, in steps of $2ms$. To correctly determine the timeout, it's important to inject packets slow enough so that their routing procedures do not overlap. Because when they overlap in the way the second packet reaches the destination before the `ACK_TIMEOUT` of the first packet elapsed, the acknowledgement compression strategy (Section 3.2.4) will combine both acknowledgements into a single packet. That in turn means that the second acknowledgement is sent before its `ACK_TIMEOUT` elapsed, and with that it may reach the originating node in time even with the `ACK_TIMEOUT` set too high, thus breaking the results.

For the experiments, we chose to transmit packets at a rate of one packet per second, which ensures independent routing of each one and makes acknowledgement compression

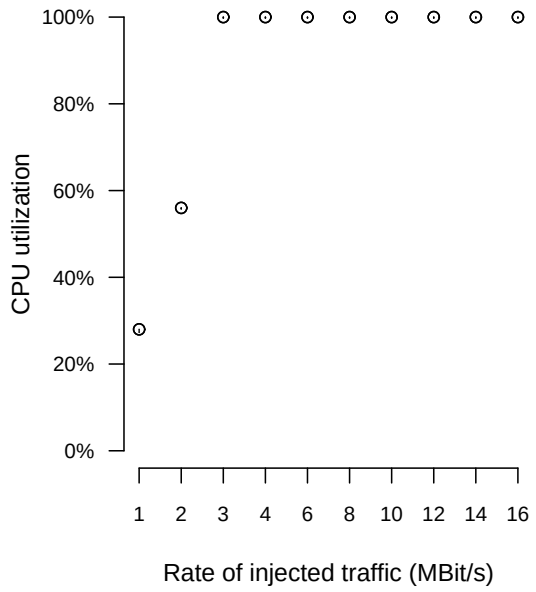


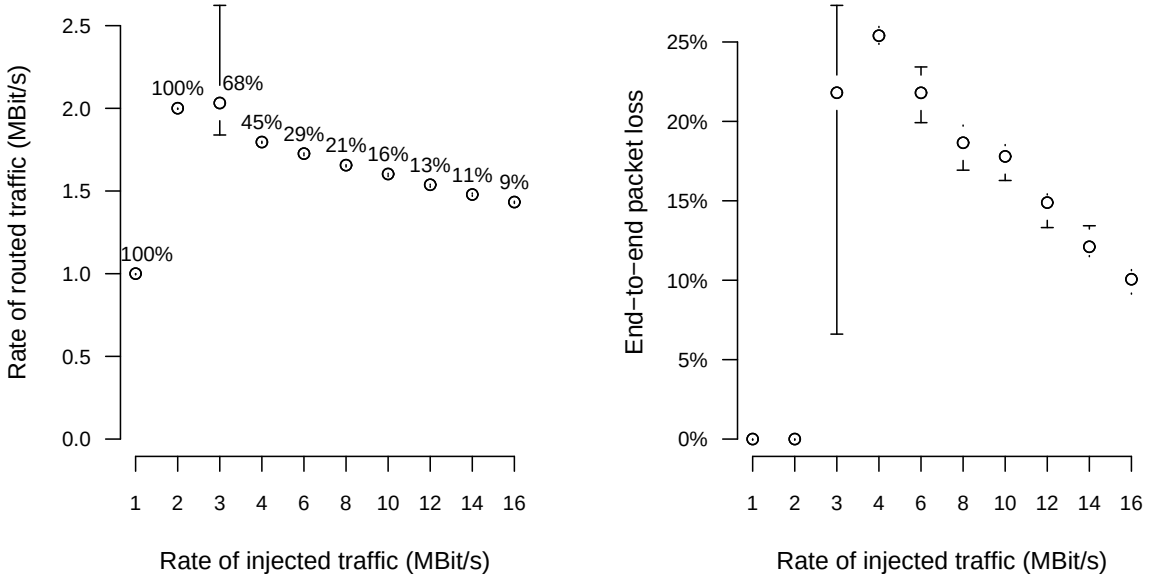
Figure 5.7: CPU utilization of the node where user traffic is injected (17.1) at different rates.

impossible. That is, because the packets are transmitted at most four times (one regular transmission, three retransmissions) while the last of the four transmissions is initiated 3∗`FORWARD_DELTA` = 135ms after the initial transmission, hence providing a significant pause period until proceeding with a new packet.

The last requirement for this experiment, the close to lossless connectivity between two nodes, is given for 17.1 and 17.2. Figure 5.6 shows the results. In this simple scenario, the `ACK_TIMEOUT` settings of 30ms and 32ms give perfect results, i.e. no retransmissions due to acknowledgements arriving too late. With higher values of `ACK_TIMEOUT`, the results show that retransmissions become very likely, and most likely with 44ms. For the remaining experiments, an `ACK_TIMEOUT` of 30ms and the default `FORWARD_DELTA` of 45ms is employed.

Data rate of workload traffic

The rate of injected data must be limited due to the CPU utilization problems experienced when the traffic load exceeds a maximum. The exact case of this problem is not perfectly clear. A runtime analysis revealed, that a great amount of CPU time is spent at the Click timer handling functions (SOAR registers at least one timer per packet) and at memory copy operations, most probably caused by copying each network packet from kernelspace



(a) Routed packets

(b) Loss

Figure 5.8: (a) Number of packets received by the SOAR router within Click. The numbers nearby the dots denote the percentage of the total number of injected packets. (b) End-to-end packet loss. Both at different rates of injected traffic.

to userspace and back. One major upcoming task will be to port the kernel version of Click to our architecture (Section 6.5), which should at least reduce the time spent on memory copies as the copies between kernel and userspace are no longer necessary.

The following experiments will demonstrate the effects caused by the CPU limitation. This time, a three node WMN consisting of nodes 16.5, 17.1, and 17.2 is used. All links were practically reliable and contention-free ($DTX = 1$). The data flows from 16.5 to 17.2 and traffic is injected at different rates while monitoring the CPU utilization at each node. The problematic CPU load is on node 16.5, because that node is the entry node for experimental traffic and starts dropping when it receives too much. Consequently, the other nodes do not see as much traffic and thus won't experience CPU load problems.

Figure 5.7 shows the CPU utilization of node 16.5 at different injected data rates, revealing that packet rates higher than 2 MBit/s already entirely exhaust the CPU. What becomes interesting now is the number of lost packets, i.e. the amount of packets that were received by the SOAR router program running on the entry node 16.5. Figure 5.8 (a) shows that with raising data rates, less data is handed over to the SOAR router, therefore it is lost somewhere on the way. That loss begins at the same data rate, at which the CPU utilization reaches a maximum. Figure 5.8 (b) shows the end-to-end packet loss,

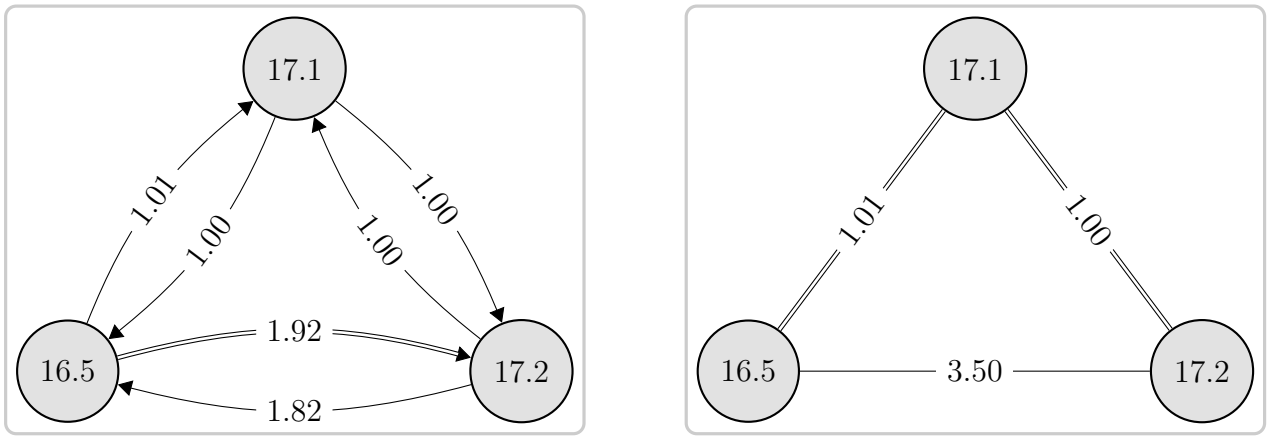


Figure 5.9: Real network topology used for the three nodes experiments. The left graph has DTX values shown along the edges (relevant for SOAR), the right graph has ETX (relevant for shortest path routing). The double lined edges represent the shortest path from node 16.5 to 17.2.

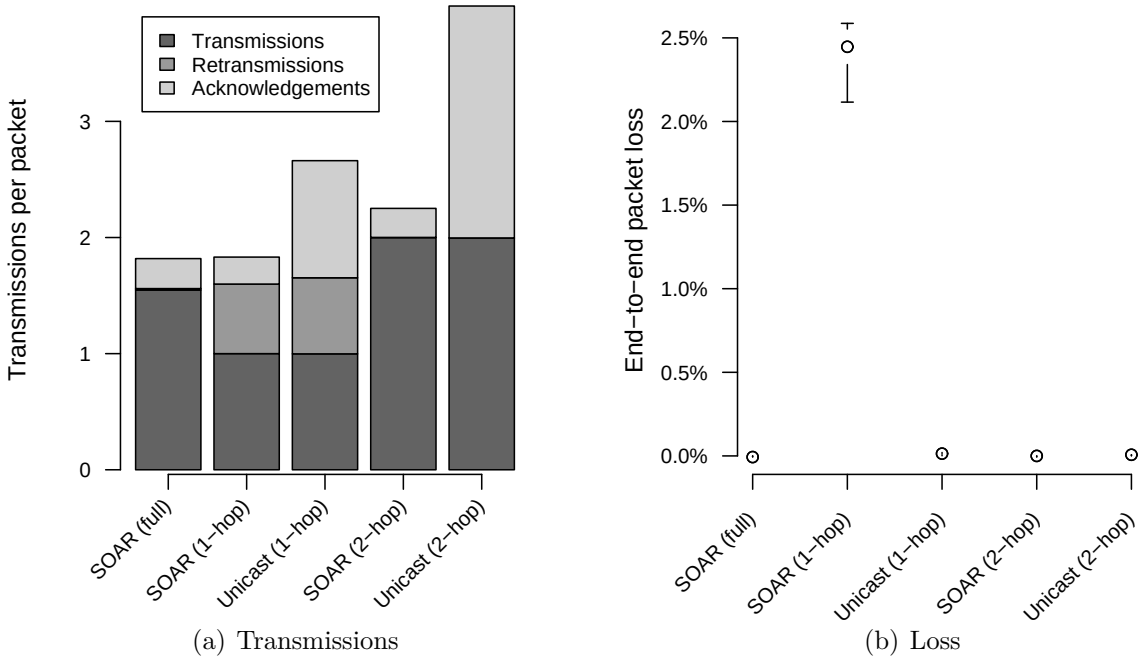
which increases as well when passing the 2 MBit/s limit, and decreases with rising data rates, simply because fewer packets are routed.

For the remaining experiments, 1 MBit/s is selected as the data rate for experimental workload traffic. We do not select 2 MBit/s as we want to reserve some of the available CPU processing power for e.g. an increasing amount of acknowledgement packets or retransmissions that may occur. Additionally, the CPU load is continuously monitored with all remaining experiments and the data rate is reduced if CPU bounds are reached.

5.3.2 Three nodes experiments

It is yet unclear, how SOAR behaves when lossy links are present and multiple possible routes to the same destination are available. This experiment is meant to emphasize the impact of opportunistic routing on route selection. For this, we decided to use only three nodes of the nine available ones of the Indoor testbed. The topology with link costs is shown on Figure 5.9. The experimental flow originates at node 16.5 and is bound to 17.2. As a comparison to SOAR the same experiment was run again with shortest path unicast routing, realized with a static Click forwarding configuration. This experiment should show, that SOAR utilizes both paths (16.5 → 17.1 → 17.2, 16.5 → 17.2) to route the packets.

Figure 5.10 shows the results of the various different experiments with this three node topology. The "SOAR (1-hop)" and "SOAR (2-hop)" experiments are restricted in route selection. "SOAR (1-hop)" only utilizes the direct lossy 16.5 → 17.2 path while "SOAR (2-hop)" does not use that direct link but is forced to route packets along 16.5 → 17.1 → 17.2. The "SOAR (full)" experiment is the one, that should demonstrate the benefits



(a) Transmissions (b) Loss

Figure 5.10: (a) Number of transmissions, retransmissions and stand-alone acknowledgements needed for delivering a packet and (b) end-to-end packet loss. Both at different scenarios.

of SOAR with this experiment, because it has no restrictions in route selection. As a comparison, the shortest path unicast approach is run for both possible routes as well.

The bars on Figure 5.10 (a) show the median total number of transmissions needed to deliver one packet end-to-end, which is the sum of the transmissions from all involved nodes. The fractions of transmissions, retransmissions and acknowledgements are visualized. When comparing the results from the "SOAR (full)" and the "SOAR (1-hop)" experiments, one can see that instead making use of retransmissions, SOAR prefers to use alternate paths, if available, thus involving more nodes in the routing process and therefore splitting the individual load. A further analysis of the evaluation data collected with "SOAR (full)" reveals that more than half of the packets (53.4%) were passing the intermediate node 17.1, the remainder (46.6%) took the direct way.

When comparing the results of all five experiments, it is clear to see that the acknowledgement compression strategy of SOAR is beneficial regarding the total number of stand-alone acknowledgement transmissions.

The "Unicast (1-hop)" experiment shows a lower number of transmissions than expected with regards to the measured ETX of that link. The reason for that assumably lays in the method of topology measurement used in this thesis. Packets with lengths of

1 KByte are used for probing the links, but IEEE 802.11 link-layer acknowledgements are significantly shorter, thus they should exhibit a considerably higher delivery success rate, which in turn reduces the expected number of retransmissions.

As expected, the 2-hop experiments show that the number of transmissions in either case is two, as both involved links are measured to be lossless. Again, SOAR is beneficial when comparing the number of acknowledgements sent.

Besides counting the number of transmissions, we want to get an idea about the involved protocol byte overhead. For that, we only take the IP layer into account, therefore the calculated overhead consists of the size of the IP option which gets added to SOAR packets, the length of the appended SOAR meta data and the total IP length of all stand-alone acknowledgements. With the "SOAR (full)" experiment, we get an overhead of 5.95% (median). Comparing that with the overhead due to IEEE 802.11 link-layer acknowledgements, which presumably is lower, is not yet feasible, because this would involve air-time measurement (Section 6.6).

Figure 5.10 (b) visualizes the end-to-end packet loss behaviour. The only experiment with noticeable packet loss is "SOAR (1-hop)". The question is, why we do not experience the same amount of packet loss with the "Unicast (1-hop)" experiment? This should be mainly due to the different maximum number of retransmissions in both experiments. Our SOAR configuration limits the number of retransmissions to a count of three (default value of `RETRANSMIT_COUNT`, Appendix A), while the MAC layer of our wireless devices is set to try up to seven retransmissions.

5.3.3 Single-flow experiments

We have seen how SOAR behaves when used with a small WMN, this next experiment will join eight of our nine indoor mesh nodes into one bigger WMN (node 16.4 was not operating well at that time) and run similar experiments. Figure 5.11 visualizes the present topology. The experimental flow starts at node 16.1 and ends at 17.2. Nodes, that are not considered for being the next hop at any point of the experiments are not shown. The shortest path is the same for both the DTX and the ETX metric: 16.1 → 16.2 → 17.3 → 17.2.

Let's have a look onto the median number of transmissions conducted by SOAR and shortest path unicast forwarding (Figure 5.12 (a)). This time, both experiments effectively utilize a similar amount of transmissions while not imposing a considerable number of retransmissions. Due to the present topology, this is expected since the shortest path route is measured to be lossless. Again, SOAR shows its strength induced by its acknowledgement compression strategy.

Figure 5.12 (b) compares the median end-to-end packet loss of both experiments.

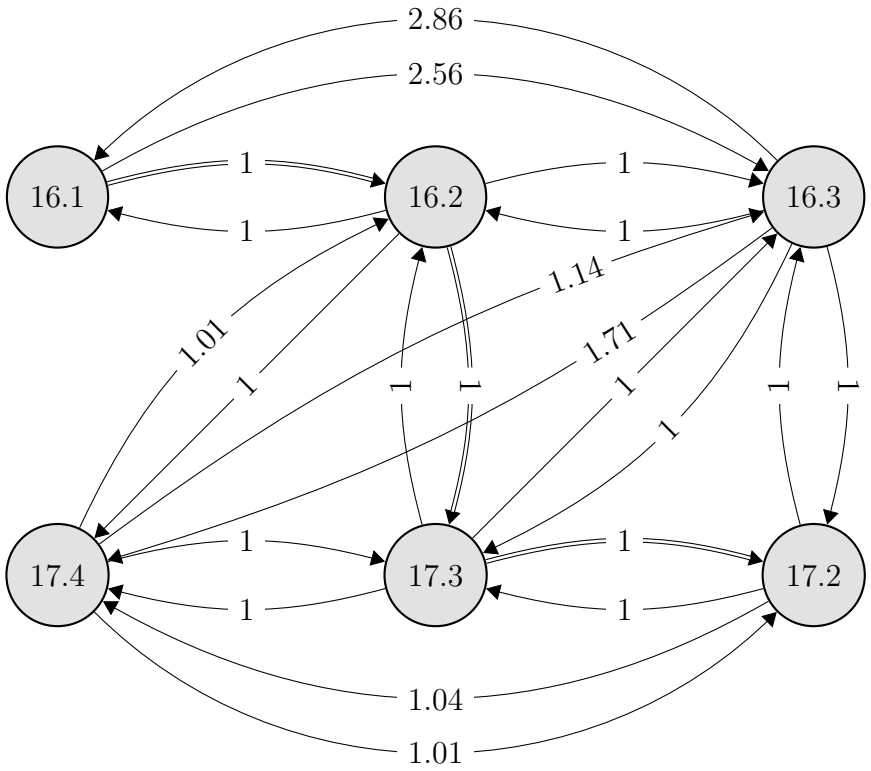


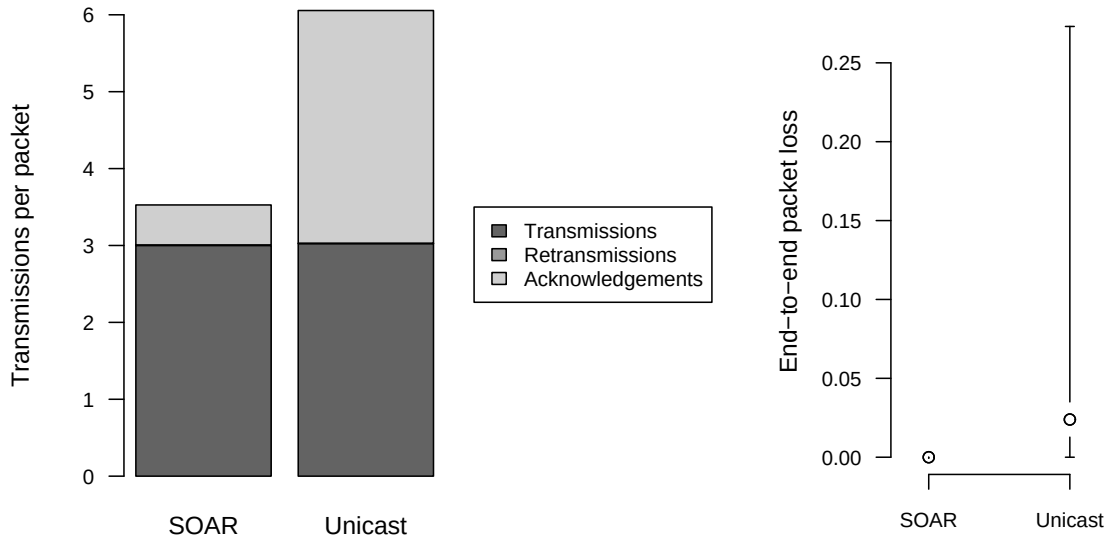
Figure 5.11: Network topology present at the single-flow experiments. DTX values are shown alone the edges.

While SOAR effectively shows no packet loss, the shortest path unicast experiment has a slight amount of end-to-end loss. This may come from short downtimes of links on the shortest path which obviously is problematic for plain shortest path routing. On the other hand, that is no problem for SOAR, since it simply bypasses periods of link downtimes by utilizing alternative routes.

5.3.4 Multi-flow experiments

For the multi-flow experiments, the same eight indoor nodes are used as before. The topology has changed since the last experiment, an updated view is shown on Figure 5.13. This time, two different flows are injected into the WMN and all nodes are involved in packet forwarding, therefore Figure 5.13 contains all running nodes.

Flow 1 starts at node 16.5 and ends at node 17.4. With this flow, we have two different shortest paths, depending on the chosen metric. SOAR uses the DTX metric, the values are shown on Figure 5.13. The shortest path then is 16.5 → 17.1 → 17.2 → 17.4 and has a total cost of 3.68. The ETX metric that shortest path routing consults yields a different route. It does not include the lossy hop 17.2 → 17.4, because its ETX is



(a) Transmissions

(b) Loss

Figure 5.12: (a) Number of transmissions, retransmissions and stand-alone acknowledgements needed for delivering a packet and (b) end-to-end packet loss. Both at different scenarios.

$(1.67^{-1} * 8.33^{-1})^{-1} \approx 13.91$, which is significantly higher than the path ETX when using node 17.3 as an intermediate node, which in turn equals to 2 for that part of the route. Accordingly, shortest path routing decides in favor of the path 16.5 → 17.1 → 17.2 → 17.3 → 17.4, with a total cost of 4.01. SOAR is expected to efficiently utilize the 17.2 → 17.4 hop, despite the fact, that the reverse direction is very lossy.

Flow 2 originates at node 17.3 and is destined for node 16.1. The shortest path for either routing approach is 17.3 → 16.2 → 16.1, with total costs of 2.17 (DTX) and 2.23 (ETX).

Figure 5.14 shows the results, separately for both flows. Matching SOAR stand-alone acknowledgement packets to a flow is not always possible, since they may contain multiple acknowledgements related to different flows at the same time. Therefore, extra bars representing the sum of acknowledgements divided by the total number of injected packets of both flows are drawn. As before, SOAR reveals its strength provoked by its acknowledgement compression.

Looking onto the number of packet transmissions belonging to flow 2, both routing protocol contestants demand similar amounts of packet transmissions. That is expected, although SOAR as compared to shortest path unicast forwarding, additionally includes

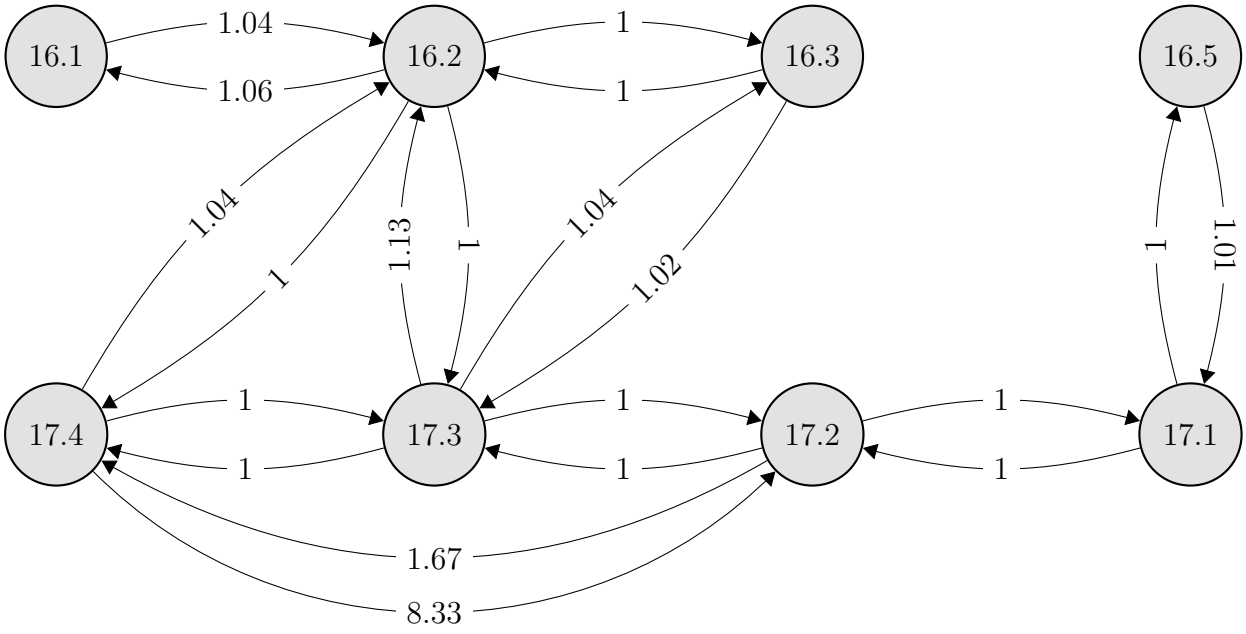


Figure 5.13: Network topology present at the multi-flow experiments.

node 16.3 in its forwarding process by adding it to the forwarding list of packets sent by 17.3. Other than a smaller amount of retransmissions, SOAR does not benefit from it. That is, because the link 17.3 → 16.2 is close to lossless, 16.2 is the highest priority node on 17.3's forwarding list and therefore most packets follow the same path as with shortest path routing.

As opposed to the results of flow 2, SOAR wins clearly when looking at the results of flow 1. As stated before, SOAR is assumed to efficiently combine both suitable paths from 17.2 to 17.4, so that the overall expected number of transmissions is lower than that of each individual path. The results show that this assumption holds in practice. To be more precise, SOAR delivers 56.3% (average) of the packets using the direct link from 17.2 to 17.4, while the others use 17.3 as an intermediate node.

With this experiment, there is close to no packet loss for each scenario. The only exception is flow 1 with shortest path unicast routing, which has a packet loss of approximately 1% (median). As already noticed, this may come from shorter downtimes of links on the path, that SOAR can handle more beneficial by routing alternatively.

5.4 TCP through Wireless Mesh Networks

The importance of TCP [20] in today's computer networks like the Internet is beyond all question. With that in mind, TCP must be well supported by networks that offer Internet connectivity. Unfortunately, TCP does not work well with WMNs due to several reasons.

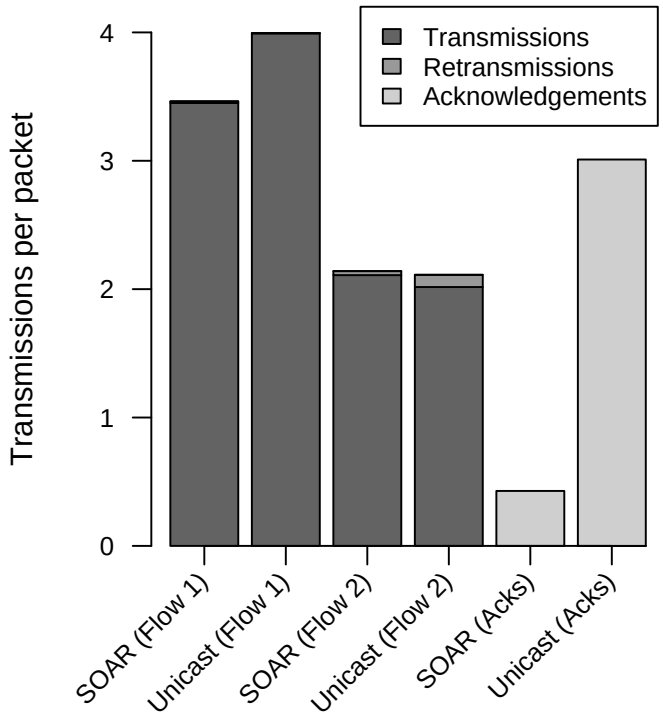


Figure 5.14: Number of transmissions, retransmissions and stand-alone acknowledgements needed for delivering a packet.

After a brief description of the problems we are facing with TCP, our approach of how to enable TCP on WMNs is described while focusing the role of the routing protocol.

TCP offers reliable data transfer for IP networks. It constantly adapts its transmission rate according to the available network bandwidth. Since the available bandwidth is unknown, TCP raises its transmission rate until it senses signs of congestion. These signs include packet loss, which is implied by out-of-order packets. It then reacts by decreasing its transmission rate. While this is legitimate and works well with wired networks, where links can be considered lossless, the same strategy fails when applied to WMNs, whose links have a non-negligible rate of packet loss.

With WMNs, the performance of TCP suffers badly, because TCP in that case cannot determine the actual reason for packet loss. As stated earlier, detected packet loss leads to reduction of the senders transmission rate, which is undesirable in cases where congestion is not present. The packet may have simply got lost in the air. Another major problem with TCP is the intense bandwidth consumption due to frequent acknowledgement packets. Furthermore, TCP does not play well with the delay jitter of WMNs, because the

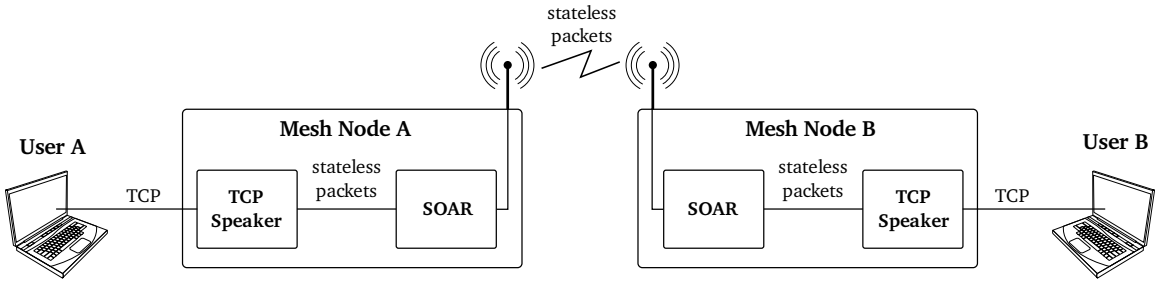


Figure 5.15: Illustration of our approach for enhancing TCP performance with WMNs.

round-trip-time (RTT, time it takes for a packet to go from A to B and back) calculation will become unreliable. This in turn may lead to premature timeouts and in the end, it may cost unnecessary retransmissions. There are publications discussing these kinds of problems in greater detail ([11, 8]).

With the Bowl framework, we aim to get around these problems by terminating TCP connections at the border of the WMN, realized by a `TCPSpeaker` element. Figure 5.15 illustrates the proposed design. Let's see how things work with the current implementation, when user A attempts to establish a TCP connection with user B. User A first transmits a TCP *connection request*-message. The `TCPSpeaker` element of mesh node A immediately answers with a *connection accepted*-message, which user A acknowledges according to the TCP handshake procedure. Note, that no packets were routed through the mesh, yet. Now, if user A sends its first data segment, the `TCPSpeaker` at mesh node A immediately acknowledges the successful reception. Then, the `TCPSpeaker` transforms the TCP segment into a stateless datagram and hands it over to `SOAR`, our mesh routing protocol implementation. Ultimately, the destination mesh node B receives the stateless packet, hands it over to its local `TCPSpeaker`, which in turn establishes a TCP connection with user B on behalf of user A and finally delivers the data segment.

With that design, we reduce the capacity utilization of the mesh, because TCP acknowledgement packets never reach the mesh routing protocol. Furthermore, we acknowledge packet reception to user TCP applications immediately, hence hiding delay jitter and packet loss. With that, we allow TCP to maintain a high throughput.

Yet unresolved problems in that area include adding full reliability to mesh routing, because the loss of a stateless packet always results in a gap in the TCP data stream. Furthermore, `SOAR` currently pulls packets from `TCPSpeaker` without making sure, that the mesh currently offers the required capacity. This effectively disables the flow control mechanism of TCP hence makes TCP increase its transmission rate to a maximum, potentially higher than the mesh can ever handle. We ultimately want to allow the user TCP side to adjust its transmission rate according to the available mesh capacity.

Experiments with TCP were not feasible at the current state of development and with respect to the experienced CPU constraints. Some initial tests revealed, that at least the setup and the various different components play well together. It is already possible to create a scenario as illustrated on Figure 5.15 and to establish TCP connections between user A and B. Although, due to the already mentioned parts that we are still missing to make the setup work properly, measurement results (approximately 3.5 MBit/s without `TCPSpeaker`, and 0.6 MBit/s with `TCPSpeaker` with a small two-node mesh network) are of no value.

Chapter 6

Future work

As Wireless Mesh Networks in general and opportunistic routing in detail both are relatively novel research areas, there exist many remaining open topics. Therefore, many ideas for future work have emerged over the course of this thesis.

6.1 Live topology measurement

Throughout this thesis, only static topology information was used. Prior to every experiment, we measured the topology and passed the result on to the nodes. For a production system, of course, the topology must be updated during runtime, reflecting any changes that may happen. The implementation of the SOAR protocol allows for live updating of the topology, but so far we have no automatic measurement application.

It seems like an obvious solution to implement a stand-alone topology measurement application that keeps the nodes up-to-date. Besides that, one could as well extend the SOAR protocol itself with a link detection and link quality measurement system. For example, packets that are sent could be used to determine the availability of links, hence potentially saving overhead.

6.2 Less SOAR meta data

The implementation of the SOAR protocol appends the shortest path and forwarding lists to each routed packet. This might be considered redundant because every single mesh node has all the information it needs to calculate these lists on its own. Nevertheless, simply leaving out that meta data without further clarification is not advisable.

Problems might arise due to an inconsistent knowledge about the topology. This could be solved e.g. by versioning the topology information and appending a topology version identifier to each packet.

On the other hand, one should consider the question about the negative effects that inconsistent topology information might have in the routing process. Given that the topology information differs only slightly and only over a shorter period of time, routing essentially should not break. The way that the forwarding list is calculated (Section 3.2.2) assures that the packet always makes progress, even if some list entries are no longer reachable or new candidates are available but not yet included.

A more serious concern remains whether forwarding loops could become a problem with inconsistent topology information. Again, this presumably is not an issue, because of two reasons. First, we use the IP header time-to-live field to stop forwarding loops eventually. Second, a node never handles a packet with the same sequence number twice, but drops it instead.

6.3 Precise timers

SOAR makes extensive use of timers. In the implementation, the accuracy is questionable, because currently there is no way to obtain exact time information from the hardware driver or the wireless interface itself. For example, when handing over a packet to the outgoing queue of the driver and starting a retransmission timer, one cannot be sure that the packet really is transmitted immediately. Therefore, the retransmission timer might be started prematurely, thus affecting the routing process. To solve this problem, timers should be based on real transmission times by implementing a notification system between the hardware driver and the routing application.

6.4 Automatic timer configuration

Besides the accuracy of timers, the timeout settings themselves are an essential part when aiming for a high-performance SOAR enabled WMN. Section 5.3.1 was about gathering a good configuration of the timeouts. For bigger networks and real user traffic with unbalanced mesh node utilization hence different packet queue lengths and hence delay variations, finding good timeout settings gets more complex and thus questionable if there even exists a good static configuration. A good configuration is a compromise between forwarding delay and superfluous packet transmissions. Future work might include developing an automatic and ideally dynamic timer configuration algorithm for SOAR and an agreement protocol for allowing settings to settle on all nodes.

6.5 Use Click kernel module

Click comes in two flavors, it can be either a userspace application or a Linux or BSD kernel module. The latter ought to achieve better performance in principle, as time consuming copy operations for every data packet from kernel to userspace memory and back are not necessary. Unfortunately, we can not make use of the Click kernel version yet, because the currently available Click kernel patches are not suitable for the Linux kernel version we use with our OpenWRT distribution on our nodes. We consider porting the Click kernel patch to work with our environment as a task for the near future. With that, we hope to be able to run experiments with considerably higher data rates.

6.6 Further measurements

A major objective is to do meaningful throughput measurements with SOAR. Hopefully this will become possible with porting the Click kernel version to our architecture. Furthermore, we expect to have everything we need to perform real air-time measurement with the new wireless measurement probe infrastructure (wprobe) currently developed at Fraunhofer FOKUS[1]. For example, this will enable us to compare the air times for transmitting a SOAR stand-alone acknowledgement with that of IEEE 802.11 link-layer acknowledgements.

[1] http://www.fokus.fraunhofer.de/

Chapter 7

Conclusion

This thesis examines the novel approach of opportunistic routing for Wireless Mesh Networks. Opportunistic routing exploits the broadcast nature of wireless networks by deferring the next hop selection to after the actual packet transmission took place. The next hop then is chosen among all nodes, that received the packet.

This thesis provides a prototype implementation of the SOAR protocol. Experiments with that reveal, that the idea behind opportunistic routing holds under realistic conditions. SOAR opportunistically choses different routes for packets with the same source and destination pair, while always aiming for the highest progress with each hop. Results show, that SOAR, as compared to shortest path routing, needs a lower total number of packet transmissions for the same routing task. That may be an indication for a lower link capacity utilization.

Due to environmental constraints, enlightening throughput measurements were not feasible. But with the evaluation results of this thesis, opportunistic routing in general and SOAR in detail without a doubt are promising approaches which should be considered for some more advanced measurement in the near future.

Appendix A

SOAR Click element parameters

Parameter	Type	Default	Modifiable at runtime	Description
IPADDR	x.x.x.x	*none*	no	IP address of the node.
BCAST	x.x.x.x	255.255.255.255	no	Destination IP address used for broadcast packets.
MTU	uint (byte)	1500	no	Maximum transmission unit for packets sent through the mesh.
PACKETS_LIMIT	uint	1000	yes	Maximum number of packets queued by the SOAR element, this can be used to limit memory utilization.
MAX_IN_FLIGHT	uint	300	no	Maximum number of packets in flight per SOAR flow.
RECV_WIN_SIZE	uint	600	no	Size of sequence number window for received packets per SOAR flow.
BURST	int	3	yes	Maximum number of packets pulled by the SOAR element during one schedule. A value of -1 sets BURST to infinite, 0 makes to SOAR element stop pulling packets.
ACK_TIMEOUT	uint (ms)	30	yes	Maximum time between packet reception and acknowledgement transmission.

Parameter	Type	Default	Modifiable at runtime	Description
ACK_ON_FORWARD	bool	true	yes	Whether to transmit an acknowlegement for packets that are forwarded right away. In theory, this is not necessary because the forwarding itself is an acknowlegement already, but it is supposed to reduce unnecessary retransmissions and thus enabled per default.
FORWARD_DELTA	uint (ms)	45	yes	Time it takes for queueing and transmitting a packet.
RETRANSMIT_COUNT	uint	3	yes	Maximum number of retransmissions.
PATH_LIMIT	uint	5	yes	Maximum length of shortest paths.
FWLIST_LIMIT	uint	5	yes	Maximum size of forwarding list.
FWLIST_THRES	uint	6	yes	Metric threshold used for calculating forwarding lists.

Appendix B

SOAR Click element handlers

The SOAR Click element offers a set of element handlers, that can be used to read data from and write data to the SOAR element. Included are handlers for each of the element parameters (see Appendix A). Additionaly, the following handlers are available:

Handler	Parameters	Kind	Description
`user_mtu`	*none*	read only	Returns the size limit for user packets, which is the MTU parameter (Appendix A) minus the least space necessary for appending the SOAR metadata.
`topology`	*topology data*	write only	This is where topology information is handed over to SOAR. The format of the argument is a simple binary format in base64 encoding [13].
`nodes`	*none*	read only	Returns a list of all known node identifiers of the topology.
`cost`	`from to`	read only	Returns the metrical link cost for the link `from`→`to`, where `from` and `to` are node identifiers.
`path`	`from to`	read only	Returns the shortest path of the route `from`→`to`, where `from` and `to` are node identifiers, with its metrical cost.
`fwlist`	`from to` [`curr`]	read only	Returns the forwarding list for the flow `from`→`to` calculated at node `curr`, where `from`, `to` and `curr` are node identifiers. If `curr` is omitted, `from` is assumed as `curr`.
`in_progress`	*none*	read only	Returns the number of packets currently delayed by a forwarding timer and the number of packets currently waiting for an acknowledgement, i.e. delayed by a retransmission timer.
`stats`	*none*	read only	Return statistics: The total number of packets seen on each of the input and output ports, the number of standalone acknowledgement packets sent and received and the count of forwarding list cache hits and misses.
`reset`	*none*	read only	Reset statistics counters.

Bibliography

[1] IEEE std 802.11-2007, wireless LAN medium access control (MAC) and physical layer (PHY) specifications, June 2007.

[2] D. Aguayo, J. Bicket, S. Biswas, G. Judd, and R. Morris. Link-level measurements from an 802.11b mesh network. *SIGCOMM Comput. Commun. Rev.*, 34(4):121–132, 2004.

[3] I. F. Akyildiz, X. Wang, and W. Wang. Wireless mesh networks: a survey. *Computer Networks*, 47(4):445–487, March 2005.

[4] S. Biswas and R. Morris. ExOR: opportunistic multi-hop routing for wireless networks. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 133–144, New York, NY, USA, 2005. ACM.

[5] S. Chachulski, M. Jennings, S. Katti, and D. Katabi. More: A network coding approach to opportunistic routing. Technical report, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, June 2006.

[6] E. Clausen and E. Jacquet. Optimized link state routing protocol (OLSR). RFC 3626, Internet Engineering Task Force, Oct. 2003.

[7] 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 ACM International Conference on Mobile Computing and Networking (MobiCom '03)*, San Diego, California, September 2003.

[8] R. de Oliveira and T. Braun. A smart TCP acknowledgment approach for multihop wireless networks. *IEEE Transactions on Mobile Computing*, 6(2):192–205, Feb. 2007.

[9] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[10] R. Draves, J. Padhye, and B. Zill. Comparison of routing metrics for static multi-hop wireless networks. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 133–144, New York, NY, USA, 2004. ACM.

[11] Z. Fu, P. Zerfos, H. Luo, S. Lu, L. Zhang, and M. Gerla. The impact of multihop wireless channel on TCP throughput and loss. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, volume 3, pages 1744–1753, Mar./Apr. 2003.

[12] Z. J. Haas and M. R. Pearlman. ZRP: a hybrid framework for routing in ad hoc networks. pages 221–253, 2001.

[13] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), Oct. 2006.

[14] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. M. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000.

[15] G. Malkin. RIP Version 2. RFC 2453 (Standard), Nov. 1998. Updated by RFC 4822.

[16] J. Moy. OSPF version 2. RFC 2328, Internet Engineering Task Force, Apr. 1998.

[17] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental), July 2003.

[18] D. C. Plummer. Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware. RFC 826 (Standard), Nov. 1982.

[19] J. Postel. Internet Protocol. RFC 791 (Standard), Sept. 1981. Updated by RFC 1349.

[20] J. Postel. Transmission Control Protocol. RFC 793 (Standard), Sept. 1981. Updated by RFCs 1122, 3168.

[21] E. Rozner, J. Seshadri, Y. Mehta, and L. Qiu. Simple opportunistic routing protocol for wireless mesh networks. pages 48–54, Sept. 2006.