

Universität Karlsruhe (TH)  
Institut für  
Betriebs- und Dialogsysteme  
Lehrstuhl Systemarchitektur

## **Efficient Detection and Utilization of Asymmetric Links in *Scalable Source Routing (SSR)***

Pascal Birnstill

Diplomarbeit

Verantwortlicher Betreuer: Prof. Dr. Frank Bellosa  
Betreuende Mitarbeiter: M.Sc. Pengfei Di,  
Dr. Thomas Fuhrmann (Technische Universität München)

May 19, 2009



Hiermit erkläre ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, May 19, 2009

---

Pascal Birnstil



## Abstract

As several empirical studies pointed out that asymmetric links occur considerably frequently in wireless networks and that network layer routing performance could potentially be significantly improved by utilizing these asymmetric links, it seems to be a reasonable feature within the Scalable Source Routing (SSR) protocol to provide support for asymmetric links.

The proposed algorithm is based on the approach of exchanging partial topology information as an extension of regular HELLO messages. Considering the network topology as a directed graph, two given nodes will only be able to communicate with each other if the network graph provides a directed cycle containing both of them. Therefore the main idea of this thesis is to find such a directed cycle, which we denote as a *loop path*.

Detecting a loop path containing an asymmetric link implies that a *reverse path* to this asymmetric link is obtained, i.e. a (multihop) source route connecting both nodes that are adjacent to the asymmetric link in the reverse direction of this asymmetric link. By this means, upon initially assuming any link to be asymmetric, source routes resolving occurring asymmetric links are gradually discovered.

Since exchanging exhaustive topology information does not scale with regard to bandwidth consumption and local storage requirements, we deployed some strategies on prioritizing and selecting appropriate topology information to be sent within periodic HELLO messages.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Basics and Related Work</b>	<b>3</b>
2.1	Scenarios Inducing Asymmetric Links . . . . .	3
2.1.1	Inherent Different Radio Capabilities . . . . .	3
2.1.2	Different Interference Levels . . . . .	3
2.1.3	Power Control Algorithms . . . . .	4
2.1.4	Impact of Asymmetric Links . . . . .	4
2.2	Related Work . . . . .	5
2.2.1	Empirical Studies on the Occurrence of Asymmetric Links in Wireless Settings . . . . .	5
2.2.2	Local Asymmetric Link Detection Based on a Regular <i>HELLO</i> Protocol . . . . .	5
2.2.3	Native Support of Asymmetric Links in <i>DSR</i> . . . . .	6
2.2.4	Extension of Conventional Distance Vector Protocols . . . . .	6
2.2.5	Approach Introducing <i>Sink Trees</i> . . . . .	6
2.2.6	Local Detection of Asymmetric Links . . . . .	7
2.2.7	Applying Reverse Path Search in AODV . . . . .	8
2.2.8	Reverse Tunneling . . . . .	9
2.3	Further Related Work on SSR in Mesh Networks . . . . .	10
<b>3</b>	<b>Description and Analysis of SSR</b>	<b>11</b>
3.1	Indirect Routing . . . . .	11
3.2	Message Forwarding . . . . .	12
3.3	Appending Source Routes and Applying Shortcuts . . . . .	13
3.4	Handling Broken Links . . . . .	15
3.5	State Maintenance . . . . .	15
<b>4</b>	<b>Approach Based on <i>Partial Sink Trees</i></b>	<b>19</b>
4.1	Assumptions and Simplifications . . . . .	19
4.2	Basic Algorithm . . . . .	20
4.2.1	<i>Sink Tree</i> Data Structure . . . . .	20
4.2.2	Sink Tree Construction and <i>Merging Sink Trees</i> . . . . .	20
4.2.3	Resolving Asymmetric Links . . . . .	21
4.2.4	Scalability Issues . . . . .	24
4.3	Scalability Enhancements . . . . .	25
4.4	Integration into SSR . . . . .	28
4.4.1	Link Substitution . . . . .	28

4.4.2	Inconsistencies of Knowledge . . . . .	32
4.5	Discussion . . . . .	35
4.5.1	Limitations of the Proposed Approach . . . . .	35
4.5.2	Detecting Broken (Asymmetric) Links . . . . .	36
<b>5</b>	<b>Implementation</b>	<b>39</b>
5.1	Overview on the <code>ssr-core</code> Library . . . . .	39
5.1.1	<code>cNode</code> Class . . . . .	39
5.1.2	<code>cMessage</code> Class and Subclasses . . . . .	40
5.1.3	Cache Classes . . . . .	41
5.1.4	Interface Store Classes . . . . .	42
5.2	Modifications within the <code>ssr-core</code> Library . . . . .	43
5.2.1	Extensions of the <code>cNode</code> Class . . . . .	43
5.2.2	<code>cMsgHelloWithTree</code> Class . . . . .	43
5.2.3	Modifications to the <code>cMsgConnect</code> Class . . . . .	44
5.2.4	<i>Early Path Optimization</i> in the <code>cMsgPayload</code> class . . . . .	44
5.2.5	<code>cSinkRouteCache</code> Class . . . . .	44
5.2.6	Incoming and Outgoing Interface Stores . . . . .	46
<b>6</b>	<b>Simulation Environment and Evaluation</b>	<b>47</b>
6.1	Overview on the Simulation Environment . . . . .	47
6.1.1	Simulator Functionality in the <code>OM_SSR</code> Class . . . . .	47
6.1.2	<code>cSsrNode</code> Wrapper Class . . . . .	48
6.1.3	Simplified MAC Layer . . . . .	48
6.1.4	Simulation Scenarios . . . . .	48
6.1.5	Further Simulation Parameters . . . . .	51
6.2	Simulation Results . . . . .	52
6.2.1	Impact of <code>SSR_EARLY_PATH_OPTIMIZATION</code> on SSR Overall Routing Performance in Absence of Asymmetric Links . . . . .	52
6.2.2	Results of <code>SSR_RANDOM_TREE</code> Mode . . . . .	54
6.2.3	Results of <code>SSR_UNRESOLVED_LINKS_FIRST</code> Mode . . . . .	56
6.2.4	Results of <code>SSR_DELTA_TREE</code> Mode . . . . .	58
6.2.5	Comparison of the Proposed HELLO Sink Tree Strategies . . . . .	60
6.2.6	Further Simulation Results . . . . .	62
6.2.7	Expanded Simulation Results Obtained with SSR Applying the <code>SSR_UNRESOLVED_LINKS_FIRST</code> Strategy . . . . .	65
6.3	Summary of the Simulation Results . . . . .	67
<b>7</b>	<b>Conclusion</b>	<b>69</b>
7.1	Final Remarks . . . . .	69

# List of Figures

2.1	Example setting of a 3-party proxy set . . . . .	7
3.1	Illustration of SSR routing process . . . . .	12
3.2	Appending further hops to a hop-by-hop message's source route . . . . .	14
3.3	Appending a shortcut to a hop-by-hop message's source route . . . . .	15
3.4	Protocol state machine visualizing SSR message forwarding . . . . .	16
4.1	Detecting a symmetric link to a physical neighbor . . . . .	21
4.2	Resolving an incoming asymmetric link(i) . . . . .	22
4.3	Resolving an incoming asymmetric link (ii) . . . . .	23
4.4	Protocol state machine of SSR message forwarding supporting asymmetric links . . . . .	29
4.5	Example for asymmetric link substitution . . . . .	30
4.6	Link substitution on a stub path after applying a shortcut . . . . .	31
4.7	Example setting inducing an inconsistency of knowledge . . . . .	32
4.8	Protocol state machine of SSR message forwarding supporting asymmetric links and SSR_EARLY_PATH_OPTIMIZATION . . . . .	34
4.9	Example setting containing an isolated node . . . . .	35
4.10	Example setting containing a partitioned network topology . . . . .	35
4.11	Example setting containing an asymmetric link that becomes unavailable . . . . .	36
5.1	Abstract class cNode . . . . .	39
5.2	Minimal class Diagram of SSR messages . . . . .	40
5.3	Simplified class diagram of route cache classes . . . . .	41
5.4	Simplified class diagram of interface store classes . . . . .	42
6.1	3 × 3 grid containing one wide transmission range node . . . . .	49
6.2	Impact of SSR_EARLY_PATH_OPTIMIZATION on the average path length per SSR_PAYLOAD message . . . . .	53
6.3	SSR overall routing performance using SSR_RANDOM_TREE mode to resolve asymmetric links . . . . .	54
6.4	SSR overall routing performance using SSR_RANDOM_TREE mode with SSR_EARLY_PATH_OPTIMIZATION extension . . . . .	55
6.5	Overall routing performance using SSR_UNRESOLVED_LINKS_FIRST mode . . . . .	56
6.6	Overall routing performance using SSR_UNRESOLVED_LINKS_FIRST mode with SSR_EARLY_PATH_OPTIMIZATION extension . . . . .	57

6.7	SSR overall routing performance using SSR_DELTA_TREE mode to resolve asymmetric links . . . . .	58
6.8	SSR overall routing performance using SSR_DELTA_TREE mode with SSR_EARLY_PATH_OPTIMIZATION extension . . . . .	59
6.9	Observed benefit of the particular HELLO sink tree strategies . . . . .	60
6.10	Observed benefit of the particular HELLO sink tree strategies using SSR_EARLY_PATH_OPTIMIZATION extension . . . . .	61
6.11	Scenario scheme, which is referred to as <i>bridge setting, type 1</i> . . . . .	62
6.12	Scenario scheme, which is referred to as <i>bridge setting, type 2</i> . . . . .	63
6.13	Applicability of SSR extension for asymmetric link extension support in bridge scenarios . . . . .	63
6.14	Associated average message hops graph of 6.13 . . . . .	64
6.15	Expanded simulation results using SSR_UNRESOLVED_LINKS_FIRST strategy . . . . .	65
6.16	Expanded simulation results using SSR_UNRESOLVED_LINKS_FIRST strategy and SSR_EARLY_PATH_OPTIMIZATION extension . . . . .	66

# List of Tables

5.1	Cache columns provided by an instance of <code>cSinkRouteCache</code> class	44
6.1	Estimated fractions of asymmetric links within the used configurations	50
6.2	Benefits and drawbacks of each particular HELLO sink tree strategy	61



# Chapter 1

## Introduction

Due to the increasing miniaturization of wireless sensor nodes (*'Smart Dust'*), more and more application scenarios for wireless sensor networks are becoming imaginable. These scenarios cover areas such as collecting of ecological data, clinical bedside monitoring or implementing intelligent office or living space.

Scalable Source Routing (SSR) [6] is a routing approach that aims at large unstructured networks like mobile ad hoc networks (*MANETs*), mesh networks and wireless sensor networks taking care of their specific properties, such as resource limitation of the used hardware platforms or frequently changing network topologies. SSR is a network layer protocol that combines source routing with Chord-like [19] indirect routing in a virtual, ring-structured address space.

Multiple empirical studies indicate that asymmetric links occur reasonably frequently in common wireless network scenarios. Thus, some benefit at the overall routing performance is anticipated when utilizing asymmetric links within routing protocols.

To the best of our knowledge, Dynamic Source Routing (DSR) [10] is the only currently implemented MANET routing protocol, which provides support for asymmetric links, but at considerable costs.

The objective of this diploma thesis hence is to develop an algorithm, which is capable of detecting asymmetric links, discovering appropriate reverse paths to occurring asymmetric links and to integrate the proposed approach into the Scalable Source Routing (SSR) protocol in order to support the utilization of asymmetric links during the SSR routing process. Finally, the proposed algorithm has to be evaluated using an existing simulation environment, which is implemented in the network simulator *OMNeT++* [20].

As this objective aims at an application of SSR in wireless scenarios, such as mobile ad hoc network or wireless sensor networks, we need to meet several context-sensitive restraints. Basically, a potential solution will have to deal with a limited storage capacity of wireless nodes as well as with comparatively small *MTUs* (*maximum transmission units*) provided by some common wireless MAC layers.

This thesis is organized as follows. In the second chapter we provide an overview on common settings inducing asymmetric links and outline other works related to the

issue of this thesis.

In chapter 3 we elaborately describe the `Scalable Source Routing` protocol. Subsequently, we analyze the `SSR` routing process in detail with regard to the posterior integration of the intended extension.

Our algorithm providing support for asymmetric links in `SSR` is introduced in chapter 4. Furthermore, we explain the modifications to the `SSR` protocol, which were required in order to be capable of utilizing asymmetric links during the routing process.

Chapter 5 describes classes of the `ssr-core` library that are relevant to the integration of the proposed extension into the `SSR` protocol.

Finally, the simulation environment as well as the obtained simulation results are depicted in chapter 6.

## Chapter 2

# Basics and Related Work

The initial sections of this chapter describe common wireless network scenarios inducing asymmetric links. Subsequently, other existing works related to the issue of utilizing asymmetric links within routing protocols are briefly summarized as well as some empirical studies, which provide some insights on the occurrence of asymmetric links in real world scenarios. Finally, we introduce some current work related to the application of the Scalable Source Routing protocol in wireless sensor networks.

### 2.1 Scenarios Inducing Asymmetric Links

As currently advised, the following sections provide an overview on common wireless settings inducing asymmetric links.

#### 2.1.1 Inherent Different Radio Capabilities

Heterogeneous nodes are the most obvious cause for asymmetric links. If the transmission range of a given node  $p$  is inherently larger than the transmission range of another node  $q$  and the distance between the two nodes is covered by the transmission range of node  $p$ , but exceeds the transmission range of node  $q$ , an asymmetric link directed from node  $p$  to node  $q$  is induced.

This kind of asymmetric link of course is a persistent phenomenon and thus quite auspicious to be utilized in routing protocols.

#### 2.1.2 Different Interference Levels

Homogeneous nodes can cause asymmetric links, too. Assume, for example, two nodes  $p$  and  $q$  that are providing an identical transmission range. Such a setting will not inevitably lead to a symmetric link between  $p$  and  $q$ . If - e.g. due to a locally increased node density within node  $q$ 's proximal environment - the given interference level at node  $q$  is significantly higher than the current interference level at node  $p$ , node  $q$  will

not be able to receive messages sent by node  $p$ .

Asymmetric links occurring due to different interference levels typically are relatively transient. Thus, asymmetric link detection mechanisms would need to work pretty efficiently in order to obtain any benefit of the utilization of this kind of asymmetric links.

### 2.1.3 Power Control Algorithms

Whenever battery lifetime is a critical issue, in particular when it comes to wireless sensor networks, there are two common techniques, which are commonly used: regularly setting the wireless node to sleep mode or reducing the wireless node's transmission power (if no traffic is present).

Reducing the transmission power has the disadvantage of increasing the likelihood of asymmetric links. That is, if the transmission power of a given node  $p$  is higher than the transmission power of a second node  $q$ , it may be possible for node  $q$  to receive a message from node  $p$ , but not for node  $p$  to receive a message that is sent by node  $q$ .

If nodes gradually reduce their transmission power depending on the decrease of their remaining battery lifetime, the induced asymmetric links will actually be reasonably persistent.

### 2.1.4 Impact of Asymmetric Links

Most network layer routing protocols assume any link to be symmetric. Exchanging topology information between neighboring nodes in general causes two issues: *knowledge asymmetry* and *routing asymmetry* [14].

The term *knowledge asymmetry* describes the observation that a source node  $p$  of an asymmetric link  $p \rightarrow q$  does not intuitively learn about the existence of this link whereas the sink node  $q$  obviously does.

Furthermore, in a setting as described above the path traversed from node  $q$  to node  $p$  necessarily needs to be different from a path that would be required to reach node  $p$  from node  $q$ . This observation is commonly referred to as *routing asymmetry*.

## 2.2 Related Work

The subsequent sections summarize some empirical studies investigating the occurrence of asymmetric links in specific wireless scenarios and provide an overview on other existing approaches concerning the issue of detection and utilization of asymmetric links within routing protocols.

### 2.2.1 Empirical Studies on the Occurrence of Asymmetric Links in Wireless Settings

Several real world studies of ad hoc networks indicate that asymmetric links make up a significant fraction of all links present.

*Ganesan et al.* observed that up to 15% of the links within their deployment, which consisted of 150 arbitrarily distributed nodes, are asymmetric even when each node is transmitting at the same transmission power and no additional radio sources exist in the given test arrangement [7].

Analogously, *De Couto et al.* [4] reported that up to 30% of all occurring links show asymmetric delivery rates in several indoor deployments of wireless nodes.

Furthermore, *Zhao et al.* [22] investigated multiple deployments of up to 60 Mica nodes and concluded that asymmetric links are quite common, i.e. at least 10% of all links have significantly asymmetric packet delivery rates (variation of  $> 50\%$ ) and thus are behaving as asymmetric links.

### 2.2.2 Local Asymmetric Link Detection Based on a Regular HELLO Protocol

In wireless settings, *HELLO* protocols are commonly employed to enable wireless nodes to gather information about their proximal environment.

Typically, each node participating in the given networking protocol periodically broadcasts a list of all nodes it received *HELLO* messages from within a defined time-out interval. Since these *HELLO* messages are designated to be exchanged between physical neighbors and not to be flooded across the whole network, they are supplied with a *TTL* (*time-to-live*) of 1.

Beyond the described functionality of discovering a node's physical neighborhood, a *HELLO* protocol could be rather beneficial at locally detecting asymmetric links and - for this additional purpose - is frequently employed within approaches concerning the utilization of asymmetric links in routing protocols as described in the subsequent sections.

Assuming an asymmetric link  $p \rightarrow q$  between two given nodes  $p$  and  $q$ , node  $p$  obviously will not receive any periodic *HELLO* messages sent by node  $q$ . Consequently, node  $q$  will not ever occur within *HELLO* messages sent by node  $p$ . As node  $q$  receives those *HELLO* messages of node  $p$  not containing its own address, node  $q$  is capable of determining that the corresponding link has to be an *incoming asymmetric link*. For node  $p$  however, there is no chance so far to even learn about the existence of node  $q$ . Obviously, this is one of the major issues that have to be solved in order to utilize asymmetric links within any routing protocol.

Within the described setting, node  $p$  would be denoted as *backward neighbor* of node  $q$  whereas node  $q$  would be called a *forward neighbor* of node  $p$ . A given link will be recognized as symmetric if and only if the adjacent nodes are backward neighbors and forward neighbors of each other at the same time.

### 2.2.3 Native Support of Asymmetric Links in DSR

Dynamic Source Routing (DSR) [10] is an ad hoc on-demand routing protocol, i.e. a source route between two given nodes  $p$  and  $q$  is not established until it is demanded by either node  $p$  or node  $q$ . A node initiates a so-called *Route Discovery* procedure by flooding the network with a *Route Request* (RREQ) message. Each time an intermediate node handles and forwards this RREQ message, it appends the incoming hop to the message's source route field. Thus, assuming that node  $p$  requires a source route to the destination node  $q$ , as soon as the RREQ message has reached node  $q$ , a source route in the direction from node  $p$  to node  $q$  is established.

In DSR the destination node  $q$  does not assume the currently obtained source route to be bidirectional, i.e. node  $q$  will not utilize the reversed source route in order to reach the issuer node  $p$ . Node  $q$  rather queries its route cache for a source route to node  $p$  or - if the route cache does not provide an active path to node  $p$  - it triggers another *Route Discovery* procedure in order to establish a new source route in the direction from node  $q$  to node  $p$ .

As DSR usually requires two *Route Discovery* procedures to establish bidirectional connectivity between two given nodes, it produces a considerable amount of control overhead and thus does not scale to larger sized networks.

### 2.2.4 Extension of Conventional Distance Vector Protocols

*Prakash* [16] investigated conventional distance vector routing algorithms and announced an extension providing support for the utilization of asymmetric links. This approach is based on exchanging  $O(n^2)$  sized matrix ( $n$  being the number of nodes in the given network topology) between neighboring nodes in order to be able to represent both possible directions of a link between any two nodes.

Although the storage requirements of this  $n \times n$  matrix representation could potentially be reduced by applying some *sparse matrix compression scheme*, this approach is not scalable with regard to increasing network sizes.

### 2.2.5 Approach Introducing Sink Trees

In [3], *Jorge A. Cobb* proposes an algorithm, which is based on the idea of periodically exchanging so-called *sink trees* between physically neighboring nodes. A node's sink tree virtually is a second route cache (here referred to as *source tree*). The semantical difference is that paths stored in the sink tree are directed towards the given node. These sink trees are used to detect and resolve potential asymmetric links.

On adding links to its sink tree, each node needs to check new or extended paths for a potentially introduced loop, i.e. the node checks whether its own address reoccurs

at some deeper level of its sink tree. Detecting a loop implies that a source route can be extracted of the considered path.

By this means, upon initially assuming any link to be asymmetric, nodes gradually discover source routes resolving these asymmetric links.

To the best of our knowledge on the current state of research, this approach produces the lowest control overhead of  $O(n)$ , where  $n$  is the number of nodes in a given network topology. Even though this would still not scale to larger networks, we decided to pick this approach up as a basis of our further investigations. Section 4.2 provides an extensive description of our variant of Cobb's algorithm.

### 2.2.6 Local Detection of Asymmetric Links

Wang et al. propose  $A^4LP$  [21], a *Location-aware and Power-aware routing protocol* for heterogeneous ad hoc networks with Asymmetric links. Basically, each node holds a set of its backward and a set of its forward neighbors. A node learns about its backward neighbors and about nodes that are backward as well as forward neighbors by employing a regular HELLO protocol as described in section 2.2.2.

The technique used to resolve backward neighbors beyond asymmetric links is delimited to so-called *3-party proxy sets*, i.e. sets of three neighboring nodes that contain at least one symmetric link and that are circularly connected among each other. Figure 2.1 depicts an example for a setting that is compliant to the above conditions for a 3-party proxy set.

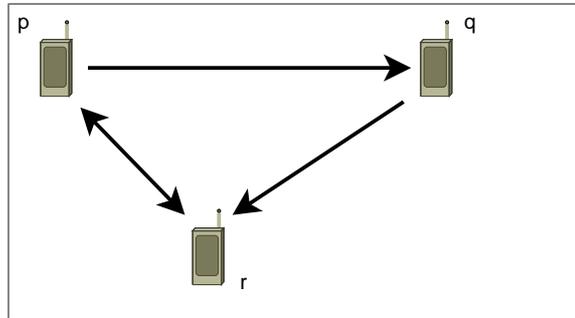


Figure 2.1: Example setting of a 3-party proxy set

Within this example, node  $r$ , which receives the HELLO messages of node  $p$  as well as the HELLO messages of node  $q$ , would notify node  $p$  of the existing asymmetric link directed from node  $p$  to node  $q$ . Henceforth, node  $p$  would be aware of this asymmetric link and capable of utilizing this link within the routing protocol.

Paths to destinations beyond a node's neighbor sets are established using an advanced flooding technique incorporating the nodes' location information, which has to be provided in any form.

In [18], *Sinha et al.* describe an extension of the *Zone Routing protocol (ZRP)*, which supports the utilization of asymmetric links.

The *zone* of a given node is defined as the set of all nodes in the current node's environment that are reachable within a certain radius measured in hops. Whereas regular HELLO messages typically are only exchanged between physical neighbors (see section 2.2.2), within this ZRP extension each node broadcasts the list of its backward neighbors within its zone, i.e. the TTL of the corresponding messages is set equal to the defined zone radius. Based on this backward neighbor information, a node is capable of calculating the shortest path to each node inside its zone. These paths are stored within a data structure that is denoted as a node's *outbound tree*.

Routes to destinations beyond a node's zone, i.e. the given node's outbound tree does not provide a path to the demanded destination, are established by applying a technique, which is referred to as *bordercasting*. A message directed to a destination outside the issuer node's zone primarily is forwarded to an appropriate border node, that is, some node in the most distant level of the issuer's outbound tree. If the current border node does not provide a route to the message's destination, it will again forward the message to a border node of its associated zone and so forth. This procedure is repeated until the considered message has reached its destination.

Bordercasting will result in a large amount of query messages, which in the worst case could be almost equally expensive as flooding the whole network. Hence, the scalability of the ZRP protocol with regard to increasing network sizes is quite disputable.

*Ramasubramanian et al.* [17] propose *BRA*, a *Bidirectional Routing Abstraction* for asymmetric mobile ad hoc networks. BRA provides a symmetric abstraction of the asymmetric network to arbitrary kinds of routing protocols, i.e. a sub-routing layer is introduced, which enables conventional routing protocols to utilize asymmetric links (the authors applied the AODV [15] protocol over BRA).

Basically, BRA employs a modified *Bellman-Ford* algorithm, which is denoted as *Reverse Distributed Bellman-Ford Algorithm (RDBFA)* [17] to discover reverse paths around occurring asymmetric links within a node's so-called *locality*, which is defined by a given radius measured in hops. This RDBFA algorithm periodically exchanges a *reversed distance-vector* reporting currently calculated distances *from* each other node within the sender's locality *to* the sender of a given distance-vector message.

The overhead for reverse route maintenance is claimed to be in  $O(n)$ , where  $n$  is the average number of nodes within a node's locality.

### 2.2.7 Applying Reverse Path Search in AODV

Similar to the DSR protocol as described in section 2.2.3, the *Ad hoc On-demand Distance Vector (AODV)* [15] routing protocol uses flooding of *Route Request* messages (abbr.: RREQ) to establish a path between a given source and the designated destination. Each hop traversed by an RREQ message is recorded within this message's source route field and thus, once the destination node receives an RREQ message, a source route between the source and the destination is obtained and propagated back to the source by sending a *Route Reply* (abbr. RREP) message containing the currently established source route back to the corresponding source node.

Obviously, if occurring asymmetric links are not being detected and blocked during this *path discovery* procedure, source routes obtained by AODV could contain asymmetric links and thus would potentially not be valid in the reverse direction from the

destination to the source.

An extension to the AODV protocol providing support for asymmetric links has been announced by *Marina et al.* [13]. Typically, several RREQ messages associated to the same *Route Discovery* procedure will reach the destination on different paths. In standard AODV, each intermediate node only handles the first RREQ message is whereas duplicates are recognized on the basis of their identical sequence number and immediately discarded.

The proposed extension to AODV suggests to handle each RREQ message at each particular intermediate node as well as at the destination in order to obtain multiple source routes in the direction from the source to the destination of a given *Route Discovery* procedure. The destination responds to each received RREQ message by sending an RREP message to the source utilizing the currently obtained source route. Furthermore the authors assume that at least one of the various paths traversed by the RREQ messages, which are received at the destination node, is completely symmetric and thus at least one RREP message would reach the source node.

Obviously, the latter assumption of at least one existing bidirectional path between any two nodes delimits the applicability of this approach. Furthermore, the scalability of the AODV protocol is significantly degraded due to the considerable additional control overhead induced by the proposed extension.

### 2.2.8 Reverse Tunneling

In [14], *Nesargi et al.* propose an approach based on the idea of tunneling MAC layer ACKs and control messages of the network layer routing protocol from the sink to the source of occurring asymmetric links.

A regular HELLO protocol as described in section 2.2.2 is employed to enable each participating node to detect incoming asymmetric links.

In order to be able to utilize asymmetric links within common MAC layer protocols demanding per frame ACKs (such as *IEEE-802.11 WLAN* [9]) as well as within the currently employed routing protocol, MAC layer ACKs and control messages of the routing protocol are tunnelled over the network layer protocol, i.e. these messages are encapsulated into regular network layer messages.

Establishing the required *reverse tunnel* from the sink to the source of an occurring asymmetric link is left to the employed routing protocol. Thus, the obtained performance and scalability of this approach is strongly dependent of the employed network layer routing protocol.

Supposing that an on-demand routing protocol like *AODV* or *DSR* (see sections 2.2.7 and 2.2.3) is employed, which is fairly common to wireless ad hoc network scenarios, the sink of an asymmetric link would have to initiate an additional route discovery procedure in order to establish an appropriate reverse path, upon receiving an RREP message propagating a source route that contains the corresponding asymmetric link.

### 2.3 Further Related Work on SSR in Mesh Networks

Towards the end of the editing time of this diploma thesis, we got in touch with *André Kaustell*, who wrote his master thesis [11] on the application of the SSR protocol in mesh networks. *Kaustell* proposed several optimizations concerning the scalability of SSR, which are quite interesting with regard to the issue of this work.

Based on the observation that the source routes that are contained in SSR payload messages occupy a considerable fraction of the given MAC layer frame, *Kaustell* proposed a technique referred to as *Virtual Route Compression (VRC)*. Typically, source routes are assembled of the particular nodes' network layer addresses. Applying VRC, each node assigns identifiers to its adjacent links. These identifiers are significantly smaller in size than common network layer addresses and thus a source route representation compressed to a fraction of its usual size could be obtained.

As these small sized link identifiers are only definite within a node's physical neighborhood, they could not be applied to represent entire source routes within a node's route cache. Thus, an additional *Route Request (RREQ)* message is required to traverse a given source route in order to assemble the link identifier representation of this source route, which is subsequently used to provide this source route within the actual SSR payload message.

Another optimization suggested by *Kaustell* is denoted as *Intermediate Node Shortcut Discovery*. This mechanism attempts to optimize the source route, which a given message has already traversed.

## Chapter 3

# Description and Analysis of SSR

This chapter gives a brief overview on Scalable Source Routing (SSR) protocol [6]. It helps to understand the considerations that are introduced in the following chapters. In particular, it explains the foundations for our proposed enhancement of SSR.

### 3.1 Indirect Routing

Indirect routing means that some node demanding some data object or service provided by any other node within the network uses a *DHT*- (*distributed hash table*, [2]) or *KBR*-based (*key based routing*, [8]) service, which maps the data object or service description to an address in the virtual ring. While DHTs deliver a node, which currently provides a certain data object or service, KBR provides a method to find the *closest* node for the requested data or service, according to some defined metric (e.g. the physical distance or the number of hops).

The overlay routing then transports the service request to the encountered address by forwarding the corresponding message in (counter-)clockwise direction of the address space until the distance between the address of the current node and the address of the requested data object or service cannot be minimized any further.

In SSR, a given node  $A$  is responsible for all data objects or services whose addresses are located between its own address and the address of some node  $B$ , whose address is the smallest of all nodes with addresses larger than node  $A$ 's address. In this constellation, node  $B$  is denoted as node  $A$ 's *successor* whereas node  $A$  is called node  $B$ 's *predecessor*. Obviously, correct and reliable overlay routing requires for each node in the network to know its actual successor. This condition is referred to as *consistency*.

Analogous to *Chord* [19], it is mandatory for each SSR node to store its physical neighbors as well as source routes to its predecessor and successor in the virtual address space within its route cache. However, whereas in *Chord* each node stores source routes to  $O(\log n)$  additional nodes at exponentially spaced distances to reduce the average request path length, in SSR, source routes to arbitrary nodes are gradually inserted into a node's route cache. Once the storage capacity of an SSR node's route cache is exhausted, entries are replaced according to *LRU* (*least recently used*) replacement policy.

### 3.2 Message Forwarding

As mentioned above, the concept of forwarding messages in Scalable Source Routing is to greedily decrease the distance in the virtual ring while preferring physically short paths. In this section we explain the SSR forwarding decision procedure in detail.

In SSR, payload messages contain a source address, a destination address and a source route. The included source route does not have to span a complete path from the originator  $s$  to the destination  $d$  as this would implicate exhaustive knowledge of the network topology to be able to send a request. In fact, a node adds a source route to the message, which, up to its current knowledge, obtains a reduction of the virtual distance to the message's destination  $d$ , i.e. the message is supplied with a source route to a *proxy destination* node, whose address resides between the address of the originator and the address of the actual destination.

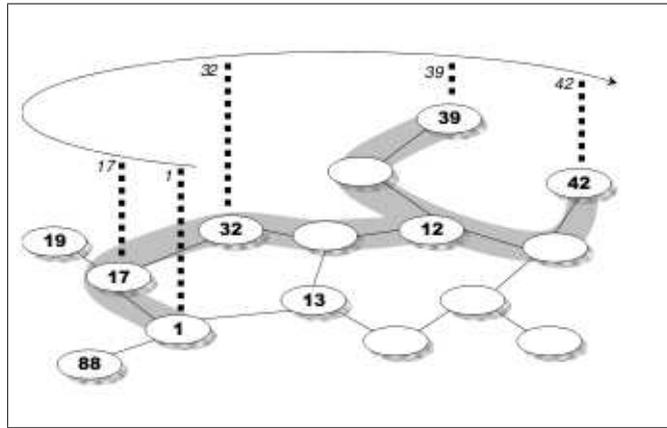


Figure 3.1: Illustration of SSR routing process

On receiving a message, each intermediate node forwards the message until the end of the contained source route is reached. Assuming that the last node  $i_1$  of the source route is not the message's destination  $d$ , the node  $i_1$  serves as a *mediator* and tries to append a source route from its route cache that will further reduce the virtual distance towards the destination  $d$  (details on appending further routing information to incomplete source routes are given in section 3.3). At this point, two distance metrics are used to proceed the forwarding decision:

1. *Physical* distance, measured in hops
2. *Virtual* distance, absolute value of the numerical difference between two nodes' addresses

The actual forwarding decision proceeds as follows. The mediator node  $i_1$  tries to append a source route attaining the message's destination  $d$ . If the route cache does not contain a path to node  $d$ , the mediator node  $i_1$  will determine another intermediate

node that is virtually closer to the destination  $d$ . As typically several such nodes exist, node  $i_1$  selects the physically closest one among those nodes. If there are still several nodes left to come into consideration, the virtually closest node to the destination  $d$  will be selected.

A brief example of SSR source routing is illustrated in figure 3.1. Node 1 demands to send a payload message to node 42. The message is forwarded to node 17, since that node is physically closest to node 1. Node 17 is preferred over node 13 due to being virtually closer to node 42. For the same reasons, the message is forwarded from node 17 to node 32. Node 32 forwards the packet to its successor, node 39, which in turn forwards the message to its successor that coincides with the destination, node 42.

### 3.3 Appending Source Routes and Applying Shortcuts

Since up to now any operation related to source routing in SSR is based on the assumption that any link between two nodes is symmetric and consequently source routes are completely bidirectional, we need to analyze the SSR routing process thoroughly. If we aim at utilizing asymmetric links as well, we obviously have to drop this assumption. Having depicted the basic principles of SSR in the previous sections, we will discuss further details of SSR source routing now, which have to be thought of accurately if the above assumption becomes invalid.

By default, *hop-by-hop* messages (payload messages actually are hop-by-hop messages) are forwarded along the source route that is contained in the message's header. On receiving this type of message, a node foremost checks if the message has hereby reached its destination. In case the source route does not contain any further hops and the message has not reached its destination yet, the current node would be a proxy destination, i.e. it has to operate as a *mediator* and thus to append a source route that will carry the message to its destination or at least virtually closer towards its destination (that is, to a further proxy destination).

In SSR, two options of appending routing information to a hop-by-hop message's source route are implemented. A mediator node could either *add further hops* towards the message's destination or *apply a shortcut* to the given source route. The latter option is depending on the mediator's knowledge of the network topology. If the mediator node knew an improved source route to the message's destination, which it is not part of itself, the remaining part of the original source route will be replaced by the new path. As expressed by the term *shortcut*, this kind of route optimization is not performed unless the number of remaining hops to the message's destination could be reduced. In order to redirect the hop-by-hop message onto its new source route, a so-called *stub path* from the mediator node to the adjacent node on this shortcut is added to the message (see figure 3.3). Eventually, hop-by-hop message forwarding proceeds as supplied before.

As we also use the currently mentioned *stub path* field in a hop-by-hop message's header to provide a path *to* the node that most recently modified the message's source route, we need to record the mediator node in this stub path field anyway, even though this mediator did not apply a shortcut to the current source route. Since on occurrence of a link failure on the message's source route we intend to inform the most recent

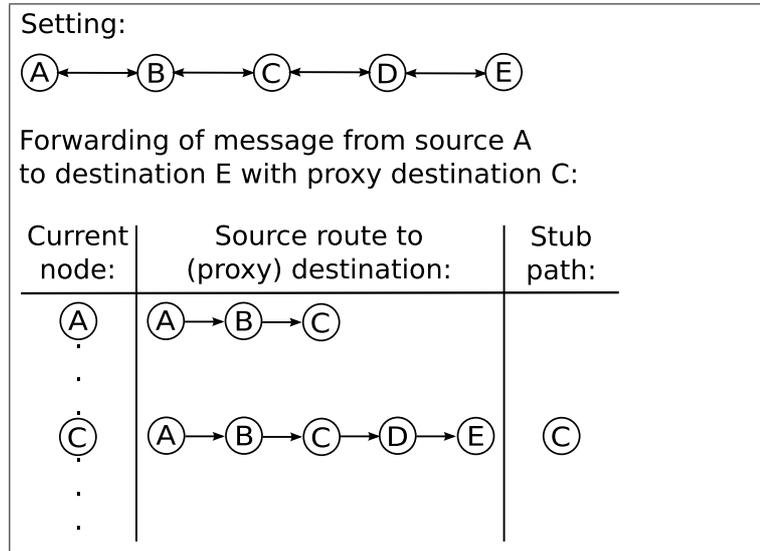


Figure 3.2: Appending further hops to a hop-by-hop message's source route

mediator node (as well as each subsequent intermediate node between the most recent mediator and the node detecting the broken link), recording the most recent mediator of a hop-by-hop message is essential (further details on reporting unavailable links are depicted in section 3.4 below).

Figure 3.2 illustrates the elementary case of appending routing information onto a message's source route. Node *A* demands to send a message to node *E*, but its route cache only provides a path to node *C*, which actually is virtually closer to the destination node *E*. In this example, node *C* is able to entirely complement the source route to the message's destination *E* and is being recorded as most recent mediator in the message's stub path field.

An example of a mediator node applying a shortcut to a message's source route is depicted in figure 3.3. The initial source route provided by the issuer node *A* carries the message to node *D*. As the mediator node *D* knows the shortcut  $C \rightarrow E \rightarrow F$  to the message's destination *F*, the hop  $\rightarrow D$  is cut off from the source route and replaced by the shortcut. Furthermore, the link  $D \rightarrow C$  is being inserted into the message's stub path field.

On appending additional routing information to an incomplete source route as depicted above, it would be reasonable to notify the message's originator about the appended path. Thereby, that node could treat similar requests by itself henceforth. Since it is not desirable to transfer routing information, which is possibly overaged, from one route cache to another, we do not send an `SSR_ROUTE_UPDATE` message until the complemented path has been entirely traversed. Thus, upon receiving the considered message, the destination node sends an `SSR_ROUTE_UPDATE` message back to the issuer along the reversed path. By this means, reporting potentially broken links directly from one node's route cache to another node's route cache is avoided.

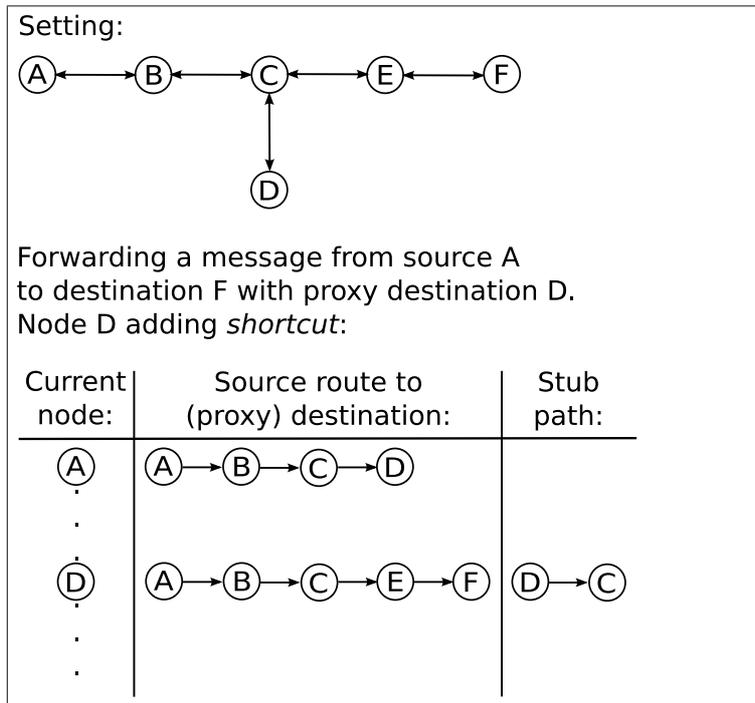


Figure 3.3: Appending a shortcut to a hop-by-hop message's source route

### 3.4 Handling Broken Links

On observing that the next hop of an incoming hop-by-hop message's source has become unavailable by now, the current intermediate node needs to inform the node that most recently modified the source route (i.e. the node that has actually appended the broken link to the message's source route) about the link failure. Thus, using the reverse path back to the message's mediator respectively to its issuer node, it sends an `SSR_ROUTE_UPDATE` message containing the broken link.

If the message's source route had previously been modified by adding a shortcut, the reversed stub path would be used to create a path carrying the `SSR_ROUTE_UPDATE` message to that most recent mediator node. Anyway, each node that could possibly have added the broken link to its route cache while forwarding the corresponding message would be notified of the link's current unavailability, i.e. the reported link will be marked as inactive in each intermediate node's route cache as well as in the mediator or issuer node's route cache.

### 3.5 State Maintenance

SSR guarantees consistent routing if and only if all nodes provide valid source routes to their virtual neighbors, that is its predecessor and successor within the virtual address ring. In this section we give a brief description of the algorithm that we currently use to establish this *consistency* condition of the virtual ring. Since this component of SSR is still an issue of research and deployment, the version described below only reflects

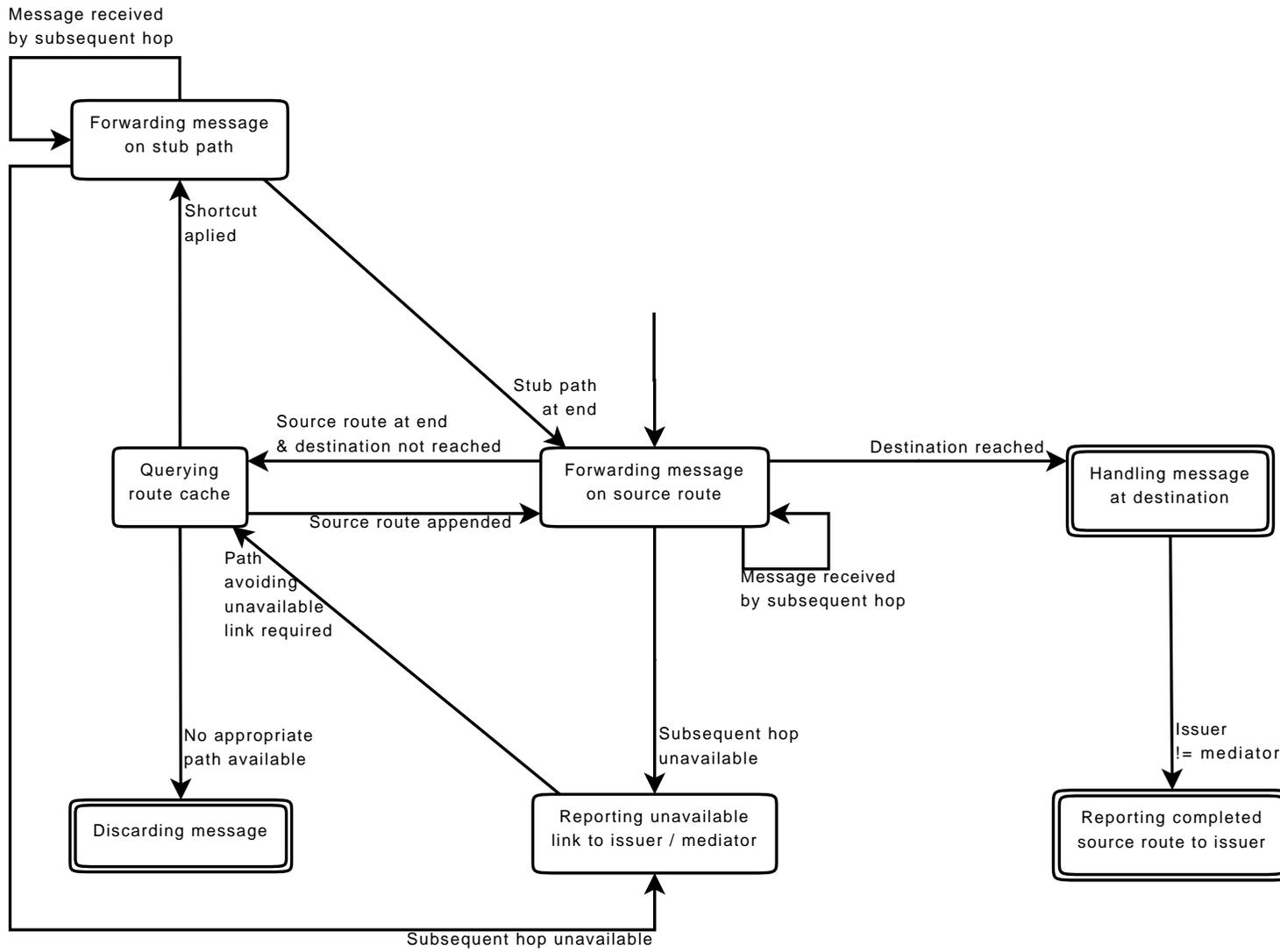


Figure 3.4: Protocol state machine visualizing SSR message forwarding

the state of the stable SSR implementation at the beginning of this work.

The following algorithm is triggered by a periodic event denoted as `SSR_NOTIFICATION` event. In the first step, a node handling this kind of event has to check - based on its local knowledge, i.e. the information stored in its route cache - whether its address could be the maximum address in the used address space, i.e. whether the current node itself could be the correct predecessor of the defined minimum address *zero*. If no other node with an address between the own address and the *zero* address was found in the route cache, an `SSR_MAX_NODE_ANNOUNCE` message would be assembled and sent per broadcast.

In case it is not possible for a given node to be the holder of the maximum address, this node queries its route cache for the node, which could most likely be its virtual neighbor and initiates an `SSR_NEIGHBOR_NOTIFICATION` message to that node. Like payload messages, `SSR_NEIGHBOR_NOTIFICATION` messages are hop-by-hop messages containing a source route to the currently assumed virtual neighbor.

On receiving an `SSR_MAX_NODE_ANNOUNCE` message, a node needs to check whether - up to its current knowledge of the network topology - there is another node whose address is bigger than the issuer's address. If such a node was found in the node's route cache, flooding of the `SSR_MAX_NODE_ANNOUNCE` message would be aborted. Otherwise, the current node also checks whether the message's issuer could be its virtual neighbor and, if so, initiates an `SSR_NEIGHBOR_NOTIFICATION` message to that node. Either way, flooding of the `SSR_MAX_NODE_ANNOUNCE` message continues.

The handling of `SSR_NEIGHBOR_NOTIFICATION` messages proceeds as follows. Each intermediate node on the message's source route queries its route cache whether it contains a better neighbor candidate (i.e. a node whose address is between the issuer's address and the message's current destination's address) of the message's issuer than the message's current destination. In case a better neighbor candidate is found the message's source route is updated. Eventually, the message is forwarded along the (new) source route to its (new) destination.

In figure 3.4 we depict a protocol state machine that summarizes the particular steps of message forwarding in SSR as described in the previous sections.



## Chapter 4

# Approach Based on *Partial Sink Trees*

In this chapter we propose an algorithm for detecting and resolving asymmetric links. It is based on an approach by *Jorge A. Cobb* [3]. We extend the basic algorithm that is described in section 4.2 by applying certain assumptions about wireless sensor network topologies (as described in section 4.3) and integrate it into the `Scalable Source Routing` protocol (see section 4.4). Finally, we discuss some limitations and special cases in section 4.5.

### 4.1 Assumptions and Simplifications

Within this work (as well as in each related work that is summarized in section 2.2), we assume that for each pair  $(p, q)$  of nodes in a given network topology, there exists a path in the direction from node  $p$  to node  $q$  as well as in the reverse direction from node  $q$  to node  $p$ . Formally expressed, we assume a *directed graph*, which moreover is *strongly connected*.

A *directed graph*  $G$  is defined as a pair  $G := (V, E)$ , whereas  $V$  represents a set of *vertices* (here: the nodes in the given network topology) and  $E$  represents a set of directed edges, i.e. a set of ordered pairs of vertices (here: the links in the given network topology). Moreover, if such a directed graph is *strongly connected*, this graph will contain at least one *directed cycle* for each particular link  $(p, q)$ , i.e. as  $(p, q)$  represents an edge, which is directed from vertex  $p$  to vertex  $q$ , the existence of a sequence of edges  $\{(q, i), \dots, (j, p)\}$  in the reverse direction is assured.

This common assumption is necessary to exclude nodes with only incoming or only outgoing links.

Furthermore, we assume that each node learns about the nodes in its physical neighborhood through periodically exchanged HELLO messages. In case of a symmetric link between two neighbors  $p$  and  $q$ , both nodes will automatically learn of the existence of this link.

If, for example, the link between the nodes  $p$  and  $q$  is an asymmetric link, which is directed from node  $p$  to node  $q$ , node  $q$  will learn of this link, i.e. node  $q$  will learn

that node  $p$  is a *backward neighbor*, on receiving node  $p$ 's HELLO messages, whereas node  $p$  will not ever learn that node  $q$  is a *forward neighbor*.

## 4.2 Basic Algorithm

In the subsequent sections we describe the basic idea of *Cobb's* approach [3], which uses so-called *sink trees* to detect and resolve occurring asymmetric links. Finally, we discuss this approach with regard to its scalability.

### 4.2.1 Sink Tree Data Structure

As mentioned above, the approach to asymmetric link detection that we pursued during this work is based on a *sink tree* data structure, which each participating node needs to hold. Physical neighbors exchange their sink trees within their periodically broadcasted HELLO messages.

Basically, a node's sink tree is a second route cache (which is also referred to as a node's *source tree*, see section 3.1), but semantically, paths stored in the sink tree are directed towards the given node, i.e. the node itself is the common sink for all paths contained in its sink tree. In a route cache/source tree, however, paths are directed from the given node to various destinations in the network, i.e. the owner of a source tree is the common source of all paths contained in its source tree.

Furthermore, within the sink tree, we store two additional properties of each link: the *asymmetric link* property and the *unresolved link* property. Virtually, the *asymmetric link* property is a flag, which we use to mark a link as asymmetric whereas the *unresolved link* property is a flag denoting (potentially asymmetric) links to whom the given node does not provide a reverse path yet.

### 4.2.2 Sink Tree Construction and Merging Sink Trees

As mentioned above, physical neighbor exchange their sink trees periodically within HELLO messages. On joining the network, a node's sink tree only contains the sink, i.e. its own address. Paths are added gradually on reception of periodic HELLO messages broadcasted by its neighbors. We denote this procedure that is described within this section as *merging* of sink trees. Since it is essential to clearly distinguish between a node's local sink tree and a foreign sink tree received in a HELLO message, we will refer to the latter as *HELLO sink tree* henceforth. Within SSR, we denote this type of HELLO messages containing a sink tree as `SSR_HELLO_WITH_TREE` messages.

Once a node receives a message of type `SSR_HELLO_WITH_TREE`, it foremost checks whether its local sink tree already contains the sender of this HELLO message. If no entry is found in the first level of the local sink tree, the given node inserts the incoming link *from* the sender, which - at least preliminary - is marked as an asymmetric link.

The second step is the previously mentioned procedure of *merging* the paths contained in the HELLO sink tree into the node's local sink tree. This is achieved with

an iteration over the set of the HELLO sink tree's leaf nodes. For each leaf node, we extract the path towards the HELLO sink tree's sink, i.e. the path to the sender of the given `SSR_HELLO_WITH_TREE` message, from the HELLO sink tree. Beginning with the link from the node next to the sink up to the sink itself, we need to check for each particular link of the current path, whether the node's local sink tree already contains the considered link or not yet. The first case indicates, that, at the most, we have to update the asymmetric link property of the corresponding entry in the local sink tree. In the latter case, we need to copy the considered entry from the received HELLO sink tree to the node's local sink tree.

Note that we initially assume each link to be asymmetric, i.e. the asymmetric link property of each entry in the local sink tree is set, but once the asymmetric link property of an entry is removed, we will not ever set it again based on information contained in a received HELLO sink tree. Since we do not assume synchronous clocks among the nodes, a node is not capable to decide whether the information contained in a received HELLO sink tree is more up to date than the information that is stored in its local sink tree. Therefore, we solve this issue using `SSR_ROUTE_UPDATE` messages (see section 3.4).

### 4.2.3 Resolving Asymmetric Links

A node uses its knowledge about the network topology, which it gathers on the basis of received `SSR_HELLO_WITH_TREE` messages from physical neighbors and which it stores in its local sink tree, to detect asymmetric links and to discover appropriate *reverse paths*. In this case, the term *reverse path* denotes a source route around an occurring asymmetric link, for instance a source route to a backward neighbor whose messages are received but that cannot be reached in the reverse direction. We refer to this procedure as *resolving asymmetric links* and describe it in detail within this section.

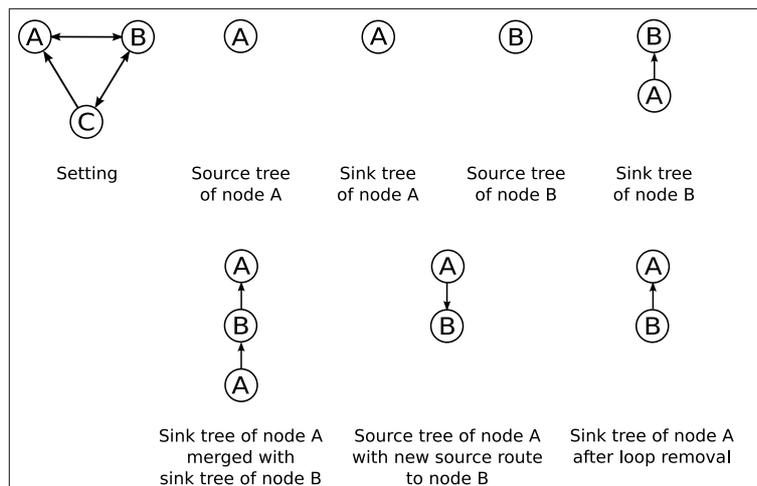


Figure 4.1: Detecting a symmetric link to a physical neighbor

The algorithm is based on the following elementary observation: If we found a *loop path* within a node's local sink tree, i.e. a path that contains the current node's address twice, we could obtain a new source route. Potentially, this new source route might be used to forward messages to some backward physical neighbor that has not been reachable so far due to an asymmetric link.

First of all, this explanation will concentrate on finding a loop within one dedicated path that has currently been added to a given node's local sink tree. As suggested above, a loop path is detected once the node's own address, i.e. the *sink* or equivalent the *root* of the sink tree, occurs again within the considered path.

A loop path  $s \leftarrow n_i \leftarrow s$  ( $s$  denoting the sink, i.e. the owner of the given local sink tree) of length 2 indicates that the link between the sink  $s$  and the physical neighbor denoted as  $n_i$  has to be a symmetric link. Hence we add the path  $s \rightarrow n_i$  to the source tree of node  $s$  whereas the duplicated entry for node  $s$  at the second level of the local sink tree will be deleted.

Figure 4.1 illustrates the described procedures employing a simple setting with three nodes  $A$ ,  $B$  and  $C$  with a single asymmetric link that is directed from node  $C$  to node  $A$ .

When being viewed from behind, a loop path with a length of  $k$  hops,  $k > 2$ , formally expressed as  $s \leftarrow n_1 \leftarrow n_2 \leftarrow \dots \leftarrow n_{k-1} \leftarrow s$  provides a source route  $s \rightarrow n_{k-1} \rightarrow \dots \rightarrow n_2 \rightarrow n_1$  to the physical neighbor  $n_1$ . Assuming an incoming asymmetric link  $s \leftarrow n_1$ , node  $s$  could resolve this incoming asymmetric link by inserting the currently discovered source route into its source tree.

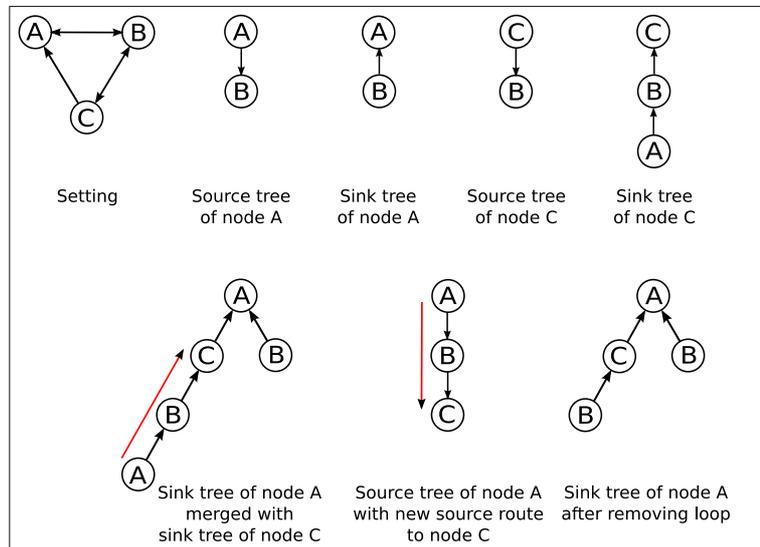


Figure 4.2: Resolving an incoming asymmetric link(i)

Continuing the example started above (see figure 4.1), this time we consider the loop path  $A \leftarrow C \leftarrow B \leftarrow A$ . Viewing this loop path from behind leads to a source route  $A \rightarrow B \rightarrow C$  to node  $C$ . Since there is an incoming asymmetric link  $A \leftarrow C$

in this examples's setting (i.e. node  $C$  is an unresolved backward neighbor of node  $A$ ), node  $A$  adds the new source route  $A \rightarrow B \rightarrow C$  to its source tree whereas the duplicated entry at the second level (*sink*  $\leftrightarrow$  *levelzero*) of the local sink tree is being deleted. We depict these steps in figure 4.2.

On detecting a loop path  $s \leftarrow n_1 \leftarrow n_2 \leftarrow \dots \leftarrow n_{k-1} \leftarrow s$  spanning  $k > 2$  hops, there is another potential of resolving an incoming asymmetric link.

Obviously, there is an - at least - outgoing link  $s \rightarrow n_{k-1}$  from node  $s$  to the second node of the loop path,  $n_{k-1}$ , i.e. node  $n_{k-1}$  is in reach of node  $s$ . If we found a disjoint path  $s \leftarrow n_n \leftarrow \dots \leftarrow n_k \leftarrow n_{k-1}$  down from this physical neighbor  $n_{k-1}$  in the local sink tree with  $s \leftarrow n_n$  being an incoming asymmetric link, we could resolve this link with the source route  $s \rightarrow n_{k-1} \rightarrow n_k \rightarrow \dots \rightarrow n_n$ . This source route is assembled of the first hop of the loop path,  $n_{k-1} \leftarrow s$ , and the disjoint path to  $n_n$  excluding the incoming asymmetric link  $s \leftarrow n_n$ .

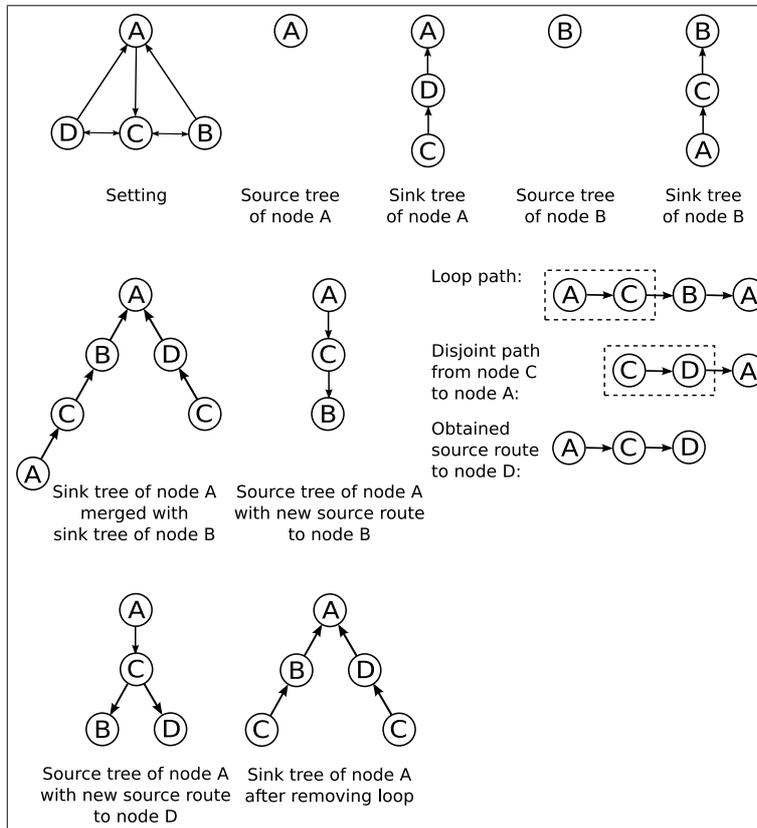


Figure 4.3: Resolving an incoming asymmetric link (ii)

Figure 4.3 shows an example setting where we apply the procedure as described above to resolve the incoming asymmetric link  $A \leftarrow D$  at node  $A$ . After merging the HELLO sink tree received from node  $B$  into the local sink tree of node  $A$ , the loop path  $A \leftarrow B \leftarrow C \leftarrow A$  is detected and thus the source route  $A \rightarrow C \rightarrow B$  is extracted as has been shown in figure 4.2 and described in the previous example.

The first hop of the loop path provides an outgoing link  $A \rightarrow C$ . Within this example, the local sink tree contains a disjoint path  $A \leftarrow D \leftarrow C$  originating at node  $C$ . Concatenating this path excluding the last hop  $D \leftarrow A$  onto the outgoing link  $A \rightarrow C$ , we assemble a new source route  $A \rightarrow C \rightarrow D$  to the unresolved neighbor  $D$ .

#### 4.2.4 Scalability Issues

If the proposed algorithm was used as described above, a node's local sink tree would grow continuously until it contained any link of the given network topology. As each symmetric link may occur twice, i.e. once for both directions, the storage complexity for the local sink tree would be in  $O(2n)$  whereas  $n$  is the number of directed edges in the network graph. Even if we assume nodes, which provide sufficient storage capacity to hold this amount of information, the given algorithm does still not scale to larger sized networks, since it is not practicable to exchange periodic HELLO messages of this dimension.

Technically, the primary constraint to meet is the *MTU* (*maximum transfer unit*), which is specific to the underlying MAC layer protocol. In practice - depending on the actual node density - even HELLO messages compliant to the given MTU could be too large. In wireless settings, the higher propagation delay of large messages implies an increasing collision probability at media access, which has to be avoided in order to prevent a grave break-in of overall network throughput. Furthermore, it is a principle of network protocol design to spend as little bandwidth as possible for control overhead.

In the subsequent section we discuss how the given algorithm could be adapted to larger sized networks while concurrently dealing with appropriately small HELLO messages.

### 4.3 Scalability Enhancements

Within this section, we propose several strategies on filling `SSR_HELLO_WITH_TREE` messages, which physical neighbors periodically exchange, with partial information of a node's local sink tree. Furthermore, we analyze these strategies with regard to their intended impact and potential side effects.

First of all, it is mandatory to include a node's currently known physical neighbors in each HELLO message. Otherwise, these HELLO messages would not be useful with regard to their intrinsic purpose of efficiently integrating new nodes into the SSR protocol anymore.

Basically, we integrated three strategies on selecting local sink tree information to be copied to `SSR_HELLO_WITH_TREE` messages into the basic algorithm that we described in section 4.2:

- `SSR_UNRESOLVED_LINKS_FIRST`
- `SSR_DELTA_TREE`
- `SSR_RANDOM_TREE`

Effectively, we only employ the third strategy to obtain comparative data during the evaluation stage of this work (as explained below).

The proposed strategy referred to as `SSR_UNRESOLVED_LINKS_FIRST` strategy is based on specific assumptions, which we derived of common wireless sensor network settings. First of all, we assume an uniform distribution of the wireless sensor nodes, which implies an at least approximately uniform node density within the network topology. Furthermore, we assume that a major fraction of the wireless sensor nodes provides an equal minimum transmission range whereas a minor fraction of the wireless sensor nodes provides an enhanced transmission range. Thus, asymmetric links induced by this kind of wide transmission range nodes usually are links that are non-essential with regard to the overall network connectivity, i.e. the network topology would still be compliant to the conditions of a connected graph (see section 4.1) if these asymmetric links were ignored respectively blocked. Nonetheless, these links could provide reasonable shortcuts when utilized in source routing. Besides that, the latter assumption suggests that the loop paths, which our algorithm requires to resolve this kind of asymmetric links, should only span a few hops.

Taking these assumptions as a basis, we propose the following approach. As long as the local sink tree's size is below or equals the space, which is provided in `SSR_HELLO_WITH_TREE` messages, we completely copy the local sink tree into the current HELLO message. Upon exceeding the space provided in HELLO messages, we apply the following prioritization to select partial information of the local sink tree:

*close unresolved links > remote unresolved links > close resolved links > remote resolved links*

According to this order, paths containing close unresolved links are primarily copied to the HELLO sink tree. If there is still some space left within the HELLO message, paths containing remote unresolved links will be selected and so forth. On the other

hand, there could be a potential for paths only containing resolved links to be helpful at resolving asymmetric links as well. Therefore two alternative perceptions of this prioritization seem to be reasonable.

If we enforce this prioritization for each particular `SSR_HELLO_WITH_TREE` message, paths containing close unresolved links will be sent in each `HELLO` message until they get resolved. Then the same procedure continues with farther unresolved links and so on. Thus, we denote this strategy on selecting partial information of the local sink tree to be copied to current `HELLO` messages as `SSR_UNRESOLVED_LINKS_FIRST` strategy. Depending on the space provided in `HELLO` messages, it could take a considerably long period of time, until paths only containing resolved links are selected and copied to a `HELLO` message - if at all.

Thus, particularly if the space provided for the local sink tree was relatively small, it would be reasonable to enforce the prioritization as described above by sending the entire local sink tree divided up to several `SSR_HELLO_WITH_TREE` messages, i.e. we send close unresolved links in the first `HELLO` message(s), subsequent `HELLO` messages contain farther unresolved links and so forth, until the local sink tree has been entirely transmitted. We refer to this variant of the currently proposed `SSR_UNRESOLVED_LINKS_FIRST` strategy as `SSR_DELTA_TREE` strategy.

Obviously, the `SSR_UNRESOLVED_LINKS_FIRST` is aimed at predominantly resolving asymmetric links within a node's local area. Consequently, there is a potential to enhance the overall routing performance of the SSR protocol with support for asymmetric links by trying to improve a hop-by-hop message's source route at each intermediate hop, i.e. each forwarding node attempts to immediately utilize currently resolved asymmetric links, which could possibly abbreviate the message's current source route. Furthermore, if this so-called `SSR_EARLY_PATH_OPTIMIZATION` extension accomplishes to improve a message's source route, we initiate an `SSR_ROUTE_UPDATE` message to the node (issuer or mediator) that most recently modified this source route. Thereby, that node could apply this source route optimization by itself in the future and thus shortcuts are gradually being distributed across the network.

As discussed in section 3.3, one of the principles of SSR is not to transfer routing information, which could possibly be overaged, from one route cache to another and thus `SSR_ROUTE_UPDATE` messages reporting route optimizations should not be initiated until the intended destination has been reached. The proposed `SSR_EARLY_PATH_OPTIMIZATION` extension of course offends this principle, but nonetheless, there is at least one argument supporting an implementation of this extension:

If we utilize asymmetric links during the SSR routing process, the source route a message has traversed upon reaching its destination will typically not be identical to the reverse path back to the message's issuer, which would be used by a subsequent `SSR_ROUTE_UPDATE` message (see section 4.4.1). Thus, the source route is not or at least not completely validated upon the reception of the `SSR_ROUTE_UPDATE` message at the precedent hop-by-hop message's issuer and therefore could possibly be overaged as well.

Clearly, the `SSR_EARLY_PATH_OPTIMIZATION` extension will also have an impact on SSR source routing without utilizing asymmetric links, which we need to observe during the evaluation stage of this work.

Our third strategy on selecting paths to be copied to `SSR_HELLO_WITH_TREE` messages virtually fills the `HELLO` message with paths that we pick of the local sink

tree at random and thus is referred to as `SSR_RANDOM_TREE` strategy. Using this strategy, each path in a node's local sink tree has the same probability to be sent in the subsequent HELLO message. As mentioned above, this is more of a *brute force* method than any kind of sophisticated strategy and thus is basically employed to compare the simulations results of the `SSR_UNRESOLVED_LINKS_FIRST` strategy as well as the `SSR_DELTA_TREE` strategy to.

Obviously, the proposed strategies on constructing HELLO sink trees according to a prioritization of local sink tree information as described above aim at efficiently resolving asymmetric links within the proximal environment of a given node. As long as occurring asymmetric links are the exception rather than the rule and the corresponding loop paths are not exceptionally long, asymmetric links should be resolved within few HELLO message intervals, i.e. the calculational best case would be:

$$time_{resolving\ asymmetric\ link} = \#hops\ of\ loop\ path * period_{HELLO\ messages}$$

## 4.4 Integration into SSR

Having introduced the proposed algorithm to detect and resolve asymmetric links in the previous sections, in the subsequent sections we describe modifications specific to SSR, which are essential in order to utilize asymmetric links within the source routing process.

### 4.4.1 Link Substitution

Before detection mechanisms for asymmetric links have been implemented in SSR, source routing was based on the assumption of paths being completely bidirectional, i.e. the destination node of a hop-by-hop message could simply reverse the contained path and send its reply to the message's issuer node. Clearly, if source routes contain asymmetric links are, this assumption will not apply any more.

We apply a principle referred to as *link substitution* in order to construct a reverse path back to the issuer while forwarding a message on the contained source route. For this purpose, we introduced a so-called *path-to-issuer* field within the header of hop-by-hop messages. On receiving a hop-by-hop message, each node has to check whether the incoming hop is a symmetric link or an asymmetric link. Obviously, we merely need to reverse symmetric links before we append them to the message's current path-to-issuer. In the latter case, i.e. the message was received over an asymmetric link, the current intermediate node needs to query its source tree for a path back to the previous hop on the message's source route in order to conveniently complement the reverse path to the hop-by-hop message's issuer. If the node's source tree does not provide an appropriate path, i.e. the message has recently been forwarded by an unresolved backward neighbor, the message will be discarded.

It is important to note that, at this particular point, initiating SSR's route update mechanism, i.e. distributing the information that the corresponding link should not be used any further, is not desirable at all. Such an approach would simply exploit this operation for asymmetric link resolving issues. Asymmetric links should only be resolved by using the proposed algorithm based on HELLO\_WITH\_TREE messages as described above (see section 4.2.3).

A solution compliant to SSR could be a time-delayed retransmission of the corresponding hop-by-hop message, assuming that the concerning unresolved link would be resolved shortly.

Figure 4.5 depicts a short example of the currently described *link substitution* principle. Node *A* is sending a message with source route  $A \rightarrow B \rightarrow C \rightarrow E$  containing an asymmetric link, which is directed from node *B* to node *C*. On receiving this message, node *C* needs to replace the incoming asymmetric link  $B \rightarrow C$  by some appropriate reverse path back to node *B*. In this example's setting, node *C* already provides the knowledge about the path  $C \rightarrow D \rightarrow B$  and thus replaces the asymmetric link  $B \rightarrow C$  by the reverse path  $C \rightarrow D \rightarrow B$ .

For the sake of clarity, this figure only shows how the path to a message's issuer is constructed by substituting each asymmetric link on the corresponding source route with an appropriate reverse path. However, it is important to note that the *source route* field and the *path-to-issuer* field actually are distinct fields within the header of hop-by-hop messages and therefore intermediate nodes do not modify the source route field



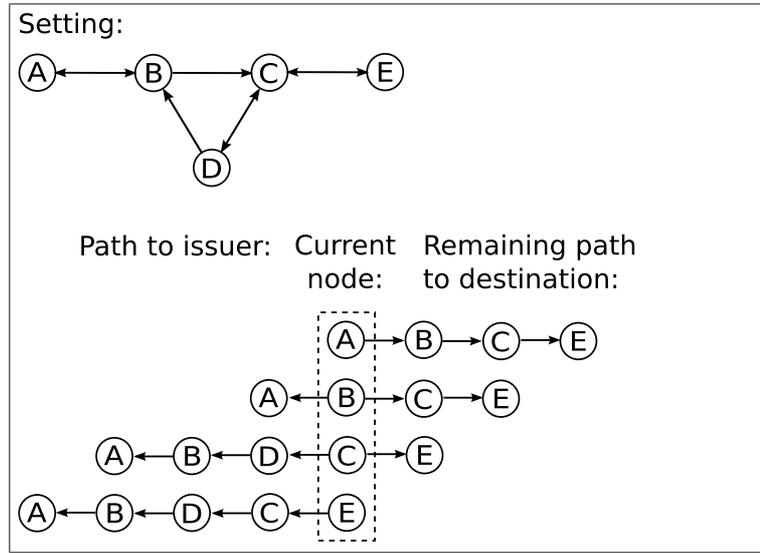


Figure 4.5: Example for asymmetric link substitution

during the link substitution procedure.

As described in section 3.3, SSR provides two options of extending incomplete source routes by some mediator node. The last node on a hop-by-hop message's source route either has to append further hops towards the message's destination or to apply a shortcut to the message's source route. In case of an occurring link failure on a currently traversed source route, we intend to notify the node that most recently modified this source route. As explained earlier, we use the *stub path* field in a hop-by-hop message's header to either record a mediator node appending further hops to the message's source route or to insert a 'real' stub path carrying the message onto its new source route after applying a shortcut.

Since naturally a stub path may contain asymmetric links, too, analogous to the previously introduced *path-to-issuer* field, we need to add a *path-to-mediator* field to hop-by-hop messages' headers. While a message is being forwarded on a stub path, we likewise employ the *link substitution* principle as described above to obtain a reverse path to the mediator node, that most recently applied a shortcut to the message's source route.

Figure 4.6 shows an example setting where the *link substitution* principle is applied on a stub path as well as on the regular source route. Node *A* demands to send a message to node *F*, but is only able to provide a source route  $A \rightarrow B \rightarrow C \rightarrow D$  to the proxy destination *D*. While message forwarding is proceeded as described above, the asymmetric link  $B \rightarrow C$  is substituted with the reverse path  $C \rightarrow E \rightarrow B$ .

The mediator node *D* applies the a shortcut to the hop-by-hop message's source route by cutting off the path  $B \rightarrow C \rightarrow D$  and appending the link  $B \rightarrow F$  instead, i.e. the message's new source route is  $A \rightarrow B \rightarrow F$ . In order to carry the message onto this source route, node *D* adds the stub path  $D \rightarrow C \rightarrow E \rightarrow B$  to the message. While traversing this stub path, the corresponding reverse path to node *D* is being constructed and recorded within the message's *path-to-mediator* field. Since the stub path contains

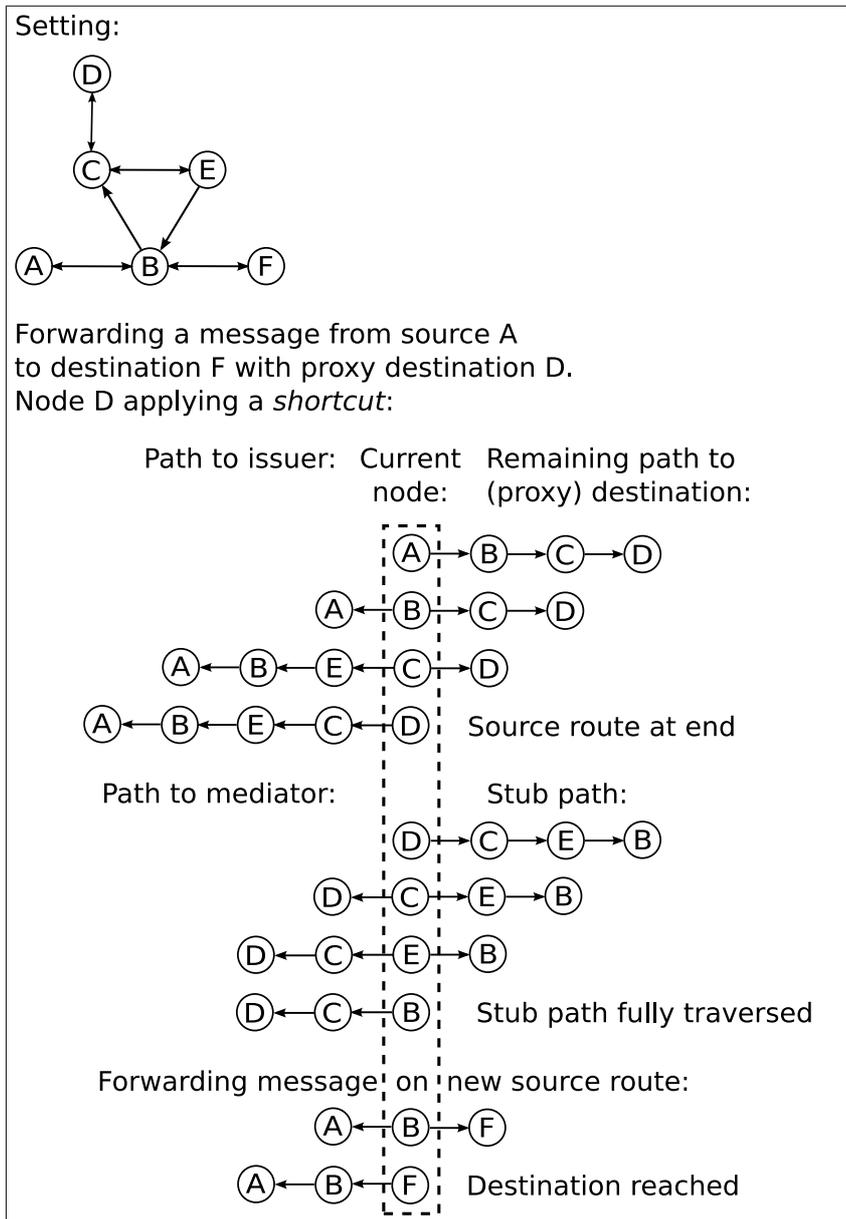


Figure 4.6: Link substitution on a stub path after applying a shortcut

the asymmetric link  $E \rightarrow B$ , node  $B$  needs to substitute this link with the reverse path  $B \rightarrow C \rightarrow E$ . This link substitution induces a loop  $C \rightarrow E \rightarrow C$  within the *path-to-mediator*  $B \rightarrow C \rightarrow E \rightarrow C \rightarrow D$ , which is detected and cut out from the current *path-to-mediator*.

Consequently, the message has reached its new source route at node  $D$  and is finally forwarded to its destination, node  $F$ .

Recapitulatory, figure 4.4 highlights the modifications to the basic SSR protocol state machine as shown in figure 3.4, which were required in order to provide support for the utilization of asymmetric links.

Furthermore, the implementation of the `SSR_EARLY_PATH_OPTIMIZATION` extension as described in section 4.3 of course requires additional modifications to the SSR protocol state machine. We emphasize these modifications in figure 4.8.

Each intermediate node handling a hop-by-hop message needs to query its route cache in order to improve the remaining part of the currently traversed source route or stub path. If an optimization is accomplished, the current mediator node needs to send an `SSR_ROUTE_UPDATE` message reporting the optimized source route to the node that most recently modified this source route, i.e. a previous mediator respectively the issuer node.

#### 4.4.2 Inconsistencies of Knowledge

An asymmetric links always has the property that it takes more time for its source to learn about its existence than it takes for its sink. In early states of SSR bootstrapping, this property could induce inconsistent knowledge of the actual network topology among neighboring nodes.

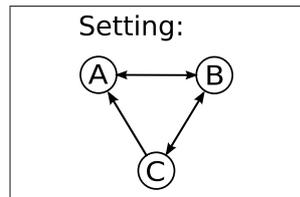


Figure 4.7: Example setting inducing an inconsistency of knowledge

Due to these inconsistencies of knowledge, it is possible for a node to receive a hop-by-hop message and not to know the next hop (i.e. an unknown forward neighbor) on the contained source route yet, although this node is actually active and thus could receive and forward the corresponding message. However, since SSR already provides the possibility to distinguish between unknown and broken links (we do not instantaneously delete a broken link from the outgoing interface store, but mark it as `unavailable`, see section 5.1.4 and 5.2.6), we can solve this issue by forwarding the message per broadcast. Based on the hop-by-hop message's source route and hop count, any node receiving the broadcasted message is capable of determining whether it is the message's intended subsequent recipient or not. Clearly, each false recipient

would discard the message.

Figure 4.7 illustrates a setting that could lead to a temporary inconsistency of knowledge as described above. As node  $B$  would definitely learn of the asymmetric link  $C \rightarrow A$  prior to node  $C$ , node  $B$  could use this link at source routing and node  $C$  would - at least for a short period of time - not even know about the existence of node  $A$ .

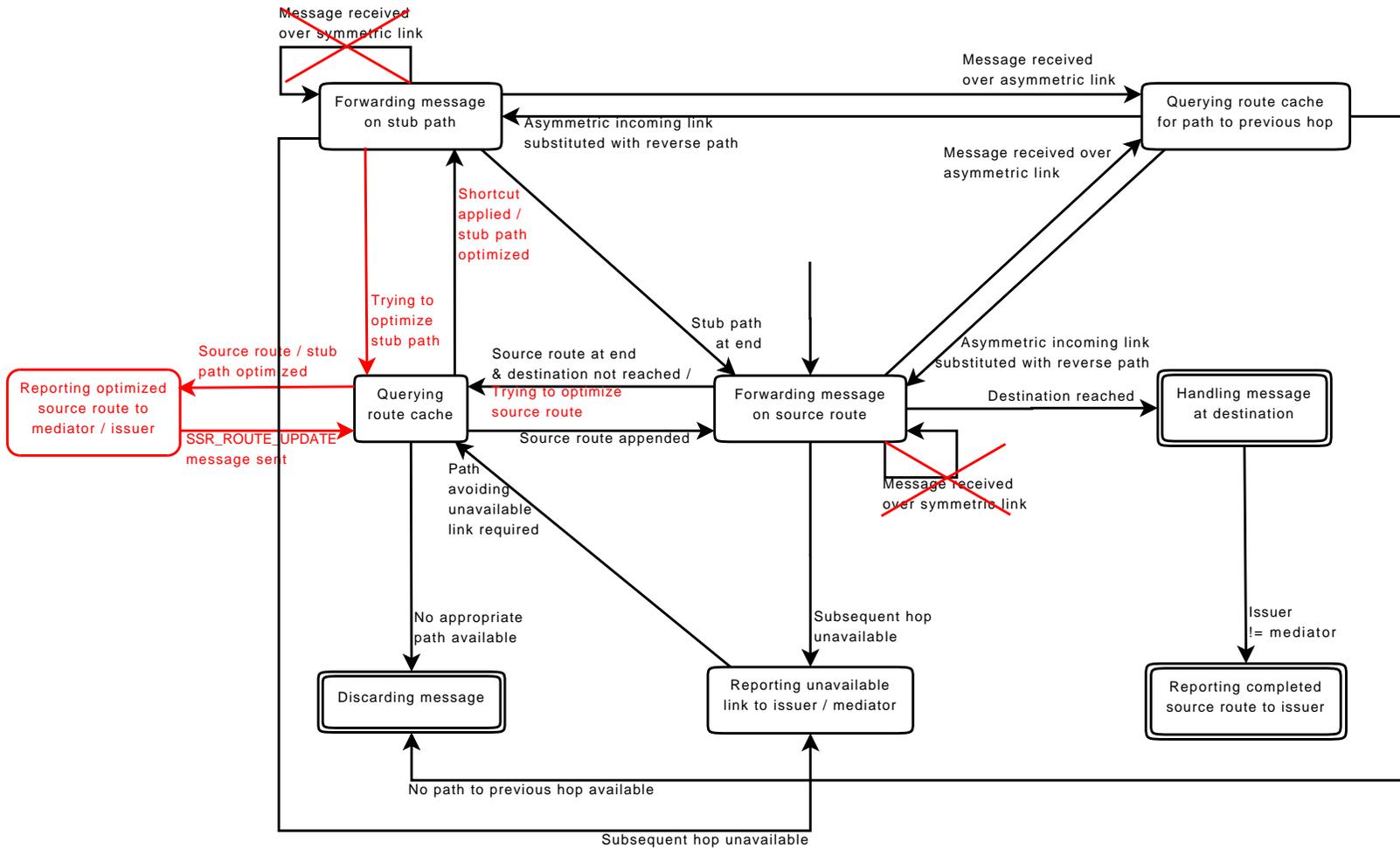


Figure 4.8: Protocol state machine of SSR message forwarding supporting asymmetric links and SSR\_EARLY\_PATH\_OPTIMIZATION

## 4.5 Discussion

In the subsequent sections we discuss some issues that are not finally solved within our approach.

### 4.5.1 Limitations of the Proposed Approach

As explained in 4.1, for each pair of adjacent nodes in the network we assume the existence of at least one directed cycle containing both of these nodes. Thus, for each occurring asymmetric link in the network topology, the existence of at least one reverse path that could resolve this asymmetric link is assured.

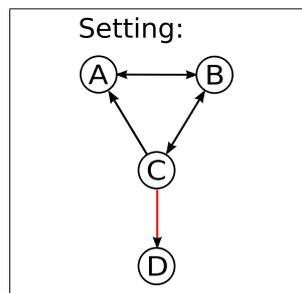


Figure 4.9: Example setting containing an isolated node

In practice, this assumption does not necessarily apply. Therefore in this section we discuss some consequences that could be anticipated if the proposed approach is employed in certain settings that are not compliant to the assumption described above.

The first setting to be discussed is depicted in figure 4.9. Due to the asymmetric link  $C \rightarrow D$ , node  $D$  will never be discovered by any other node in this example's network topology. Node  $D$  receives the periodic HELLO messages of node  $C$ , but the asymmetric link  $C \rightarrow D$  cannot be resolved since no reverse path to node  $C$  exists.

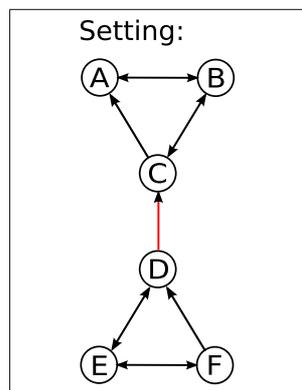


Figure 4.10: Example setting containing a partitioned network topology

Figure 4.10 illustrates a network topology, which is partitioned due to the unresolvable asymmetric link  $D \rightarrow C$ . Analogous to the previous setting, the lower partition of the given network does not learn of the upper partition's existence at all. Nevertheless, neither source routing nor the algorithm resolving asymmetric links is affected within the lower partition. As node  $D$  actually is a backward neighbor of node  $C$ , this kind of network partition, which is hidden behind an asymmetric link, is also referred to as a *backward partition*.

However, within the upper partition (which analogously is denoted as a *forward partition*), the algorithm resolving asymmetric links indeed is affected, since node  $C$  receives the HELLO messages of node  $D$  and merges the contained HELLO sink tree into its local sink tree (see section 4.2). Consequently, node  $C$  will propagate the asymmetric link  $D \rightarrow C$  (and, subsequently, the asymmetric link  $F \rightarrow D$  as well), which actually is a close unresolved link of node  $C$ , across the forward partition of the network topology. Obviously, this information is of no use for any of the nodes within the forward partition, since that link is not resolvable at all. Furthermore, some space in the local sink trees of the forward partition's nodes as well as in the HELLO sink trees that are exchanged between those nodes is wasted by including (superfluous) topology information of the backward partition.

Assuming a large backward partition containing plenty of asymmetric links that are relatively close to the forward partition, the forward partition will constantly get flushed with superfluous sink tree information of the backward partition. Consequently, the proposed algorithm will not be able to utilize the space provided in HELLO messages optimally and thus will take a considerably longer period of time to resolve occurring asymmetric links within the forward partition.

### 4.5.2 Detecting Broken (Asymmetric) Links

If we utilize asymmetric links during the routing process, a given source route and its reverse path typically are - at least partially - different from each other.

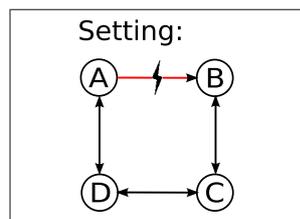


Figure 4.11: Example setting containing an asymmetric link that becomes unavailable

Within the SSR protocol, we detect a failing symmetric link once no HELLO message of the corresponding physical neighbor is received within a defined link timeout interval.

Considering an asymmetric link  $A \rightarrow B$  as depicted in figure 4.11, node  $A$  of course does not receive HELLO messages that are sent by node  $B$  at all. Thus, on the basis of HELLO message reception, only the sink, node  $B$ , is capable of detecting a failing asymmetric link.

In order to solve this issue, node  $B$  could employ the reverse path  $B \rightarrow C \rightarrow D \rightarrow A$  to send an `SSR_ROUTE_UPDATE` message reporting the broken link  $A \rightarrow B$  to its former source, node  $A$  (as well as to each intermediate node on the reverse path). However, in the worst case, a failing asymmetric link could lead to a partitioned network topology if there is no redundant directed cycle containing both of the nodes that are adjacent to this failing asymmetric link (see section 4.1). Clearly, the same consideration applies to a failing link on the reverse path: If a redundant reverse path existed, it would eventually be discovered, otherwise SSR would necessarily fail.

As long as an unavailable asymmetric link has not been detected due to absent HELLO messages, there is no possibility to efficiently detect this link failure during the routing process. Ordinary per hop acknowledgements are not efficiently applicable when utilizing asymmetric links. If, referring to the above example, per hop acknowledgements from node  $B$  to node  $A$  were tunnelled over the reverse path  $B \rightarrow C \rightarrow D \rightarrow A$  (as proposed by *Nesargi et al.* [14]) or if end-to-end acknowledgements were applied, node  $A$  would not be able to definitely identify the link that actually has become unavailable on the basis of an absent acknowledgement of node  $B$ . Thus, the tunnelled or end-to-end acknowledgement is required to be hop-wise acknowledged as well and - if a single per hop acknowledgement gets lost - the routing protocol has to be employed to discover an alternative reverse path that could carry the tunnelled or end-to-end acknowledgement to its destination (node  $A$ ).

Due to the considerable complexity of such enhanced acknowledgement mechanisms, it seems to be rather recommendable (within the SSR protocol at least) to rely on local detection of link failures, discarding potentially overaged source routes and time-delayed message retransmissions instead.

In case of a broken asymmetric link, the sink needs to notify the source by employing the corresponding reverse path or - if the current reverse path is overaged - by employing the routing protocol to discover a new reverse path. Basically, any technique of reporting broken (asymmetric) links will only work if the network topology still provides a directed cycle containing both of the nodes that are adjacent to the broken link.



# Chapter 5

## Implementation

The scope of this chapter is to provide as much documentation as necessary to orient oneself within the implementation and to be able to use or to extend it.

Foremost, we give a brief overview on all classes of the `ssr-core` library that are relevant to the integration of the proposed algorithm and extensions as described in the previous chapter 4.

In the subsequent sections we locate specific implementation parts within the `ssr-core` library and explain major extensions or modifications that have been integrated during the implementation stage of this work.

A complete and exhaustive analysis of the `ssr-core` library is given in the study thesis *Scalable Source Routing in the AmbiComp Environment* by *Fabian Knittel* [12].

### 5.1 Overview on the `ssr-core` Library

Within the following sections, we outline the purpose of each class of the `ssr-core` library that has been added or extended during the implementation stage of this work. Furthermore, we provide some basic information on the architecture of the `ssr-core` library, which is crucial to understand how the accomplished modifications fit into the context of this architecture.

#### 5.1.1 `cNode` Class



Figure 5.1: Abstract class `cNode`

The `cNode` class is an abstract class representing an SSR node in the network. It contains any functionality related to interaction within the implementation of SSR and defines interfaces for interaction with upper and lower layers of the protocol stack. The latter is implemented providing a set of pure virtual functions (for example `sendToNic`, `sendUp`). By implementing these functions within a child class of

`cNode`, the `ssr-core` library could be adapted to various environments and applications.

### 5.1.2 `cMessage` Class and Subclasses

This section will give a brief description of the hierarchical architecture of message classes within the `ssr-core` library, which is depicted in the class diagram 5.2.

The abstract superclass `cMessage` of all message classes provides common properties of all derived types of messages, such as fields for *network layer address* of the source node and the message's *hop count*.

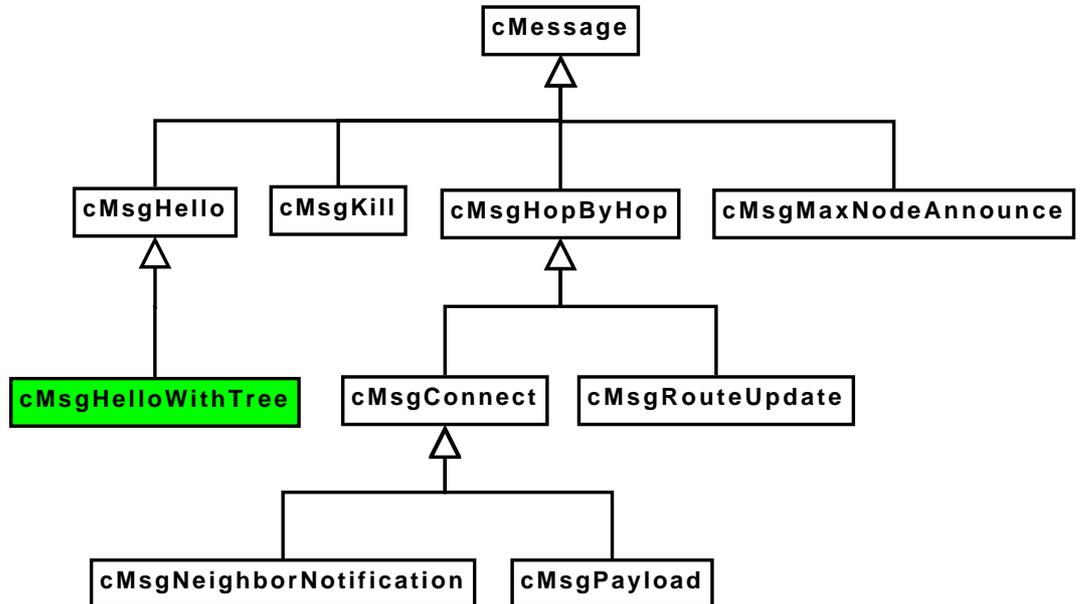


Figure 5.2: Minimal class Diagram of SSR messages

SSR\_HELLO messages as derived from the `cMsgHello` class are sent per broadcast in order to distribute information about the physical environment and contain a list of the sender's currently known physical neighbors.

We added the subclass `cMsgHelloWithTree` in order to create extended HELLO messages of type `SSR_HELLO_WITH_TREE` containing a sink tree (see sections 4.2 and 4.3). We provide further details concerning this class in section 5.2.2.

Once a node running SSR is going down, a message of type `SSR_KILL` (an instance of the class `cMsgKill`) is sent out per broadcast in order to preliminary inform the node's physical neighbors.

Messages of type `SSR_MAX_NODE ANNOUNCE` (`cMsgMaxNodeAnnounce`) are - as well as `SSR_NEIGHBOR_NOTIFICATION` messages, which we describe below - used by the SSR *state maintenance* algorithm (see section 3.5) in order to establish the

consistency state of the virtual ring, i.e. to discover each node's virtual neighbors.

The `cMsgHopByHop` class is the abstract superclass of any message type containing a source route to the intended destination or at least a source route towards a proxy destination. These messages contain a path and potentially a stub path (see section 3.3).

As described in chapter 3, we employ messages of type `SSR_ROUTE_UPDATE`, which are created as instances of the `cMsgRouteUpdate` class, for at least two reasons. We either use these messages to report a broken link to the - issuer or mediator - node that most recently modified a previous hop-by-hop message's source route or to report a complemented or improved source route to a previous hop-by-hop message's destination to the corresponding issuer node.

Furthermore, if we enable the proposed `SSR_EARLY_PATH_OPTIMIZATION` extension (see section 4.3), each time an intermediate node optimizes a hop-by-hop message's source route, it will send an `SSR_ROUTE_UPDATE` message to the node that most recently modified this source route. Consequently, messages of this type contain complete source routes, which should not be modified by the SSR routing algorithm (see sections 3.3 and 3.4).

In contrast to that, a node sending an `SSR_NEIGHBOR_NOTIFICATION` or `SSR_PAYLOAD` message occasionally cannot provide a complete source route to reach the intended destination. Thus we implemented the basic mechanisms of appending source routes and applying shortcuts (see 3.3) within the abstract `cMsgConnect` superclass of the `cMsgNeighborNotification` and `cMsgPayload` classes.

When sending an `SSR_NEIGHBOR_NOTIFICATION` message, the issuer node selects the - up to its local knowledge - most likely virtual neighbor candidate and provides a source route to this potential virtual neighbor. As each intermediate checks whether it could update the message's current destination with a better virtual neighbor candidate, `SSR_NEIGHBOR_NOTIFICATION` messages usually contain a relative destination, i.e. we aim at discovering the virtually closest node to the `SSR_NEIGHBOR_NOTIFICATION` message's source.

### 5.1.3 Cache Classes

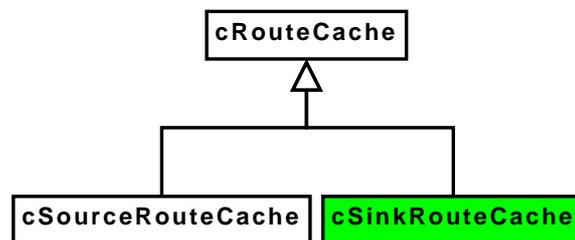


Figure 5.3: Simplified class diagram of route cache classes

Within the hitherto existing implementation of the `ssr-core` library, a node's route cache has been created as an instance of the `cRouteCache` class. This class contains the actual route cache functionality, such as assembling a path to or towards

a given virtual address, inserting, removing or refreshing links (i.e. cache lines) and a replacement algorithm applying *LRU* (*least recently used*) replacement policy (see section 3.1).

#### 5.1.4 Interface Store Classes

An instance of the `cNode` contains one instance of the `cInterfaceStore` class, which an SSR node uses to hold its physical neighbors. This interface store basically provides an *ARP*-like mapping of the physical neighbors' network layer addresses to their MAC layer addresses.

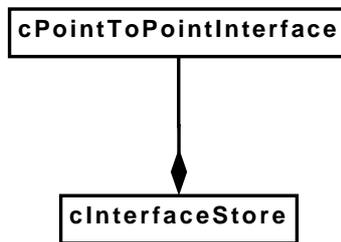


Figure 5.4: Simplified class diagram of interface store classes

For each physical neighbor, an instance of the `cPointToPointInterface` class is created, which the SSR node uses to store the physical neighbor's MAC address and to observe its state, i.e. a physical neighbor is marked as *active* as long as its periodic HELLO messages are received within a defined link timeout interval.

In order to support the utilization of asymmetric links in SSR, we need to distinguish between *incoming* and *outgoing* physical neighbors. We provide further details on this issue in section 5.2.6.

## 5.2 Modifications within the `ssr-core` Library

In the subsequent sections we describe major modifications and extensions to the `ssr-core` library, which have been integrated during the implementation stage of this work.

Thereby we outline how the novel functionality of the algorithm and extensions as proposed in the previous chapter 4 has been implemented and explain where these particular components or modifications are located within the `ssr-core` library.

### 5.2.1 Extensions of the `cNode` Class

Basically, we extended the `cNode` class by adding a local sink tree, i.e. an instance of the `cSinkRouteCache` class (see section 5.2.5), as well as a second instance of the `cInterfaceStore` class, since it is required to distinguish between incoming and outgoing interfaces henceforth (refer to section 5.2.6 for further details).

The novel function `checkForUnresolvedLinks()` performs an iteration over the links stored in the local sink tree updating the `isUnresolved` property of each particular entry by querying the node's source tree for a matching reverse path. We employ this functionality to implement a prioritization of unresolved links when selecting paths of the local sink tree to be copied to an `SSR_HELLO_WITH_TREE` message. Thus, we call the function `checkForUnresolvedLinks()` each time after having processed a received `HELLO` sink tree.

Furthermore, some minor modifications were required to store some parameters, which we need to configure the `HELLO` sink tree construction procedure, namely the maximum `HELLO` sink tree size provided in `SSR_HELLO_WITH_TREE` messages, the strategy that is used to select local sink tree information to be copied to the `HELLO` sink tree, or to enable/disable the `SSR_EARLY_PATH_OPTIMIZATION` extension (see sections 4.3 and 5.2.5).

### 5.2.2 `cMsgHelloWithTree` Class

The novel `cMsgHelloWithTree` class is derived of the `cMsgHello` class of regular `HELLO` messages. Instead of a list of a node's physical neighbors, `HELLO` messages of this type contain a `HELLO` sink tree, i.e. an instance of the `cSinkRouteCache` class, which we create by using a dedicated constructor for `HELLO` sink trees (see section 5.2.5).

Within this class, the mandatory `handle(...)` function of the `cMsgHello` class is overwritten. First of all, we either have to add the sender of a currently received `SSR_HELLO_WITH_TREE` message to the incoming interface store or we have to refresh the corresponding instance of the `cPointToPointInterface` class. If the `HELLO` sink tree contains a link directed from the current node towards the sender of the `HELLO` sink tree, we will add the sender to the outgoing interface store as well (see section 5.2.6). Furthermore, this function provides the implementation of the algorithm searching for loops within paths that have recently been merged into the local

sink tree (see sections 4.2.3 and 5.2.5).

### 5.2.3 Modifications to the `cMsgConnect` Class

As described in section 4.4.1, each intermediate node handling a message of type `SSR_NEIGHBOR_NOTIFICATION` or `SSR_PAYLOAD` needs to extend the message's *path-to-issuer* field and potentially its *path-to-mediator* field by appending a reverse path to the previous hop on the message's source route or stub path.

Since this operation is required for both, `SSR_NEIGHBOR_NOTIFICATION` and `SSR_PAYLOAD` messages, it has been implemented within the common superclass `cMsgConnect`.

### 5.2.4 Early Path Optimization in the `cMsgPayload` class

If we enable the application of the `SSR_EARLY_PATH_OPTIMIZATION` extension, each intermediate node receiving an `SSR_PAYLOAD` message will need to query its route cache in order to attempt to improve the message's source route or a currently traversed stub path (see section 4.3). Regular SSR message forwarding would only append routing information or apply a shortcut if either the message's source route was at its end or the subsequent hop on the message's source route had become unavailable (see sections 3.2 and 3.4).

### 5.2.5 `cSinkRouteCache` Class

As described in section 4.2.1, a sink tree virtually is a route cache and thus the `cSinkRouteCache` class is derived from the `cRouteCache` class. Since HELLO sink trees are created as instances of the `cSinkRouteCache` class as well as nodes' local sink trees, but have different attributes, we implemented two different constructors within this class.

address	uplink	...	asymLink	isUnresolved	sentInPrevHelloMsg	time

Table 5.1: Cache columns provided by an instance of `cSinkRouteCache` class

The constructor used to instantiate local sink trees practically creates an empty route cache object of type `SINK_TREE`. Optionally, a maximum number of cache lines can be passed to this constructor. Table 5.1 illustrates the additional cache columns that had to be established in order to implement specific functionality of the local sink tree data structure, namely the properties `asymLink`, `isUnresolved` and `sentInPrevHelloMsg`.

On receiving a message of type `SSR_HELLO_WITH_TREE`, we use a node's local sink tree's member function `mergePathIntoLocalSinkTree(path, helloSinkTree)` in order to merge the currently selected path of the given HELLO sink tree into the node's local sink tree (see section 4.2.2). For each link that is already

contained in the local sink tree, we increment the time stamp of the corresponding cache line.

We will update the asymmetric link property as well if and only if the considered link is marked as symmetric in the HELLO sink tree, but is marked as asymmetric in the local sink tree. Since we do not assume synchronized clocks, a node is not capable of distinguishing whether the information stored in the local sink tree or the information received with the HELLO sink tree is more up-to-date. Thus, once we have marked a link as symmetric in the local sink tree, we assume this link as symmetric until we receive an SSR\_ROUTE\_UPDATE message, reporting that the link has become unavailable in a given direction.

Once a node's local sink tree is running short of space, we use a cache line replacement algorithm enforcing an *LRU* (*least recently used*) replacement policy, i.e. entries are discarded in ascending order of their time stamps (see section 5.1.3).

As described in the sections 4.2 and 4.3, HELLO sink trees may be constructed as identical copies of a node's local sink tree. This is the default mode (referred to as SSR\_SINK\_TREE\_COPY), which we apply as long as the current size of the local sink tree is below or equals the maximum size of provided for a HELLO sink tree.

As soon as the local sink tree's size exceeds the space SSR\_HELLO\_WITH\_TREE messages provide for the the HELLO sink tree, this SSR\_SINK\_TREE\_COPY mode is not applicable any longer and thus we employ one of the following strategies (see section 4.3):

- (SSR\_SINK\_TREE\_COPY)
- SSR\_UNRESOLVED\_LINKS\_FIRST
- SSR\_DELTA\_TREE
- SSR\_RANDOM\_TREE

Independent of the configured strategy, it is mandatory to include a node's physical neighbors in the HELLO sink tree (this is equivalent to copying the first level of the local sink tree to the HELLO sink tree). Once this step is finished, the constructor branches out depending on the selected strategy ( $\leftrightarrow$  SSR\_SINK\_TREE\_MODE).

The functions `getClosestUnsentUnresolvedLink(localSinkTree)` and `getClosestUnsentResolvedLink(localSinkTree)` are used for the SSR\_UNRESOLVED\_LINKS\_FIRST strategy as well as for the SSR\_DELTA\_TREE strategy. Basically, these functions deliver paths containing the closest unresolved respectively resolved link of the local sink tree that has not been copied to the currently constructed HELLO sink tree yet.

Furthermore, these functions utilize the `hasBeenSentInPreviousHelloMsg` property of the local sink tree's sink tree cache lines in order to delimit their selection procedure on links that have not been sent within the current series of HELLO messages. We only need this property while applying the SSR\_DELTA\_TREE strategy. Thus it is being ignored by other strategies.

Employing the SSR\_RANDOM\_TREE strategy, random paths of the local sink tree (that have not been copied to the HELLO sink tree yet) are delivered using the HELLO sink tree's member function `getFurtherRandomPathOfLocalSinkTree(maxHopDepth, localSinkTree)` and copied to the HELLO sink tree until the

provided space is entirely filled up.

The function `copyPathToHelloSinkTree(path, localSinkTree, maxHopDepth)` is used to copy a selected path from the local sink tree to a HELLO sink tree. Once the remaining space  $k$  in the currently constructed HELLO sink tree is not sufficient for a selected path of length  $l > k$  to be copied to the HELLO sink tree completely, only the first  $k$  hops of the given path will be copied. We need this functionality in order to be able to optimally utilize the space provided in `SSR_HELLO_WITH_TREE` messages.

Furthermore, if a *maximum hop depth* parameter  $k$  is passed to this function, only the first  $k$  hops of the given path will be copied to the HELLO sink tree. Using this optional parameter we are able to create hop depth restricted HELLO sink trees.

## 5.2.6 Incoming and Outgoing Interface Stores

In order to support the utilization of asymmetric links, an SSR node needs to distinguish between *incoming* and *outgoing* interfaces. Therefore it is required to maintain a second instance of the `cInterfaceStore` class.

If a given node receives an `SSR_HELLO_WITH_TREE` message, it will add the sender to its incoming interface store. Moreover, if the current node itself is contained in the first level of the received HELLO sink tree, the sender will be added to the outgoing interface store as well. Consequently, in case of a symmetric link, the corresponding neighbor finally occurs in the incoming interface store as well as in the outgoing interface store.

Contrariwise, if a HELLO sink tree contains a link directed from the current node towards another node whose HELLO messages are not being received by the current node, this *forward neighbor* will be added to the outgoing interface store and it will be tagged with the MAC address *zero*. This specific MAC address denotes that there is an outgoing asymmetric link, which we could utilize in source routing by forwarding a corresponding hop-by-hop message per broadcast.

Actually, utilization of asymmetric links in source routes implies that we have to send messages per broadcast across occurring asymmetric links. In the first place, this is due to the fact that a given node will not ever learn the MAC layer address of a forward neighbor beyond an asymmetric link. Furthermore, if unicast at the underlying MAC layer relied on acknowledged transmission (as for example within the *IEEE-802.11* standard [9]), it would not be possible to use unicast across asymmetric links, anyway.

## Chapter 6

# Simulation Environment and Evaluation

The initial sections of this chapter give a brief overview on the simulation environment and the test arrangements that we used during the evaluation stage of this work. Enhancements and extensions to this simulation environment are subjects of other current works, thus we will concentrate on information that is relevant to the interpretation of our simulation results, which we present and discuss subsequently.

### 6.1 Overview on the Simulation Environment

The simulation environment that we used to evaluate the proposed SSR extension supporting asymmetric links has been implemented in *OMNeT++* [20]. This simulation environment already existed for various evaluation purposes concerning the SSR protocol and thus we only had to apply some minor modifications in order to be able to configurate additional parameters specific to the extension and to record further statistical data during simulations runs.

#### 6.1.1 Simulator Functionality in the `OM_SSR` Class

The `OM_SSR` class, which is derived of the *OMNeT++* `cSimpleModule` class, represents the central *OMNeT++* simulation module containing any simulation related logic, such as initializing a simulation run, event registering and handling or recording and calculating of statistical data during simulation runs.

Basically, we had to slightly modify the simulation initialization step in order to be able to import additional parameters that we need to configurate our SSR extension for support of asymmetric links (see section 6.1.4).

Since the simulator has already been capable of generating statistics of routing performance characteristics like the average accumulated hops declined by payload messages and the average payload message delay, we furthermore had to add the functionality of printing information concerning the period of time the SSR protocol needed to converge, i.e. to establish the *consistency* condition described in

section 3.5, into an additional result file, which is named according to the scheme `ssrConverge-[\dots].txt`.

### 6.1.2 cSsrNode Wrapper Class

The `cOmSsrNode` class is derived of the `cNode` class of the `ssr-core` library (see section 5.1.1). Thus, it virtually implements an SSR node within the simulation environment and routes events and messages between actual SSR protocol and the central *OMNeT++* simulation module, which we described in section 6.1.1 above.

### 6.1.3 Simplified MAC Layer

The primary concern of the evaluation of this work is to investigate whether the algorithm described in section 4.2 could be adapted to scale in medium size wireless networks using the extensions proposed in section 4.3 and to determine the impact of certain parameters, which are specific to these extensions, on the overall routing performance of the SSR protocol.

Therefore and since implementations of realistic wireless MAC layers (like *IEEE 802.11 (WLAN)*, [9], *IEEE-802.15.1 (Bluetooth)*, [1] or *IEEE-802.15.4 (ZigBee)*, [5]) are still in a stage of development respectively have only recently been finished (*IEEE-802.11*), we employed a simplified MAC layer implementation denoted as `easyOmMac` during the evaluation stage of this work, which is based on unidirectional point-to-point connections.

However, an evaluation of the proposed algorithm on the basis of realistic MAC layers will be a subject of our future work.

### 6.1.4 Simulation Scenarios

The scenarios that we used during this evaluation stage were generated on the basis of random  $n \times n$  grids, i.e. network layer addresses are arbitrarily distributed all over the grid. Furthermore, we supplied these grids with varying fractions of nodes providing a larger transmission range. Whereas nodes with a default transmission range do only reach (and thus are being reached by) physical neighbors that are located in the same line or in the same column of the grid, these enhanced transmission range nodes are capable to transcend the diagonals of adjacent grid cells.

Figure 6.1 shows a simple example of a  $3 \times 3$  grid containing one single node with an enhanced transmission range.

Consequently, such a wide transmission range node induces up to four asymmetric links (one asymmetric links for each wide transmission range node located at a corner of the grid and two asymmetric links for each wide transmission range node located at a border of the given grid). However, the fraction of wide transmission range nodes is not equivalent to the actual fraction of asymmetric links within the resulting physical network topology.

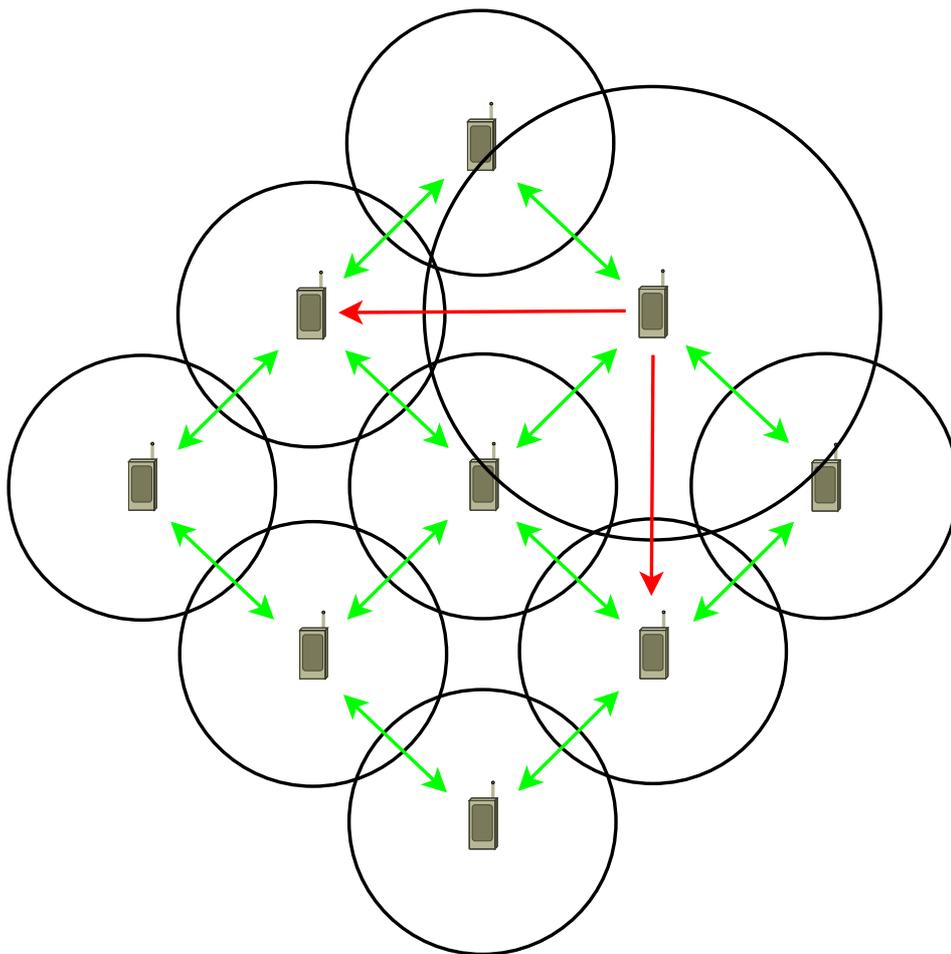


Figure 6.1:  $3 \times 3$  grid containing one wide transmission range node

The number of symmetric links, i.e. the number of regular edges of an  $n \times n$  grid, can be calculated using the formula:

$$\#_{edges} = 4(n - 2)^2 + 4(n - 1) = 4(n^2 + 3n + 3)$$

Assuming for example that there are 20% of wide transmission range nodes within a  $10 \times 10$  grid, i.e. there are up to 20 nodes each inducing up to 4 asymmetric links, the given grid has a total of  $\leq 612$  links whereof  $\leq 80$  are asymmetric links. Hence the resulting fraction of asymmetric links in the given grid could be estimated to  $\leq 13\%$ .

Fraction of high range nodes:	0%	10%	20%	30%
Fraction of asymmetric links:	0%	$\leq 7\%$	$\leq 13\%$	$\leq 18,5\%$

Table 6.1: Estimated fractions of asymmetric links within the used configurations

Within this evaluation stage, we employed scenarios containing fractions of 0%, 10%, 20% and 30% of wide transmission range nodes. Table 6.1 shows the fractions of asymmetric links corresponding to these values.

We wrote a *Python*-script to automatically generate scenarios as described above. This script demands the following parameters:

- Number of nodes
- Distance between adjacent nodes [m]
- Fraction of wide transmission range nodes [%]
- Local sink tree size [# of cache lines]
- HELLO sink tree size [# of cache lines]
- HELLO sink tree mode [SSR\_SINK\_TREE\_MODE]
- HELLO sink tree maximum hop depth [# of hops]
- Enable/disable SSR\_EARLY\_PATH\_OPTIMIZATION [BOOL]
- Seed [numeric] (optional)

On the basis of these parameters, the script generates a `positions-[\dots].ini` file defining the position of each node within the grid topology, a `ranges-[\dots].ini` file defining each node as default or wide transmission range node and a `scenario-[\dots].ini` file providing further parameters configuring the SSR protocol respectively the simulation environment, which are briefly described in the subsequent section 6.1.5. Furthermore, this script appends the source code, which is required to interpret the described `*.ini` files, to the `NetGrid.ned` file of the *OMNeT++* simulation environment.

This `NetGrid.ned` file basically contains source code that is required to build the described scenarios within the *OMNeT++* simulator and thus is written in the NED language of *OMNeT++*.

### 6.1.5 Further Simulation Parameters

This section describes some further parameters occurring within the `scenario-[...] .ini` file that are used to configure the corresponding simulations run. We did not modify these parameters during the evaluation stage of this work and thus have them set to certain default values by our scenario generator script (see section 6.1.4). As these parameters specify our test arrangement, we provide a list as well as a brief description of each parameter below.

Basically, parameters of the form `ssr.*` denote parameters that are specific to the SSR protocol whereas the remaining parameters define properties of the *OMNeT++* simulation environment.

- `ssr.broadcastInterval`
- `ssr.notifyInterval`
- `ssr.cacheSize`
- `ssr.useAsymLink`
- `simTimeLimit`
- `activeTime`
- `activeTimeEnd`
- `packetSize`
- `requestRate`

The parameter `ssr.broadcastInterval` defines the period of time between subsequent HELLO messages in seconds (default value: *2s*). Analogously, `ssr.notifyInterval` denotes the interval between subsequent initiations of the SSR state maintenance algorithm, which is extensively described in section 3.5 (default value: *10s*). `ssr.cacheSize` defines the size of a node's route cache measured in cache lines (default value: *255*). The boolean value `ssr.useAsymLink` is used to enable/disable the extension providing support for asymmetric links within the SSR protocol (default value: *true*).

The first parameter concerning the simulation environment, `simTimeLimit`, denotes the virtual time span of SSR protocol activity that is being simulated within the specified simulation run. The default value is derived of observed convergence time values of particular network sizes: *90s* for  $\#_{nodes} \leq 49$  respectively *120s* for  $64 \leq \#_{nodes} \leq 100$ . The same applies for the subsequent time values `activeTime` and `activeTimeEnd`, which define the points of time each node starts respectively stops generating random requests, i.e. sending messages of type `SSR_PAYLOAD` to arbitrary destinations (default values: *30s / 90s* for  $\#_{nodes} \leq 49$  and *60s / 120s* for  $64 \leq \#_{nodes} \leq 100$ ). Finally, the parameter `requestRate` denotes the number of these random requests each node generates and sends per second during the active time (default value: *1*).

## 6.2 Simulation Results

Within the subsequent sections we present the simulation results that we gathered during the evaluation stage of this work. Predominantly, we illustrate the obtained overall routing performance on the basis of the average path length of `SSR_PAYLOAD` messages measured in hops. Since we have been using a simplified MAC layer during these simulations (see section 6.1.3), the also recorded average message delay values are not equally meaningful.

We furthermore recorded the period of time the SSR protocol required to converge, i.e. to establish the *consistency* condition as described in 3.5. But since for any scenarios as described in section 6.1.4 the corresponding results do - as expected - not significantly vary from simulations result obtained using standard SSR, this data is only crucial to observe in specific scenarios as considered in section 6.2.6.

Foremost, we investigate the inherent impact of the proposed `SSR_EARLY_PATH_OPTIMIZATION` extension on the overall routing performance of SSR by using standard SSR, i.e. support for asymmetric links has been disabled during these simulation runs. We need this evaluation step in order to be able to estimate a potential additional benefit of the `SSR_EARLY_PATH_OPTIMIZATION` extension in scenarios containing asymmetric links.

We simulated each HELLO sink tree strategy as proposed in section 4.3 using scenarios as described in section 6.1.4 with up to 100 nodes and fractions of 10%, 20% and 30% of wide transmission range nodes. Furthermore, we performed each of this simulation series having the `SSR_EARLY_PATH_OPTIMIZATION` extension disabled as well as enabled.

Finally, in section 6.2.6 we investigate the applicability of our proposed algorithm in specific scenarios containing asymmetric links that require exceptionally long loop paths to be resolved.

### 6.2.1 Impact of `SSR_EARLY_PATH_OPTIMIZATION` on SSR Overall Routing Performance in Absence of Asymmetric Links

Clearly, the `SSR_EARLY_PATH_OPTIMIZATION` extension as described in section 4.3 could obtain a positive impact on the overall routing performance of SSR.

Originally, within the SSR routing process, source routes of hop-by-hop messages would only have been modified if necessary, i.e. if the message's source route was at its end or if the subsequent hop had become unavailable in the meantime. If now each intermediate node attempts to improve a hop-by-hop message's source route, it is reasonably likely that the length of the resulting source route will be reduced, but on the cost of enforcing each particular intermediate node to query its route cache and - in case of a successful route optimization - an additional `SSR_ROUTE_UPDATE` message, which has to be sent to the message's mediator respectively issuer.

Indeed, these additional `SSR_ROUTE_UPDATE` messages are optional, but seem quite reasonable in order to propagate source route optimizations back to the most recent mediator node, which consequently could apply the corresponding optimization by itself henceforth.

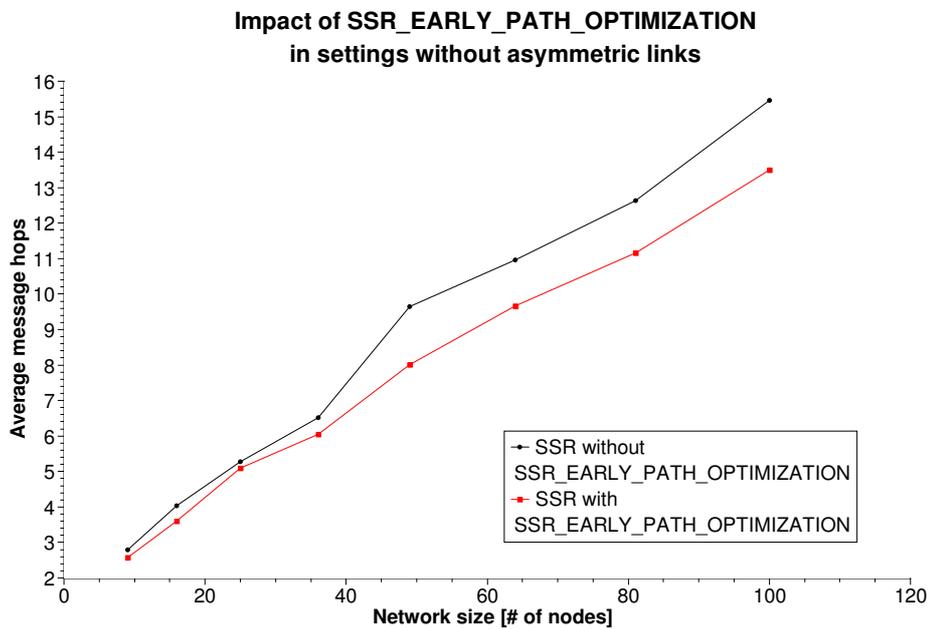


Figure 6.2: Impact of `SSR_EARLY_PATH_OPTIMIZATION` on the average path length per `SSR_PAYLOAD` message

In order to estimate the impact of the `SSR_EARLY_PATH_OPTIMIZATION` extension on the overall routing performance of SSR independent of the utilization of asymmetric links, support for asymmetric links has been disabled within this simulation series.

The graph depicted in figure 6.2 shows that the application of the `SSR_EARLY_PATH_OPTIMIZATION` extension could reduce the number of hops traversed by `SSR_PAYLOAD` messages up to about 15%.

## 6.2.2 Results of SSR\_RANDOM\_TREE Mode

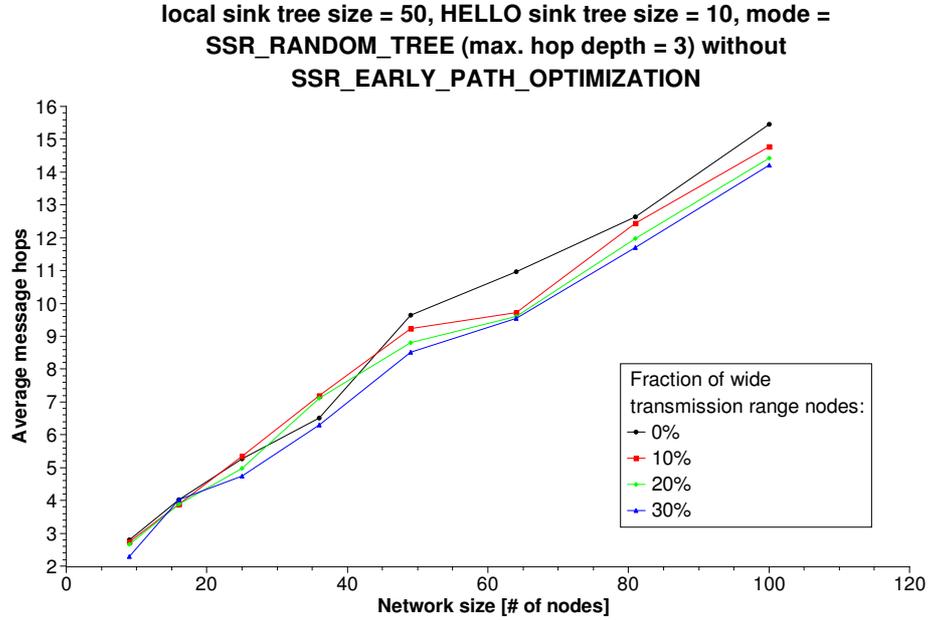


Figure 6.3: SSR overall routing performance using SSR\_RANDOM\_TREE mode to resolve asymmetric links

Using the SSR\_RANDOM\_TREE strategy, the local sink tree information to be copied to a HELLO sink tree is selected at random. Thus, this mode rather tends to fill up the nodes' route caches with source routes to various destinations than to efficiently resolve occurring asymmetric links. Consequently, this could possibly result in a positive impact on SSR's overall routing performance, which we cannot clearly put down to the utilization of asymmetric links.

In order to delimit this bias, we used an additional *maximum hop depth* parameter during this series of simulation runs:

As in scenarios as described in 6.1.4 each asymmetric link could be resolved by a loop path of 3 hops length, we predefined a maximum hop depth of 3 hops for the HELLO sink trees during this series of simulations. The figures 6.3 and 6.4 visualize the SSR overall routing performance obtained in SSR\_RANDOM\_TREE mode depending on the application of the SSR\_EARLY\_PATH\_OPTIMIZATION extension.

Subsequently, we use the results of this non-sophisticated strategy as reference values for the simulation results of the SSR\_UNRESOLVED\_LINKS\_FIRST strategy and the SSR\_DELTA\_TREE strategy.

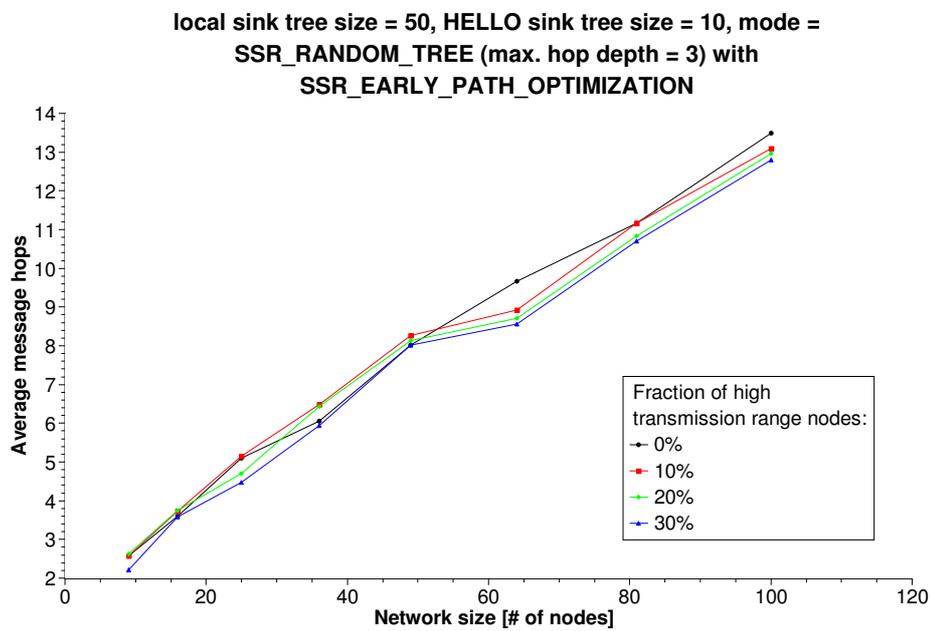


Figure 6.4: SSR overall routing performance using SSR\_RANDOM\_TREE mode with SSR\_EARLY\_PATH\_OPTIMIZATION extension

### 6.2.3 Results of SSR\_UNRESOLVED\_LINKS\_FIRST Mode

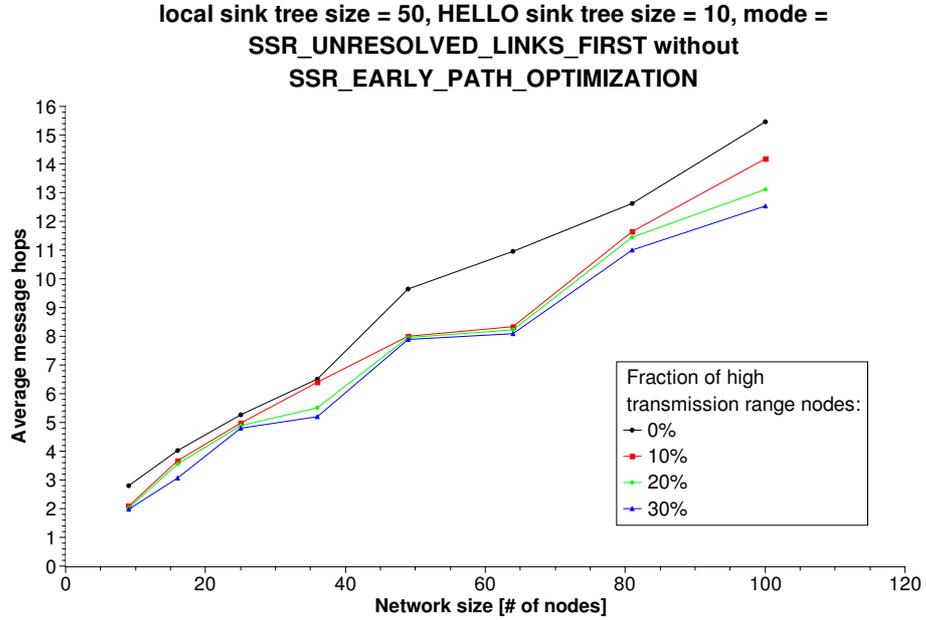


Figure 6.5: Overall routing performance using SSR\_UNRESOLVED\_LINKS\_FIRST mode

The results of this series of simulations show the performance of the SSR\_UNRESOLVED\_LINKS\_FIRST mode. This strategy is the most focussed on efficiently resolving asymmetric links and thus should least of all strategies affect the overall routing performance of SSR by distributing source routes to various destinations across the network.

Compared to the simulation results presented in the previous section, which we obtained while applying the SSR\_RANDOM\_TREE strategy, the relative benefit at the overall routing performance of SSR dependent on the fraction of wide transmission range nodes is significantly increased.

Observing the graphs depicted in figure 6.5 and 6.6, it is furthermore remarkable that the relative benefit of utilizing the occurring asymmetric links within SSR source routing is not considerably increased when using the SSR\_EARLY\_PATH\_OPTIMIZATION extension as proposed in section 4.3.

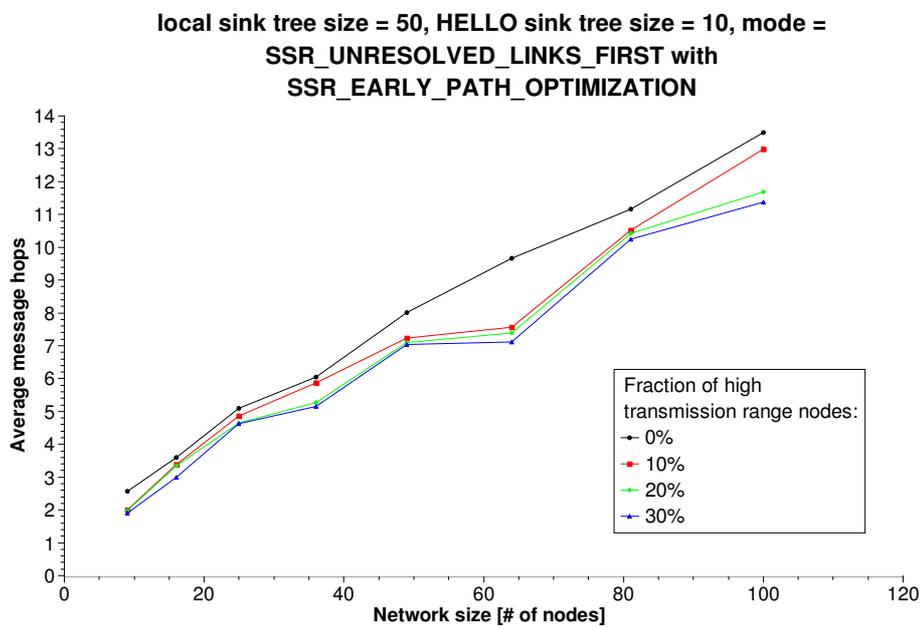


Figure 6.6: Overall routing performance using SSR\_UNRESOLVED\_LINKS\_FIRST mode with SSR\_EARLY\_PATH\_OPTIMIZATION extension

### 6.2.4 Results of SSR\_DELTA\_TREE Mode

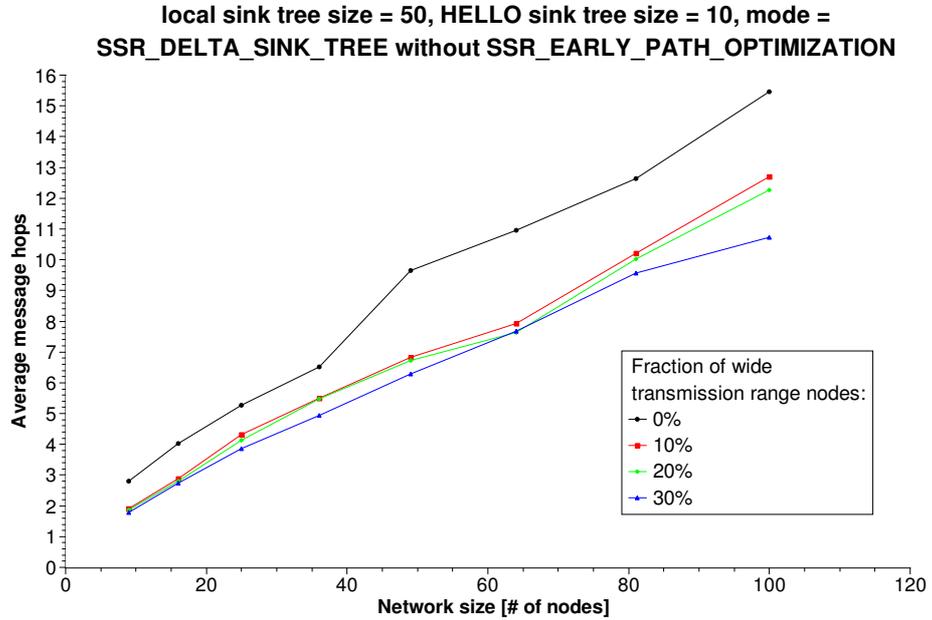


Figure 6.7: SSR overall routing performance using SSR\_DELTA\_TREE mode to resolve asymmetric links

This section presents the simulation results that we obtained with SSR while constructing HELLO sink trees with the SSR\_DELTA\_TREE strategy. Since physical neighbors exchange their entire local sink trees within several SSR\_HELLO\_WITH\_TREE messages, there certainly is a considerable impact on the overall routing performance beyond the utilization of asymmetric links. Based on the distribution of extensive sink tree information, the algorithm as proposed in section 4.2.3 is able to extract many additional source routes to various destinations within the given network topology.

The graphs depicted in figure 6.7 and figure 6.8 illustrate the simulation results, which we obtained using the SSR\_DELTA\_TREE strategy with as well as without the SSR\_EARLY\_PATH\_OPTIMIZATION extension.

As the overall routing performance of SSR is again increased compared to the simulation results of the SSR\_UNRESOLVED\_LINKS\_FIRST mode presented in the previous section, the supposed impact of distributing extensive sink tree information seems to be confirmed.

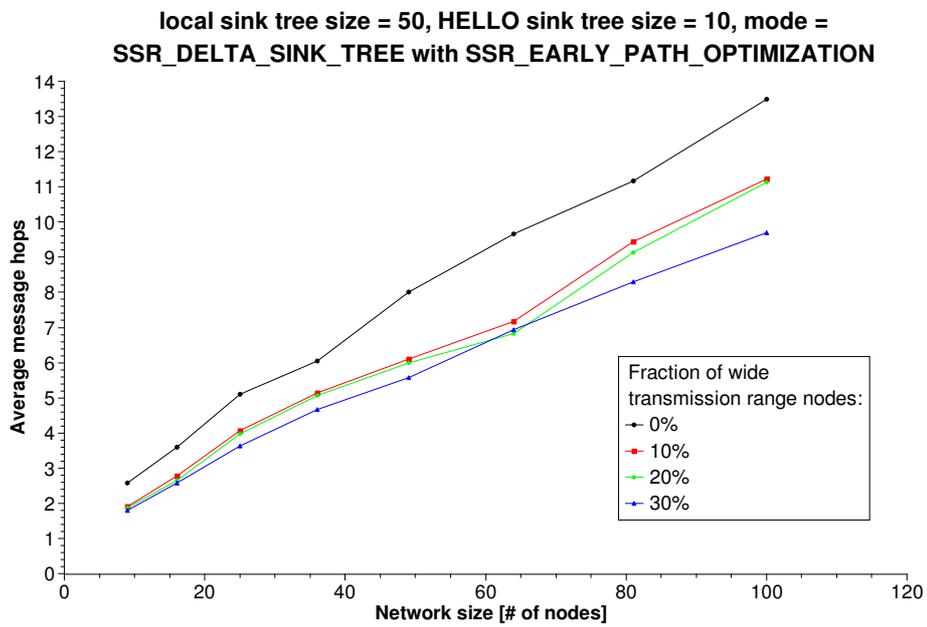


Figure 6.8: SSR overall routing performance using SSR\_DELTA\_TREE mode with SSR\_EARLY\_PATH\_OPTIMIZATION extension

### 6.2.5 Comparison of the Proposed HELLO Sink Tree Strategies

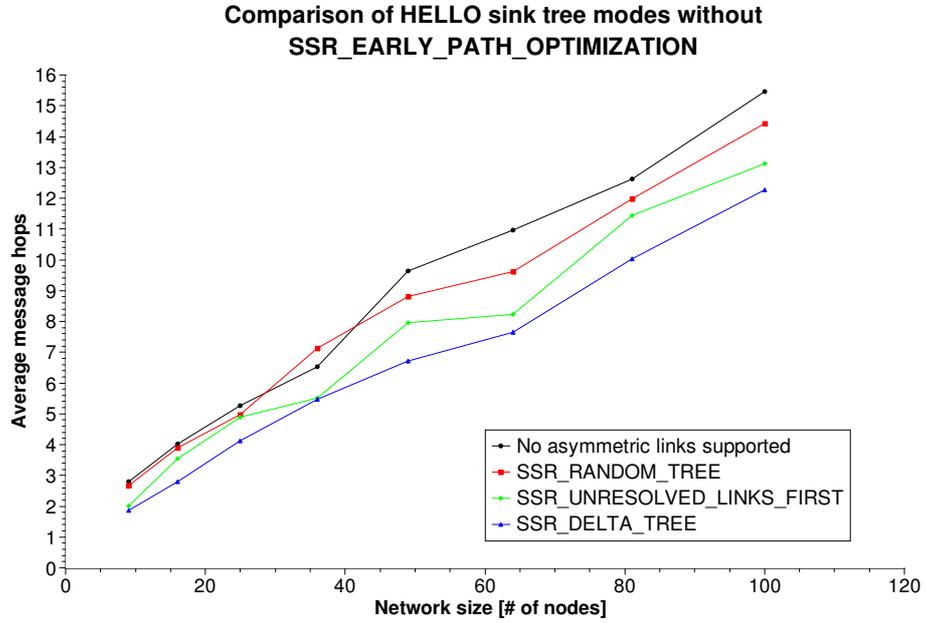


Figure 6.9: Observed benefit of the particular HELLO sink tree strategies

In order to be able to compare the overall routing performance obtained by the proposed HELLO sink tree strategies, all simulation results of scenarios containing a fraction of 20% of wide transmission range nodes are accumulated and depicted within the graphs 6.9 and 6.10. As usual, the latter illustrates the corresponding simulation results when additionally employing the `SSR_EARLY_PATH_OPTIMIZATION` extension.

The remarkable fact that the routing performance obtained when using the `SSR_DELTA_TREE` strategy exceeds the routing performance obtained with the `SSR_UNRESOLVED_LINKS_FIRST` strategy is not completely unexpected. As discussed in the previous section as well as in section 4.3, a positive impact on the overall routing performance beyond the utilization of asymmetric links could reasonably be assumed.

Although the obtained overall routing performance while employing the `SSR_DELTA_TREE` strategy exceeds the obtained overall routing performance while employing the `SSR_UNRESOLVED_LINKS_FIRST` strategy, the latter strategy is rather recommended to be applied within the SSR protocol, as we will point out in the subsequent section.

Table 6.2 summarizes the observed benefits and drawbacks of the previously compared HELLO sink tree strategies.

HELLO sink tree modes	Benefits	Drawbacks
SSR_RANDOM_TREE		Poor efficiency at resolving asymmetric links
SSR_UNRESOLVED_LINKS_FIRST	Good efficiency at resolving asymmetric links	
SSR_DELTA_TREE	Most significant benefit at overall routing performance	Not as efficient as SSR_UNRESOLVED_LINKS_FIRST mode at resolving asymmetric links

Table 6.2: Benefits and drawbacks of each particular HELLO sink tree strategy

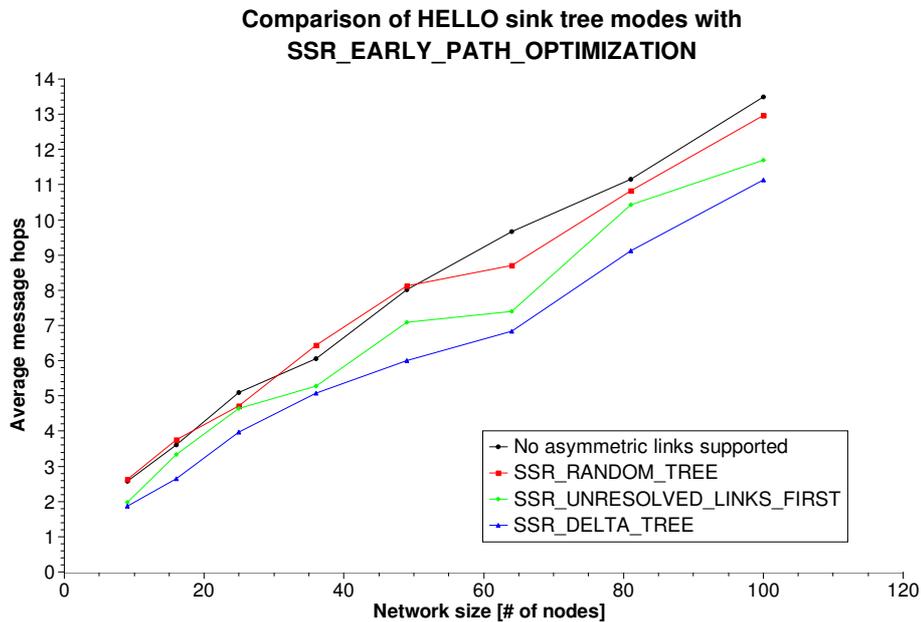


Figure 6.10: Observed benefit of the particular HELLO sink tree strategies using SSR\_EARLY\_PATH\_OPTIMIZATION extension

### 6.2.6 Further Simulation Results

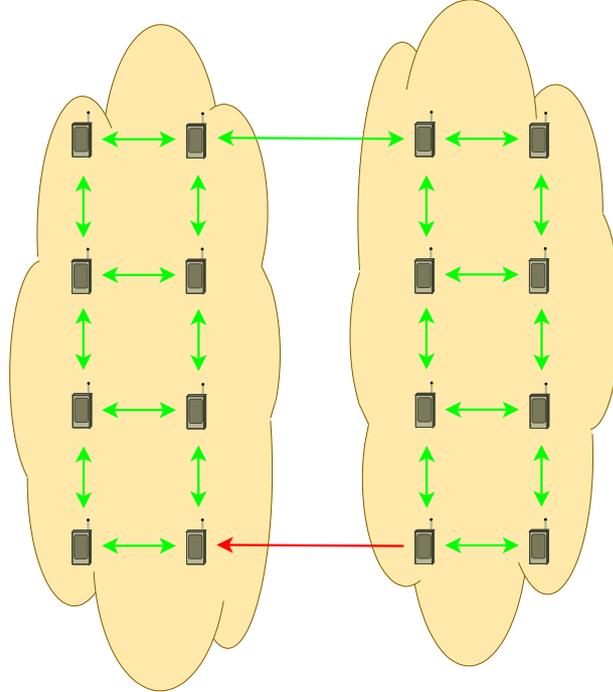


Figure 6.11: Scenario scheme, which is referred to as *bridge setting, type 1*

In addition to the simulations scenarios as described in section 6.1.4, we built several further scenarios, which are kind of worst case scenarios to the SSR protocol. Basically, these additional scenarios follow two explicit schemes, which we outlined in the figures 6.11 and 6.12.

Both scenario schemes provide a network topology, which is almost divided into two partitions. In the first scheme, which is referred to as *bridge setting, type 1*, these partitions are connected through a single symmetric and a single asymmetric link. As we placed these links at opposite borders of the given topology, the loop path resolving the asymmetric link is exceptionally long.

However, in theory even standard SSR without support for asymmetric links should converge (and thus provide reliable source routing) within a tolerable period of time as at least one symmetric connection between the two partitions exists. In practice though, standard SSR did not converge in a *type 1* bridge setting containing 36 nodes within a time span of 300s.

Within the second scheme, which is denoted as *bridge setting, type 2*, we replaced the remaining symmetric connection between the two network partitions by a second asymmetric link, which is directed contrariwise to the first asymmetric links that is already contained in the described *type 1* bridge setting scheme.

Obviously, it is impossible for standard SSR to converge at all since virtual neighbors that are located in different partitions of the network have no chance to ever discover each other.

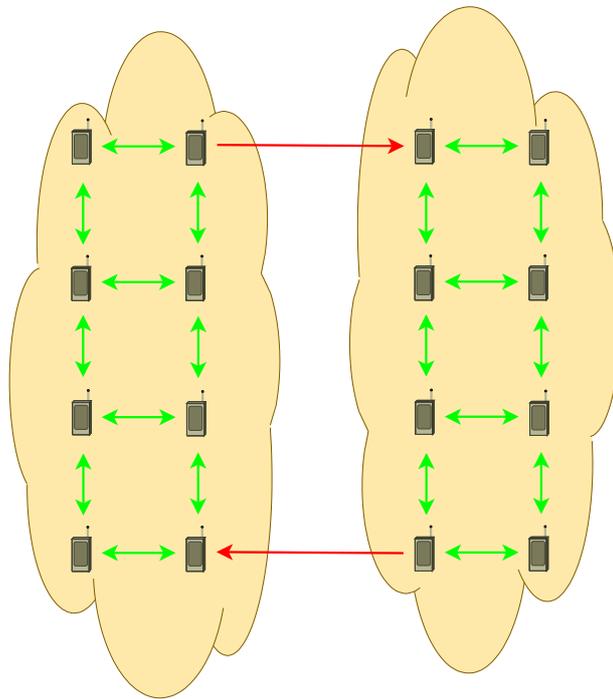


Figure 6.12: Scenario scheme, which is referred to as *bridge setting, type 2*

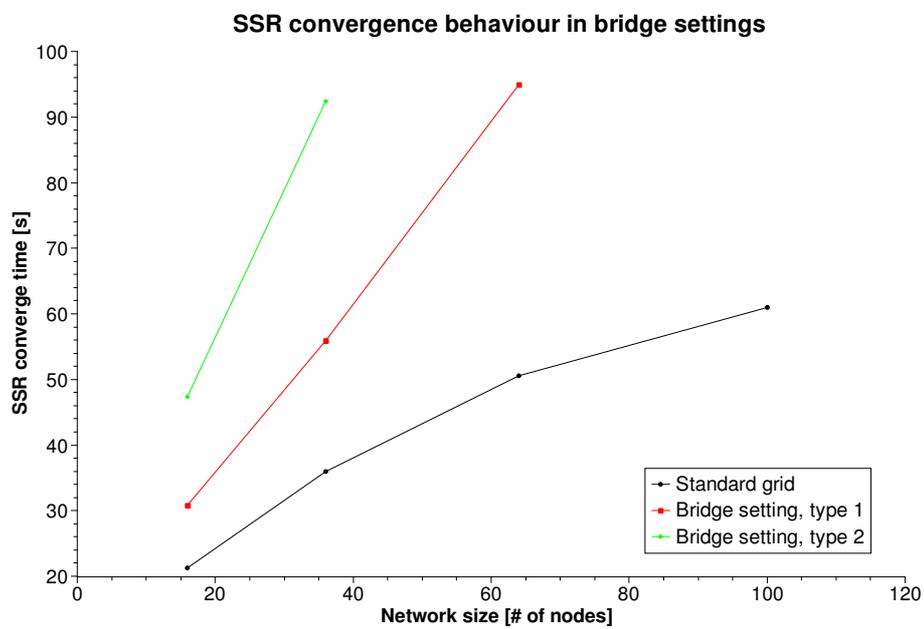


Figure 6.13: Applicability of SSR extension for asymmetric link extension support in bridge scenarios

The graph depicted in figure 6.13 shows the simulation results concerning the convergence behaviour that we obtained with SSR while employing the proposed `SSR_UNRESOLVED_LINKS_FIRST` strategy to construct HELLO sink trees. Missing results indicate that the SSR protocol did not converge within a time span of 300s. Figure 6.14 depicts the associated *average message hops* graph.

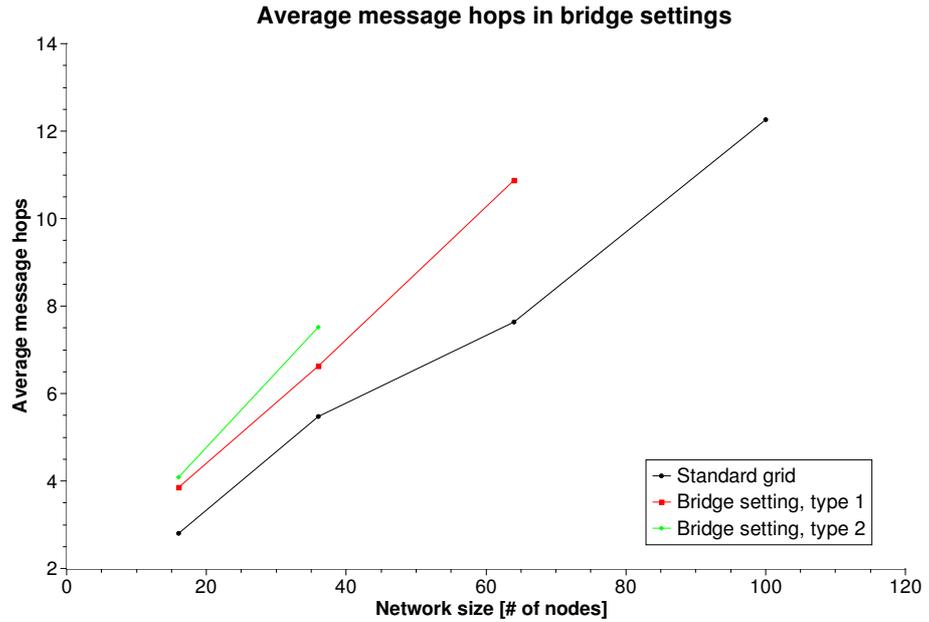


Figure 6.14: Associated average message hops graph of 6.13

Applying the `SSR_DELTA_TREE` or `SSR_RANDOM_TREE` strategy, the obtained simulation results were significantly worse, i.e. the SSR protocol did not converge within a time span of 300s in *type 1* or *type 2* bridge scenarios containing 36 nodes.

### 6.2.7 Expanded Simulation Results Obtained with SSR Applying the `SSR_UNRESOLVED_LINKS_FIRST` Strategy

As illustrated in the previous sections, the `SSR_UNRESOLVED_LINKS_FIRST` strategy seems to be the most recommendable strategy to be employed in order to support the utilization of asymmetric links within the SSR protocol. Thus, we expanded the corresponding simulation series to 5 seeds per particular setting, i.e. 5 random scenarios of each network size containing a fraction of 20% of wide transmission range nodes have been simulated. The graphs depicted in figure 6.15 and figure 6.16 show the obtained simulation results depending on the application of the `SSR_EARLY_PATH_OPTIMIZATION` extension.

These simulation results basically confirm the simulation results introduced in section 6.2.3. Furthermore, this expanded study shows the impact of the actual physical topology on the overall routing performance of SSR.

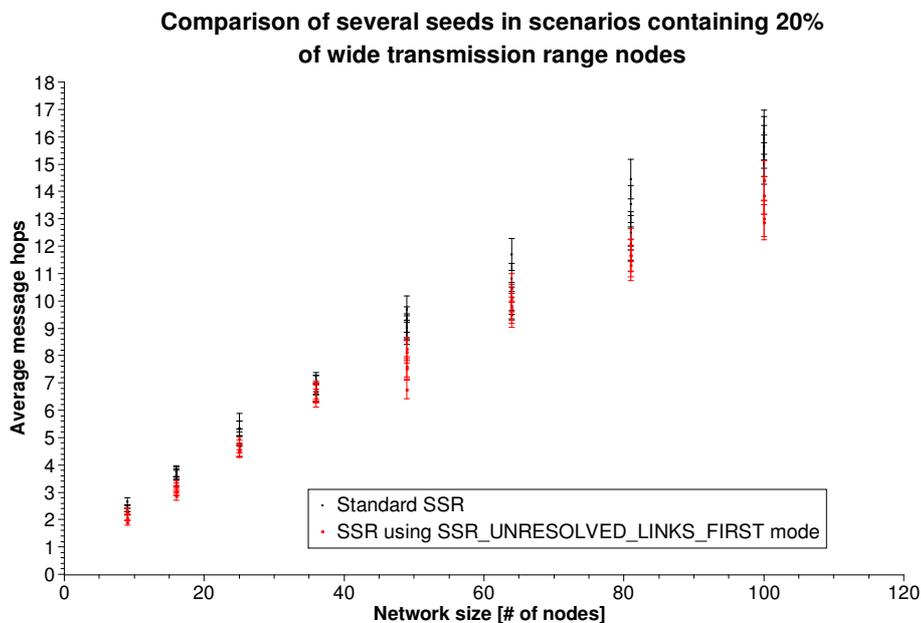


Figure 6.15: Expanded simulation results using `SSR_UNRESOLVED_LINKS_FIRST` strategy

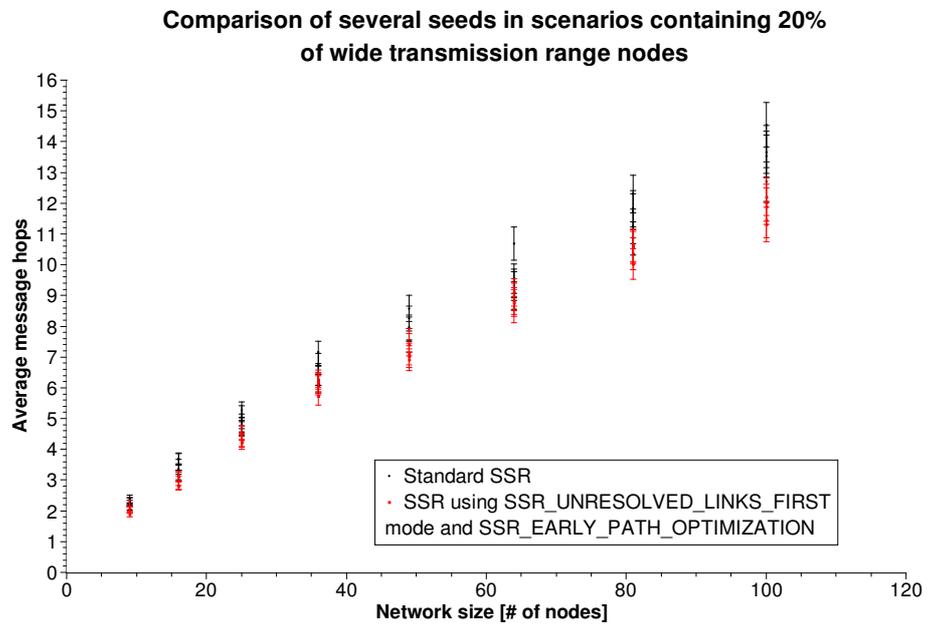


Figure 6.16: Expanded simulation results using `SSR_UNRESOLVED_LINKS_FIRST` strategy and `SSR_EARLY_PATH_OPTIMIZATION` extension

### 6.3 Summary of the Simulation Results

Summarizing the simulation results introduced and visualized within the sections 6.2.2 to 6.2.4, we showed that the proposed algorithm based on exchanging partial sink tree information between physical neighbors is capable of locally resolving relatively close asymmetric links at an acceptable cost, i.e. physical neighbors exchange HELLO messages containing a total of 10 entries, which are selected from a node's local sink tree according to one of the proposed HELLO sink tree strategies.

As the simulation results of each particular HELLO sink tree strategy indicate, the application of the `SSR_EARLY_PATH_OPTIMIZATION` extension proposed in section 4.3 does not obtain a significant additional benefit in scenarios containing asymmetric links compared to scenarios not containing any asymmetric links, i.e. the extension seems not to achieve a more extensive utilization of shortcuts induced by occurring asymmetric links. However, as the overall routing performance actually is significantly better (up to about 20% in certain scenarios) when utilizing occurring asymmetric links, we could argue that asymmetric links are already used quite frequently without employing the `SSR_EARLY_PATH_OPTIMIZATION` extension.

Thus it is arguable whether to use this extension within the SSR protocol at all, since - as previously explained in section 4.3 - it produces additional control overhead.

The simulation results as presented in section 6.2.6 show that the proposed strategies on utilizing the delimited space provided in HELLO messages do not provide a convenient performance at resolving asymmetric links when exceptionally long loop paths are required. We could alleviate this issue somewhat by increasing the space provided for HELLO sink trees, but that does not seem to be a practicable solution with regard to the MTUs of some MAC layers, which are commonly used in wireless sensor networks. Nevertheless, using our extension for support of asymmetric links, SSR actually worked in some special scenarios where standard SSR did not converge.

One of the most promising approaches on establishing reliable routing within these kinds of scenarios seems to be the *Dynamic Source Routing* protocol proposed by *Johnson et al.* [10], which is briefly described in section 2.2.3 - but on the considerable cost of flooding the network twice.

Maybe the basic idea of this approach could be borrowed to SSR in order to be applied for asymmetric links that could not be resolved within a considerably long period of time.



## Chapter 7

# Conclusion

Within this diploma thesis, we proposed a scalable approach on resolving asymmetric links in ad hoc network settings, which is based on the idea of exchanging partial sink trees between neighboring nodes within a regular HELLO protocol. We introduced a prioritization of local sink tree information that aims at resolving asymmetric links in ascending order of their distance to a given node. This prioritization strategy allows us to deal with relatively small HELLO message sizes as required in practice due to the small MTUs of MAC layers that are commonly used in wireless sensor networks.

Our algorithm has been integrated into the `Scalable Source Routing` protocol and evaluated in a simulation environment based on the network simulator `OMNeT++`. We thereby obtained a benefit of up to about 20% at the overall routing performance of SSR in scenarios containing asymmetric links.

Furthermore, we were to some extent successful at employing our extended SSR protocol in certain scenarios where standard SSR did not converge at all.

While applying the proposed `SSR_EARLY_PATH_OPTIMIZATION` extension, we could enhance the overall routing performance of SSR up to about 15% even in scenarios without asymmetric links.

### 7.1 Final Remarks

Having empirically validated the algorithmically correct operability of the proposed approach, our efforts have to be expanded towards more realistic studies, i.e. we need to generate scenarios, which are closer to real world settings of mobile ad hoc networks and wireless sensor networks and we further have to verify our results and observations using realistic abstractions of common MAC layers during future simulations. The latter implicitly denotes that we have to accurately adapt the parameterization of our simulation environment based on restraints, which have to be derived of specific real world settings and MAC layers.

Furthermore, it seems to be quite promising to combine the proposed approach on utilizing asymmetric links in `Scalable Source Routing` with the *Virtual Route Compression* technique developed by *André Kaustell* [11] (a short summary is given in

section 2.3) in order to improve the fraction of actual payload data that could be carried in `SSR_PAYLOAD` messages.

Finally, as proposed in section 6.3, it could be reasonable to employ a DSR-like flooding technique in order to resolve asymmetric links whose corresponding loop path is exceptionally long and thus would not be discovered by our algorithm within an acceptable period of time.

# Bibliography

- [1] Ieee std 802.15.1 - 2005 ieee standard for information technology - telecommunications and information exchange between systems - local and metropolitan area networks - specific requirements. - part 15.1: Wireless medium access control (mac) and physical layer (phy) specifications for wireless personal area networks (wpans). *IEEE Std 802.15.1-2005 (Revision of IEEE Std 802.15.1-2002)*, pages 001–580, 2005.
- [2] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in p2p systems. *Commun. ACM*, 46(2):43–48, 2003.
- [3] J. A. Cobb. Forward-only uni-directional routing. In *Proceedings of International Conference on Computer Communications and Networks '02*, pages 370–375, University of Texas at Dallas, 2002.
- [4] D. De Couto, D. Aguayo, B. Chamber, and R. Morris. Performance of multihop wireless networks: Shortest path is not enough. In *Workshop on Hot Topics in Networkin (HotNets)*, Princeton, NJ, 2002.
- [5] L. De Nardis and M.-G. Di Benedetto. Overview of the ieee 802.15.4/4a standards for low data rate wireless personal data networks. pages 285–289, March 2007.
- [6] Thomas Fuhrmann. Scalable routing for networked sensors and actuators. In *Proc. 2nd Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, September 2005.
- [7] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. An empirical study of epidemic algorithms in large scale multihop wireless networks. Intel research, tech. rep., 2002.
- [8] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of dht routing geometry on resilience and proximity. In *Proceedings of the SIGCOMM 2003 conference*, pages 381–394, ACM Press, 2003.
- [9] IEEE 802.11 Working Group. IEEE Std. 802.11-1999: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications., 1999.
- [10] D. B. Johnson, D. A. Maltz, and J. Broch. Dsr: The dynamic source routing protocol for multi-hop wireless ad hoc networks. In *Ad Hoc Networking*, pages 139–172, 2001.
- [11] A. Kaustell. Scalable routing in mesh networks for devices with limited resources: Scalable source routing and optimization proposals, 2009.

- [12] F. Knittel. Scalable source routing in the ambicomp environment. Study Thesis, 2009.
- [13] M. K. Marina and S. R. Das. Routing performance in the presence of unidirectional links in multihop wireless networks. In *MOBIHOC '02: Proceedings of the 3rd ACM International Symposium on Mobile Ad Hoc Networking & Computing*, pages 12–23, Lausanne, Switzerland, June 9–11 2002.
- [14] S. Nesargi and R. Prakash. A tunneling approach to routing with unidirectional links in mobile ad-hoc networks. In *Proceedings of International Conference on Computer Communications and Networks '00*, pages 522–527, University of Texas at Dallas, 2000.
- [15] Charles E. Perkins and Elizabeth M. Royer. Ad hoc On-Demand Distance Vector Routing. In *Proc. 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, New Orleans, LA, USA, February 1999.
- [16] R. Prakash. A routing algorithm for wireless ad hoc networks with unidirectional links. In *Wireless Networks 7*, pages 617–625, University of Texas at Dallas, 2001.
- [17] V. Ramasubramanian and D. Mossé. Bra: A bidirectional routing abstraction for asymmetric mobile ad hoc networks. In *IEEE/ACM Transactions on Networking 16*, pages 116–129, 2008.
- [18] P. Sinha, S. V. Krishnamurthy, and S. Dao. Scalable unidirectional routing with zone routing protocol (zrp) extensions for mobile ad-hoc networks. In *Proceedings of Wireless Communications and Networking Conference (WCNC)*, pages 1329–1339, 2000.
- [19] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the SIGCOMM 2001 conference*, pages 149–160, ACM Press, 2001.
- [20] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Simutools '08: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, pages 1–10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).
- [21] G. Wang, Y. Ji, D. C. Marinescu, and D. Turgut. A routing protocol for power constrained networks with asymmetric links. In *PE-WASUN '04: Proceedings of the 1st ACM international workshop on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks*, pages 69–76, Venezia, Italy, October 7 2004.
- [22] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *Proc. Conf. Embedded Networked Sensor Systems (SenSys)*, Los Angeles, CA, 2003.