# A Process Algebra for Wireless Mesh Networks

**used for**

# Modelling, Verifying and Analysing AODV


Ansgar Fehnker
NICTA*
Sydney, Australia
Computer Science and Engineering
University of New South Wales
Sydney, Australia

Rob van Glabbeek
NICTA*
Sydney, Australia
Computer Science and Engineering
University of New South Wales
Sydney, Australia

Peter Höfner
NICTA*
Sydney, Australia
Computer Science and Engineering
University of New South Wales
Sydney, Australia

Annabelle McIver
Department of Computing
Macquarie University
Sydney, Australia
NICTA*
Sydney, Australia

Marius Portmann
NICTA*
Brisbane, Australia
Information Technology and
Electrical Engineering
University of Queensland
Brisbane, Australia

Wee Lum Tan
NICTA*
Brisbane, Australia
Information Technology and
Electrical Engineering
University of Queensland
Brisbane, Australia



Route finding and maintenance are critical for the performance of networked systems, particularly when mobility can lead to highly dynamic and unpredictable environments; such operating contexts are typical in wireless mesh networks. Hence correctness and good performance are strong requirements of routing protocols.

In this paper we propose AWN (Algebra for Wireless Networks), a process algebra tailored to the modelling of Mobile Ad hoc Network (MANET) and Wireless Mesh Network (WMN) protocols. It combines novel treatments of local broadcast, conditional unicast and data structures.

In this framework, we present a rigorous analysis of the Ad hoc On-Demand Distance Vector (AODV) protocol, a popular routing protocol designed for MANETs and WMNs, and one of the four protocols currently defined as an RFC (request for comments) by the IETF MANET working group.

We give a complete and unambiguous specification of this protocol, thereby formalising the RFC of AODV, the de facto standard specification, given in English prose. In doing so, we had to make non-evident assumptions to resolve ambiguities occurring in that specification. Our formalisation models the exact details of the core functionality of AODV, such as route maintenance and error handling, and only omits timing aspects.

The process algebra allows us to formalise and (dis)prove crucial properties of mesh network routing protocols such as loop freedom and packet delivery. We are the first to provide a detailed proof of loop freedom of AODV. In contrast to evaluations using simulation or other formal methods such as model checking, our proof is generic and holds for any possible network scenario in terms of network topology, node mobility, traffic pattern, etc. Due to ambiguities and contradictions the RFC specification allows several readings. For this reason, we analyse multiple interpretations. In fact we show for more than 5000 interpretations whether they are loop free or not. Thereby we demonstrate how the reasoning and proofs can relatively easily be adapted to protocol variants.

Using our formal and unambiguous specification, we find some shortcomings of AODV that can easily affect performance. Examples are non-optimal routes established by AODV and the fact that some routes are not found at all. These problems are analysed and improvements are suggested. As the improvements are formalised in the same process algebra, carrying over the proofs is again relatively easy.



*NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.



**Technical Report 5513**, NICTA, 2013
http://www.nicta.com.au/pub?id=5513

# Contents

# 1 Introduction

Wireless Mesh Networks (WMNs) have gained considerable popularity and are increasingly deployed in a wide range of application scenarios, including emergency response communication, intelligent transportation systems, mining, video surveillance, etc. They are self-organising wireless multi-hop networks that can provide broadband communication without relying on a wired backhaul infrastructure, a benefit for rapid and low-cost network deployment. WMNs can be considered a superset of Mobile Ad hoc Networks (MANETs), where a network consists exclusively of mobile end user devices such as laptops or smartphones. In contrast to MANETs, WMNs typically also contain stationary infrastructure devices called mesh routers.

An important characteristic of WMNs is that they operate in unpredictable environments with highly dynamic network topologies—due to node mobility and the variable nature of wireless links. Because of this, route finding and maintenance are critical for the performance of WMNs. Usually, a routing protocol is used to establish and maintain network connectivity through paths between source and destination node pairs. As a consequence, the routing protocol is one of the key factors determining the performance and reliability of WMNs. One of the most popular routing protocols that is widely used in WMNs is the Ad hoc On-Demand Distance Vector (AODV) routing protocol [79]. It is one of the four protocols currently standardised by the IETF MANET working group, and it also forms the basis of new WMN routing protocols, including HWMP in the IEEE 802.11s wireless mesh network standard [53]. The details of the AODV protocol are laid out in the request-for-comments-document (RFC 3561 [79]), a de facto standard. However, due to the use of English prose, this specification contains ambiguities and contradictions. This can lead to significantly different implementations of the AODV routing protocol, depending on the developer's understanding and reading of the AODV RFC. In the worst case scenario, an AODV implementation may contain serious flaws, such as routing loops.

Traditional approaches to the analysis of AODV and many other AODV-based protocols [81, 53, 89, 99, 83] are simulation and test-bed experiments. While these are important and valid methods for protocol evaluation, in particular for quantitative performance evaluation, they have limitations in regards to the evaluation of basic protocol correctness properties. Experimental evaluation is resource intensive and time-consuming, and, even after a very long time of evaluation, only a finite set of network scenarios can be considered—no general guarantee can be given about correct protocol behaviour for a wide range of unpredictable deployment scenarios [5]. This problem is illustrated by recent discoveries of limitations in AODV-like protocols that have been under intense scrutiny over many years [72].

We believe that formal methods can help in this regard; they complement simulation and test-bed experiments as methods for protocol evaluation and verification, and provide stronger and more general assurances about protocol properties and behaviour. The overall goal is to reduce the "time-to-market" for better (new or modified) WMN protocols, and to increase the reliability and performance of the corresponding networks.

The *first contribution* of this paper is AWN (Algebra of Wireless Networks), a process algebra that provides a step towards this goal. It combines novel treatments of data structures, conditional unicast and local broadcast, and allows formalisation of all important aspects of a routing protocol. All these features are necessary to model "real life" WMNs. Data structures are used to store and maintain information such as routing tables. The conditional unicast construct allows us to model that a node in a network sends a message to a particular neighbour, and if this fails—for example because the receiver has moved out of transmission range—error handling is initiated. Finally, the local broadcast primitive, which allows a node to send messages to all its immediate neighbours, models the wireless broadcast mechanism implemented by the physical and data link layer of wireless standards relevant for WMNs. The formalisation assumes that any broadcast message *is* received by all nodes within transmission range.[1] This abstraction enables us to interpret a failure of route discovery (see our eighth contribution below) as an imperfection in the protocol, rather than as a result of a chosen formalism not ensuring guaranteed receipt.

As a *second contribution*, we give a complete and accurate formal specification of the core functionality of the AODV routing protocol using AWN. Our model covers all core components of AODV, but none of the optional features, and abstracts from timing issues. The algebra provides the right level of abstraction to model key features such as unicast and broadcast, while abstracting from implementation-related details. As its semantics is completely unambiguous, specifying a protocol in such a framework enforces total precision and the removal of any ambiguity.

The *third contribution* is to demonstrate how AWN can be used to support reasoning about protocol behaviour and to provide rigorous proofs of key protocol properties, using the examples of route correctness and loop freedom. In contrast to what can be achieved by model checking or test-bed experiments, our proofs apply to all conceivable dynamic network topologies. Route correctness is a minimal sanity requirement for a routing protocol; it is the property that the routing table entries stored at a node are entirely based on information on routes to other nodes that either is currently valid or was valid at some point in the past. Loop freedom is a critical property for any routing protocol, but it is particularly relevant and challenging for WMNs. Descriptions as in [32] capture the common understanding of loop freedom: "A routing-table loop is a path specified in the nodes' routing tables at a particular point in time that visits the same node more than once before reaching the intended destination." Packets caught in a routing loop, until they are discarded by the IP Time-To-Live (TTL) mechanism, can quickly saturate the links and have a detrimental impact on network performance. It is therefore critical to ensure that protocols prevent routing loops. We show that loop freedom can be guaranteed only if sequence numbers are used in a careful way, considering further rules and assumptions on the behaviour of the protocol. The problem is, as shown in the case of AODV, that these additional rules and assumptions are not explicitly stated in the RFC, and that the RFC has significant ambiguities in regards to this. To the best of our knowledge we are the first to give a complete and detailed proof of loop freedom.[2] This is our *fourth contribution*.

As a *fifth contribution*, we show details of several ambiguities and contradictions found in the AODV RFC, and discuss which interpretations (plausible and consistent readings of the RFC) will lead to routing loops, and which are loop free. In fact we analyse more than 5000 interpretations. Hereby we demonstrate how our reasoning and proofs can relatively easily be adapted to protocol variants. In particular, our *sixth contribution*, we demonstrate that routing loops can be created—while fully complying with the RFC, and making reasonable assumptions when the RFC allows different interpretations. As our next contribution, we also analyse five key implementations of the AODV protocol and show that three of them can produce routing loops.

As an *eighth contribution*, we apply linear-time temporal logic (LTL) to formulate temporal properties of routing protocols, such as *route discovery*: "if a route discovery process is initiated in a state where the source node is connected to the destination and during this process no (relevant) link breaks,

[1] In reality, communication is only half-duplex: a single-interface network node cannot receive messages while sending and hence messages can be lost. However, the CSMA protocol used at the link layer—not modelled by AWN—keeps the probability of packet loss due to two nodes (within range) sending at the same time rather low. Since we are examining imperfect protocols, we first of all want to establish how they behave under optimal conditions. For this reason we abstract from probabilistic reasoning by assuming no message loss at all, rather than working with a lossy broadcast formalism that offers no guarantees that any message will ever arrive.

[2] Loop freedom of AODV has been "proven" at least twice [82, 106], but the proof in [82] is not correct, and the one in [106] is based on a simple subset of AODV only, not including the "intermediate route reply" feature—a most likely source of loops.

then the source will eventually discover a route to the destination" and *packet delivery*, saying that under certain circumstances a packet will surely be delivered to its destination. We moreover show that AODV does not satisfy these properties.

In order for the last result to be meaningful, we first develop a general method to augment a protocol specification with a *fairness component* that requires that certain fairness properties are met, and apply this method to our specification of AODV. We also adapt the semantics of LTL in order to make a protocol specification satisfy natural progress and justness properties. Without ensuring these properties, temporal properties like route discovery and packet delivery would trivially fail to hold. The same would apply if we had not assumed guaranteed receipt of broadcast messages by nodes within transmission range (cf. Footnote 1).

Last but not least, we discuss several limitations of the AODV protocol and propose solutions to them. We show how our formal specification can be used to analyse the proposed modifications and show that the resulting AODV variants are loop free.

The rigorous protocol analysis discussed in this paper has the potential to save a significant amount of time in the development and evaluation of new network protocols, can provide increased levels of assurance of protocol correctness, and complements simulation and other experimental protocol evaluation approaches.

This paper is organised as follows: Section 2 gives an informal introduction to AODV. Section 3 describes which features of the AODV protocol are modelled in this paper, and which are not. In Section 4 we introduce the process algebra AWN.[3] Section 6 provides a detailed formal specification of AODV in AWN.[4] To achieve this, we present the basic data structure needed in Section 5. In Section 7 we formally prove some properties of AODV that can be expressed as invariants, in particular loop freedom and route correctness.[6]

In Section 8 we discuss and formalise many ambiguities, contradictions and cases of unspecified behaviour in the RFC, and present an inventory of their plausible resolutions. Combining the resolutions of the various ambiguities leads to 5184 possible interpretations of the RFC. We show which of these interpretations lead to routing loops or other unacceptable behaviour. For the remaining interpretations we show loop freedom and route correctness, through small adaptations in the proofs given in Section 7. We also analyse five implementations of AODV.[7]

In Section 9 we propose a general framework to ensure progress, fairness and justness properties, and apply the proposal to augment our AODV specification with a fairness component. Subsequently, we formulate two temporal properties (route discovery and packet delivery) that AODV-like protocols should satisfy, and demonstrate that AODV does not enjoy these properties. Section 10 discusses several shortcomings of AODV and proposes five ways in which the protocol can be improved. All improvements are formalised in AWN, and we show that they enjoy loop freedom and route correctness.[8] Section 11 describes related work, and in Section 12 we summarise our findings and point at work that is yet to be done.

[3]Major parts of this section have been published in "A Process Algebra for Wireless Mesh Networks" [26].[5]

[4]Parts of the specification are published in [26], in "Automated Analysis of AODV using UPPAAL" [25] and in "A Rigorous Analysis of AODV and its Variants" [48].[5]

[5]The references in [26, 48] to Prop 7.10(b), Sect. 8 and Sect. 9.1 of this paper, are now to Prop. 7.14(b), Sect. 9 and Sect. 8.1.

[6]A sketch of the loop freedom proof is given in [26] and in [48].

[7]A summary of this section appeared in "Sequence Numbers Do Not Guarantee Loop Freedom—AODV Can Yield Routing Loops" [39].

[8]Two of the improvements from this section are presented in [48].

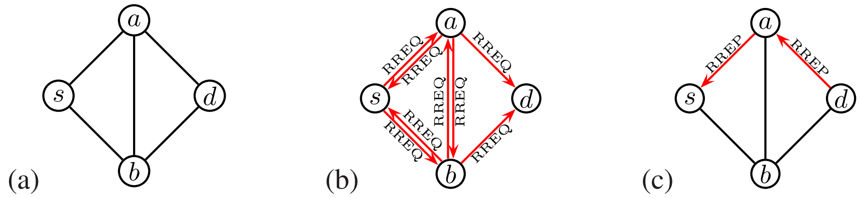



Figure 1: Example network topology

# 2 Ad hoc On-Demand Distance Vector Routing Protocol

AODV [79] is a widely-used routing protocol designed for MANETs, and is one of the four protocols currently standardised by the IETF MANET working group[9]. It also forms the basis of new WMN routing protocols, including the upcoming IEEE 802.11s wireless mesh network standard [53].

## 2.1 Basic Protocol

AODV is a reactive protocol: routes are established only on demand. A route from a source node $s$ to a destination node $d$ is a sequence of nodes $[s, n_1, \ldots, n_k, d]$, where $n_1$, …, $n_k$ are intermediate nodes located on the path from $s$ to $d$. Its basic operation can best be explained using a simple example topology shown in Figure 1(a), where edges connect nodes within transmission range. We assume node $s$ wants to send a data packet to node $d$, but $s$ does not have a valid routing table entry for $d$. Node $s$ initiates a route discovery mechanism by broadcasting a route request (RREQ) message, which is received by $s$'s immediate neighbours $a$ and $b$. We assume that neither $a$ nor $b$ knows a route to the destination node $d$.[10] Therefore, they simply re-broadcast the message, as shown in Figure 1(b). Each RREQ message has a unique identifier which allows nodes to ignore duplicate RREQ messages that they have handled before.

When forwarding the RREQ message, each intermediate node updates its routing table and adds a "reverse route" entry to $s$, indicating via which next hop the node $s$ can be reached, and the distance in number of hops. Once the first RREQ message is received by the destination node $d$ (we assume via $a$), $d$ also adds a reverse route entry in its routing table, saying that node $s$ can be reached via node $a$, at a distance of 2 hops.

Node $d$ then responds by sending a route reply (RREP) message back to node $s$, as shown in Figure 1(c). In contrast to the RREQ message, the RREP is unicast, i.e., it is sent to an individual next-hop node only. The RREP is sent from $d$ to $a$, and then to $s$, using the reverse routing table entries created during the forwarding of the RREQ message. When processing the RREP message, a node creates a "forward route" entry into its routing table. For example, upon receiving the RREP via $a$, node $s$ creates an entry saying that $d$ can be reached via $a$, at a distance of 2 hops. At the completion of the route discovery process, a route has been established from $s$ to $d$, and data packets can start to flow.

In the event of link and route breaks, AODV uses route error (RERR) messages to inform affected nodes. Sequence numbers are another important aspect of AODV, and are used to indicate the freshness of routing table entries for the purpose of preventing routing loops.

## 2.2 Detailed Examples

Each node *ip* stores and maintains its own sequence number and its own routing table, which consists of exactly one entry for each known destination *dip*. In this paper we represent a routing table entry as a tuple $(\mathit{dip}, \mathit{dsn}, \mathit{dsk}, \mathit{flag}, \mathit{hops}, \mathit{nhip}, \mathit{pre})$, indicating that *nhip* is the next hop on a route to *dip* of length

[9] `http://datatracker.ietf.org/wg/manet/charter/`

[10] In case an intermediate node knows a route to $d$, it directly sends a route reply back.

*hops*; *dsn* is a sequence number measuring the freshness of this information. The flag *dsk* indicates if the sequence number is known (`kno`) or unknown (`unk`). In the former case the sequence number *dsn* can be used to measure the freshness; in the latter the value of *dsn* cannot be used since one cannot "trust" the value. The flag *flag* indicates if the route is *valid* (`val`)—it can be used to forward packets—or if it is outdated (`inv`). Finally, *pre* is the set of neighbours who are "interested" in the route to *dip*—they are expected to use *ip* as the next hop in their own routes to *dip*.

We illustrate the AODV routing protocol in the example of Figure 2, where AODV is used to establish a route between nodes *a* and *c*. The small numbers inside the nodes denote the nodes' sequence numbers. Initially all these numbers are set to 1. For simplicity, we leave out the last component *pre* of routing table entries; hence each entry is a 6-tuple here.

Figure 2(a) shows the initial state. We assume that node *a* wants to send a data packet to node *c*. First, *a* checks its routing table and finds that it does not have a (valid) routing table entry for the destination node *c*. In fact its routing table is empty. Therefore it initiates a route discovery process by generating a RREQ message. For ease of explanation, we represent the generated RREQ message as $\texttt{rreq}(hops\,,rreqid\,,dip\,,dsn\,,dsk\,,oip\,,osn\,,sip)$, indicating that the route request originates from node *oip* with sequence number *osn*, searching for a route to destination *dip* with sequence number at least *dsn*. This sequence number is taken from the entry for *dip* in the routing table maintained by node *a*. If no entry for *dip* is available, *dsn* is set to 0. If there is no entry for *dip* or the sequence number is marked as unknown in the routing table, *dsk* is set to `unk` ("unknown"); otherwise it is set to `kno` ("known"). In addition, *hops* is the number of hops the message has already travelled from *oip*, *rreqid* is the unique identifier of the route request, and *sip* denotes the sender of the message.[11]

When generating a new RREQ message, the originator node must increment its own sequence number before copying it into the RREQ message. Therefore, the RREQ message from node *a* is $\texttt{rreq}(0\,,rreqid\,,c\,,0\,,\texttt{unk}\,,a\,,2\,,a)$. This RREQ message is broadcast to all its neighbours (Figure 2(b)).

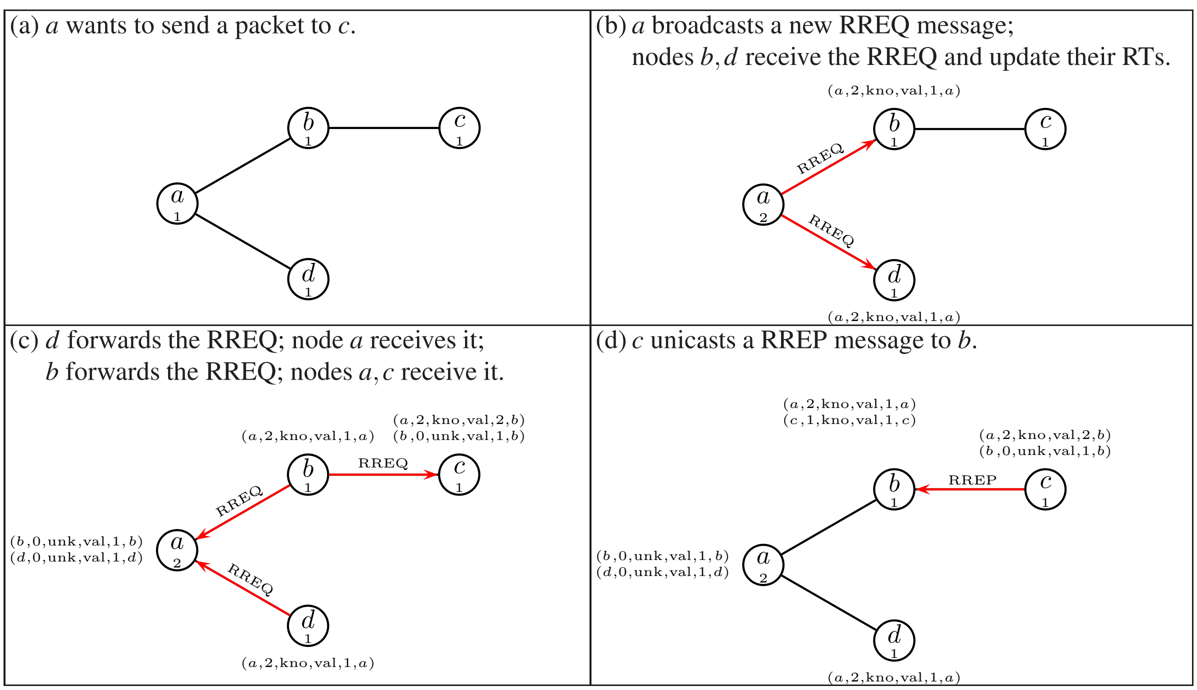



Figure 2: Simple example

[11]Following the RFC specification of AODV, the sender address *sip* is not part of the message itself; however a node that receives a message is able to obtain it from the source IP address field in the IP header of the message.

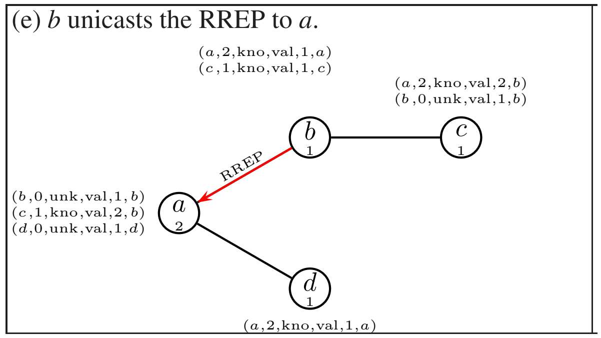



Figure 2 (cont'd): Simple example

Nodes $b$ and $d$ receive the request and update their routing tables to insert an entry for node $a$. Since nodes $b$ and $d$ do not know a route to node $c$ (they have no routing table entry with destination $c$), they both re-broadcast the RREQ message, as shown in Figure 2(c). Before forwarding the message, nodes $b$ and $d$ increment the *hops* information in the RREQ message from 0 to 1, meaning that the distance to $a$ is now 1.

The forwarded RREQ messages from nodes $b$ and $d$ are then received by node $a$. Through these messages node $a$ knows that nodes $b$ and $d$ are 1-hop neighbours, but node $a$ does not know their sequence numbers, hence they are set to "unknown". Therefore, node $a$ creates routing table entries for its neighbours, but with unknown sequence number 0, and sequence-number-status flag set to `unk`. Apart from this, since node $a$ is the originator of the RREQ, it will ignore these messages.

The same RREQ message forwarded by node $b$ is also received by node $c$. Node $c$ reacts by creating routing table entries for both its previous-hop neighbour (node $b$) and the originator of the RREQ (node $a$). It then responds by generating a RREP message. Again, for ease of explanation, we represent the RREP message as $\texttt{rrep}(\mathit{hops}\,,\mathit{dip}\,,\mathit{dsn}\,,\mathit{oip}\,,\mathit{sip})$, where *hops* now indicates the distance to *dip*. As before, *sip* is the sender of the message. Since the destination node's sequence number specified in the received RREQ message is unknown ($\mathit{dsn} = 0$ and $\mathit{dsk} = \texttt{unk}$), node $c$ copies its own sequence number into the RREP message. Hence the RREP message from node $c$ is $\texttt{rrep}(0\,,c\,,1\,,a\,,c)$.

From node $c$, the RREP message is unicast back to its previous-hop node $b$, on the path back towards the originator node $a$ (Figure 2(d)). Node $b$ processes the RREP message and updates its routing table to insert an entry for node $c$. It also increments the *hops* information in the RREP message from 0 to 1 before forwarding it to node $a$ (Figure 2(e)). When node $a$ receives the RREP message, this completes the route discovery process and a route is now established from node $a$ to node $c$. Data packets from node $a$ can now be sent to node $c$.

We next describe a more interesting example of how AODV operates in a changing network topology. In this example, we will show that due to the changing network topology and subsequent updates to the routing table, a route reply message is not necessarily sent back to the node which had forwarded the route request previously.

Figure 3(a) shows the initial network topology, and the initial state of the nodes in the topology. We assume that node $s$ wants to send a data packet to node $d$; hence it generates and broadcasts a route request message $\mathrm{RREQ}_1$ ($\texttt{rreq}(0\,,\mathit{rreqid}\,,d\,,0\,,\texttt{unk}\,,s\,,2\,,s)$), as shown in Figure 3(b).

Next, the network topology changes whereby node $s$ is now within transmission range of node $d$. This change in the network topology can be due to node mobility (i.e., node $s$ moves into transmission range of node $d$), or due to the improved quality of the wireless link between nodes $s$ and $d$. Figure 3(d) shows a situation where node $s$ wants to send a data packet to node $a$, thereby generating and broadcasting a new route request message $\mathrm{RREQ}_2$ ($\texttt{rreq}(0\,,\mathit{rreqid}\,,a\,,0\,,\texttt{unk}\,,s\,,3\,,s)$) destined to node $a$. Note that

$RREQ_2$ is received by node $d$, which results in the insertion of an entry for node $s$ with sequence number 3 in node $d$'s routing table. At the same time, the previous route request $RREQ_1$ is forwarded to node $b$ on its path towards node $d$.

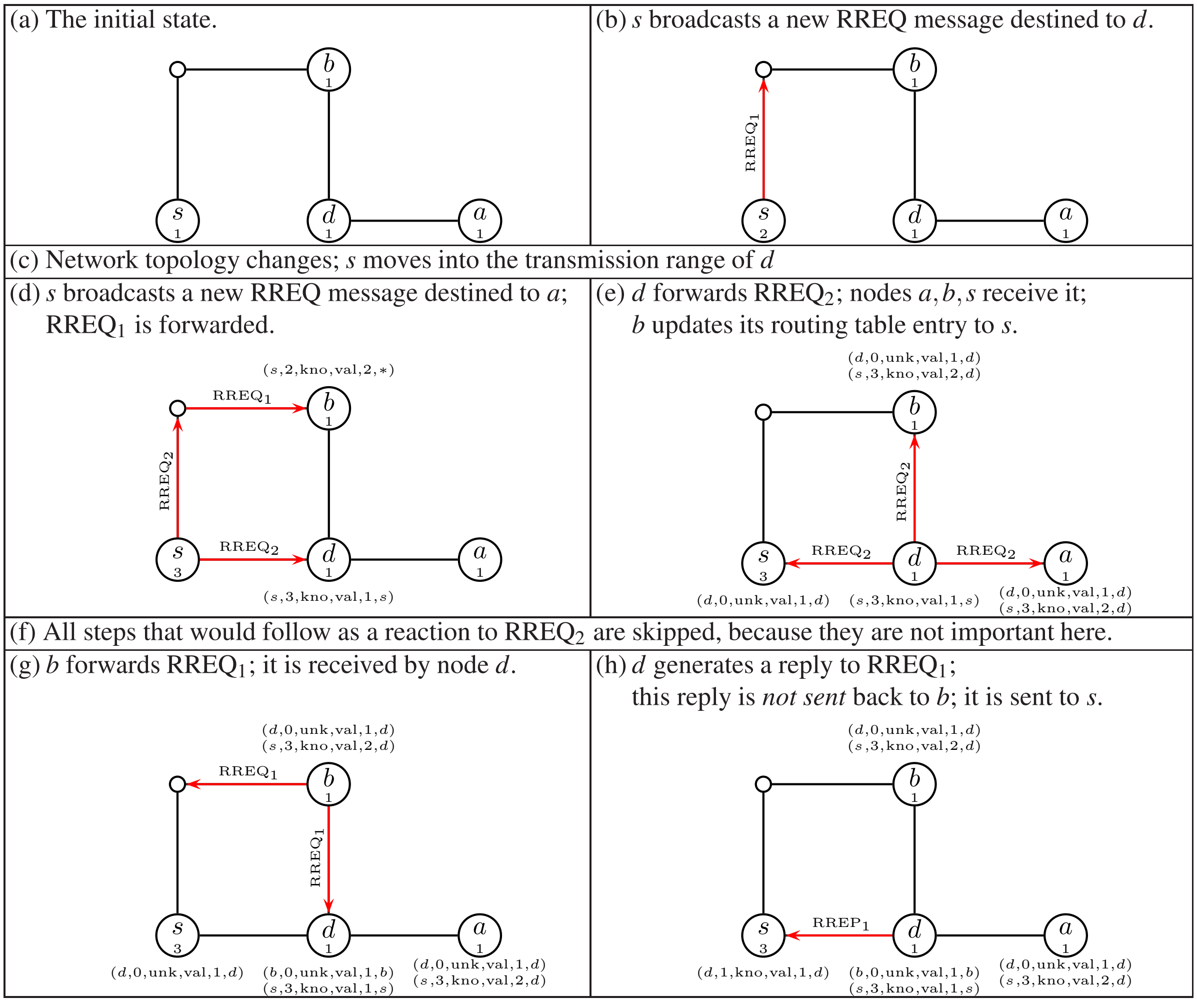



Figure 3: An example with changing network topology

Figure 3(e) shows that node $d$ forwards $RREQ_2$, which is received by nodes $a$, $b$, and $s$. The subsequent steps in response to $RREQ_2$, i.e. the generation of a RREP message by node $a$, and its unicast to node $d$ and subsequent forwarding to the originator node $s$, are not shown in Figure 3 as they do not contribute towards the objective of this example.

Figure 3(g) shows that $RREQ_1$ is forwarded by node $b$ and finally received by the destination node $d$. Since the destination sequence number for node $s$ in $RREQ_1$ ($dsn = 2$) is older than the corresponding destination sequence number information in node $d$'s routing table entry for node $s$ ($dsn = 3$), the routing table entry for node $s$ is not updated. Node $d$ then generates a RREP message in response to $RREQ_1$. The destination node $d$ searches in its routing table for a reverse route entry for node $s$, and finds that the next hop *nhip* for the route towards node $s$ is node $s$ itself. Therefore, the RREP message is not sent back to node $b$ (from which the $RREQ_1$ message is received), but instead is sent back directly to node $s$ (Figure 3(h)).

# 3 Abstractions Chosen

Our formalisation of AODV tries to accurately model the protocol as defined in the IETF RFC 3561 specification [79]. The model focusses on layer 3 of the protocol stack, i.e., the routing and forwarding of messages and packets, and abstracts from lower layer network protocols and mechanisms such as the Carrier Sense Multiple Access (CSMA) protocol. The presented formalisation includes all core components of the protocol, but, at the moment, abstracts from timing issues and optional protocol features. This keeps our specification manageable. Our plan is to extend our model step by step. Even though our model currently does not cover all aspects, it allows us to point to shortcomings in AODV and to discuss some possible improvements. The model also allows us to reason about protocol behaviour and to prove critical protocol characteristics.

In this section, we list all items that are not yet part of our formal model.

## 3.1 Timing

We abstract from *all timing issues*. Surely, this is a big decision and there are good reasons to add time as a next step. However, this abstraction makes the verification of properties much easier:

No entry of a routing table or route reply message has the field *lifetime* that maintains the expiration or deletion time of the route in AODV. Informally this means that no valid route is set to invalid due to timing, that no invalid route disappears from the routing table (except when it is overwritten), and that we never delete elements of the set `rreqs` of already seen requests (described in Section 5.6). In terms of the RFC that means that `ACTIVE_ROUTE_TIMEOUT`, `DELETE_PERIOD` and `PATH_DISCOVERY_TIME` are set to infinity.

## 3.2 Optional Protocol Features

A route may be *locally repaired* if a link break in a valid route occurs. In that case, the node upstream of that break may choose to initiate a local repair if the destination was no farther than `MAX_REPAIR_TTL` hops away. Local repair is optional; therefore we do not model this feature here.

To avoid unnecessary network-wide dissemination of RREQs, the originating node should use an *expanding ring search* technique. This is again an optional feature, which is not modelled here; we can say that the `RING_TRAVERSAL_TIME` is set to infinity.

A route request may be sent multiple times. This happens if a node, after broadcasting a RREQ, does not receive the corresponding RREP within a given amount of time. In that case the node may broadcast another RREQ, up to a maximum of `RREQ_RETRIES`. Since the default value for `RREQ_RETRIES` is only two, and moreover this whole procedure is optional, we have not modelled this resending of RREQ messages.

If a route discovery has been attempted `RREQ_RETRIES` times without receiving any RREP, a *destination unreachable message* should be delivered to the client (application) hooked up at the originator. This interaction between different layers of the protocol stack has not been modelled here since it is not a core part of the protocol itself.

When a node wants to increment its sequence number, but the largest possible number ($2^{32}-1$) has already been assigned to it, a *sequence number rollover* has to be accomplished. This rollover violates the property that sequence numbers are monotonically increased over time; therefore it would be possible to create routing loops. It appears that loops as a consequence of rollover are rare in practice and therefore we decided to model sequence numbers by the unbounded set of natural numbers.

*Interfaces*, as part of routing table entries, store information concerning the network link, e.g., that the node is connected via Ethernet. This is because AODV should operate smoothly over wired as well as wireless networks. Here we assume that nodes have only one type of network interface and consequently leave out this field.

Another phenomenon which may yield complications and possibly routing loops, are node crashes. For now, we have neither modelled crashes nor *actions after reboot*.

By default, our process algebra establishes only bidirectional links[12]. We will point out how by a trivial change it can model unidirectional links (Section 4). We have decided not to make this our default here, since, by doing so, fundamental properties such as route correctness would not hold for AODV any longer (see Section 7.7). Unidirectional links come along with "blacklist" sets, which we also do not model.

We further do not model the optional support for aggregate networks and the use of AODV with other networks, as loosely discussed in Sections 7 and 8 of the AODV RFC [79].

Finally, *hello messages* can be used as an optional feature to offer connectivity information to a node's neighbours. Since in our model all optional parts are skipped, we do not model hello messages either; information about 1-hop neighbours is established by receiving AODV control messages.

### 3.3 Flags

Following the RFC [79], AODV control messages and routing table entries have to maintain a series of state and routing flags such as the repair flag, the unknown sequence number flag, and the gratuitous RREP flag. For most of these flags there is no compulsion to ever set them. An exception is the *unknown sequence number ('U') flag*. In some implementations, such as AODV-UU [2], this flag is omitted in favour of a special element denoting the unknown sequence number. In our model, we follow the RFC and model the sequence number as well as the 'U' flag. We speak of a *sequence-number-status flag*, with values "known" and "unknown".

Besides the 'U' flag, each route request has the *join* ('J'), the *repair* ('R'), the *gratuitous RREP* ('G') and the *destination only* ('D') *flag*. The 'J' and 'R' flag are reserved for *multicast*, an optional feature not fully specified in the RFC. We do not model the multicast feature, and hence ignore these two flags. The 'G' flag indicates whether a gratuitous RREP should be unicast, by an intermediate node answering the RREQ message, to the destination node of the original message; the 'D' flag indicates that only the destination may respond to this RREQ. Both flags may be set when a request is initiated. Since this is also optional, we have decided to skip these features for the moment. However, their inclusion should be straightforward.

A route reply carries two flags: the *repair* ('R') flag, used for the multicast feature, and the *acknowledgment* ('A') *flag*, which indicates that a route reply acknowledgment message must be sent in response to a RREP message. We do not model these flags: the former since we do not model multicast at all; the latter since this flag is optional. Consequently, we have no need to model the *route reply acknowledgment (RREP-ACK) message*, which—next to RREQ, RREP and RERR—constitutes a fourth kind of AODV control message.

Finally, an error message only maintains the *no delete* ('N') *flag*. It is set if a node has performed a local repair. Since we do not model local repair, we are able to abstract from that flag.

Flags pertaining to local repair, but stored in the routing tables, are the *repairable* and the *being repaired flags*. For the same reasons, we skip these flags as well.

[12] A bidirectional link means that if a node $b$ is in transmission range of $a$ ($a$ can send messages to $b$), then $a$ is also in range of $b$. A bidirectional link does *not* mean that if $a$ knows a route to $b$, then $b$ knows a route to $a$.

# 4 A Process Algebra for Wireless Mesh Routing Protocols

In this section we propose AWN (Algebra of Wireless Networks), a process algebra for the specification of WMN routing protocols, such as AODV. It is a variant of standard process algebras [71, 47, 4, 8], adapted to the problem at hand. For example, it allows us to embed data structures. In AWN, a WMN is modelled as an encapsulated parallel composition of network nodes. On each node several sequential processes may be running in parallel. Network nodes communicate with their direct neighbours—those nodes that are in transmission range—using either broadcast or unicast. Our formalism maintains for each node the set of nodes that are currently in transmission range. Due to mobility of nodes and variability of wireless links, nodes can move in or out of transmission range. The encapsulation of the entire network inhibits communications between network nodes and the outside world, with the exception of the receipt and delivery of data packets from or to clients[13] of the modelled protocol that may be hooked up to various nodes.

## 4.1 A Language for Sequential Processes

The internal state of a process is determined, in part, by the values of certain data variables that are maintained by that process. To this end, we assume a data structure with several types, variables ranging over these types, operators and predicates. First order predicate logic yields terms (or *data expressions*) and formulas to denote data values and statements about them.[14] Our data structure always contains the types $\mathtt{DATA}$, $\mathtt{MSG}$, $\mathtt{IP}$ and $\mathscr{P}(\mathtt{IP})$ of *application layer data*, *messages*, *IP addresses*—or any other node identifiers—and *sets of IP addresses*. We further assume that there is a function $\mathtt{newpkt}: \mathtt{DATA} \times \mathtt{IP} \rightarrow \mathtt{MSG}$ that generates a message with new application layer data for a particular destination. The purpose of this function is to inject data to the protocol; details will be given later.

In addition, we assume a type $\mathtt{SPROC}$ of *sequential processes*, and a collection of *process names*, each being an operator of type $\mathtt{TYPE}_1 \times \cdots \times \mathtt{TYPE}_n \rightarrow \mathtt{SPROC}$ for certain data types $\mathtt{TYPE}_i$. Each process name $X$ comes with a *defining equation*

$$X(\mathtt{var}_1, \ldots, \mathtt{var}_n) \stackrel{def}{=} p\ ,$$

in which, for each $i = 1, \ldots, n$, $\mathtt{var}_i$ is a variable of type $\mathtt{TYPE}_i$ and $p$ a *sequential process expression* defined by the grammar below. $p$ may contain the variables $\mathtt{var}_i$ as well as $X$; however, all occurrences of data variables in $p$ have to be *bound*.[15] The choice of the underlying data structure and the process names with their defining equations can be tailored to any particular application of our language; our decisions made for modelling AODV are presented in Sections 5 and 6. The process names are used to denote the processes that feature in this application, with their arguments $\mathtt{var}_i$ binding the current values of the data variables maintained by these processes.

The *sequential process expressions* are given by the following grammar:

$$SP ::= X(exp_1, \ldots, exp_n) \mid [\varphi]SP \mid [\![\mathtt{var} := exp]\!]SP \mid SP + SP \mid \alpha.SP \mid \mathbf{unicast}(dest, ms).SP \blacktriangleright SP$$
$$\alpha ::= \mathbf{broadcast}(ms) \mid \mathbf{groupcast}(dests, ms) \mid \mathbf{send}(ms) \mid \mathbf{deliver}(data) \mid \mathbf{receive}(\mathtt{msg})$$

Here $X$ is a process name, $exp_i$ a data expression of the same type as $\mathtt{var}_i$, $\varphi$ a data formula, $\mathtt{var} := exp$ an assignment of a data expression $exp$ to a variable $\mathtt{var}$ of the same type, $dest$, $dests$, $data$ and $ms$ data expressions of types $\mathtt{IP}$, $\mathscr{P}(\mathtt{IP})$, $\mathtt{DATA}$ and $\mathtt{MSG}$, respectively, and $\mathtt{msg}$ a data variable of type $\mathtt{MSG}$.

[13] The application layer that initiates packet sending and awaits receipt of a packet.

[14] As operators we also allow *partial* functions with the convention that any atomic formula containing an undefined subterm evaluates to $\mathtt{false}$.

[15] An occurrence of a data variable in $p$ is *bound* if it is one of the variables $\mathtt{var}_i$, a variable $\mathtt{msg}$ occurring in a subexpression $\mathbf{receive}(\mathtt{msg}).q$, a variable $\mathtt{var}$ occurring in a subexpression $[\![\mathtt{var} := exp]\!]q$, or an occurrence in a subexpression $[\varphi]q$ of a variable occurring free in $\varphi$. Here $q$ is an arbitrary sequential process expression.

$$\xi, \mathbf{broadcast}(ms).p \xrightarrow{\mathbf{broadcast}(\xi(ms))} \xi, p$$

$$\xi, \mathbf{groupcast}(dests, ms).p \xrightarrow{\mathbf{groupcast}(\xi(dests),\xi(ms))} \xi, p$$

$$\xi, \mathbf{unicast}(dest, ms).p \blacktriangleright q \xrightarrow{\mathbf{unicast}(\xi(dest),\xi(ms))} \xi, p$$

$$\xi, \mathbf{unicast}(dest, ms).p \blacktriangleright q \xrightarrow{\neg\mathbf{unicast}(\xi(dest),\xi(ms))} \xi, q$$

$$\xi, \mathbf{send}(ms).p \xrightarrow{\mathbf{send}(\xi(ms))} \xi, p$$

$$\xi, \mathbf{deliver}(data).p \xrightarrow{\mathbf{deliver}(\xi(data))} \xi, p$$

$$\xi, \mathbf{receive}(\mathtt{msg}).p \xrightarrow{\mathbf{receive}(m)} \xi[\mathtt{msg} := m], p \qquad (\forall m \in \mathtt{MSG})$$

$$\xi, [\![\mathtt{var} := exp]\!]p \xrightarrow{\tau} \xi[\mathtt{var} := \xi(exp)], p$$

$$\frac{\emptyset[\mathtt{var}_i := \xi(exp_i)]_{i=1}^{n}, p \xrightarrow{a} \zeta, p'}{\xi, X(exp_1, \ldots, exp_n) \xrightarrow{a} \zeta, p'} \; (X(\mathtt{var}_1, \ldots, \mathtt{var}_n) \stackrel{def}{=} p) \qquad (\forall a \in \mathrm{Act})$$

$$\frac{\xi, p \xrightarrow{a} \zeta, p'}{\xi, p+q \xrightarrow{a} \zeta, p'} \qquad \frac{\xi, q \xrightarrow{a} \zeta, q'}{\xi, p+q \xrightarrow{a} \zeta, q'} \qquad \frac{\xi \xrightarrow{\varphi} \zeta}{\xi, [\varphi]p \xrightarrow{\tau} \zeta, p} \qquad (\forall a \in \mathrm{Act})$$

Table 1: *Structural operational semantics for sequential process expressions*

Given a valuation of the data variables by concrete data values, the sequential process $[\varphi]p$ acts as $p$ if $\varphi$ evaluates to `true`, and deadlocks if $\varphi$ evaluates to `false`. In case $\varphi$ contains free variables that are not yet interpreted as data values, values are assigned to these variables in any way that satisfies $\varphi$, if possible. The sequential process $[\![\mathtt{var} := exp]\!]p$ acts as $p$, but under an updated valuation of the data variable `var`. The sequential process $p+q$ may act either as $p$ or as $q$, depending on which of the two processes is able to act at all. In a context where both are able to act, it is not specified how the choice is made. The sequential process $\alpha.p$ first performs the action $\alpha$ and subsequently acts as $p$. The action **broadcast**$(ms)$ broadcasts (the data value bound to the expression) $ms$ to the other network nodes within transmission range, whereas **unicast**$(dest, ms).p \blacktriangleright q$ is a sequential process that tries to unicast the message $ms$ to the destination $dest$; if successful it continues to act as $p$ and otherwise as $q$. In other words, **unicast**$(dest, ms).p$ is prioritised over $q$; only if the action **unicast**$(dest, ms)$ is not possible, the alternative $q$ will happen. It models an abstraction of an acknowledgment-of-receipt mechanism that is typical for unicast communication but absent in broadcast communication, as implemented by the link layer of relevant wireless standards such as IEEE 802.11. The process **groupcast**$(dests, ms).p$ tries to transmit $ms$ to all destinations $dests$, and proceeds as $p$ regardless of whether any of the transmissions is successful. Unlike **unicast** and **broadcast**, the expression **groupcast** does not have a unique counterpart in networking. Depending on the protocol and the implementation it can be an iteratively unicast, a broadcast, or a multicast; thus **groupcast** abstracts from implementation details. The action **send**$(ms)$ synchronously transmits a message to another process running on the same network node; this action can occur only when this other sequential process is able to receive the message. The sequential process **receive**$(\mathtt{msg}).p$ receives any message $m$ (a data value of type `MSG`) either from another node, from another sequential process running on the same node or from the client hooked up to the local node. It then proceeds as $p$, but with the data variable `msg` bound to the value $m$. The submission of data from a client is modelled by the receipt of a message $\mathtt{newpkt}(d, dip)$, where the function `newpkt` generates a message containing the data $d$ and the intended destination $dip$. Data is delivered to the client by **deliver**$(data)$.

The internal state of a sequential process described by an expression $p$ in this language is determined by $p$, together with a *valuation* $\xi$ associating data values $\xi(\mathtt{var})$ to the data variables `var` maintained by this process. Valuations naturally extend to $\xi$*-closed* data expressions—those in which all variables are either bound or in the domain of $\xi$. The structural operational semantics of Table 1 is in the style of Plotkin [85] and describes how one internal state can evolve into another by performing an *action*.[16] The set Act of actions consists of $\textbf{broadcast}(m)$, $\textbf{groupcast}(D,m)$, $\textbf{unicast}(dip,m)$, $\neg\textbf{unicast}(dip,m)$, $\textbf{send}(m)$, $\textbf{deliver}(d)$, $\textbf{receive}(m)$ and internal actions $\tau$, for each choice of $m\in\texttt{MSG}$, $dip\in\texttt{IP}$, $D\in\mathscr{P}(\texttt{IP})$ and $d\in\texttt{DATA}$. Here, $\neg\textbf{unicast}(dip,m)$ denotes a failed unicast. Moreover $\xi[\texttt{var} := v]$ denotes the valuation that assigns the value $v$ to the variable $\texttt{var}$, and agrees with $\xi$ on all other variables. The empty valuation $\emptyset$ assigns values to no variables. Hence $\emptyset[\texttt{var}_i := v_i]_{i=1}^n$ is the valuation that *only* assigns the values $v_i$ to the variables $\texttt{var}_i$ for $i = 1,\dots,n$. The rule for process names in Table 1 (Line 9) says that a process, named $X$, has the same transitions as the body $p$ of its defining equation. In CCS [71], such a rule is $\dfrac{p \stackrel{a}{\longrightarrow} p'}{X \stackrel{a}{\longrightarrow} p'}$. Adding data variables as arguments of process names would yield $\dfrac{\xi,p \stackrel{a}{\longrightarrow} \zeta,p'}{\xi,X(\texttt{var}_1,\dots,\texttt{var}_n) \stackrel{a}{\longrightarrow} \zeta,p'}$. However, a sequential process expression may call a process name with data expressions filled in for these variables. This necessitates a translation from a given valuation $\xi$ of the variables that may occur in these data expressions to a new valuation $\xi^{\#}$ of the variables $\texttt{var}_i$ that occur in the defining equation of $X$:

$$\frac{\xi^{\#},X(\texttt{var}_1,\dots,\texttt{var}_n) \stackrel{a}{\longrightarrow} \zeta,p'}{\xi,X(exp_1,\dots,exp_n) \stackrel{a}{\longrightarrow} \zeta,p'}\ .$$

Here $\xi^{\#}(\texttt{var}_i) = \xi(exp_i)$. Moreover, in defining $\xi^{\#}$ we drop all bindings of variables other than the $\texttt{var}_i$.

**Example 4.1** Given the defining equation

$$X(\texttt{numa}) \stackrel{def}{=} \textbf{send}(\texttt{numa}+1)\,.\,\textbf{receive}(\texttt{numb})\,.\,X(\texttt{numa}+\texttt{numb})$$

and the valuation given by $\xi(\texttt{numa}) = 3$ and $\xi(\texttt{numb}) = 4$, with numa and numb data variables of type $\mathbb{N}$, we have

$$\xi,X(\texttt{numa}+\texttt{numb}) \xrightarrow{\textbf{send}(8)} \zeta,\textbf{receive}(\texttt{numb})\,.\,X(\texttt{numa}+\texttt{numb})\ ,$$

where $\zeta(\texttt{numa}) = 7$ and $\zeta(\texttt{numb})$ is undefined.

An alternative and more traditional rule for process names would be $\dfrac{\xi,p[exp_i/\texttt{var}_i]_{i=1}^n \stackrel{a}{\longrightarrow} \zeta,p'}{\xi,X(exp_1,\dots,exp_n) \stackrel{a}{\longrightarrow} \zeta,p'}$ where $p[exp_i/\texttt{var}_i]_{i=1}^n$ denotes the expression $p$ in which each variable $\texttt{var}_i$ is replaced by the expression $exp_i$, for $i = 1,\dots,n$. This would modify the derivation of Example 4.1 into

$$\xi,X(\texttt{numa}+\texttt{numb}) \xrightarrow{\textbf{send}(8)} \xi,\textbf{receive}(\texttt{numc})\,.\,X(\texttt{numa}+\texttt{numb}+\texttt{numc})\ ,$$

in which one applies $\alpha$*-conversion* when renaming the argument numb of **receive** into numc to avoid a name clash. In this paper we avoid casual application of $\alpha$-conversion, since in our invariant proofs in Section 7 we track the value of variables that are identified by name only. With this in mind we formulated our rule for process names.

The rules defining the choice operator (Table 1, Line 10) are standard and imply immediately that $+$ is associative.

Finally, $\xi \stackrel{\varphi}{\to} \zeta$ says that $\zeta$ is an extension of $\xi$, i.e., a valuation that agrees with $\xi$ on all variables on which $\xi$ is defined, and valuates the other variables occurring free in $\varphi$, such that the formula $\varphi$ holds under $\zeta$. All variables not free in $\varphi$ and not evaluated by $\xi$ are also not evaluated by $\zeta$.

[16] Eight of the transition rules feature statements of the form $\xi(exp)$ where $exp$ is a data expression. Here the application of the rule depends on $\xi(exp)$ being defined. In case $\xi(exp)$ is undefined—either because $exp$ contains a variable that is not in the domain of $\xi$ or because $exp$ contains a partial function that is given an argument for which it is not defined—the transition cannot be taken, possibly leading to a deadlock of the represented process.

$$\frac{P \xrightarrow{a} P'}{P \langle\!\langle\, Q \xrightarrow{a} P' \langle\!\langle\, Q} \quad (\forall a \neq \mathbf{receive}(m)) \qquad \frac{Q \xrightarrow{a} Q'}{P \langle\!\langle\, Q \xrightarrow{a} P \langle\!\langle\, Q'} \quad (\forall a \neq \mathbf{send}(m))$$

$$\frac{P \xrightarrow{\mathbf{receive}(m)} P' \qquad Q \xrightarrow{\mathbf{send}(m)} Q'}{P \langle\!\langle\, Q \xrightarrow{\tau} P' \langle\!\langle\, Q'} \quad (\forall m \in \mathtt{MSG})$$

Table 2: *Structural operational semantics for parallel process expressions*

**Example 4.2** Let $\xi(\mathtt{numa}) = 7$ and $\xi(\mathtt{numb})$, $\xi(\mathtt{numc})$ be undefined. Then the sequential process given by the pair $\xi, [\mathtt{numa} = \mathtt{numb} + \mathtt{numc}]p$ admits several transitions of the form

$$\xi, [\mathtt{numa} = \mathtt{numb} + \mathtt{numc}]p \xrightarrow{\tau} \zeta, p$$

such as the one with $\zeta(\mathtt{numb}) = 2$ and $\zeta(\mathtt{numc}) = 5$. On the other hand, $\xi, [\mathtt{numa} = \mathtt{numb} + 8]p$ admits no transitions, since $\mathtt{numb} \in \mathbb{N}$.

### 4.2 A Language for Parallel Processes

*Parallel process expressions* are given by the grammar

$$PP \ ::= \ \xi, SP \ \mid \ PP \langle\!\langle\, PP$$

where $SP$ is a sequential process expression and $\xi$ a valuation. An expression $\xi, p$ denotes a sequential process expression equipped with a valuation of the variables it maintains. The process $P \langle\!\langle\, Q$ is a parallel composition of $P$ and $Q$, running on the same network node. As formalised in Table 2, an action $\mathbf{receive}(m)$ of $P$ synchronises with an action $\mathbf{send}(m)$ of $Q$ into an internal action $\tau$. These receive actions of $P$ and send actions of $Q$ cannot happen separately. All other actions of $P$ and $Q$, including receive actions of $Q$ and send actions of $P$, occur interleaved in $P \langle\!\langle\, Q$. Thus, in an expression $(P \langle\!\langle\, Q) \langle\!\langle\, R$, for example, the send and receive actions of $Q$ can communicate only with $P$ and $R$, respectively, but the receive actions of $R$, as well as the send actions of $P$, remain available for communication with the environment. Therefore, a parallel process expression denotes a parallel composition of sequential processes $\xi, P$ with information flowing from right to left. The variables of different sequential processes running on the same node are maintained separately, and thus cannot be shared.

Instead of introducing the novel operator $\langle\!\langle$, we could have used the partially synchronous parallel composition operator $\|$ of ACP [4], $|$ of CCS [71] or $\|_A$ of CSP [77]. However, those operators are normally used in conjunction with restriction and/or concealment operators, which are not needed when using $\langle\!\langle$. In ACP a *restriction* or *encapsulation* operator is used to prevent read and send actions of the components of a parallel composition to occur by themselves, without synchronising with an action from another other component. Furthermore, a *concealment* or *abstraction* operator is used to convert the results of successful synchronisation into internal actions, thereby making sure that they will not take part in further synchronisations with the environment. In CCS, the concealment operator is not needed, as the parallel composition directly produces internal actions as the results of synchronisation; however, the restriction operator is indispensable. In CSP, on the other hand, the restriction operator is made redundant by incorporating its function within the parallel composition. In this framework matching read and send actions have the same name, which is also the name of the result of their synchronisation. This makes the concealment operator indispensable. It appears to be impossible to combine the ideas of CCS and CSP directly to make both the restriction and the concealment operator redundant, while maintaining associativity of the parallel composition. Our operator $\langle\!\langle$ is the first that does not need such auxiliary operators, sacrificing commutativity, but not associativity, to make this possible.

Though $\langle\!\langle$ only allows information flow in one direction, it reflects reality of WMNs. Usually two sequential processes run on the same node: $P \langle\!\langle Q$. The main process $P$ deals with all protocol details of the node, e.g., message handling and maintaining the data such as routing tables. The process $Q$ manages the queueing of messages as they arrive; it is always able to receive a message even if $P$ is busy. The use of message queueing in combination with $\langle\!\langle$ is crucial, since otherwise incoming messages would be lost when the process is busy dealing with other messages[17], which would not be an accurate model of what happens in real implementations.

### 4.3 A Language for Networks

We model network nodes in the context of a wireless mesh network by *node expressions* of the form $ip : PP : R$. Here $ip \in \mathtt{IP}$ is the *address* of the node, $PP$ is a parallel process expression, and $R \in \mathscr{P}(\mathtt{IP})$ is the *range* of the node—the set of nodes that are currently within transmission range of $ip$.

$$\frac{P \xrightarrow{\textbf{broadcast}(m)} P'}{ip : P : R \xrightarrow{R:\textbf{*cast}(m)} ip : P' : R} \qquad \frac{P \xrightarrow{\textbf{groupcast}(D,m)} P'}{ip : P : R \xrightarrow{R\cap D:\textbf{*cast}(m)} ip : P' : R}$$

$$\frac{P \xrightarrow{\textbf{unicast}(dip,m)} P' \quad dip \in R}{ip : P : R \xrightarrow{\{dip\}:\textbf{*cast}(m)} ip : P' : R} \qquad \frac{P \xrightarrow{\neg\textbf{unicast}(dip,m)} P' \quad dip \notin R}{ip : P : R \xrightarrow{\tau} ip : P' : R}$$

$$\frac{P \xrightarrow{\textbf{deliver}(d)} P'}{ip : P : R \xrightarrow{ip:\textbf{deliver}(d)} ip : P' : R} \qquad \frac{P \xrightarrow{\textbf{receive}(m)} P'}{ip : P : R \xrightarrow{\{ip\}\neg\emptyset:\textbf{arrive}(m)} ip : P' : R}$$

$$\frac{P \xrightarrow{\tau} P'}{ip : P : R \xrightarrow{\tau} ip : P' : R} \qquad ip : P : R \xrightarrow{\emptyset\neg\{ip\}:\textbf{arrive}(m)} ip : P : R$$

$$ip : P : R \xrightarrow{\textbf{connect}(ip,ip')} ip : P : R \cup \{ip'\} \qquad ip : P : R \xrightarrow{\textbf{disconnect}(ip,ip')} ip : P : R - \{ip'\}$$

$$ip : P : R \xrightarrow{\textbf{connect}(ip',ip)} ip : P : R \cup \{ip'\} \qquad ip : P : R \xrightarrow{\textbf{disconnect}(ip',ip)} ip : P : R - \{ip'\}$$

$$\frac{ip \notin \{ip', ip''\}}{ip : P : R \xrightarrow{\textbf{connect}(ip',ip'')} ip : P : R} \qquad \frac{ip \notin \{ip', ip''\}}{ip : P : R \xrightarrow{\textbf{disconnect}(ip',ip'')} ip : P : R}$$

Table 3: *Structural operational semantics for node expressions*

A *partial network* is then modelled by a *parallel composition* $\|$ of node expressions, one for every node in the network, and a *complete network* is a partial network within an *encapsulation operator* $[\_]$ that limits the communication of network nodes and the outside world to the receipt and the delivery of data packets to and from the application layer attached to the modelled protocol in the network nodes. This yields the following grammar for network expressions:

$$N ::= [M] \qquad\qquad M ::= \ ip : PP : R \ \mid \ M \| M \ .$$

The operational semantics of node and network expressions of Tables 3 and 4 uses transition labels $R:\textbf{*cast}(m)$, $H\neg K:\textbf{arrive}(m)$, $\textbf{connect}(ip,ip')$, $\textbf{disconnect}(ip,ip')$, $ip:\textbf{newpkt}(d,dip)$, $ip:\textbf{deliver}(d)$ and $\tau$. As before, $m \in \mathtt{MSG}$, $d \in \mathtt{DATA}$, $R \in \mathscr{P}(\mathtt{IP})$, and $ip, ip' \in \mathtt{IP}$. Moreover, $H, K \in \mathscr{P}(\mathtt{IP})$ are

[17] assuming that one employs the optional augmentation of Section 4.5

$$\frac{M \xrightarrow{R:*\mathbf{cast}(m)} M' \quad N \xrightarrow{H\neg K:\mathbf{arrive}(m)} N'}{M\|N \xrightarrow{R:*\mathbf{cast}(m)} M'\|N'} \begin{pmatrix} H \subseteq R \\ K \cap R = \emptyset \end{pmatrix} \qquad \frac{M \xrightarrow{H\neg K:\mathbf{arrive}(m)} M' \quad N \xrightarrow{R:*\mathbf{cast}(m)} N'}{M\|N \xrightarrow{R:*\mathbf{cast}(m)} M'\|N'} \begin{pmatrix} H \subseteq R \\ K \cap R = \emptyset \end{pmatrix}$$

$$\frac{M \xrightarrow{H\neg K:\mathbf{arrive}(m)} M' \quad N \xrightarrow{H'\neg K':\mathbf{arrive}(m)} N'}{M\|N \xrightarrow{(H\cup H')\neg(K\cup K'):\mathbf{arrive}(m)} M'\|N'}$$

$$\frac{M \xrightarrow{ip:\mathbf{deliver}(d)} M'}{M\|N \xrightarrow{ip:\mathbf{deliver}(d)} M'\|N} \qquad \frac{N \xrightarrow{ip:\mathbf{deliver}(d)} N'}{M\|N \xrightarrow{ip:\mathbf{deliver}(d)} M\|N'} \qquad \frac{M \xrightarrow{\tau} M'}{M\|N \xrightarrow{\tau} M'\|N} \qquad \frac{N \xrightarrow{\tau} N'}{M\|N \xrightarrow{\tau} M\|N'}$$

$$\frac{M \xrightarrow{\mathbf{connect}(ip,ip')} M' \quad N \xrightarrow{\mathbf{connect}(ip,ip')} N'}{M\|N \xrightarrow{\mathbf{connect}(ip,ip')} M'\|N'} \qquad \frac{M \xrightarrow{\mathbf{disconnect}(ip,ip')} M' \quad N \xrightarrow{\mathbf{disconnect}(ip,ip')} N'}{M\|N \xrightarrow{\mathbf{disconnect}(ip,ip')} M'\|N'}$$

$$\frac{M \xrightarrow{\mathbf{connect}(ip,ip')} M'}{[M] \xrightarrow{\mathbf{connect}(ip,ip')} [M']} \qquad \frac{M \xrightarrow{\mathbf{disconnect}(ip,ip')} M'}{[M] \xrightarrow{\mathbf{disconnect}(ip,ip')} [M']} \qquad \frac{M \xrightarrow{R:*\mathbf{cast}(m)} M'}{[M] \xrightarrow{\tau} [M']} \qquad \frac{M \xrightarrow{\tau} M'}{[M] \xrightarrow{\tau} [M']}$$

$$\frac{M \xrightarrow{ip:\mathbf{deliver}(d)} M'}{[M] \xrightarrow{ip:\mathbf{deliver}(d)} [M']} \qquad \frac{M \xrightarrow{\{ip\}\neg K:\mathbf{arrive}(\mathtt{newpkt}(d,dip))} M'}{[M] \xrightarrow{ip:\mathbf{newpkt}(d,dip)} [M']}$$

Table 4: *Structural operational semantics for network expressions*

sets of IP addresses. The action $R\!:\!*\mathbf{cast}(m)$ casts a message $m$ that can be received by the set $R$ of network nodes. We do not distinguish whether this message has been broadcast, groupcast or unicast—the differences show up merely in the value of $R$. Recall that $D \in \mathscr{P}(\mathtt{IP})$ denotes a set of intended destinations, and $dip \in \mathtt{IP}$ a single destination. A failed unicast attempt on the part of its process is modelled as an internal action $\tau$ on the part of a node expression. The action **send**$(m)$ of a process does not give rise to any action of the corresponding node—this action of a sequential process cannot occur without communicating with a receive action of another sequential process running on the same node.

The action $H\neg K\!:\!\mathbf{arrive}(m)$ states that the message $m$ simultaneously arrives at all addresses $ip \in H$, and fails to arrive at all addresses $ip \in K$. The rules of Table 4 let a $R\!:\!*\mathbf{cast}(m)$-action of one node synchronise with an **arrive**$(m)$ of all other nodes, where this **arrive**$(m)$ amalgamates the arrival of message $m$ at the nodes in the transmission range $R$ of the $*\mathbf{cast}(m)$, and the non-arrival at the other nodes. The rules for **arrive**$(m)$ in Table 3 state that arrival of a message at a node happens if and only if the node receives it, whereas non-arrival can happen at any time. This embodies our assumption that, at any time, any message that is transmitted to a node within range of the sender is actually received by that node. (The eighth rule in Table 3, having no premises, may appear to say that any node $ip$ has the option to disregard any message at any time. However, the encapsulation operator (below) prunes away all such disregard-transitions that do not synchronise with a cast action for which $ip$ is out of range.)

Internal actions $\tau$ and the action $ip\!:\!\mathbf{deliver}(d)$ are simply inherited by node expressions from the processes that run on these nodes, and are interleaved in the parallel composition of nodes that makes up a network. Finally, we allow actions **connect**$(ip,ip')$ and **disconnect**$(ip,ip')$ for $ip,ip' \in \mathtt{IP}$ modelling a change in network topology. Each node needs to synchronise with such an action. These actions can be thought of as occurring nondeterministically, or as actions instigated by the environment of the modelled network protocol. In this formalisation node $ip'$ is in the range of node $ip$, meaning that $ip'$ can receive messages sent by $ip$, if and only if $ip$ is in the range of $ip'$. To break this symmetry, one just skips the last four rules of Table 3 and replaces the synchronisation rules for **connect** and **disconnect** in Table 4 by interleaving rules (like the ones for **deliver** and $\tau$).

The main purpose of the encapsulation operator is to ensure that no messages will be received that have never been sent. In a parallel composition of network nodes, any action $\textbf{receive}(m)$ of one of the nodes $ip$ manifests itself as an action $H\neg K\!:\!\textbf{arrive}(m)$ of the parallel composition, with $ip \in H$. Such actions can happen (even) if within the parallel composition they do not communicate with an action $\textbf{*cast}(m)$ of another component, because they might communicate with a $\textbf{*cast}(m)$ of a node that is yet to be added to the parallel composition. However, once all nodes of the network are accounted for, we need to inhibit unmatched arrive actions, as otherwise our formalism would allow any node at any time to receive any message. One exception however are those arrive actions that stem from an action $\textbf{receive}(\texttt{newpkt}(d, dip))$ of a sequential process running on a node, as those actions represent communication with the environment. Here, we use the function `newpkt`, which we assumed to exist.[18] It models the injection of new data $d$ for destination `dip`.

The encapsulation operator passes through internal actions, as well as delivery of data to destination nodes, this being an interaction with the outside world. $\textbf{*cast}(m)$-actions are declared internal actions at this level; they cannot be steered by the outside world. The connect and disconnect actions are passed through in Table 4, thereby placing them under control of the environment; to make them nondeterministic, their rules should have a $\tau$-label in the conclusion, or alternatively $\textbf{connect}(ip,ip')$ and $\textbf{disconnect}(ip,ip')$ should be thought of as internal actions. Finally, actions $\textbf{arrive}(m)$ are simply blocked by the encapsulation—they cannot occur without synchronising with a $\textbf{*cast}(m)$—except for $\{ip\}\neg K\!:\!\textbf{arrive}(\texttt{newpkt}(d, dip))$ with $d \in \texttt{DATA}$ and $dip \in \texttt{IP}$. This action represents new data $d$ that is submitted by a client of the modelled protocol to node $ip$, for delivery at destination $dip$.

## 4.4 Results on the Process Algebra

Our process algebra admits translation into one without data structures (although we cannot *describe* the target algebra without using data structures). The idea is to replace any variable by all possible values it can take. Formally, processes $\xi, p$ are replaced by $\mathcal{T}_\xi(p)$, where $\mathcal{T}_\xi$ is defined inductively by

$$
\begin{aligned}
&\mathcal{T}_\xi(\textbf{broadcast}(ms).p) = \textbf{broadcast}(\xi(ms)).\mathcal{T}_\xi(p)\ ,\\
&\mathcal{T}_\xi(\textbf{groupcast}(dests,ms).p) = \textbf{groupcast}(\xi(dests),\xi(ms)).\mathcal{T}_\xi(p)\ ,\\
&\mathcal{T}_\xi(\textbf{unicast}(dest,ms).p \blacktriangleright q) = \textbf{unicast}(\xi(dest),\xi(ms)).\mathcal{T}_\xi(p) \blacktriangleright \mathcal{T}_\xi(q)\ ,\\
&\mathcal{T}_\xi(\textbf{send}(ms).p) = \textbf{send}(\xi(ms)).\mathcal{T}_\xi(p)\ ,\\
&\mathcal{T}_\xi(\textbf{deliver}(data).p) = \textbf{deliver}(\xi(data)).\mathcal{T}_\xi(p)\ ,\\
&\mathcal{T}_\xi(\textbf{receive}(\texttt{msg}).p) = \textstyle\sum_{m\in\texttt{MSG}} \textbf{receive}(m).\mathcal{T}_{\xi[\texttt{msg}:=m]}(p)\ ,\\
&\mathcal{T}_\xi([\![\texttt{var} := exp]\!]p) = \tau.\mathcal{T}_{\xi[\texttt{var}:=\xi(exp)]}(p)\ ,\\
&\mathcal{T}_\xi([\varphi]p) = \textstyle\sum_{\{\zeta\mid\xi\stackrel{\varphi}{\to}\zeta\}} \tau.\mathcal{T}_\zeta(p)\ ,\\
&\mathcal{T}_\xi(p+q) = \mathcal{T}_\xi(p) + \mathcal{T}_\xi(q)\ ,\\
&\mathcal{T}_\xi(X(exp_1,\dots,exp_n)) = X_{\xi(exp_1),\dots,\xi(exp_n)}\ .
\end{aligned}
$$

The last equation requires the introduction of a process name $X_{\vec{v}}$ for every substitution instance $\vec{v}$ of the arguments of $X$. The resulting process algebra has a structural operational semantics in the *de Simone* format, generating the same transition system—up to strong bisimilarity, $\leftrightarrow$—as the original. Only the rules for sequential process expressions are different; these are displayed in Table 5. It follows that $\leftrightarrow$, and many other semantic equivalences, are congruences on our language.

**Theorem 4.3** *Strong bisimilarity is a congruence for all operators of our language.*

This is a deep result that usually takes many pages to establish (e.g., [94]). Here we get it directly from the existing theory on structural operational semantics, as a result of carefully designing our language within the disciplined framework described by de Simone [20]. □

[18] To avoid the function `newpkt` we could have introduced a new primitive **newpkt**, which is dual to **deliver**.

$$\mathbf{broadcast}(m).p \xrightarrow{\mathbf{broadcast}(m)} p \qquad \mathbf{send}(m).p \xrightarrow{\mathbf{send}(m)} p$$
$$\mathbf{groupcast}(D,m).p \xrightarrow{\mathbf{groupcast}(D,m)} p \qquad \mathbf{deliver}(d).p \xrightarrow{\mathbf{deliver}(d)} p$$
$$\mathbf{unicast}(dip,m).p \blacktriangleright q \xrightarrow{\mathbf{unicast}(dip,m)} p \qquad \mathbf{receive}(m).p \xrightarrow{\mathbf{receive}(m)} p$$
$$\mathbf{unicast}(dip,m).p \blacktriangleright q \xrightarrow{\neg\mathbf{unicast}(dip,m)} q \qquad \tau.p \xrightarrow{\tau} p$$
$$\frac{p \xrightarrow{a} p'}{X \xrightarrow{a} p'}\ (X \stackrel{def}{=} p) \qquad \frac{p \xrightarrow{a} p'}{p+q \xrightarrow{a} p'} \qquad \frac{q \xrightarrow{a} q'}{p+q \xrightarrow{a} q'} \qquad \frac{p_i \xrightarrow{a} p'}{\sum_{i\in I} p_i \xrightarrow{a} p'} \qquad (\forall a \in \mathrm{Act})$$

Table 5: *Structural operational semantics for sequential processes after elimination of data structures*

**Theorem 4.4** *$\langle\!\langle$ is associative, and $\|$ is associative and commutative, up to $\underline{\leftrightarrow}$.*

*Proof.* The operational rules for these operators fit a format presented in [17], guaranteeing associativity up to $\underline{\leftrightarrow}$. The *ASSOC-de Simone format* of [17] applies to all transition system specifications (TSSs) in de Simone format, and allows 7 different types of rules (named 1–7) for the operators in question. Our TSS is in De Simone format; the three rules for $\langle\!\langle$ of Table 2 are of types 1, 2 and 7, respectively. To be precise, it has rules $1_a$ for $a \in \mathrm{Act} - \{\mathbf{receive}(m) \mid m\in\mathtt{MSG}\}$, rules $2_a$ for $a \in \mathrm{Act} - \{\mathbf{send}(m) \mid m\in\mathtt{MSG}\}$, and rules $7_{(a,b)}$ for $(a,b) \in \{(\mathbf{receive}(m),\mathbf{send}(m)) \mid m\in\mathtt{MSG}\}$. Moreover, the partial *communication function* $\gamma : \mathrm{Act}\times\mathrm{Act} \rightharpoonup \mathrm{Act}$ is given by $\gamma(\mathbf{receive}(m),\mathbf{send}(m)) = \tau$. The main result of [17] is that an operator is guaranteed to be associative, provided that $\gamma$ is associative and six conditions are fulfilled. In the absence of rules of types 3, 4, 5 and 6, five of these conditions are trivially fulfilled, and the remaining one reduces to

$$7_{(a,b)} \;\Rightarrow\; (1_a \Leftrightarrow 2_b) \;\wedge\; (2_a \Leftrightarrow 2_{\gamma(a,b)}) \;\wedge\; (1_b \Leftrightarrow 1_{\gamma(a,b)}) \;.$$

Here $1_a$ says that rule $1_a$ is present, etc. This condition is met for $\langle\!\langle$ because the antecedent holds only when taking $(a,b) = (\mathbf{receive}(m),\mathbf{send}(m))$ for some $m\in\mathtt{MSG}$. In that case $1_a$ is false, $2_b$ is false, and $2_a$, $2_\tau$, $1_b$ and $1_\tau$ are true. Moreover, $\gamma(\gamma(a,b),c)$ and $\gamma(a,\gamma(b,c))$ are never defined, thus making $\gamma$ trivially associative. The argument for $\|$ being associative proceeds likewise. Here the only non-trivial condition is the associativity of $\gamma$, given by

$$\gamma(R\,{:}\,{*}\mathbf{cast}(m), H\neg K\,{:}\,\mathbf{arrive}(m)) = \gamma(H\neg K\,{:}\,\mathbf{arrive}(m), R\,{:}\,{*}\mathbf{cast}(m)) = R\,{:}\,{*}\mathbf{cast}(m)\;,$$

provided $H \subseteq R$ and $K \cap R = \emptyset$, and

$$\gamma(H\neg K\,{:}\,\mathbf{arrive}(m), H'\neg K'\,{:}\,\mathbf{arrive}(m)) = (H\cup H')\neg(K\cup K')\,{:}\,\mathbf{arrive}(m)\;.$$

Commutativity of $\|$ follows by symmetry. □

### 4.5 Optional Augmentation to Ensure Non-Blocking Broadcast

Our process algebra, as presented above, is intended for networks in which each node is *input enabled* [61], meaning that it is always ready to receive any message, i.e., able to engage in the transition $\mathbf{receive}(m)$ for any $m \in \mathtt{MSG}$. In our model of AODV (Section 6) we will ensure this by equipping each node with a message queue that is always able to accept messages for later handling—even when the main sequential process is currently busy. This makes our model *non-blocking*, meaning that no sender can be delayed in transmitting a message simply because one of the potential recipients is not ready to receive it.

However, the operational semantics does allow blocking if one would (mis)use the process algebra to model nodes that are not input enabled. This is a logical consequence of insisting that any broadcast message *is* received by all nodes within transmission range.

Since the possibility of blocking can regarded as a bad property of broadcast formalisms, one may wish to take away the expressiveness of the language that allows modelling a blocking broadcast. This is the purpose of the following optional augmentations of our operational semantics.

The first possibility is the addition of the rule

$$\frac{P \overset{\textbf{receive}(m)}{\not\longrightarrow}}{ip : P : R \xrightarrow{\{ip\}\neg\emptyset : \textbf{arrive}(m)} ip : P : R}\,.$$

It states that a message may arrive at a node *ip* regardless whether the node is ready to receive it; if it is not ready, the message is simply ignored, and the process running on the node remains in the same state.

A variation on the same idea stems from the *Calculus of Broadcasting Systems* (CBS) [88]. It consists in eliminating the negative premise in the above rule in favour of actions $\textbf{ignore}(m) \in \text{Act}$—in [88] called *discard* actions $w$: —which can be performed by a process exactly when it is not ready to do a $\textbf{receive}(m)$. The rule above then becomes

$$\frac{P \xrightarrow{\textbf{ignore}(m)} P'}{ip : P : R \xrightarrow{\{ip\}\neg\emptyset : \textbf{arrive}(m)} ip : P' : R}$$

and we need the extra rules:

$$\xi, \textbf{broadcast}(ms).p \xrightarrow{\textbf{ignore}(m)} \xi, \textbf{broadcast}(ms).p$$

$$\xi, \textbf{groupcast}(dests, ms).p \xrightarrow{\textbf{ignore}(m)} \xi, \textbf{groupcast}(dests, ms).p$$

$$\xi, \textbf{unicast}(dest, ms).p \blacktriangleright q \xrightarrow{\textbf{ignore}(m)} \xi, \textbf{unicast}(dest, ms).p \blacktriangleright q$$

$$\xi, \textbf{send}(ms).p \xrightarrow{\textbf{ignore}(m)} \xi, \textbf{send}(ms).p$$

$$\xi, \textbf{deliver}(data).p \xrightarrow{\textbf{ignore}(m)} \xi, \textbf{deliver}(data).p$$

$$\xi, [\![\texttt{var} := exp]\!]p \xrightarrow{\textbf{ignore}(m)} \xi, [\![\texttt{var} := exp]\!]p$$

$$\xi, [\varphi]p \xrightarrow{\textbf{ignore}(m)} \xi, [\varphi]p$$

$$\frac{\xi, p \xrightarrow{\textbf{ignore}(m)} \xi, p' \quad \xi, q \xrightarrow{\textbf{ignore}(m)} \xi, q'}{\xi, p+q \xrightarrow{\textbf{ignore}(m)} \xi, p'+q'}$$

for all $m \in \texttt{MSG}$. Furthermore, the first rule for $\langle\!\langle$ from Table 3 is replaced by

$$\frac{P \xrightarrow{a} P'}{P \langle\!\langle Q \xrightarrow{a} P' \langle\!\langle Q} \quad (\forall a \neq \textbf{receive}(m),\ \textbf{ignore}(m)).$$

These rules ensure that for all $P$ and $m$ we always have $P \xrightarrow{\textbf{ignore}(m)} Q \;\Leftrightarrow\; (Q = P \wedge P \overset{\textbf{receive}(m)}{\not\longrightarrow})$. After elimination of the data structures as described in Section 4.5, this operational semantics is again in the de Simone format.

Either of these two optional augmentations of our semantics gives rise to the same transition system. Moreover, when modelling networks in which all nodes are input enabled—as we do in this paper—the added rule for node expressions will never be used, and the resulting transition system is the same whether we use augmentation or not.

## 4.6 Illustrative Example

To illustrate the use of our process algebra AWN, we consider a network of two nodes $a$ and $b$ ($a, b \in \texttt{IP}$) on which the same process is running, although starting in different states. The process describes a simply (toy-)protocol: whenever a new data packet for destination *dip* "appears",[19] the data is broadcast through the network until it finally reaches *dip*. A node alternates between broadcasting, and receiving and handling a message. The *data* stemming from a message received by node *ip* will be delivered to the application layer if the message is destined for *ip* itself. Otherwise the node forwards the message. Every message travelling through the network and handled by the protocol has the form $\mathtt{mg}(data, dip)$, where $data \in \mathtt{DATA}$ is the data to be sent and $dip \in \mathtt{IP}$ is its destination. The behaviour of each node can be modelled by:

$$\begin{aligned}
&\mathtt{X}(\mathtt{ip}\,,\mathtt{data},\mathtt{dip}) \stackrel{def}{=} \textbf{broadcast}(\mathtt{mg}(\mathtt{data},\mathtt{dip})).\mathtt{Y}(\mathtt{ip})\\
&\mathtt{Y}(\mathtt{ip}) \stackrel{def}{=} \textbf{receive}(\mathtt{m}).([\mathtt{m}{=}\mathtt{mg}(\mathtt{data},\mathtt{dip}) \wedge \mathtt{dip}{=}\mathtt{ip}]\textbf{deliver}(\mathtt{data}).\mathtt{Y}(\mathtt{ip})\\
&\qquad\qquad\qquad\qquad + [\mathtt{m}{=}\mathtt{mg}(\mathtt{data},\mathtt{dip}) \wedge \mathtt{dip}{\neq}\mathtt{ip}]\mathtt{X}(\mathtt{ip}\,,\mathtt{data},\mathtt{dip}))\ .
\end{aligned}$$

If a node is in a state $\mathtt{X}(ip\,, data\,, dip)$, where $ip \in \mathtt{IP}$ is the node's stored value of its ow IP address, it will broadcast $\mathtt{mg}(data, dip)$ and continue in state $\mathtt{Y}(ip)$, meaning that all information about the message is dropped. If a node in state $\mathtt{Y}(ip)$ receives a message *m*—a value that will be assigned to the variable $\mathtt{m}$—it has two ways to continue: process $[\mathtt{m}{=}\mathtt{mg}(\mathtt{data},\mathtt{dip}) \wedge \mathtt{dip}{=}\mathtt{ip}]$ $\textbf{deliver}(\mathtt{data}).\mathtt{Y}(\mathtt{ip})$ is enabled if the incoming message has the form $\mathtt{mg}(data, dip)$ and the node itself is the destination of the message ($\mathtt{dip}{=}\mathtt{ip}$). In that case the data distilled from *m* will be delivered to the application layer, and the process returns to $\mathtt{Y}(ip)$. Alternatively, if $[\mathtt{m}{=}\mathtt{mg}(\mathtt{data},\mathtt{dip}) \wedge \mathtt{dip}{\neq}\mathtt{ip}]$, the process continues as $\mathtt{X}(ip\,, data, dip)$, which will then broadcast another message with contents *data* and *dip*. Note that calls to processes use expressions as parameters.

Let us have a look at three scenarios. First, assume that the nodes *a* and *b* are within transmission range of each other; node *a* in state $\mathtt{X}(a\,, d\,, a)$, and node *b* in $\mathtt{Y}(b)$. This is formally expressed as $[a\!:\!\mathtt{X}(a\,, d\,, a)\!:\!\{b\} \,\|\, b\!:\!\mathtt{Y}(b)\!:\!\{a\}]$, although for compactness of presentation, we just write $[\mathtt{X}(a\,, d\,, a) \,\|\, \mathtt{Y}(b)]$ below. In this case, node *a* broadcasts the message $\mathtt{mg}(d, a)$ and continues as $\mathtt{Y}(a)$. Node *b* receives the message, and continues (after evaluation of the message) as $\mathtt{X}(b\,, d\,, a)$. Next *b* broadcasts (forwards) the message, and continues as $\mathtt{Y}(b)$, while node *a* receives $\mathtt{mg}(d, b)$, and, due to evaluation, **deliver**s *d* and continues as $\mathtt{Y}(a)$. Formally, we get transitions from one state to the other:

$$[\mathtt{X}(a\,, d\,, a) \,\|\, \mathtt{Y}(b)] \xrightarrow{a:\textbf{*cast}(\mathtt{mg}(d,a))} \xrightarrow{\tau} [\mathtt{Y}(a) \,\|\, \mathtt{X}(b\,, d\,, a)] \xrightarrow{b:\textbf{*cast}(\mathtt{mg}(d,a))} \xrightarrow{\tau} \xrightarrow{a:\textbf{deliver}(d)} [\mathtt{Y}(a) \,\|\, \mathtt{Y}(b)].$$

Here, the $\tau$-transitions are the actions of evaluating one of the two guards of a process $\mathtt{Y}$, and we left out three intermediate expressions.

Second, assume that the nodes are not within transmission range, with the initial process of *a* and *b* the same as above; formally $[a\!:\!\mathtt{X}(a\,, d\,, a)\!:\!\emptyset \,\|\, b\!:\!\mathtt{Y}(b)\!:\!\emptyset]$. As before, node *a* broadcasts $\mathtt{mg}(d, a)$ and continues in $\mathtt{Y}(a)$; but this time the message is not received by any node; hence no message is forwarded or delivered and both nodes end up running process $\mathtt{Y}$.

For the last scenario, we assume that *a* and *b* are within transmission range and that they have the initial states $\mathtt{X}(a\,, d\,, b)$ and $\mathtt{X}(b\,, e\,, a)$. Without the augmentation of Section 4.5, the network expression $[\mathtt{X}(a\,, d\,, b) \,\|\, \mathtt{X}(b\,, e\,, a)]$ admits no transitions at all; neither node can broadcast its message, because the other node is not listening. With the optional augmentation, assuming that node *a* sends first:

$$[\mathtt{X}(a\,, d\,, b) \,\|\, \mathtt{X}(b\,, e\,, a)] \xrightarrow{a:\textbf{*cast}((\mathtt{mg}(d,b))} [\mathtt{Y}(a) \,\|\, \mathtt{X}(b\,, e\,, a)] \xrightarrow{b:\textbf{*cast}(\mathtt{mg}(e,a))} \xrightarrow{\tau} \xrightarrow{a:\textbf{deliver}(e)} [\mathtt{Y}(a) \,\|\, \mathtt{Y}(b)].$$

Unfortunately, node *b* is initially in a state where it cannot receive a message, so *a*'s message "remains unheard" and *b* will never deliver that message. To avoid this behaviour, and ensure that both messages get delivered, as happens in real WMNs, a message queue can be introduced (see Section 6.6). Using a message queue, the optional augmentation is not needed, since any node is always in a state where it can receive a message.

[19] In this small example, we assume that new data packets just appear "magically"; of course one could use the message $\mathtt{newpkt}(\mathtt{data},\mathtt{dip})$ instead.

# 5 Data Structure for AODV

In this section we set out the basic data structure needed for the detailed formal specification of AODV. As well as describing *types* for the information handled at the nodes during the execution of the protocol we also define functions which will be used to describe the precise intention—and overall effect—of the various update mechanisms in an AODV implementation. The definitions are grouped roughly according to the various "aspects" of AODV and the host network.

## 5.1 Mandatory Types

As stated in the previous section, the data structure always consists of application layer data, messages, IP addresses and sets of IP addresses.

(a) The ultimate purpose of AODV is to deliver *application layer data*. The type DATA describes a set of application layer data items. An item of data is thus a particular element of that set, denoted by the variable $\texttt{data} \in \texttt{DATA}$.

(b) *Messages* are used to send information via the network. In our specification we use the variable msg of the type MSG. We distinguish AODV control messages (route request, route reply, and route error) as well as *data packets*: messages for sending application layer data (see Section 5.8).

(c) The type IP describes a set of IP addresses or, more generally, a *set of node identifiers*. In the RFC 3561 [79], IP is defined as the set of all IP addresses. We assume that each node has a unique identifier $ip \in \texttt{IP}$. Moreover, in our model, each node *ip* maintains a variable ip which always has the value *ip*. In any AODV control message, the variable sip holds the IP address of the sender, and if the message is part of the *route discovery process*—a route request or route reply message—we use oip and dip for the origin and destination of the route sought. Furthermore, rip denotes an unreachable destination and nhip the next hop on some route.

## 5.2 Sequence Numbers

As explained in Section 2, any node maintains its own *sequence number*— the value of the variable sn— and a routing table whose entries describe routes to other nodes. The value of sn increases over time. AODV equips each routing table entry with a sequence number to constitute a measure approximating the relative freshness of the information held—a smaller number denotes older information. All sequence numbers of routes to $dip \in \texttt{IP}$ stored in routing tables are ultimately derived from *dip*'s own sequence number at the time such a route was discovered.

We denote the set of sequence numbers by SQN and assume it to be totally ordered. By default we take SQN to be $\mathbb{N}$, and use standard functions such as max. The initial sequence number of any node is 1. We reserve a special element $0 \in \texttt{SQN}$ to be used for the sequence number of a route, whose semantics is that no sequence number for that route is known. Sequence numbers are incremented by the function

$$\begin{aligned} \texttt{inc} : \texttt{SQN} &\rightarrow \texttt{SQN} \\ \texttt{inc}(sn) &= \begin{cases} sn+1 & \text{if } sn \neq 0 \\ sn & \text{otherwise .} \end{cases} \end{aligned}$$

The variables osn, dsn and rsn of type SQN are used to denote the sequence numbers of routes leading to the nodes oip, dip and rip.

AODV tags sequence numbers of routes as "known" or "unknown". This indicates whether the value of the sequence number can be trusted. The sequence-number-status flag is set to unknown (unk) when a routing table entry is updated with information that is not equipped with a sequence number itself. In such a case the old sequence number of the entry is maintained; hence the value unk does not indicate that no sequence number for the entry is known. Here we use the set $\mathtt{K} = \{\mathtt{kno}, \mathtt{unk}\}$ for the possible values of the sequence-number-status flag; we use the variable dsk to range over type K.

### 5.3 Modelling Routes

In a network, pairs $(ip_0, ip_k) \in \mathtt{IP} \times \mathtt{IP}$ of nodes are considered to be "connected" if $ip_0$ can send to $ip_k$ directly, i.e., $ip_0$ is in transmission range of $ip_k$ and vice versa. We say that such nodes are connected by a single *hop*. When $ip_0$ is not connected to $ip_k$ then messages from $ip_0$ directed to $ip_k$ need to be "routed" through intermediate nodes. We say that a *route* (from $ip_0$ to $ip_k$) is made up of a sequence $[ip_0, ip_1, ip_2, \dots, ip_{k-1}, ip_k]$, where $(ip_i, ip_{i+1})$, $i = 0, \dots, k-1$, are connected pairs; the *length* or *hop count* of the route is the number of single hops, and any node $ip_i$ needs only to know the "next hop" address $ip_{i+1}$ in order to be able to route messages intended for the final destination $ip_k$.

In operation, routes to a particular destination are requested and, when finally established, need to be re-evaluated in regard to their "validity". Routes may become *invalid* if one of the pairs $(ip_i, ip_{i+1})$ in the hop-to-hop sequence gets disconnected. Then AODV may be reinvoked, as the need arises, to discover alternative routes. Meanwhile, an invalid route remains invalid until fresh information is received which establishes a valid replacement route.

In addition to the next hop and hop count, AODV also "tags" a route with its validity, sequence number and sequence-number status. For every route, a node moreover stores a list of *precursors*, modelled as a set of IP addresses. This set collects all nodes which are currently potential users of the route, and are located one hop further "upstream". When the interest of other nodes emerges, these nodes are added to the precursor list[20]; the main purpose of recording this information is to inform those nodes when the route becomes invalid.

In summary, following the RFC, a routing table entry (or entry for short) is given by 7 components:

(a) The destination IP address, which is an element of IP;

(b) The destination sequence number—an element of SQN;

(c) The sequence-number-status flag—an element of the set $\mathtt{K} = \{\mathtt{kno}, \mathtt{unk}\}$;

(d) A flag tagging the route as being valid or invalid—an element of the set $\mathtt{F} = \{\mathtt{val}, \mathtt{inv}\}$. We use the variable flag to range over type F;

(e) The hop count, which is an element of $\mathbb{N}$. The variable hops ranges over the type $\mathbb{N}$ and we make use of the standard function $+1$;

(f) The next hop, which is again an element of IP; and

(g) A precursor list, which is modelled as an element of $\mathscr{P}(\mathtt{IP})$.[21] We use the variable pre to range over $\mathscr{P}(\mathtt{IP})$.

We denote the type of routing table entries by R, use the variable r, and define a generation function

$$(\_,\_,\_,\_,\_,\_,\_) : \mathtt{IP} \times \mathtt{SQN} \times \mathtt{K} \times \mathtt{F} \times \mathbb{N} \times \mathtt{IP} \times \mathscr{P}(\mathtt{IP}) \;\rightarrow\; \mathtt{R}\,.$$

A tuple $(dip\,, dsn\,, dsk\,, flag\,, hops\,, nhip\,, pre)$ describes a route to *dip* of length *hops* and validity *flag*; the very next node on this route is *nhip*; the last time the entry was updated the destination sequence number was *dsn*; *dsk* denotes whether the sequence number is "outdated" or can be used to reason about freshness of the route. Finally, *pre* is a set of all neighbours who are "interested" in the route to *dip*. A

[20] The RFC does not mention a situation where nodes are dropped from the list, which seems curious.
[21] The word "precursor list" is used in the RFC, but no properties of lists are used.

node being "interested" in the route is somewhat sketchily defined as one which has previously used the current node to route messages to *dip*. Interested nodes are recorded in case the route to *dip* should ever become invalid, so that they may subsequently be informed. We use projections $\pi_1,\dots\pi_7$ to select the corresponding component from the 7-tuple: For example, $\pi_6:\mathtt{R}\to\mathtt{IP}$ determines the next hop.

### 5.4 Routing Tables

Nodes store all their information about routes in their *routing tables*; a node *ip*'s routing table consists of a set of routing table entries, exactly one for each known destination. Thus, a routing table is defined as a set of entries, with the restriction that each has a different destination *dip*, i.e., the first component of each entry in a routing table is unique.[22] Formally, we define the type RT of routing tables by

$$\mathtt{RT} := \{rt \mid rt\in\mathscr{P}(\mathtt{R}) \wedge \forall r,s\in rt: r\neq s\Rightarrow \pi_1(r)\neq\pi_1(s)\}\ .$$

In the specification and implementation of AODV during route finding nodes choose between alternative routes if necessary to ensure that only one route per destination ends up in their routing table. In our model, each node *ip* maintains a variable `rt`, whose value is the current routing table of the node.

In the formal model (and indeed in any AODV implementation) we need to extract the components of the entry for any given destination from a routing table. To this end, we define the following partial functions—they are partial because the routing table need not have an entry for the given destination. We begin by selecting the entry in a routing table corresponding to a given destination *dip*:

$$\begin{aligned}\sigma_{route}:\mathtt{RT}\times\mathtt{IP}&\rightharpoonup\mathtt{R}\\ \sigma_{route}(rt,dip)&:=\begin{cases}r & \text{if } r\in rt\wedge\pi_1(r)=dip\\ \text{undefined} & \text{otherwise}\ .\end{cases}\end{aligned}$$

Through the projections $\pi_1,\dots,\pi_7$, defined above, we can now select the components of a selected entry:

(a) The *destination sequence number* relative to the destination *dip*:

$$\begin{aligned}\mathtt{sqn}:\mathtt{RT}\times\mathtt{IP}&\to\mathtt{SQN}\\ \mathtt{sqn}(rt,dip)&:=\begin{cases}\pi_2(\sigma_{route}(rt,dip)) & \text{if } \sigma_{route}(rt,dip) \text{ is defined}\\ 0 & \text{otherwise}\end{cases}\end{aligned}$$

(b) The *"known" status* of the sequence number of a route:

$$\begin{aligned}\mathtt{sqnf}:\mathtt{RT}\times\mathtt{IP}&\to\mathtt{K}\\ \mathtt{sqnf}(rt,dip)&:=\begin{cases}\pi_3(\sigma_{route}(rt,dip)) & \text{if } \sigma_{route}(rt,dip) \text{ is defined}\\ \mathtt{unk} & \text{otherwise}\end{cases}\end{aligned}$$

(c) The *validity status* of a recorded route:

$$\begin{aligned}\mathtt{flag}:\mathtt{RT}\times\mathtt{IP}&\rightharpoonup\mathtt{F}\\ \mathtt{flag}(rt,dip)&:=\pi_4(\sigma_{route}(rt,dip))\end{aligned}$$

(d) The *hop count* of the route from the current node (hosting *rt*) to *dip*:

$$\begin{aligned}\mathtt{dhops}:\mathtt{RT}\times\mathtt{IP}&\rightharpoonup\mathbb{N}\\ \mathtt{dhops}(rt,dip)&:=\pi_5(\sigma_{route}(rt,dip))\end{aligned}$$

[22] As an alternative to restricting the set, we could have defined routing tables as partial functions from IP to R, in which case it makes more sense to define an entry as a 6-tuple, not including the the destination IP as the first component.

(e) The *identity of the next node on the route to dip* (if such a route is known):

$$\begin{aligned} \mathtt{nhop} : \mathtt{RT} \times \mathtt{IP} &\rightharpoonup \mathtt{IP} \\ \mathtt{nhop}(rt, dip) &:= \pi_6(\sigma_{route}(rt, dip)) \end{aligned}$$

(f) The set of *precursors* or neighbours interested in using the route from *ip* to *dip*:

$$\begin{aligned} \mathtt{precs} : \mathtt{RT} \times \mathtt{IP} &\rightharpoonup \mathscr{P}(\mathtt{IP}) \\ \mathtt{precs}(rt, dip) &:= \pi_7(\sigma_{route}(rt, dip)) \end{aligned}$$

The domain of these partial functions changes during the operation of AODV as more routes are discovered and recorded in the routing table *rt*. The first two functions are extended to be total functions: whenever there is no route to *dip* inside the routing table under consideration, the sequence number is set to "unknown" (0) and the sequence-number-status flag is set to "unknown" (unk), respectively. In the same style each partial function could be turned into a total one. However, in the specification we use these functions only when they are defined.

We are not only interested in information about a single route, but also in general information on a routing table:

(a) The set of destination IP addresses for *valid* routes in *rt* is given by

$$\begin{aligned} \mathtt{vD} : \mathtt{RT} &\to \mathscr{P}(\mathtt{IP}) \\ \mathtt{vD}(rt) &:= \{dip \mid (dip, *, *, \mathtt{val}, *, *, *) \in rt\} \end{aligned}$$

(b) The set of destination IP addresses for *invalid* routes in *rt* is

$$\begin{aligned} \mathtt{iD} : \mathtt{RT} &\to \mathscr{P}(\mathtt{IP}) \\ \mathtt{iD}(rt) &:= \{dip \mid (dip, *, *, \mathtt{inv}, *, *, *) \in rt\} \end{aligned}$$

(c) Last, we define the set of destination IP addresses for *known* routes by

$$\begin{aligned} \mathtt{kD} : \mathtt{RT} &\to \mathscr{P}(\mathtt{IP}) \\ \mathtt{kD}(rt) &:= \mathtt{vD}(rt) \cup \mathtt{iD}(rt) = \{dip \mid (dip, *, *, *, *, *, *) \in rt\} \end{aligned}$$

Obviously, the partial functions $\sigma_{route}$, flag, dhops, nhop and precs are defined for *rt* and *dip* exactly when $dip \in \mathtt{kD}(rt)$.

## 5.5 Updating Routing Tables

Routing tables can be updated for three principal reasons. The first is when a node needs to adjust its list of precursors relative to a given destination; the second is when a received request or response carries information about network connectivity; and the last when information is received to the effect that a previously valid route should now be considered invalid. We define an update function for each case.

### 5.5.1 Updating Precursor Lists

Recall that the precursors of a given node *ip* relative to a particular destination *dip* are the nodes that are "interested" in a route to *dip* via *ip*. The function addpre takes a routing table entry and a set of IP addresses *npre* and updates the entry by adding *npre* to the list of precursors already present:

$$\begin{aligned} \mathtt{addpre} : \mathtt{R} \times \mathscr{P}(\mathtt{IP}) &\to \mathtt{R} \\ \mathtt{addpre}((dip, dsn, dsk, flag, hops, nhip, pre), npre) &:= (dip, dsn, dsk, flag, hops, nhip, pre \cup npre)\ . \end{aligned}$$

Often it is necessary to add precursors to an entry of a given routing table. For that, we define the function addpreRT, which takes a routing table *rt*, a destination *dip* and a set of IP addresses *npre* and updates the entry with destination *dip* by adding *npre* to the list of precursors already present. It is only defined if an entry for destination *dip* exists.

$$\begin{aligned}&\mathtt{addpreRT} : \mathtt{RT} \times \mathtt{IP} \times \mathscr{P}(\mathtt{IP}) \rightharpoonup \mathtt{RT}\\ &\quad\mathtt{addpreRT}(rt\,,dip\,,npre) := (rt - \{\sigma_{route}(rt\,,dip)\}) \cup \{\mathtt{addpre}(\sigma_{route}(rt\,,dip)\,,npre)\}\ .\end{aligned}$$

Formally, we remove the entry with destination *dip* from the routing table and insert a new entry for that destination. This new entry is the same as before—only the precursors have been added.

### 5.5.2 Inserting New Information in Routing Tables

If a node gathers new information about a route to a destination *dip*, then it updates its routing table depending on its existing information on a route to *dip*. If no route to *dip* was known at all, it inserts a new entry in its routing table recording the information received. If it already has some (partial) information then it may update this information, depending on whether the new route is fresher or shorter than the one it has already. We define an update function $\mathtt{update}(rt\,,r)$ of a routing table *rt* with an entry *r* only when *r* is valid, i.e., $\pi_4(r) = \mathtt{val}$, $\pi_2(r) = 0 \Leftrightarrow \pi_3(r) = \mathtt{unk}$, and $\pi_3(r) = \mathtt{unk} \Rightarrow \pi_5(r) = 1$.[23]

$$\mathtt{update} : \mathtt{RT} \times \mathtt{R} \rightharpoonup \mathtt{RT}$$

$$\mathtt{update}(rt,r) := \begin{cases} rt \cup \{r\} & \text{if } \pi_1(r) \notin \mathtt{kD}(rt)\\ nrt \cup \{nr\} & \text{if } \pi_1(r) \in \mathtt{kD}(rt) \wedge \mathtt{sqn}(rt\,,\pi_1(r)) < \pi_2(r)\\ nrt \cup \{nr\} & \text{if } \pi_1(r) \in \mathtt{kD}(rt) \wedge \mathtt{sqn}(rt\,,\pi_1(r)) = \pi_2(r) \wedge \mathtt{dhops}(rt\,,\pi_1(r)) > \pi_5(r)\\ nrt \cup \{nr\} & \text{if } \pi_1(r) \in \mathtt{kD}(rt) \wedge \mathtt{sqn}(rt\,,\pi_1(r)) = \pi_2(r) \wedge \mathtt{flag}(rt\,,\pi_1(r)) = \mathtt{inv}\\ nrt \cup \{nr'\} & \text{if } \pi_1(r) \in \mathtt{kD}(rt) \wedge \pi_3(r) = \mathtt{unk}\\ nrt \cup \{ns\} & \text{otherwise}\ , \end{cases}$$

where $s := \sigma_{route}(rt\,,\pi_1(r))$ is the current entry in the routing table for the destination of *r* (if it exists), and $nrt := rt - \{s\}$ is the routing table without that entry. The entry $nr := \mathtt{addpre}(r\,,\pi_7(s))$ is identical to *r* except that the precursors from *s* are added and $ns := \mathtt{addpre}(s\,,\pi_7(r))$ is generated from *s* by adding the precursors from *r*. Lastly, $nr'$ is identical to *nr* except that the sequence number is replaced by the one from the route *s*. More precisely, $nr' := (dip_{nr}\,,\pi_2(s)\,,dsk_{nr}\,,flag_{nr}\,,hops_{nr}\,,nhip_{nr}\,,pre_{nr})$ if $nr = (dip_{nr}\,,*\,,dsk_{nr}\,,flag_{nr}\,,hops_{nr}\,,nhip_{nr}\,,pre_{nr})$. In the situation where $\mathtt{sqn}(rt,\pi_1(r)) = \pi_2(r)$ both routes *nr* and $nr'$ are equal. Therefore, though the cases of the above definition are not mutually exclusive, the function is well defined.

The first case describes the situation where the routing table does not contain any information on a route to *dip*. The second case models the situation where the new route has a greater sequence number. As a consequence all the information from the incoming information is copied into the routing table. In the third and fourth case the sequence numbers are the same and cannot be used to identify better information. Hence other measures are used. The route inside the routing table is only replaced if either the new hop count is strictly smaller—a shorter route has been found—or if the route inside the routing table is marked as invalid. The fifth case deals with the situation where a new route to a known destination has been found without any information on its sequence number ($\pi_2(r) = 0 \wedge \pi_3(r) = \mathtt{unk}$). In that case the routing table entry to that destination is always updated, but the existing sequence number is maintained, and marked as "unknown".

Note that we do not update if we receive a new entry where the sequence number and the hop count are identical to the current entry in the routing table. Following the RFC, the time period (till the valid route becomes invalid) should be reset; however at the moment we do not model timing aspects.

[23] After we have introduced our specification for AODV in Section 6, we will justify that this definition is sufficient.

### 5.5.3 Invalidating Routes

Invalidating routes is a main feature of AODV; if a route is not valid any longer its validity flag has to be set to invalid. By doing this, the stored information about the route, such as the sequence number or the hop count, remains accessible. The process of invalidating a routing table entry follows four rules: (a) any sequence number is incremented by 1, except (b) the truly unknown sequence number ($sqn = 0$, which will only occur if $dsk = \texttt{unk}$) is not incremented, (c) the validity flag of the entry is set to `inv`, and (d) an invalid entry cannot be invalidated again. However, in exception to (a) and (b), when the invalidation is in response to an error message, this message also contains a new (and already incremented) sequence number for each destination to be invalidated.

The function for invalidating routing table entries takes as arguments a routing table and a set of destinations $dests \in \mathscr{P}(\texttt{IP} \times \texttt{SQN})$. Elements of this set are $(rip, rsn)$-pairs that not only identify an unreachable destination *rip*, but also a sequence number that describes the freshness of the faulty route. As for routing tables, we restrict ourselves to sets that have at most one entry for each destination; this time we formally define *dests* as a *partial function* from `IP` to `SQN`, i.e. a subset of $\texttt{IP} \times \texttt{SQN}$ satisfying

$$(rip, rsn), (rip, rsn') \in dests \Rightarrow rsn = rsn' \ .$$

We use the variable `dests` to range over such sets. When invoking `invalidate` we either distil *dests* from an error message, or determine *dests* as a set of pairs $(rip, \texttt{inc}(\texttt{sqn}(rt, rip)))$, where the operator `inc` (from Section 5.2) takes care of (a) and (b). Moreover, we will distil or construct *dests* in such a way that it only lists destinations for which there is a valid entry in the routing table—this takes care of (d).

$$\begin{aligned}
&\texttt{invalidate} : \texttt{RT} \times (\texttt{IP} \rightharpoonup \texttt{SQN}) \to \texttt{RT}\\
&\texttt{invalidate}(rt, dests) \ := \ \{r \,|\, r \in rt \ \wedge \ (\pi_1(r), *) \notin dests\}\\
&\qquad\qquad \cup \ \{(\pi_1(r), rsn, \pi_3(r), \texttt{inv}, \pi_5(r), \pi_6(r), \pi_7(r)) \mid r \in rt \ \wedge \ (\pi_1(r), rsn) \in dests\}
\end{aligned}$$

All entries in the routing table for a destination *rip* in *dests* are modified. The modification replaces the value `val` by `inv` and the sequence number in the entry by the corresponding sequence number from *dests*.

Copying the sequence number from *dests* leaves the possibility that the destination sequence number of an entry is decreased, which violates one of the fundamental assumption of AODV and may yield unexpected behaviour (cf. Section 8.1). To guarantee an increase of the sequence number, *rsn* in Line 3 of the above definition could be replaced by taking the maximum of the sequence number that was already in the routing table $+1$, and the sequence number from *dests*, i.e., $\max(\texttt{inc}(\pi_2(r)), rsn)$.

## 5.6 Route Requests

A route request—RREQ—for a destination *dip* is initiated by a node (with routing table *rt*) if this node wants to transmit a data packet to *dip* but there is no valid entry for *dip* in the routing table, i.e. $dip \notin \texttt{vD}(rt)$. When a new route request is sent out it contains the identity of the originating node *oip*, and a *route request identifier* (RREQ ID); the type of all such identifiers is denoted by `RREQID`, and the variable `rreqid` ranges over this type. This information does not change, even when the request is re-broadcast by any receiving node that does not already know a route to the requested destination. In this way any request still circulating through the network can be uniquely identified by the pair $(oip, rreqid) \in \texttt{IP} \times \texttt{RREQID}$. For our specification we set $\texttt{RREQID} = \mathbb{N}$. In our model, each node maintains a variable `rreqs` of type

$$\mathscr{P}(\texttt{IP} \times \texttt{RREQID})$$

of sets of such pairs to store the sets of route requests seen by the node so far. Within this set, the node records the requests it has previously initiated itself. To ensure a fresh *rreqid* for each new RREQ it generates, the node *ip* applies the following function:

$$\begin{aligned} \mathtt{nrreqid} : \mathscr{P}(\mathtt{IP} \times \mathtt{RREQID}) \times \mathtt{IP} &\rightarrow \mathtt{RREQID} \\ \mathtt{nrreqid}(rreqs\,, ip) &:= \max\{n \mid (ip, n) \in rreqs\} + 1\ , \end{aligned}$$

where we take the maximum of the empty set to be 0.

## 5.7 Queued Packets

Strictly speaking the task of sending data packets is not regarded as part of the AODV protocol—however, failure to send a packet because either a route to the destination is unknown, or a previously known route has become invalid, prompts AODV to be activated. In our modelling we describe this interaction between packet sending and AODV, providing the minimal infrastructure for our specification.

If a new packet is submitted by a client of AODV to a node, it may need to be stored until a route to the packet's destination has been found and the node is not busy carrying out other AODV tasks. We use a queue-style data structure for modelling the store of packets at a node, noting that at each node there may be many data queues, one for each destination. In general, we denote queues of type TYPE by [TYPE], denote the empty queue by [], and make use of the standard (partial) functions $\mathtt{head} : [\mathtt{TYPE}] \rightharpoonup \mathtt{TYPE}$, $\mathtt{tail} : [\mathtt{TYPE}] \rightharpoonup [\mathtt{TYPE}]$ and $\mathtt{append} : \mathtt{TYPE} \times [\mathtt{TYPE}] \to [\mathtt{TYPE}]$ that return the "oldest" element in the queue, remove the "oldest" element, and add a packet to the queue, respectively.

The data type

$$\mathtt{STORE} := \left\{ store \;\middle|\; \begin{array}{l} store \in \mathscr{P}(\mathtt{IP} \times \mathtt{P} \times [\mathtt{DATA}]) \ \wedge \\ \big((dip, p, q), (dip, p', q') \in store \Rightarrow p = p' \wedge q = q'\big) \end{array} \right\}$$

describes stores of enqueued data packets for various destinations, where $\mathtt{P} := \{\mathtt{no\text{-}req}, \mathtt{req}\}$. An element $(dip, p, q) \in \mathtt{IP} \times \mathtt{P} \times [\mathtt{DATA}]$ denotes the queue $q$ of packets destined for *dip*; the request-required flag $p$ is `req` if a new route discovery process for *dip* still needs to be initiated, i.e., a route request message needs to be sent. The value `no-req` indicates that such a RREQ message has been sent already, and either the reply is still pending or a route to *dip* has been established. The flag is set to `req` when a routing table entry is invalidated.

As for routing tables, we require that there is at most one entry for every IP address. In our model, each node maintains a variable `store` of type STORE to record its current store of data packets.

We define some functions for inspecting a store:

(a) Similar to $\sigma_{route}$, we need a function that is able to extract the queue for a given destination.

$$\begin{aligned} \sigma_{queue} : \mathtt{STORE} \times \mathtt{IP} &\rightarrow [\mathtt{DATA}] \\ \sigma_{queue}(store\,, dip) &:= \begin{cases} q & \text{if } (dip, *, q) \in store \\ [\,] & \text{otherwise} \end{cases} \end{aligned}$$

(b) We define a function qD to extract the destinations for which there are unsent packets:

$$\begin{aligned} \mathtt{qD} : \mathtt{STORE} &\rightarrow \mathscr{P}(\mathtt{IP}) \\ \mathtt{qD}(store) &:= \{dip \mid (dip, *, *) \in store\}\ . \end{aligned}$$

Next, we define operations for adding and removing data packets from a store.

(c) Adding a data packet for a particular destination to a store is defined by:

$$\mathtt{add} : \mathtt{DATA} \times \mathtt{IP} \times \mathtt{STORE} \to \mathtt{STORE}$$
$$\mathtt{add}(d\,, dip\,, store) := \begin{cases} store \cup \{(dip\,, \mathtt{req}, \mathtt{append}(d\,, [\,]))\} & \text{if } (dip, *, *) \notin store \\ store - \{(dip, p, q)\} & \\ \qquad \cup \{(dip\,, p\,, \mathtt{append}(d\,, q))\} & \text{if } (dip, p, q) \in store\ . \end{cases}$$

Informally, the process selects the entry $(dip\,,p\,,q) \in store \in \mathtt{STORE}$, where $dip$ is the destination of the application layer data $d$, and appends $d$ to the queue $q$ of $dip$ in that triple; the request-required flag $p$ remains unchanged. In case there is no entry for $dip$ in $store$, the process creates a new queue $[d]$ of stored packets that only contains the data packet under consideration and inserts it—together with $dip$—into the store; the request-required flag is set to `req`, since a route request needs to be sent.

(d) To delete the oldest packet for a particular destination from a store , we define:

$$\begin{aligned} \mathtt{drop} : \mathtt{IP} \times \mathtt{STORE} &\rightharpoonup \mathtt{STORE} \\ \mathtt{drop}(dip\,,store) &:= \begin{cases} store - \{(dip,*,q)\} & \text{if } \mathtt{tail}(q) = [\,] \\ store - \{(dip,p,q)\} & \\ \qquad \cup \{(dip,p,\mathtt{tail}(q))\} & \text{otherwise ,} \end{cases} \end{aligned}$$

where $q = \sigma_{queue}(store\,,dip)$ is the selected queue for destination $dip$. If $dip \notin \mathtt{qD}(store)$ then $q = [\,]$. Therefore $\mathtt{tail}(q)$ and hence also $\mathtt{drop}(dip\,,store)$ is undefined. Note that if $d$ is the last queued packet for a specific destination, the whole entry for the destination is removed from $store$.

In our model of AODV we use only `add` and `drop` to update a store. This ensures that the store will never contain a triple $(dip,*,[\,])$ with an empty data queue, i.e.,

$$dip \in \mathtt{qD}(store) \Rightarrow \sigma_{queue}(store\,,dip) \neq [\,]\ . \tag{1}$$

Finally, we define operations for reading and manipulating the request-required flag of a queue.

(e) We define a partial function $\sigma_{p\text{-}flag}$ to extract the flag for a destination for which there are unsent packets:

$$\begin{aligned} \sigma_{p\text{-}flag} : \mathtt{STORE} \times \mathtt{IP} &\rightharpoonup \mathtt{P} \\ \sigma_{p\text{-}flag}(store\,,dip) &:= \begin{cases} p & \text{if } (dip,p,*) \in store \\ \text{undefined} & \text{otherwise .} \end{cases} \end{aligned}$$

(f) We define functions `setRRF` and `unsetRRF` to change the request-required flag. After a route request for destination $dip$ has been initiated, the request-required flag for $dip$ has to be set to `no-req`.

$$\begin{aligned} \mathtt{unsetRRF} : \mathtt{STORE} \times \mathtt{IP} &\rightarrow \mathtt{STORE} \\ \mathtt{unsetRRF}(store\,,dip) &:= \begin{cases} store - \{(dip,*,q)\} \cup \{(dip,\mathtt{no\text{-}req},q)\} & \text{if } \{(dip,*,q)\} \in store \\ store & \text{otherwise .} \end{cases} \end{aligned}$$

In case that there is no queued data for destination $dip$, the $store$ remains unchanged.

Whenever a route is invalidated the corresponding request-required flag has to be set to `req`; this indicates that the protocol might need to initiate a new route discovery process. Since the function `invalidate` invalidates sets of routing table entries, we define a function with a set of destinations $dests \in \mathscr{P}(\mathtt{IP} \times \mathtt{SQN})$ as one of its arguments (annotated with sequence numbers, which are not used here).

$$\begin{aligned} \mathtt{setRRF} : \mathtt{STORE} \times (\mathtt{IP} \rightharpoonup \mathtt{SQN}) &\rightarrow \mathtt{STORE} \\ \mathtt{setRRF}(store\,,dests) &:= \{(dip,p,q) \mid (dip,p,q) \in store \wedge (dip,*) \notin dests\} \\ &\cup\ \ \{(dip,\mathtt{req},q) \mid (dip,p,q) \in store \wedge (dip,*) \in dests\}\ . \end{aligned}$$

## 5.8 Messages and Message Queues

Messages are the main ingredient of any routing protocol. The message types used in the AODV protocol are route request, route reply, and route error. To generate theses messages, we use functions

$$\begin{aligned}&\texttt{rreq} : \mathbb{N} \times \texttt{RREQID} \times \texttt{IP} \times \texttt{SQN} \times \texttt{K} \times \texttt{IP} \times \texttt{SQN} \times \texttt{IP} \to \texttt{MSG}\\&\texttt{rrep} : \mathbb{N} \times \texttt{IP} \times \texttt{SQN} \times \texttt{IP} \times \texttt{IP} \to \texttt{MSG}\\&\texttt{rerr} : (\texttt{IP} \rightharpoonup \texttt{SQN}) \times \texttt{IP} \to \texttt{MSG}\,.\end{aligned}$$[24]

The function $\texttt{rreq}(\mathit{hops},\mathit{rreqid},\mathit{dip},\mathit{dsn},\mathit{dsk},\mathit{oip},\mathit{osn},\mathit{sip})$ generates a route request. Here, *hops* indicates the hop count from the originator *oip*—that, at the time of sending, had the sequence number *osn*—to the sender of the message *sip*; *rreqid* uniquely identifies the route request; *dsn* is the least level of freshness of a route to dip that is acceptable to *oip*—it has been obtained by incrementing the latest sequence number received in the past by *oip* for a route towards *dip*; and *dsk* indicates whether we can trust that number. In case no sequence number is known, *dsn* is set to 0 and *dsk* to `unk`. By $\texttt{rrep}(\mathit{hops},\mathit{dip},\mathit{dsn},\mathit{oip},\mathit{sip})$ a route reply message is obtained. Originally, it was generated by *dip*—where *dsn* denotes the sequence number of *dip* at the time of sending—and is destined for *oip*; the last sender of the message was the node with IP address *sip* and the distance between *dip* and *sip* is given by *hops*. The error message is generated by $\texttt{rerr}(\mathit{dests},\mathit{sip})$, where $\mathit{dests} : \texttt{IP} \rightharpoonup \texttt{SQN}$ is the list of unreachable destinations and *sip* denotes the sender. Every unreachable destination *rip* comes together with the incremented last-known sequence number *rsn*.

Next to these AODV control messages, we use for our specification also data packets: messages that carry application layer data.

$$\begin{aligned}&\texttt{newpkt} : \texttt{DATA} \times \texttt{IP} \to \texttt{MSG}\\&\texttt{pkt} : \texttt{DATA} \times \texttt{IP} \times \texttt{IP} \to \texttt{MSG}\end{aligned}$$

Although these messages are not part of the protocol itself, they are necessary to initiate error messages, and to trigger the route discovery process. $\texttt{newpkt}(d,\mathit{dip})$ generates a message containing new application layer data *d* destined for a particular destination *dip*. Such a message is submitted to a node by a client of the AODV protocol hooked up to that node. The function $\texttt{pkt}(d,\mathit{dip},\mathit{sip})$ generates a message containing application layer data *d*, that is sent by the sender *sip* to the next hop on the route towards *dip*.

All messages received by a particular node are first stored in a queue (see Section 6.6 for a detailed description). To model this behaviour we use a message queue, denoted by the variable `msgs` of type [`MSG`]. As for every other queue, we will freely use the functions `head`, `tail` and `append`.

### 5.9 Summary

The following table describes the entire data structure we use.

| **Basic Type** | **Variables** | **Description** |
|---|---|---|
| `IP` | `ip, dip, oip, rip, sip, nhip` | node identifiers |
| `SQN` | `dsn, osn, rsn, sn` | sequence numbers |
| `K` | `dsk` | sequence-number-status flag |
| `F` | `flag` | route validity |
| $\mathbb{N}$ | `hops` | hop counts |
| `R` | `r` | routing table entries |
| `RT` | `rt` | routing tables |
| `RREQID` | `rreqid` | request identifiers |
| `P` | | request-required flag |
| `DATA` | `data` | application layer data |
| `STORE` | `store` | store of queued data packets |
| `MSG` | `msg` | messages |

[24] The ordering of the arguments follows the RFC.

| **Complex Type** | **Variables** | **Description** |
|---|---|---|
| $[\mathtt{TYPE}]$ | | queues with elements of type TYPE |
| $[\mathtt{MSG}]$ | msgs | message queues |
| $\mathscr{P}(\mathtt{TYPE})$ | | sets consisting of elements of type TYPE |
| $\mathscr{P}(\mathtt{IP})$ | pre | sets of identifiers (precursors, destinations, ...) |
| $\mathscr{P}(\mathtt{IP}\times\mathtt{RREQID})$ | rreqs | sets of request identifiers with originator IP |
| $\mathtt{TYPE}_1 \rightharpoonup \mathtt{TYPE}_2$ | | partial functions from $\mathtt{TYPE}_1$ to $\mathtt{TYPE}_2$ |
| $\mathtt{IP}\rightharpoonup\mathtt{SQN}$ | dests | sets of destinations with sequence numbers |
| **Constant/Predicate** | | **Description** |
| $0:\mathtt{SQN},\ 1:\mathtt{SQN}$ | | unknown, smallest sequence number |
| $<\subseteq\mathtt{SQN}\times\mathtt{SQN}$ | | strict order on sequence numbers |
| $\mathtt{kno},\mathtt{unk}:\mathtt{K}$ | | constants to distinguish known and unknown sqns |
| $\mathtt{val},\mathtt{inv}:\mathtt{F}$ | | constants to distinguish valid and invalid routes |
| $\mathtt{no\text{-}req},\mathtt{req}:\mathtt{P}$ | | constants indicating whether a RREQ is required |
| $0:\mathbb{N},\ 1:\mathbb{N},\ <\subseteq\mathbb{N}\times\mathbb{N}$ | | standard constants/predicates of natural numbers |
| $[\,]:[\mathtt{TYPE}],\ \emptyset:\mathscr{P}(\mathtt{TYPE})$ | | empty queue, empty set |
| $\in\subseteq\mathtt{TYPE}\times\mathscr{P}(\mathtt{TYPE})$ | | membership, standard set theory |
| **Function** | | **Description** |
| $\mathtt{head}:[\mathtt{TYPE}]\rightharpoonup\mathtt{TYPE}$ | | returns the "oldest" element in the queue |
| $\mathtt{tail}:[\mathtt{TYPE}]\rightharpoonup[\mathtt{TYPE}]$ | | removes the "oldest" element in the queue |
| $\mathtt{append}:\mathtt{TYPE}\times[\mathtt{TYPE}]\to[\mathtt{TYPE}]$ | | inserts a new element into the queue |
| $\mathtt{drop}:\mathtt{IP}\times\mathtt{STORE}\rightharpoonup\mathtt{STORE}$ | | deletes a packet from the queued data packets |
| $\mathtt{add}:\mathtt{DATA}\times\mathtt{IP}\times\mathtt{STORE}\to\mathtt{STORE}$ | | adds a packet to the queued data packets |
| $\mathtt{unsetRRF}:\mathtt{STORE}\times\mathtt{IP}\to\mathtt{STORE}$ | | set the request-required flag to no-req |
| $\mathtt{setRRF}:\mathtt{STORE}\times(\mathtt{IP}\rightharpoonup\mathtt{SQN})\to\mathtt{STORE}$ | | set the request-required flag to req |
| $\sigma_{queue}:\mathtt{STORE}\times\mathtt{IP}\to[\mathtt{DATA}]$ | | selects the data queue for a particular destination |
| $\sigma_{p\text{-}flag}:\mathtt{STORE}\times\mathtt{IP}\rightharpoonup\mathtt{P}$ | | selects the flag for a destination from the store |
| $\sigma_{route}:\mathtt{RT}\times\mathtt{IP}\rightharpoonup\mathtt{R}$ | | selects the route for a particular destination |
| $(\_,\_,\_,\_,\_,\_,\_):\mathtt{IP}\times\mathtt{SQN}\times\mathtt{K}\times\mathtt{F}\times\mathbb{N}\times\mathtt{IP}\times\mathscr{P}(\mathtt{IP})\to\mathtt{R}$ | | generates a routing table entry |
| $\mathtt{inc}:\mathtt{SQN}\to\mathtt{SQN}$ | | increments the sequence number |
| $\max:\mathtt{SQN}\times\mathtt{SQN}\to\mathtt{SQN}$ | | returns the larger sequence number |
| $\mathtt{sqn}:\mathtt{RT}\times\mathtt{IP}\to\mathtt{SQN}$ | | returns the sequence number of a particular route |
| $\mathtt{sqnf}:\mathtt{RT}\times\mathtt{IP}\to\mathtt{K}$ | | determines whether the sequence number is known |
| $\mathtt{flag}:\mathtt{RT}\times\mathtt{IP}\rightharpoonup\mathtt{F}$ | | returns the validity of a particular route |
| $+1:\mathbb{N}\to\mathbb{N}$ | | increments the hop count |
| $\mathtt{dhops}:\mathtt{RT}\times\mathtt{IP}\rightharpoonup\mathbb{N}$ | | returns the hop count of a particular route |
| $\mathtt{nhop}:\mathtt{RT}\times\mathtt{IP}\rightharpoonup\mathtt{IP}$ | | returns the next hop of a particular route |
| $\mathtt{precs}:\mathtt{RT}\times\mathtt{IP}\rightharpoonup\mathscr{P}(\mathtt{IP})$ | | returns the set of precursors of a particular route |
| $\mathtt{vD},\mathtt{iD},\mathtt{kD}:\mathtt{RT}\to\mathscr{P}(\mathtt{IP})$ | | returns the set of valid, invalid, known destinations |
| $\mathtt{qD}:\mathtt{STORE}\to\mathscr{P}(\mathtt{IP})$ | | returns the set of destinations with unsent packets |
| $\cap,\ \cup,\ \bigcup\{\ldots\},\ \ldots$ | | standard set-theoretic functions |
| $\mathtt{addpre}:\mathtt{R}\times\mathscr{P}(\mathtt{IP})\to\mathtt{R}$ | | adds a set of precursors to a routing table entry |
| $\mathtt{addpreRT}:\mathtt{RT}\times\mathtt{IP}\times\mathscr{P}(\mathtt{IP})\rightharpoonup\mathtt{RT}$ | | adds a set of precursors to an entry inside a table |
| $\mathtt{update}:\mathtt{RT}\times\mathtt{R}\rightharpoonup\mathtt{RT}$ | | updates a routing table with a route (if fresh enough) |
| $\mathtt{invalidate}:\mathtt{RT}\times(\mathtt{IP}\rightharpoonup\mathtt{SQN})\to\mathtt{RT}$ | | invalidates a set of routes within a routing table |
| $\mathtt{nrreqid}:\mathscr{P}(\mathtt{IP}\times\mathtt{RREQID})\times\mathtt{IP}\to\mathtt{RREQID}$ | | generates a new route request identifier |
| $\mathtt{newpkt}:\mathtt{DATA}\times\mathtt{IP}\to\mathtt{MSG}$ | | generates a message with new application layer data |
| $\mathtt{pkt}:\mathtt{DATA}\times\mathtt{IP}\times\mathtt{IP}\to\mathtt{MSG}$ | | generates a message containing application layer data |
| $\mathtt{rreq}:\mathbb{N}\times\mathtt{RREQID}\times\mathtt{IP}\times\mathtt{SQN}\times\mathtt{K}\times\mathtt{IP}\times\mathtt{SQN}\times\mathtt{IP}\to\mathtt{MSG}$ | | generates a route request |
| $\mathtt{rrep}:\mathbb{N}\times\mathtt{IP}\times\mathtt{SQN}\times\mathtt{IP}\times\mathtt{IP}\to\mathtt{MSG}$ | | generates a route reply |
| $\mathtt{rerr}:(\mathtt{IP}\rightharpoonup\mathtt{SQN})\times\mathtt{IP}\to\mathtt{MSG}$ | | generates a route error message |

Table 6: *Data structure of AODV*

# 6 Modelling AODV

In this section, we present a specification of the AODV protocol using process algebra. The model includes a mechanism to describe the delivery of data packets; though this is not part of the protocol itself it is necessary to trigger any AODV activity. Our model consists of 7 processes, named `AODV`, `NEWPKT`, `PKT`, `RREQ`, `RREP`, `RERR` and `QMSG`:

- The basic process `AODV` reads a message from the message queue and, depending on the type of the message, calls other processes. When there is no message handling going on, the process initiates the transmission of queued data packets or generates a new route request (if packets are stored for a destination, no route to this destination is known and no route request for this destination is pending).
- The processes `NEWPKT` and `PKT` describe all actions performed by a node when a data packet is received. The former process handles a newly injected packet. The latter describes all actions performed when a node receives data from another node via the protocol. This includes accepting the packet (if the node is the destination), forwarding the packet (if the node is not the destination) and sending an error message (if forwarding fails).
- The process `RREQ` models all events that might occur after a route request has been received. This includes updating the node's routing table, forwarding the route request as well as the initiation of a route reply if a route to the destination is known.
- Similarly, the `RREP` process describes the reaction of the protocol to an incoming route reply.
- The process `RERR` models the part of AODV which handles error messages. In particular, it describes the modification and forwarding of the AODV error message.
- The last process `QMSG` concerns message handling. Whenever a message is received, it is first stored in a message queue. If the corresponding node is able to handle a message it pops the oldest message from the queue and handles it. An example where a node is not ready to process an incoming message immediately is when it is already handling a message.

In the remainder of the section, we provide a formal specification for each of these processes and explain them step by step. Our specification can be split into three parts: the brown lines describe updates to be performed on the node's data, e.g., its routing table; the black lines are other process algebra constructs (cf. Section 4); and the blue lines are ordinary comments.

## 6.1 The Basic Routine

The basic process `AODV` either reads a message from the corresponding queue, sends a queued data packet if a route to the destination has been established, or initiates a new route discovery process in case of queued data packets with invalid or unknown routes. This process maintains five data variables, `ip`, `sn`, `rt`, `rreqs` and `store`, in which it stores its own identity, its own sequence number, its current routing table, the list of route requests seen so far, and its current store of queued data packets that await transmission (cf. Section 5).

The message handling is described in Lines 1–20. First, the message has to be read from the queue of stored messages (**receive**(`msg`)). After that, the process `AODV` checks the type of the message and calls a process that can handle the message: in case of a newly injected data packet, the process `NEWPKT` is called; in case of an incoming data packet, the process `PKT` is called; in case that the incoming message is an AODV control message (route request, route reply or route error), the node updates its routing table. More precisely, if there is no entry to the message's sender `sip`, the receiver-node creates an entry with the unknown sequence number 0 and hop count 1; in case there is already a routing table entry $(\mathtt{sip},\mathtt{dsn},*,*,*,*,\mathtt{pre})$, then this entry is updated to $(\mathtt{sip},\mathtt{dsn},\mathtt{unk},\mathtt{val},1,\mathtt{sip},\mathtt{pre})$ (cf. Lines 10, 14 and 18). Afterwards, the processes RREQ, RREP and RERR are called, respectively.

**Process 1** The basic routine

```
AODV(ip,sn,rt,rreqs,store) ≝
1.   receive(msg) .
2.   /* depending on the message, the node calls different processes */
3.   (
4.     [ msg = newpkt(data,dip) ]      /* new DATA packet */
5.       NEWPKT(data,dip , ip,sn,rt,rreqs,store)
6.     + [ msg = pkt(data,dip,oip) ]      /* incoming DATA packet */
7.       PKT(data,dip,oip , ip,sn,rt,rreqs,store)
8.     + [ msg = rreq(hops,rreqid,dip,dsn,dsk,oip,osn,sip) ]      /* RREQ */
9.       /* update the route to sip in rt */
10.      [[rt := update(rt,(sip,0,unk,val,1,sip,∅))]]      /* 0 is used since no sequence number is known */
11.      RREQ(hops,rreqid,dip,dsn,dsk,oip,osn,sip , ip,sn,rt,rreqs,store)
12.    + [ msg = rrep(hops,dip,dsn,oip,sip) ]      /* RREP */
13.      /* update the route to sip in rt */
14.      [[rt := update(rt,(sip,0,unk,val,1,sip,∅))]]
15.      RREP(hops,dip,dsn,oip,sip , ip,sn,rt,rreqs,store)
16.    + [ msg = rerr(dests,sip) ]      /* RERR */
17.      /* update the route to sip in rt */
18.      [[rt := update(rt,(sip,0,unk,val,1,sip,∅))]]
19.      RERR(dests,sip , ip,sn,rt,rreqs,store)
20.  )
21. + [ Let dip ∈ qD(store) ∩ vD(rt) ]      /* send a queued data packet if a valid route is known */
22.   [[data := head(σ_queue(store,dip))]]
23.   unicast(nhop(rt,dip),pkt(data,dip,ip)) .
24.     [[store := drop(dip,store)]]      /* drop data from the store for dip if the transmission was successful */
25.     AODV(ip,sn,rt,rreqs,store)
26.   ▶ /* an error is produced and the routing table is updated */
27.     [[dests := {(rip,inc(sqn(rt,rip))) | rip ∈ vD(rt) ∧ nhop(rt,rip) = nhop(rt,dip)}]]
28.     [[rt := invalidate(rt,dests)]]
29.     [[store := setRRF(store,dests)]]
30.     [[pre := ⋃{precs(rt,rip) | (rip,∗) ∈ dests}]]
31.     [[dests := {(rip,rsn) | (rip,rsn) ∈ dests ∧ precs(rt,rip) ≠ ∅}]]
32.     groupcast(pre,rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
33. + [ Let dip ∈ qD(store) − vD(rt) ∧ σ_p-flag(store,dip) = req ]      /* a route discovery process is initiated */
34.   [[store := unsetRRF(store,dip)]]      /* set request-required flag to no-req */
35.   [[sn := inc(sn)]]      /* increment own sequence number */
36.   /* update rreqs by adding (ip,nrreqid(rreqs,ip)) */
37.   [[rreqid := nrreqid(rreqs,ip)]]
38.   [[rreqs := rreqs ∪ {(ip,rreqid)}]]
39.   broadcast(rreq(0,rreqid,dip,sqn(rt,dip),sqnf(rt,dip),ip,sn,ip)) . AODV(ip,sn,rt,rreqs,store)
```

The second part of AODV (Lines 21–32) initiates the sending of a data packet. For that, it has to be checked if there is a queued data packet for a destination that has a known and valid route in the routing table ($\mathtt{qD}(\mathtt{store})\cap\mathtt{vD}(\mathtt{rt})\neq\emptyset$). In case that there is more than one destination with stored data and a known route, an arbitrary destination is chosen and denoted by dip (Line 21).[25] Moreover data is set to the first queued data packet from the application layer that should be sent ($\mathtt{data}:=\mathtt{head}(\sigma_{queue}(\mathtt{store},\mathtt{dip}))$).[26] This data packet is unicast to the next hop on the route to dip.

[25] Although the word "let" is not part of the syntax, we add it to stress the nondeterminism happening here.
[26] Following the RFC, data packets waiting for a route should be buffered "first-in, first-out" (FIFO).

If the unicast is successful, the data packet `data` is removed from `store` (Line 24). Finally, the process calls itself—stating that the node is ready for handling a new message, initiating the sending of another packet towards a destination, etc. In case the unicast is not successful, the data packet has not been transmitted. Therefore `data` is not removed from `store`. Moreover, the node knows that the link to the next hop on the route to `dip` is faulty and, most probably, broken. An error message is initiated. Generally, route error and link breakage processing requires the following steps: (a) invalidating existing routing table entries, (b) listing affected destinations, (c) determining which neighbours may be affected (if any), and (d) delivering an appropriate AODV error message to such neighbours [79]. Therefore, the process determines all valid destinations `dests` that have this unreachable node as next hop (Line 27) and marks the routing table entries for these destinations as invalid (Line 28), while incrementing their sequence numbers (Line 27). In Line 29, we set, for all invalidated routing table entries, the request-required flag to `req`, thereby indicating that a new route discovery process may need to be initiated. In Line 30 the recipients of the error message are determined. These are the precursors of the invalidated destinations, i.e., the neighbouring nodes listed as having a route to one of the affected destinations passing through the broken link. Finally, an error message is sent to them (Line 32), listing only those invalidated destinations with a non-empty set of precursors (Line 31).

The third and final part of AODV (Lines 33–39) initiates a route discovery process. This is done when there is at least one queued data packet for a destination without a valid routing table entry, that is not waiting for a reply in response to a route request process initiated before. Following the RFC, the process generates a new route request. This is achieved in four steps: First, the request-required flag is set to `no-req` (Line 34), meaning that no further route discovery processes for this destination need to be initiated.[27] Second, the node's own sequence number is increased by 1 (Line 35). Third, by determining `nrreqid(rreqs,ip)`, a new route request identifier is created and stored—together with the node's `ip`—in the set `rreqs` of route requests already seen (Line 38). Fourth, the message itself is sent (Line 39) using broadcast. In contrast to **unicast**, transmissions via **broadcast** are not checked on success. The information inside the message follows strictly the RFC. In particular, the hop count is set to 0, the route request identifier previously created is used, etc. This ends the initiation of the route discovery process.

## 6.2 Data Packet Handling

The processes NEWPKT and PKT describe all actions performed by a node when a data packet is injected by a client hooked up to the local node or received via the protocol, respectively. For the process PKT, this includes the acceptance (if the node is the destination), the forwarding (if the node is not the destination), as well as the sending of an error message in case something went wrong. The process NEWPKT does not include the initiation of a new route request; this is part of the process AODV. Although packet handling itself is not part of AODV, it is necessary to include it in our formalisation, since a failure to transmit a data packet triggers AODV activity.

The process NEWPKT first checks whether the node is the intended addressee of the data packet. If this is the case, it delivers the data and returns to the basic routine AODV. If the node is not the intended destination ($\mathtt{dip} \neq \mathtt{ip}$, Line 3), the `data` is added to the data queue for `dip` (Line 4),[28] which finishes the handling of a newly injected data packet. The further handling of queued data (forwarding it to the next hop on the way to the destination in case a valid route to the destination is known, and otherwise initiating a new route request if still required) is the responsibility of the main process AODV.

[27]The RFC does not describe packet handling in detail; hence the request-required flag is not part of the RFC's RREQ generation process.

[28]If no data for destination `dip` was already queued, the function `add` creates a fresh queue for `dip`, and set the request-required flag to `req`; otherwise, the request-required flag keeps the value it had already.

**Process 2** Routine for handling a newly injected data packet

```
NEWPKT(data,dip,ip,sn,rt,rreqs,store) =def
1.  [ dip = ip ]      /* the DATA packet is intended for this node */
2.    deliver(data) . AODV(ip,sn,rt,rreqs,store)
3.  + [ dip ≠ ip ]      /* the DATA packet is not intended for this node */
4.    [[store := add(data,dip,store)]] . AODV(ip,sn,rt,rreqs,store)
```

Similar to NEWPKT, the process PKT first checks whether it is the intended addressee of the data packet. If this is the case, it delivers the data and returns to the basic routine AODV. If the node is not the intended destination ($\texttt{dip} \neq \texttt{ip}$, Line 3) more activity is needed.

In case that the node has a valid route to the data's destination dip ($\texttt{dip} \in \texttt{vD}(\texttt{rt})$), it forwards the packet using a unicast to the next hop $\texttt{nhop}(\texttt{rt},\texttt{dip})$ on the way to dip. Similar to the unicast of the process AODV, it has to be checked whether the transmission is successful: no further action is necessary if the transmission succeeds, and the node returns to the basic routine AODV. If the transmission fails, the link to the next hop $\texttt{nhop}(\texttt{rt},\texttt{dip})$ is assumed to be broken. As before, all destinations dests that are reached via that broken link are determined (Line 9) and all precursors interested in at least one of these destinations are informed via an error message (Line 14). Moreover, all the routing table entries using the broken link have to be invalidated in the node's routing table rt (Line 10), and all corresponding request-required flags are set to req (Line 11).

In case that the node has no valid route to the destination dip ($\texttt{dip} \notin \texttt{vD}(\texttt{rt})$), the data packet is lost and possibly an error message is sent. If there is an (invalid) route to the data's destination dip in the routing table (Line 18), the possibly affected neighbours can be determined and the error message is sent to these precursors (Line 20). If there is no information about a route towards dip nothing happens (and the basic process AODV is called again).

**Process 3** Routine for handling a received data packet

```
PKT(data,dip,oip,ip,sn,rt,rreqs,store) =def
1.  [ dip = ip ]      /* the DATA packet is intended for this node */
2.    deliver(data) . AODV(ip,sn,rt,rreqs,store)
3.  + [ dip ≠ ip ]      /* the DATA packet is not intended for this node */
4.    (
5.      [ dip ∈ vD(rt) ]      /* valid route to dip */
6.        /* forward packet */
7.        unicast(nhop(rt,dip),pkt(data,dip,oip)) . AODV(ip,sn,rt,rreqs,store)
8.        ▶ /* If the packet transmission is unsuccessful, a RERR message is generated */
9.          [[dests := {(rip,inc(sqn(rt,rip)))|rip ∈ vD(rt) ∧ nhop(rt,rip) = nhop(rt,dip)}]]
10.         [[rt := invalidate(rt,dests)]]
11.         [[store := setRRF(store,dests)]]
12.         [[pre := ⋃{precs(rt,rip)|(rip,∗) ∈ dests}]]
13.         [[dests := {(rip,rsn)|(rip,rsn) ∈ dests ∧ precs(rt,rip) ≠ ∅}]]
14.         groupcast(pre,rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
15.     + [ dip ∉ vD(rt) ]      /* no valid route to dip */
16.       /* no local repair occurs; data is lost */
17.       (
18.         [ dip ∈ iD(rt) ]      /* invalid route to dip */
19.           /* if the route is invalid, a RERR is sent to the precursors */
20.           groupcast(precs(rt,dip),rerr({(dip,sqn(rt,dip))},ip)) . AODV(ip,sn,rt,rreqs,store)
21.         + [ dip ∉ iD(rt) ]      /* route not in rt */
22.           AODV(ip,sn,rt,rreqs,store)
23.       )
24.   )
```

### 6.3 Receiving Route Requests

The process RREQ models all events that may occur after a route request has been received.

The process first reads the unique identifier (oip, rreqid) of the route request received. If this pair is already stored in the node's data rreqs, the route request has been handled before and the message can silently be ignored (Lines 1–2).

**Process 4** RREQ handling

```
RREQ(hops,rreqid,dip,dsn,dsk,oip,osn,sip,ip,sn,rt,rreqs,store) =def
1.  [ (oip,rreqid) ∈ rreqs ]     /* the RREQ has been received previously */
2.    AODV(ip,sn,rt,rreqs,store)     /* silently ignore RREQ, i.e. do nothing */
3.  + [ (oip,rreqid) ∉ rreqs ]     /* the RREQ is new to this node */
4.    [[rt := update(rt,(oip,osn,kno,val,hops+1,sip,∅))]]     /* update the route to oip in rt */
5.    [[rreqs := rreqs ∪ {(oip,rreqid)}]]     /* update rreqs by adding (oip,rreqid) */
6.    (
7.      [ dip = ip ]     /* this node is the destination node */
8.        [[sn := max(sn,dsn)]]     /* update the sqn of ip */
9.        /* unicast a RREP towards oip of the RREQ */
10.       unicast(nhop(rt,oip),rrep(0,dip,sn,oip,ip)) . AODV(ip,sn,rt,rreqs,store)
11.       ▶ /* If the transmission is unsuccessful, a RERR message is generated */
12.         [[dests := {(rip,inc(sqn(rt,rip)))|rip ∈ vD(rt) ∧ nhop(rt,rip) = nhop(rt,oip)}]]
13.         [[rt := invalidate(rt,dests)]]
14.         [[store := setRRF(store,dests)]]
15.         [[pre := ⋃{precs(rt,rip)|(rip,∗) ∈ dests}]]
16.         [[dests := {(rip,rsn)|(rip,rsn) ∈ dests ∧ precs(rt,rip) ≠ ∅}]]
17.         groupcast(pre,rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
18.     + [ dip ≠ ip ]     /* this node is not the destination node */
19.       (
20.         [ dip ∈ vD(rt) ∧ dsn ≤ sqn(rt,dip) ∧ sqnf(rt,dip) = kno ]     /* valid route to dip that is fresh enough */
21.           /* update rt by adding precursors */
22.           [[rt := addpreRT(rt,dip,{sip})]]
23.           [[rt := addpreRT(rt,oip,{nhop(rt,dip)})]]
24.           /* unicast a RREP towards the oip of the RREQ */
25.           unicast(nhop(rt,oip),rrep(dhops(rt,dip),dip,sqn(rt,dip),oip,ip)) .
26.             AODV(ip,sn,rt,rreqs,store)
27.           ▶ /* If the transmission is unsuccessful, a RERR message is generated */
28.             [[dests := {(rip,inc(sqn(rt,rip)))|rip ∈ vD(rt) ∧ nhop(rt,rip) = nhop(rt,oip)}]]
29.             [[rt := invalidate(rt,dests)]]
30.             [[store := setRRF(store,dests)]]
31.             [[pre := ⋃{precs(rt,rip)|(rip,∗) ∈ dests}]]
32.             [[dests := {(rip,rsn)|(rip,rsn) ∈ dests ∧ precs(rt,rip) ≠ ∅}]]
33.             groupcast(pre,rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
34.       + [ dip ∉ vD(rt) ∨ sqn(rt,dip) < dsn ∨ sqnf(rt,dip) = unk ]     /* no valid route that is fresh enough */
35.           /* no further update of rt */
36.           broadcast(rreq(hops+1,rreqid,dip,max(sqn(rt,dip),dsn),dsk,oip,osn,ip)) .
37.           AODV(ip,sn,rt,rreqs,store)
38.       )
39.   )
```

If the received message is new to this node ((oip, rreqid) $\notin$ rreqs, Line 3), the node establishes a route of length hops+1 back to the originator oip of the message. If this route is "better" than the route to oip in the current routing table, the routing table is updated by this route (Line 4). Moreover the unique identifier has to be added to the set rreqs of already seen (and handled) route requests (Line 5).

After these updates the process checks if the node is the intended destination (dip = ip, Line 7). In that case, a route reply must be initiated: first, the node's sequence number is—according to the RFC—set to the maximum of the current sequence number and the destination sequence number in the RREQ packet (Line 8).[29] Then the reply is unicast to the next hop on the route back to the originator oip of the route request. The content of the new route reply is as follows: the hop count is set to 0, the destination and originator are copied from the route request received and the destination's sequence number is the node's own sequence number sn; of course the sender's IP of this message has to be set to the node's ip. As before (cf. Sections 6.1 and 6.2), the process invalidates the corresponding routing table entries, sets request-required flags and sends an error message to all relevant precursors if the unicast transmission fails (Lines 12–17).

If the node is not the destination dip of the message but an intermediate hop along the path from the originator to the destination, it is allowed to generate a route reply only if the information in its own routing table is fresh enough. This means that (a) the node has a valid route to the destination, (b) the destination sequence number in the node's existing routing table entry for the destination ($\mathtt{sqn}(\mathtt{rt},\mathtt{dip})$) is greater than or equal to the requested destination sequence number dsn of the message and (c) the sequence number $\mathtt{sqn}(\mathtt{rt},\mathtt{dip})$ is known, i.e., $\mathtt{sqnf}(\mathtt{rt},\mathtt{dip}) = \mathtt{kno}$. If these three conditions are satisfied—the check is done in Line 20—the node generates a new route reply and sends it to the next hop on the way back to the originator oip of the received route request.[30]. To this end, it copies the sequence number for the destination dip from the routing table rt into the destination sequence number field of the RREP message and it places its distance in hops from the destination ($\mathtt{dhops}(\mathtt{rt},\mathtt{dip})$) in the corresponding field of the new reply (Line 25). The unicast might fail, which causes the usual error handling (Lines 28–33). Just before transmitting the unicast, the intermediate node updates the forward route entry to dip by placing the last hop node (sip)[31] into the precursor list for the forward route entry (Line 22). Likewise, it updates the reverse route entry to oip by placing the first hop $\mathtt{nhop}(\mathtt{rt},\mathtt{dip})$ towards dip in the precursor list for that entry (Line 23).[32]

If the node is not the destination and there is either no route to the destination dip inside the routing table or the route is not fresh enough, the route request received has to be forwarded. This happens in Line 36. The information inside the forwarded request is mostly copied from the request received. Only the hop count is increased by 1 and the destination sequence number is set to the maximum of the destination sequence number in the RREQ packet and the current sequence number for dip in the routing table. In case dip is an unknown destination, $\mathtt{sqn}(\mathtt{rt},\mathtt{dip})$ returns the unknown sequence number 0.

## 6.4 Receiving Route Replies

The process RREP describes the reaction of the protocol to an incoming route reply. Our model first checks if a forward routing table entry is going to be created or updated (Line 1). This is the case if (a) the node has no known route to the destination, or (b) the destination sequence number in the node's existing routing table entry for the destination ($\mathtt{sqn}(\mathtt{rt},\mathtt{dip})$) is smaller than the destination sequence number dsn in the RREP message, or (c) the two destination sequence numbers are equal and, in addition, either the incremented hop count of the RREP received is strictly smaller than the one in the routing table, or the entry for dip in the routing table is invalid. Hence Line 1 could be replaced by

$$[\mathtt{dip} \notin \mathtt{kD}(\mathtt{rt}) \vee \mathtt{sqn}(\mathtt{rt},\mathtt{dip}) < \mathtt{dsn} \vee (\mathtt{sqn}(\mathtt{rt},\mathtt{dip}) = \mathtt{dsn} \wedge (\mathtt{dhops}(\mathtt{rt},\mathtt{dip}) > \mathtt{hops}+1 \vee \mathtt{flag}(\mathtt{rt},\mathtt{dip}) = \mathtt{inv}))\,]\,.^{33}$$

[29] According to I. Chakeres on the IETF MANET mailing list (http://www.ietf.org/mail-archive/web/manet/current/msg02589.html) Line 8 ought to be replaced by $[\![\mathtt{sn} := \max(\mathtt{sn},\mathtt{inc}(\mathtt{dsn}))]\!]$.

[30] This next hop will often, but not always, be sip; see Figure 3 in Section 2.

[31] This is a mistake in the RFC; it should have been $\mathtt{nhop}(\mathtt{rt},\mathtt{oip})$.

[32] Unless the *gratuitous RREP flag* is set, which we do not model in this paper, this update is rather useless, as the precursor $\mathtt{nhop}(\mathtt{rt},\mathtt{dip})$ in general is not aware that it has a route to oip.

[33] In case $\mathtt{dip} \notin \mathtt{kD}(\mathtt{rt})$, the terms $\mathtt{dhops}(\mathtt{rt},\mathtt{dip})$ and $\mathtt{flag}(\mathtt{rt},\mathtt{dip})$ are not defined. In such a case, according to the convention of Footnote 14 in Section 4, the atomic formulas $\mathtt{dhops}(\mathtt{rt},\mathtt{dip}) > \mathtt{hops}+1$ and $\mathtt{flag}(\mathtt{rt},\mathtt{dip}) = \mathtt{inv}$ evaluate to false. However, in case one would use lazy evaluation of the outermost disjunction, the evaluation of the expression would be independent of the choice of a convention for interpreting undefined terms appearing in formulas.

**Process 5** RREP handling

```
RREP(hops,dip,dsn,oip,sip,ip,sn,rt,rreqs,store) ≝
1.  [ rt ≠ update(rt,(dip,dsn,kno,val,hops+1,sip,∅)) ]      /* the routing table has to be updated */
2.    ⟦rt := update(rt,(dip,dsn,kno,val,hops+1,sip,∅))⟧
3.    (
4.      [ oip = ip ]      /* this node is the originator of the corresponding RREQ */
5.        /* a packet may now be sent; this is done in the process AODV */
6.        AODV(ip,sn,rt,rreqs,store)
7.      + [ oip ≠ ip ]      /* this node is not the originator; forward RREP */
8.        (
9.          [ oip ∈ vD(rt) ]      /* valid route to oip */
10.           /* add next hop towards oip as precursor and forward the route reply */
11.           ⟦rt := addpreRT(rt,dip,{nhop(rt,oip)})⟧
12.           ⟦rt := addpreRT(rt,nhop(rt,dip),{nhop(rt,oip)})⟧
13.           unicast(nhop(rt,oip),rrep(hops+1,dip,dsn,oip,ip)) .
14.             AODV(ip,sn,rt,rreqs,store)
15.           ▶ /* If the transmission is unsuccessful, a RERR message is generated */
16.             ⟦dests := {(rip,inc(sqn(rt,rip)))|rip ∈ vD(rt) ∧ nhop(rt,rip) = nhop(rt,oip)}⟧
17.             ⟦rt := invalidate(rt,dests)⟧
18.             ⟦store := setRRF(store,dests)⟧
19.             ⟦pre := ⋃{precs(rt,rip)|(rip,∗) ∈ dests}⟧
20.             ⟦dests := {(rip,rsn)|(rip,rsn) ∈ dests ∧ precs(rt,rip) ≠ ∅}⟧
21.             groupcast(pre,rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
22.         + [ oip ∉ vD(rt) ]      /* no valid route to oip */
23.           AODV(ip,sn,rt,rreqs,store)
24.       )
25.   )
26. + [ rt = update(rt,(dip,dsn,kno,val,hops+1,sip,∅)) ]      /* the routing table is not updated */
27.   AODV(ip,sn,rt,rreqs,store)
```

In case that one of these conditions is true, the routing table is updated in Line 2. If the node is the intended addressee of the route reply ($\mathtt{oip} = \mathtt{ip}$) the protocol returns to its basic process AODV. Otherwise ($\mathtt{oip} \neq \mathtt{ip}$) the message should be forwarded. Following the RFC [79], “If the current node is not the node indicated by the Originator IP Address in the RREP message AND a forward route has been created or updated [...], the node consults its route table entry for the originating node to determine the next hop for the RREP packet, and then forwards the RREP towards the originator using the information in that route table entry.” This action needs a valid route to the originator oip of the route request to which the current message is a reply ($\mathtt{oip} \in \mathtt{vD}(\mathtt{rt})$, Line 9). The content of the RREP message to be sent is mostly copied from the RREP received; only the sender has to be changed (it is now the node’s ip) and the hop count is incremented. Prior to the unicast, the node $\mathtt{nhop}(\mathtt{rt},\mathtt{oip})$, to which the message is sent, is added to the list of precursors for the routes to dip (Line 11) and to the next hop on the route to dip (Line 12). Although not specified in the RFC, it would make sense to also add a precursor to the reverse route by $[\![\mathtt{rt} := \mathtt{addpreRT}(\mathtt{rt},\mathtt{oip},\{\mathtt{nhop}(\mathtt{rt},\mathtt{dip})\})]\!]$. As usual, if the unicast fails, the affected routing table entries are invalidated and the precursors of all routes using the broken link are determined and an error message is sent (Lines 16–21). In the unlikely situation that a reply should be forwarded but no valid route is known by the node, nothing happens. Following the RFC, no precursor has to be notified and no error message has to be sent—even if there is an invalid route.

If a forward routing table entry is not created nor updated, the reply is silently ignored and the basic process is called (Lines 26–27).

### 6.5 Receiving Route Errors

The process RERR models the part of AODV which handles error messages. An error message consists of a set dests of pairs of an unreachable destination IP address rip and the corresponding unreachable destination sequence number rsn.

**Process 6** RERR handling

```
RERR(dests,sip,ip,sn,rt,rreqs,store) ≝
1. /* invalidate broken routes */
2. [[dests := {(rip,rsn) | (rip,rsn) ∈ dests ∧ rip ∈ vD(rt) ∧ nhop(rt,rip) = sip ∧ sqn(rt,rip) < rsn}]]
3. [[rt := invalidate(rt,dests)]]
4. [[store := setRRF(store,dests)]]
5. /* forward the RERR to all precursors for rt entries for broken connections */
6. [[pre := ⋃{precs(rt,rip) | (rip,∗) ∈ dests}]]
7. [[dests := {(rip,rsn) | (rip,rsn) ∈ dests ∧ precs(rt,rip) ≠ ∅}]]
8. groupcast(pre,rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
```

If a node receives an AODV error message from a neighbour for one or more valid routes, it has—under some conditions—to invalidate the entries for those routes in its own routing table and forward the error message. The node compares the set dests of unavailable destinations from the incoming error message with its own entries in the routing table. If the routing table lists a valid route with a $(\mathtt{rip},\mathtt{rsn})$-combination from dests and if the next hop on this route is the sender sip of the error message, this entry may be affected by the error message. In our formalisation, we have added the requirement $\mathtt{sqn}(\mathtt{rt},\mathtt{rip}) < \mathtt{rsn}$, saying that the entry is affected by the error message only if the "incoming" sequence number is larger than the one stored in the routing table, meaning that it is based on fresher information.[34] In this case, the entry has to be invalidated and all precursors of this particular route have to be informed. This has to be done for all affected routes.

In fact, the process first determines all $(\mathtt{rip},\mathtt{rsn})$-pairs that have effects on its own routing table and that may have to be forwarded as content of a new error message (Line 2). After that, all entries to unavailable destinations are invalidated (Line 3), and as usual when routing table entries are invalidated, the request-required flags are set to req (Line 4). In Line 6 the set of all precursors (affected neighbours) of the unavailable destinations are summarised in the set pre. Then, the set dests is "thinned out" to only those destinations that have at least one precursor— only these destinations are transmitted in the forwarded error message (Line 7). Finally, the message is sent (Line 8).

### 6.6 The Message Queue and Synchronisation

We assume that any message sent by a node *sip* to a node *ip* that happens to be within transmission range of *sip* is actually received by *ip*. For this reason, *ip* should always be able to perform a receive action, regardless of which state it is in. However, the main process AODV that runs on the node *ip* can reach a state, such as PKT, RREQ, RREP or RERR, in which it is not ready to perform a receive action. For this reason we introduce a process QMSG, modelling a message queue, that runs in parallel with AODV or any other process that might be called. Every incoming message is first stored in this queue, and piped from there to the process AODV, whenever AODV is ready to handle a new message. The process QMSG is always ready to receive a new message, even when AODV is not. The whole parallel process running on a node is then given by an expression of the form

$$(\xi,\mathtt{AODV}(\mathtt{ip},\mathtt{sn},\mathtt{rt},\mathtt{rreqs},\mathtt{store}))\ \langle\!\langle\ (\zeta,\mathtt{QMSG}(\mathtt{msgs}))\ .$$

[34]This additional requirement is in the spirit of Section 6.2 of the RFC [79] on updating routing table entries, but in contradiction with Section 6.11 of the RFC on handling RERR messages. In Section 8 we will show that the reading of Section 6.11 of the RFC gives rise to routing loops.

**Process 7** Message queue

```
QMSG(msgs) ≝
    /* store incoming message at the end of msgs */
    receive(msg) . QMSG(append(msg,msgs))
  + [ msgs ≠ [] ]      /* the queue is not empty */
    (
      /* pop top message and send it to another sequential process */
      send(head(msgs)) . QMSG(tail(msgs))
      /* or receive and store an incoming message */
      + receive(msg) . QMSG(append(msg,msgs))
    )
```

### 6.7 Initial State

To finish our specification, we have to define an initial state. The initial network expression is an encapsulated parallel composition of node expressions $ip : P : R$, where the (finite) number of nodes and the range $R$ of each node expression is left unspecified (can be anything). However, each node in the parallel composition is required to have a unique IP address $ip$. The initial process $P$ of $ip$ is given by the expression $(\xi, \mathtt{AODV(ip,sn,rt,rreqs,store)}) \langle\langle\ (\zeta, \mathtt{QMSG(msgs)})$, with

$$\xi(\mathtt{ip}) = ip \wedge \xi(\mathtt{sn}) = 1 \wedge \xi(\mathtt{rt}) = \emptyset \wedge \xi(\mathtt{rreqs}) = \emptyset \wedge \xi(\mathtt{store}) = \emptyset \wedge \zeta(\mathtt{msgs}) = [\,] . \quad (2)$$

This says that initially each node is correctly informed about its own identity; its own sequence number is initialised with 1 and its routing table, the list of RREQs seen, the store of queued data packets as well as the message queue are empty.

## 7 Invariants

Using our process algebra for wireless mesh networks and the proposed model of AODV we can now formalise and prove crucial properties of AODV. In this section we verify properties that can be expressed as invariants, i.e., statements that hold all the time when the protocol is executed.

The most important invariant we establish is *loop freedom*; most prior results can be regarded as stepping stones towards this goal. Next to that we also formalise and discuss *route correctness*.

### 7.1 State and Transition Invariants

A *(state) invariant* is a statement that holds for all reachable states of our model. Here states are network expressions, as defined in Section 4.3. An invariant is usually verified by showing that it holds for all possible initial states, and that, for any transition $N \xrightarrow{\ell} N'$ between (encapsulated) network expressions derived by our operational semantics, if it holds for state $N$ then it also holds for state $N'$.

Besides (state) invariants, we also establish statements we call *transition invariants*. A transition invariant is a statement that holds for each reachable transition $N \xrightarrow{\ell} N'$ between (encapsulated) network expressions derived by the operational semantics (Table 4). In establishing a transition invariant for a particular transition, we usually assume it has already been obtained for all *prior transitions*, those that occurred beforehand. Since the transition system generated by our operational semantics may have cycles, we need to give a well-founded definition of "beforehand". To this end we treat a statement about a transition as one about a *transition occurrence*, defined as a path in our transition system, stating in an initial state, and ending with the transition under consideration. This way the induction is performed on the length of such a path. We speak of *induction on reachability*.

To facilitate formalising transition invariants, we present a taxonomy of the transitions that can be generated by our operational semantics, along with some notation: the label $\ell$ of a transition $N \xrightarrow{\ell} N'$ can be either **connect**$(ip,ip')$, **disconnect**$(ip,ip')$, $ip$:**newpkt**$(d,dip)$, $ip$:**deliver**$(d)$ or $\tau$. We are most interested in the last case. A transition $N \xrightarrow{\tau} N'$ either arises from a transition $R$:***cast**$(m)$ performed by a network node $ip$, synchronising with receive actions of all nodes $dip \in R$ in transmission range, or stems from a $\tau$-transition of a network node $ip$.

In the former case, we write $N \xrightarrow{R:\textbf{*cast}(m)}_{ip} N'$. This means that $N = [M]$ and $N' = [M']$ are network expressions such that $M \xrightarrow{R:\textbf{*cast}(m)} M'$, and the cast action is performed by node $ip$. This transition originates from an action **broadcast**$(ms)$, **groupcast**$(dests, ms)$, or **unicast**$(dest, ms)$ (cf. Section 4). Each such action can be identified by a line number in one of the processes of Section 6.

In the latter case, a $\tau$-transition of a node $ip$ stems either from a failed unicast, an evaluation $[\varphi]$, an assignment $[\![\texttt{var} := exp]\!]$, or a synchronisation of two actions **send**$(ms)$ and **receive**$(\texttt{msg})$ performed by sequential processes running on that node. In our model these processes are AODV and QMSG, and these actions can also be identified by line numbers in the processes of Section 6.

The following observations are crucial in establishing many of our invariants.

**Proposition 7.1**

(a) With the exception of new packets that are submitted to a node by a client of AODV, every message received and handled by the main routine of AODV has to be sent by some node before. More formally, we consider an arbitrary path $N_0 \xrightarrow{\ell_1} N_1 \xrightarrow{\ell_2} \ldots \xrightarrow{\ell_k} N_k$ with $N_0$ an initial state in our model of AODV. If the transition $N_{k-1} \xrightarrow{\ell_k} N_k$ results from a synchronisation involving the action **receive**$(\texttt{msg})$ from Line 1 of Pro. 1—performed by the node $ip$—, where the variable msg is assigned the value $m$, then either $m = \texttt{newpkt}(d,dip)$ or one of the $\ell_i$ with $i < k$ stems from an action ***cast**$(m)$ of a node $ip'$ of the network.

(b) No node can receive a message directly from itself. Using the formalisation above, we must have $ip \neq ip'$.

*Proof.* The only way Line 1 of Pro. 1 can be executed, is through a synchronisation of the main process AODV with the message queue QMSG (Pro. 6) running on the same node. This involves the action **send**$(m)$ of QMSG. Here $m$ is popped from the message queue msgs, which started out empty. So at some point QMSG must have performed the action **receive**$(m)$. However, this action is blocked by the encapsulation operator $[\_]$ of Table 4, except when $m$ has the form $\texttt{newpkt}(d, dip)$ or when it synchronises with an action ***cast**$(m)$ of another node. □

At first glance Part(b) does not seem to reflect reality. Of course, an application running on a local node has to be able to send data packets to another application running on the same node. However, in any practical implementation, when a node sends a message to itself, the message will be delivered to the corresponding application on the local node without ever being “seen” by AODV or any other routing protocol. Therefore, from AODV’s perspective, no node can receive a message (directly) from itself.

## 7.2 Notions and Notations

Before formalising and proving invariants, we introduce some useful notions and notations.

All processes except QMSG maintain the five data variables ip, sn, rt, rreqs and store. Next to that QMSG maintains the variable msgs. Hence, these 6 variables can be evaluated at any time. Moreover, every node expression in the transition system looks like

$$ip : (\xi, P \ \langle\langle \ \zeta, \texttt{QMSG}(\texttt{msgs})) : R \ ,$$

where $P$ is a state in one of the following sequential processes:

$\texttt{AODV}(\texttt{ip},\texttt{sn},\texttt{rt},\texttt{rreqs},\texttt{store})$,
$\texttt{NEWPKT}(\texttt{data},\texttt{dip},\texttt{ip},\texttt{sn},\texttt{rt},\texttt{rreqs},\texttt{store})$,
$\texttt{PKT}(\texttt{data},\texttt{dip},\texttt{oip},\texttt{ip},\texttt{sn},\texttt{rt},\texttt{rreqs},\texttt{store})$,
$\texttt{RREQ}(\texttt{hops},\texttt{rreqid},\texttt{dip},\texttt{dsn},\texttt{dsk},\texttt{oip},\texttt{osn},\texttt{sip},\texttt{ip},\texttt{sn},\texttt{rt},\texttt{rreqs},\texttt{store})$,
$\texttt{RREP}(\texttt{hops},\texttt{dip},\texttt{dsn},\texttt{oip},\texttt{sip},\texttt{ip},\texttt{sn},\texttt{rt},\texttt{rreqs},\texttt{store})$ or
$\texttt{RERR}(\texttt{dests},\texttt{sip},\texttt{ip},\texttt{sn},\texttt{rt},\texttt{rreqs},\texttt{store})$.

Hence the state of the transition system for a node *ip* is determined by the process $P$, the range $R$, and the two valuations $\xi$ and $\zeta$. If a network consists of a (finite) set $\mathbf{IP} \subseteq \texttt{IP}$ of nodes, a reachable network expression $N$ is an encapsulated parallel composition of node expressions—one for each $ip \in \mathbf{IP}$. In this section, we assume $N$ and $N'$ to be reachable network expressions in our model of AODV. To distill current information about a node from $N$, we define the following projections:

$P_N^{ip} := P$, where $ip : (*, P \langle\langle *, *) : *$ is a node expression of $N$,
$R_N^{ip} := R$, where $ip : (*, * \langle\langle *, *) : R$ is a node expression of $N$,
$\xi_N^{ip} := \xi$, where $ip : (\xi, * \langle\langle *, *) : *$ is a node expression of $N$,
$\zeta_N^{ip} := \zeta$, where $ip : (*, * \langle\langle \zeta, *) : *$ is a node expression of $N$.

For example, $P_N^{ip}$ determines the sequential process the node is currently working in, $R_N^{ip}$ denotes the set of all nodes currently within transmission range of *ip*, and $\xi_N^{ip}(\texttt{rt})$ evaluates the current routing table maintained by node *ip* in the network expression $N$. In the forthcoming proofs, when discussing the effects of an action, identified by a line number in one of the processes of our model, $\xi$ denotes the current valuation $\xi_N^{ip}$, where *ip* is the address of the local node, executing the action under consideration, and $N$ is the network expression obtained right before this action occurs, corresponding with the line number under consideration. When consider the effects of several actions, corresponding to several line numbers, $\xi$ is always interpreted most locally. For instance, in the proof of Proposition 7.14(a), case **Pro. 4, Line 36**, we write

> Hence ... $ip_c := \xi(\texttt{ip}) = ip$ and $\xi_N^{ip_c} = \xi$ (by (3)). At Line 4 we update the routing table using $r := \xi(\texttt{oip},\texttt{osn},\texttt{kno},\texttt{val},\texttt{hops}{+}1,\texttt{sip},\emptyset)$ as new entry. The routing table does not change between Lines 4 and 36; nor do the values of the variables $\texttt{hops}$, $\texttt{oip}$ and $\texttt{osn}$.

Writing $N_k$ for a network expression in which the local node *ip* is about to execute Line $k$, this passage can be reworded as

> Hence ... $ip_c := \xi_{N_{36}}^{ip}(\texttt{ip}) = ip$ and $\xi_{N_{36}}^{ip_c} = \xi_{N_{36}}^{ip}$ (by (3)).
> $\xi_{N_5}^{ip}(\texttt{rt}) := \xi_{N_4}^{ip}(\texttt{update}(\texttt{rt},(\texttt{oip},\texttt{osn},\texttt{kno},\texttt{val},\texttt{hops}{+}1,\texttt{sip},\emptyset)))$
> $\qquad := \texttt{update}(\xi_{N_4}^{ip}(\texttt{rt}),(\xi_{N_4}^{ip}(\texttt{oip}),\xi_{N_4}^{ip}(\texttt{osn}),\ldots))$.
> $\xi_{N_5}^{ip}(\texttt{rt}) = \xi_{N_{36}}^{ip}(\texttt{rt}) \wedge \xi_{N_4}^{ip}(\texttt{hops}) = \xi_{N_{36}}^{ip}(\texttt{hops}) \wedge \xi_{N_4}^{ip}(\texttt{oip}) = \xi_{N_{36}}^{ip}(\texttt{oip}) \wedge \xi_{N_4}^{ip}(\texttt{osn}) = \xi_{N_{36}}^{ip}(\texttt{osn})$.

In all of case **Pro. 4, Line 36**, through the statement of the proposition, $N$ is bound to $N_{36}$, so that $\xi_N^{ip} = \xi_{N_{36}}^{ip}$.

In Section 5.4 we have defined functions that work on evaluated routing tables $\xi_N^{ip}(\texttt{rt})$, such as $\texttt{nhop}$. To ease readability, we abbreviate $\texttt{nhop}(\xi_N^{ip}(\texttt{rt}),dip)$ by $\texttt{nhop}_N^{ip}(dip)$. Similarly, we use $\texttt{sqn}_N^{ip}(dip)$, $\texttt{dhops}_N^{ip}(dip)$, $\texttt{flag}_N^{ip}(dip)$, $\sigma_{route}{}_N^{ip}(dip)$, $\texttt{kD}_N^{ip}$, $\texttt{vD}_N^{ip}$ and $\texttt{iD}_N^{ip}$ for $\texttt{sqn}(\xi_N^{ip}(\texttt{rt}),dip)$, $\texttt{dhops}(\xi_N^{ip}(\texttt{rt}),dip)$, $\texttt{flag}(\xi_N^{ip}(\texttt{rt}),dip)$, $\sigma_{route}(\xi_N^{ip}(\texttt{rt}),ip)$, $\texttt{kD}(\xi_N^{ip}(\texttt{rt}))$, $\texttt{vD}(\xi_N^{ip}(\texttt{rt}))$ and $\texttt{iD}(\xi_N^{ip}(\texttt{rt}))$, respectively.

### 7.3 Basic Properties

In this section we show some of the most fundamental invariants for AODV. The first one is already stated in the RFC [79, Sect. 3].

**Proposition 7.2** Each sequence number of any given node $ip$ increases monotonically, i.e., never decreases, and is never unknown. That is, for $ip \in \mathbf{IP}$, if $N \xrightarrow{\ell} N'$ then $1 \leq \xi_N^{ip}(\mathtt{sn}) \leq \xi_{N'}^{ip}(\mathtt{sn})$.

*Proof.* In all initial states the invariant is satisfied, as all sequence numbers of all nodes are set to 1 (see (2) in Section 6.7). The Processes 1–7 of Section 6 change a node's sequence number only through the functions `inc` and max. This occurs at two places only:

**Pro. 1, Line 35:** Here $\xi_N^{ip}(\mathtt{sn}) \leq \mathtt{inc}(\xi_N^{ip}(\mathtt{sn})) = \xi_{N'}^{ip}(\mathtt{sn})$.

**Pro. 4, Line 8:** Here $\xi_N^{ip}(\mathtt{sn}) \leq \max(\xi_N^{ip}(\mathtt{sn}), *) = \xi_{N'}^{ip}(\mathtt{sn})$.

From this and the fact that all sequence numbers are initialised with 1 we get $1 \leq \xi_N^{ip}(\mathtt{sn})$. □

The proof strategy used above can be generalised.

**Remark 7.3** *Most of the forthcoming proofs can be done by showing the statement for each initial state and then checking all locations in the processes where the validity of the invariant is possibly changed. Note that routing table entries are only changed by the functions* `update`*,* `invalidate` *or* `addpreRT`*. Thus we have to show that an invariant dealing with routing tables is satisfied after the execution of these functions if it was valid before. In our proofs, we go through all occurrences of these functions. In case the invariant does not make statements about precursors, the function* `addpreRT` *need not be considered.*

**Proposition 7.4** The set of known destinations of a node increases monotonically. That is, for $ip \in \mathbf{IP}$, if $N \xrightarrow{\ell} N'$ then $\mathtt{kD}_N^{ip} \subseteq \mathtt{kD}_{N'}^{ip}$.

*Proof.* None of the functions used to change routing tables removes an entry altogether. □

**Proposition 7.5** The set of already seen route requests of a node increases monotonically. That is, for $ip \in \mathbf{IP}$, if $N \xrightarrow{\ell} N'$ then $\xi_N^{ip}(\mathtt{rreqs}) \subseteq \xi_{N'}^{ip}(\mathtt{rreqs})$.

*Proof.* None of the functions used in the specification ever removes an entry from `rreqs`. □

**Proposition 7.6** In each node's routing table, the sequence number for any given destination increases monotonically, i.e., never decreases. That is, for $ip, dip \in \mathbf{IP}$, if $N \xrightarrow{\ell} N'$ then $\mathtt{sqn}_N^{ip}(dip) \leq \mathtt{sqn}_{N'}^{ip}(dip)$.

*Proof.* The only function that can decrease a sequence number is `invalidate`. When invalidating routing table entries using the function `invalidate(rt,dests)`, sequence numbers are copied from `dests` to the corresponding entry in `rt`. It is sufficient to show that for all $(rip, rsn) \in \xi_N^{ip}(\mathtt{dests})$ $\mathtt{sqn}_N^{ip}(rip) \leq rsn$, as all other sequence numbers in routing table entries remain unchanged.

**Pro. 1, Line 28; Pro. 3, Line 10; Pro. 4, Lines 13, 29; Pro. 5, Line 17:**
The set `dests` is constructed immediately before the invalidation procedure. For $(rip, rsn) \in \xi_N^{ip}(\mathtt{dests})$, we have $\mathtt{sqn}_N^{ip}(rip) \leq \mathtt{inc}(\mathtt{sqn}_N^{ip}(rip)) = rsn$.

**Pro. 6, Line 3:** When constructing `dests` in Line 2, the side condition $\xi_{N_2}^{ip}(\mathtt{sqn}(\mathtt{rt},\mathtt{rip})) < \xi_{N_2}^{ip}(\mathtt{rsn})$ is taken into account, which immediately yields the claim for $(rip, rsn) \in \xi_N^{ip}(\mathtt{dests})$. □

Our next invariant tells that each node is correctly informed about its own identity.

**Proposition 7.7** For each $ip \in \mathbf{IP}$ and each reachable state $N$ we have $\xi_N^{ip}(\mathtt{ip}) = ip$.

*Proof.* According to Section 6.7 the claim is assumed to hold for each initial state, and none of our processes has an assignment changing the value of the variable `ip`. □

This proposition will be used implicitly in many of the proofs to follow. In particular, for all $ip', ip'' \in \mathbf{IP}$

$$\xi_N^{ip'}(\mathtt{ip}) = ip'' \;\Rightarrow\; ip' = ip'' \;\wedge\; \xi_N^{ip'} = \xi_N^{ip''} \; . \tag{3}$$

Next, we show that every AODV control message contains the IP address of the sender.

**Proposition 7.8** If an AODV control message is sent by node $ip \in \mathbf{IP}$, the node sending this message identifies itself correctly:

$$N \xrightarrow{R:*\mathbf{cast}(m)}_{ip} N' \;\Rightarrow\; ip = ip_c\ ,$$

where the message $m$ is either $\texttt{rreq}(*,*,*,*,*,*,*,ip_c)$, $\texttt{rrep}(*,*,*,*,ip_c)$, or $\texttt{rerr}(*,ip_c)$.

The proof is straightforward: whenever such a message is sent in one of the processes of Section 6, $\xi(\texttt{ip})$ is set as the last argument. □

**Corollary 7.9** At no point will the variable $\texttt{sip}$ maintained by node $ip$ have the value $ip$.

$$\xi_N^{ip}(\texttt{sip}) \neq ip$$

*Proof.* The value of $\texttt{sip}$ stems, through Lines 8, 12 or 16 of Pro. 1, from an incoming AODV control message of the form $\xi_N^{ip}(\texttt{rreq}(*,*,*,*,*,*,*,\texttt{sip}))$, $\xi_N^{ip}(\texttt{rrep}(*,*,*,*,\texttt{sip}))$, or $\xi_N^{ip}(\texttt{rerr}(*,\texttt{sip}))$ (Pro. 1, Line 1); the value of $\texttt{sip}$ is never changed. By Proposition 7.1, this message must have been sent before by a node $ip' \neq ip$. By Proposition 7.8, $\xi_N^{ip}(\texttt{sip}) = ip'$. □

**Proposition 7.10** All routing table entries have a hop count greater or equal than 1.

$$(*,*,*,*,hops,*,*) \in \xi_N^{ip}(\texttt{rt}) \;\Rightarrow\; hops \geq 1 \tag{4}$$

*Proof.* All initial states trivially satisfy the invariant since all routing tables are empty. The functions $\texttt{invalidate}$ and $\texttt{addpreRT}$ do not affect the invariant, since they do not change the hop count of a routing table entry. Therefore, we only have to look at the application calls of $\texttt{update}$. In each case, if the update does not change the routing table entry beyond its precursors (the last clause of $\texttt{update}$), the invariant is trivially preserved; hence we examine the cases that an update actually occurs.

**Pro. 1, Lines 10, 14, 18:** All these updates have a hop count equals to 1; hence the invariant is preserved.

**Pro. 4, Line 4; Pro. 5, Line 2:** Here, $\xi(\texttt{hops})+1$ is used for the update. Since $\xi(\texttt{hops}) \in \mathbb{N}$, the invariant is maintained. □

**Proposition 7.11**

(a) If a route request with hop count 0 is sent by a node $ip_c \in \mathbf{IP}$ , the sender must be the originator.

$$N \xrightarrow{R:*\mathbf{cast}(\texttt{rreq}(0,*,*,*,*,oip_c,*,ip_c))}_{ip} N' \;\Rightarrow\; oip_c = ip_c (= ip) \tag{5}$$

(b) If a route reply with hop count 0 is sent by a node $ip_c \in \mathbf{IP}$, the sender must be the destination.

$$N \xrightarrow{R:*\mathbf{cast}(\texttt{rrep}(0,dip_c,*,*,ip_c))}_{ip} N' \;\Rightarrow\; dip_c = ip_c (= ip) \tag{6}$$

*Proof.*

(a) We have to check that the consequent holds whenever a route request is sent. In all the processes there are only two locations where this happens.

**Pro. 1, Line 39:** A request with content $\xi(0,*,*,*,*,\texttt{ip},*,\texttt{ip})$ is sent. Since the sixth and the eighth component are the same ($\xi(\texttt{ip})$), the claim holds.

**Pro. 4, Line 36:** The message has the form $\texttt{rreq}(\xi(\texttt{hops})+1,*,*,*,*,*,*,*)$. Since $\xi(\texttt{hops}) \in \mathbb{N}$, $\xi(\texttt{hops})+1 \neq 0$ and hence the antecedent does not hold.

(b) We have to check that the consequent holds whenever a route reply is sent. In all the processes there are only three locations where this happens.

**Pro. 4, Line 10:** A reply with content $\xi(0,\mathtt{dip},*,*,\mathtt{ip})$ is sent. By Line 7 we have $\xi(\mathtt{dip}) = \xi(\mathtt{ip})$, so the claim holds.

**Pro. 4, Line 25:** The message has the form $\mathtt{rrep}(\mathtt{dhops}(\mathtt{rt},\mathtt{dip}),*,*,*,*)$. By Proposition 7.10, $\mathtt{dhops}(\mathtt{rt},\mathtt{dip}) > 0$, so the antecedent does not hold.

**Pro. 5, Line 13:** The message has the form $\mathtt{rrep}(\xi(\mathtt{hops}){+}1,*,*,*,*)$. Since $\xi(\mathtt{hops}) \in \mathbb{N}$, $\xi(\mathtt{hops})+1 \neq 0$ and hence the antecedent does not hold. □

**Proposition 7.12**

(a) Each routing table entry with 0 as its destination sequence number has a sequence-number-status flag valued unknown.

$$(dip,0,f,*,*,*,*) \in \xi_N^{ip}(\mathtt{rt}) \;\Rightarrow\; f = \mathtt{unk} \tag{7}$$

(b) Unknown sequence numbers can only occur at 1-hop connections.

$$(*,*,\mathtt{unk},*,hops,*,*) \in \xi_N^{ip}(\mathtt{rt}) \;\Rightarrow\; hops = 1 \tag{8}$$

(c) 1-hop connections must contain the destination as next hop.

$$(dip,*,*,*,1,nhip,*) \in \xi_N^{ip}(\mathtt{rt}) \;\Rightarrow\; dip = nhip \tag{9}$$

(d) If the sequence number 0 occurs within a routing table entry, the hop count as well as the next hop can be determined.

$$(dip,0,f,*,hops,nhip,*) \in \xi_N^{ip}(\mathtt{rt}) \;\Rightarrow\; f = \mathtt{unk} \,\wedge\, hops = 1 \,\wedge\, dip = nhip \tag{10}$$

*Proof.* At the initial states all routing tables are empty. Since `invalidate` and `addpreRT` change neither the sequence-number-status flag, nor the next hop or the hop count of a routing table entry, and—by Proposition 7.6—cannot decrease the sequence number of a destination, we only have to look at the application calls of `update`. As before, we only examine the cases that an update actually occurs.

(a) Function calls of the form $\mathtt{update}(rt,r)$ always preserve the invariant: in case `update` is given an argument for which it is not defined, the process algebra blocks and no change of the routing table is performed (cf. Footnote 16 in Section 4); in case one of the first four clauses in the definition of `update` is used, this follows because $\mathtt{update}(rt,r)$ is defined only when $\pi_2(r) = 0 \Leftrightarrow \pi_3(r) = \mathtt{unk}$; in case the fifth clause is used it follows because $\pi_3(r) = \mathtt{unk}$; and in case the last clause is used, it follows by induction, since the invariant was already valid before the update.

(b) **Pro. 1, Lines 10, 14, 18:** All these updates have an unknown sequence number and hop count equal to 1. By Clause 5 of `update`, these sequence-number-status flag and hop count are transferred literally into the routing table; hence the invariant is preserved.

**Pro. 4, Line 4 and Pro. 5, Line 2:** In these updates the sequence-number-status flag is set to `kno`. By the definition of `update`, this value ends up in the routing table. Hence the assumption of the invariant to be proven is not satisfied.

(c) **Pro. 1, Lines 10, 14, 18:** The new entries $(\xi(\mathtt{sip},0,\mathtt{unk},\mathtt{val},1,\mathtt{sip},\emptyset))$ satisfy the invariant; even if the routing table is actually updated with one of the new routes, the invariant holds afterwards.

**Pro. 4, Line 4; Pro. 5, Line 2:** The route which might be inserted into the routing table has hop count $\mathtt{hops}{+}1$, $\mathtt{hops} \in \mathbb{N}$. It can only be equal to 1 if the received message had hop count $\mathtt{hops} = 0$. In that case Invariant (5), resp. (6), guarantees that the invariant remains unchanged.

(d) Immediate from Parts (a) to (c). □

**Proposition 7.13**

(a) Whenever an originator sequence number is sent as part of a route request message, it is known, i.e., it is greater or equal than 1.

$$N \xrightarrow{R:*\mathbf{cast}(\mathtt{rreq}(*,*,*,*,*,*,osn_c,*))}_{ip} N' \;\Rightarrow\; osn_c \geq 1 \tag{11}$$

(b) Whenever a destination sequence number is sent as part of a route reply message, it is known, i.e., it is greater or equal than 1.

$$N \xrightarrow{R:*\mathbf{cast}(\mathtt{rrep}(*,*,dsn_c,*,*))}_{ip} N' \;\Rightarrow\; dsn_c \geq 1 \tag{12}$$

*Proof.*

(a) We have to check that the consequent holds whenever a route request is sent.

**Pro. 1, Line 39:** A route request is initiated. The originator sequence number is a copy of the node's own sequence number, i.e., $osn_c = \xi(\mathtt{sn})$. By Proposition 7.2, we get $osn_c \geq 1$.

**Pro. 4, Line 36:** Here, $osn_c := \xi(\mathtt{osn})$. $\xi(\mathtt{osn})$ is not changed within Pro. 4; it stems, through Line 8 of Pro. 1, from an incoming RREQ message (Pro. 1, Line 1). For this incoming RREQ message, using Proposition 7.1(a) and induction on reachability, the invariant holds and hence the claim follows immediately.

(b) We have to check that the consequent holds whenever a route reply is sent.

**Pro. 4, Line 10:** The destination initiates a route reply. The sequence number is a copy of the node's own sequence number, i.e., $dsn_c = \xi(\mathtt{sn})$. By Proposition 7.2, we get $dsn_c \geq 1$.

**Pro. 4, Line 25:** The sequence number used for the message is copied from the routing table; its value is $dsn_c := \mathtt{sqn}(\xi(\mathtt{rt}),\xi(\mathtt{dip}))$. By Line 20, we know that $\mathtt{flag}(\xi(\mathtt{rt}),\xi(\mathtt{dip})) = \mathtt{kno}$ and hence, by Invariant (7), $dsn_c \geq 1$. Thus the invariant is maintained.

**Pro. 5, Line 13:** Here, $dsn_c := \xi(\mathtt{dsn})$. $\xi(\mathtt{dsn})$ is not changed within Pro. 5; it stems, through Line 12 of Pro. 1, from an incoming RREP message (Pro. 1, Line 1). For this incoming RREP message the invariant holds and hence the claim follows immediately. □

**Proposition 7.14**

(a) If a route request is sent (forwarded) by a node $ip_c$ different from the originator of the request then the content of $ip_c$'s routing table must be fresher or at least as good as the information inside the message.

$$\begin{aligned} & N \xrightarrow{R:*\mathbf{cast}(\mathtt{rreq}(hops_c,*,*,*,*,oip_c,osn_c,ip_c))}_{ip} N' \wedge ip_c \neq oip_c \\ \Rightarrow\; & oip_c \in \mathtt{kD}_N^{ip_c} \wedge \big(\mathtt{sqn}_N^{ip_c}(oip_c) > osn_c \\ & \vee (\mathtt{sqn}_N^{ip_c}(oip_c) = osn_c \wedge \mathtt{dhops}_N^{ip_c}(oip_c) \leq hops_c \wedge \mathtt{flag}_N^{ip_c}(oip_c) = \mathtt{val})\big) \end{aligned} \tag{13}$$

(b) If a route reply is sent by a node $ip_c$, different from the destination of the route, then the content of $ip_c$'s routing table must be consistent with the information inside the message.

$$\begin{aligned} & N \xrightarrow{R:*\mathbf{cast}(\mathtt{rrep}(hops_c,dip_c,dsn_c,*,ip_c))}_{ip} N' \wedge ip_c \neq dip_c \\ \Rightarrow\; & dip_c \in \mathtt{kD}_N^{ip_c} \wedge \mathtt{sqn}_N^{ip_c}(dip_c) = dsn_c \wedge \mathtt{dhops}_N^{ip_c}(dip_c) = hops_c \wedge \mathtt{flag}_N^{ip_c}(dip_c) = \mathtt{val} \end{aligned} \tag{14}$$

*Proof.*

(a) We have to check all cases where a route request is sent:

**Pro. 1, Line 39:** A new route request is initiated with $ip_c = oip_c := \xi(\mathtt{ip}) = ip$. Here the antecedent of (13) is not satisfied.

**Pro. 4, Line 36:** The broadcast message has the form

$$\xi(\mathtt{rreq}(\mathtt{hops}{+}1,\mathtt{rreqid},\mathtt{dip},\max(\mathtt{sqn}(\mathtt{rt},\mathtt{dip}),\mathtt{dsn}),\mathtt{dsk},\mathtt{oip},\mathtt{osn},\mathtt{ip}))\ .$$

Hence $hops_c := \xi(\mathtt{hops}){+}1$, $oip_c := \xi(\mathtt{oip})$, $osn_c := \xi(\mathtt{osn})$, $ip_c := \xi(\mathtt{ip}) = ip$ and $\xi_N^{ip_c} = \xi$ (by (3)). At Line 4 we update the routing table using $r{:=}\xi(\mathtt{oip},\mathtt{osn},\mathtt{kno},\mathtt{val},\mathtt{hops}{+}1,\mathtt{sip},\emptyset)$ as new entry. The routing table does not change between Lines 4 and 36; nor do the values of the variables hops, oip and osn. If the new (valid) entry is inserted into the routing table, then one of the first four cases in the definition of update must have applied—the fifth case cannot apply, since $\pi_3(r) = \mathtt{kno}$. Thus, using that $oip_c \neq ip_c$,

$$\begin{aligned}
\mathtt{sqn}_N^{ip_c}(oip_c) &= \mathtt{sqn}(\xi(\mathtt{rt}),\xi(\mathtt{oip})) &&= \xi(\mathtt{osn}) &&= osn_c\\
\mathtt{dhops}_N^{ip_c}(oip_c) &= \mathtt{dhops}(\xi(\mathtt{rt}),\xi(\mathtt{oip})) &&= \xi(\mathtt{hops})+1 &&= hops_c\\
\mathtt{flag}_N^{ip_c}(oip_c) &= \mathtt{flag}(\xi(\mathtt{rt}),\xi(\mathtt{oip})) &&= \xi(\mathtt{val}) &&= \mathtt{val}\ .
\end{aligned}$$

In case the new entry is not inserted into the routing table (the sixth case of update), we have $\mathtt{sqn}_N^{ip_c}(oip_c) = \mathtt{sqn}(\xi(\mathtt{rt}),\xi(\mathtt{oip})) \geq \xi(\mathtt{osn}) = osn_c$, and in case that $\mathtt{sqn}_N^{ip_c}(oip_c) = osn_c$ we see that $\mathtt{dhops}_N^{ip_c}(oip_c) = \mathtt{dhops}(\xi(\mathtt{rt}),\xi(\mathtt{oip})) \leq \xi(\mathtt{hops})+1 = hops_c$ and moreover $\mathtt{flag}_N^{ip_c}(oip_c) = \mathtt{val}$. Therefore the invariant holds.

(b) We have to check all cases where a route reply is sent.

**Pro. 4, Line 10:** A new route reply with $ip_c := \xi(\mathtt{ip}) = ip$ is initiated. Moreover, by Line 7, $dip_c := \xi(\mathtt{dip}) = \xi(\mathtt{ip}) = ip$ and thus $ip_c = dip_c$. Hence, the antecedent of (14) is not satisfied.

**Pro. 4, Line 25:** We have $ip_c := \xi(\mathtt{ip}) = ip$, so $\xi_N^{ip_c} = \xi$. This time, by Line 18, $dip_c := \xi(\mathtt{dip}) \neq \xi(\mathtt{ip}) = ip_c$. By Line 20 there is a valid routing table entry for $dip_c := \xi(\mathtt{dip})$.

$$\begin{aligned}
dsn_c &:= \mathtt{sqn}(\xi(\mathtt{rt}),\xi(\mathtt{dip})) &&= \mathtt{sqn}_N^{ip_c}(dip_c)\ ,\\
hops_c &:= \mathtt{dhops}(\xi(\mathtt{rt}),\xi(\mathtt{dip})) &&= \mathtt{dhops}_N^{ip_c}(dip_c)\ .
\end{aligned}$$

**Pro. 5, Line 13:** The RREP message has the form

$$\xi(\mathtt{rrep}(\mathtt{hops}+1,\mathtt{dip},\mathtt{dsn},\mathtt{oip},\mathtt{ip}))\ .$$

Hence $hops_c := \xi(\mathtt{hops}){+}1$, $dip_c := \xi(\mathtt{dip})$, $dsn_c := \xi(\mathtt{dsn})$, $ip_c := \xi(\mathtt{ip}) = ip$ and $\xi_N^{ip_c} = \xi$. Using $(\xi(\mathtt{dip}),\xi(\mathtt{dsn}),\mathtt{kno},\mathtt{val},\xi(\mathtt{hops}){+}1,\xi(\mathtt{sip}),\emptyset)$ as new entry, the routing table is updated at Line 2. With exception of its precursors, which are irrelevant here, the routing table does not change between Lines 2 and 13; nor do the values of the variables hops, dip and dsn. Line 1 guarantees that during the update in Line 2, the new entry is inserted into the routing table, so

$$\begin{aligned}
\mathtt{sqn}_N^{ip_c}(dip_c) &= \mathtt{sqn}(\xi(\mathtt{rt}),\xi(\mathtt{dip})) &&= \xi(\mathtt{dsn}) &&= dsn_c\\
\mathtt{dhops}_N^{ip_c}(dip_c) &= \mathtt{dhops}(\xi(\mathtt{rt}),\xi(\mathtt{dip})) &&= \xi(\mathtt{hops})+1 &&= hops_c\\
\mathtt{flag}_N^{ip_c}(dip_c) &= \mathtt{flag}(\xi(\mathtt{rt}),\xi(\mathtt{dip})) &&= \xi(\mathtt{val}) &&= \mathtt{val}\ .
\end{aligned}$$

□

**Proposition 7.15** Any sequence number appearing in a route error message stems from an invalid destination and is equal to the sequence number for that destination in the sender's routing table at the time of sending.

$$N \xrightarrow{R:\textbf{*cast}(\mathtt{rerr}(dests_c,ip_c))}_{ip} N' \ \wedge\ (rip_c,rsn_c) \in dests_c \ \Rightarrow\ rip_c \in \mathtt{iD}_N^{ip} \ \wedge\ rsn_c = \mathtt{sqn}_N^{ip}(rip_c) \tag{15}$$

*Proof.* We have to check that the consequent holds whenever a route error message is sent. In all the processes there are only seven locations where this happens.

**Pro. 1, Line 32:** The set $dests_c$ is constructed in Line 31 as a subset of $\xi_{N_{31}}^{ip}(\mathtt{dests}) = \xi_{N_{28}}^{ip}(\mathtt{dests})$. For each pair $(rip_c, rsn_c) \in \xi_{N_{28}}^{ip}(\mathtt{dests})$ one has $rip_c = \xi_{N_{27}}^{ip}(\mathtt{rip}) \in \mathtt{vD}_{N_{27}}^{ip}$. Then in Line 28, using the function $\mathtt{invalidate}$, $\mathtt{flag}(\xi(\mathtt{rt}), rip_c)$ is set to $\mathtt{inv}$ and $\mathtt{sqn}(\xi(\mathtt{rt}), rip_c)$ to $rsn_c$. Thus we obtain $rip_c \in \mathtt{iD}_N^{ip}$ and $\mathtt{sqn}_N^{ip}(rip_c) = rsn_c$.

**Pro. 3, Line 14; Pro. 4, Lines 17, 33; Pro. 5, Line 21; Pro. 6, Line 8:** Exactly as above.

**Pro. 3, Line 20:** The set $dests_c$ contains only one single element. Hence $rip_c := \xi_N^{ip}(\mathtt{dip})$ and $rsn_c := \xi_N^{ip}(\mathtt{sqn}(\mathtt{rt}, \mathtt{dip}))$. By Line 18, we have $rip_c = \xi_N^{ip}(\mathtt{dip}) \in \mathtt{iD}_N^{ip}$. The remaining claim follows by $rsn_c = \xi_N^{ip}(\mathtt{sqn}(\mathtt{rt}, \mathtt{dip})) = \mathtt{sqn}(\xi_N^{ip}(\mathtt{rt}), \xi_N^{ip}(\mathtt{dip})) = \mathtt{sqn}_N^{ip}(rip_c)$. □

### 7.4 Well-Definedness

We have to ensure that our specification of AODV is actually well defined. Since many functions introduced in Section 5 are only partial, it has to be checked that these functions are either defined when they are used, or are subterms of atomic formulas. In the latter case, those formula would evaluate to `false` (cf. Footnote 14 on Page 10).

The first proposition shows that the functions defined in Section 5 respect the data structure. In fact, these properties are required (or implied) by our data structure.

**Proposition 7.16**

(a) In each routing table there is at most one entry for each destination.

(b) In each store of queued data packets there is at most one data queue for each destination.

(c) Whenever a set of pairs $(rip, rsn)$ is assigned to the variable `dests` of type $\mathtt{IP} \rightharpoonup \mathtt{SQN}$, or to the first argument of the function `rerr`, this set is a partial function, i.e., there is at most one entry $(rip, rsn)$ for each destination $rip$.

*Proof.*

(a) In all initial states the invariant is satisfied, as a routing table starts out empty (see (2) in Section 6.7). None of the Processes 1–7 of Section 6 changes a routing table directly; the only way a routing table can be changed is through the functions `update`, `invalidate` and `addpreRT`. The latter two only change the sequence number, the validity status and the precursors of an existing route. This kind of update has no effect on the invariant. The first function inserts a new entry into a routing table only if the destination is unknown, that is, if no entry for this destination already exists in the routing table; otherwise the existing entry is replaced. Therefore the invariant is maintained.

(b) In any initial state the invariant is satisfied, as each store of queued data packets starts out empty. In Processes 1–7 of Section 6 a store is updated only through the functions `add` and `drop`. These functions respect the invariant.

(c) This is checked by inspecting all assignments to `dests` in Processes 1–7.

**Pro. 1, Line 16:** The message $\xi(\mathtt{msg})$ is received in Line 1, and hence, by Proposition 7.1(a), sent by some node before. The content of the message does not change during transmission, and we assume there is only one way to read a message $\xi(\mathtt{msg})$ as $\mathtt{rerr}(\xi(\mathtt{dests}), \xi(\mathtt{sip}))$. By induction, we may assume that when the other node composed the message, a partial function was assigned to the first argument $\xi(\mathtt{dests})$ of `rerr`.

- **Pro. 1, Line 27; Pro. 3, Line 9; Pro. 4, Lines 12, 28; Pro. 5, Line 16:** The assigned sets have the form $\{(\xi(\texttt{rip}), \texttt{inc}(\texttt{sqn}(\xi(\texttt{rt}), \xi(\texttt{rip})))) \mid \ldots\})$. Since inc and sqn are functions, for each $\xi(\texttt{rip})$ there is only one pair $(\xi(\texttt{rip}), \texttt{inc}(\texttt{sqn}(\xi(\texttt{rt}), \xi(\texttt{rip}))))$.
- **Pro. 1, Line 31; Pro. 3, Line 13; Pro. 4, Lines 16, 32; Pro. 5, Line 20; Pro. 6, Line 7:** In each of these cases a set $\xi(\texttt{dests})$ constructed four lines before is used to construct a new set. By the invariant to be proven, these sets are already partial functions. From these sets some values are removed. Since subsets of partial functions are again partial functions, the claim follows immediately.
- **Pro. 6, Line 2:** Similar to the previous case except that the set $\xi(\texttt{dests})$ to be thinned out is not constructed before but stems from an incoming RERR message.
- **Pro. 3, Lines 20:** The set is explicitly given and consists of only one element; thus the claim is trivial. □

Property (a) is stated in the RFC [79].

**Proposition 7.17** In our specification of AODV, the functions $\sigma_{route}$ and flag are only used when they are defined.

*Proof.* In our entire specification we do not use these functions at all; they are only used for defining other functions. □

**Proposition 7.18** In our specification of AODV, the function dhops is only used when it is defined.

*Proof.* The function $\texttt{dhops}(rt, dip)$ is defined iff $dip \in \texttt{kD}(rt)$.

**Pro. 4, Line 25:** By Line 20 $\xi(\texttt{dip}) \in \texttt{vD}(\xi(\texttt{rt})) \subseteq \texttt{kD}(\xi(\texttt{rt}))$; so $\texttt{dhops}(\xi(\texttt{rt}), \xi(\texttt{dip}))$ is defined. □

**Proposition 7.19** In our specification of AODV, the function nhop is either used within formulas or if it is defined; hence it is only used in a meaningful way.

*Proof.* As in Proposition 7.18, the function $\texttt{nhop}(rt, dip)$ is defined iff $dip \in \texttt{kD}(rt)$.

- **Pro. 1, Line 27; Pro. 3, Line 9; Pro. 4, Lines 12, 28; Pro. 5, Line 16; Pro. 6, Line 2:** The function is used within a formula.
- **Pro. 1, Line 23:** Line 21 states $\xi(\texttt{dip}) \in \texttt{vD}(\xi(\texttt{rt}))$; hence $\texttt{nhop}(\xi(\texttt{rt}), \xi(\texttt{dip}))$ is defined.
- **Pro. 3, Line 7:** By Line 5, $\xi(\texttt{dip}) \in \texttt{vD}(\xi(\texttt{rt}))$.
- **Pro. 4, Lines 10, 25:** In Line 4 the entry for destination $\xi(\texttt{oip})$ is updated; by this $\xi(\texttt{oip}) \in \texttt{kD}(\xi(\texttt{rt}))$.
- **Pro. 4, Line 23:** By Line 20 $\xi(\texttt{dip}) \in \texttt{vD}(\xi(\texttt{rt}))$.
- **Pro. 5, Lines 11, 13:** By Line 9 $\xi(\texttt{oip}) \in \texttt{vD}(\xi(\texttt{rt}))$.
- **Pro. 5, Line 12:** In Line 2 the entry for destination $\xi(\texttt{dip})$ is updated; by this $\xi(\texttt{dip}) \in \texttt{kD}(\xi(\texttt{rt}))$. By Line 9 $\xi(\texttt{oip}) \in \texttt{vD}(\xi(\texttt{rt}))$.

If nhop is used within a formula, then $\texttt{nhop}(rt, rip)$ may not be defined, namely if $rip \notin \texttt{kD}(rt)$. In such a case, according to the convention of Footnote 14 in Section 4, the atomic formula in which this term occurs evaluates to false, and thereby is defined properly. □

If one chooses to use lazy evaluation for conjunction, then nhop is only used where it is defined.

**Proposition 7.20** In our specification of AODV, the function precs is only used when it is defined.

*Proof.* As in Proposition 7.18, the function $\texttt{precs}(rt, dip)$ is defined iff $dip \in \texttt{kD}(rt)$.

**Pro. 1, Line 30; Pro. 3, Line 12; Pro. 4, Lines 15, 31; Pro. 5, Line 19:** Three lines before the $\texttt{precs}$ is used, the set $\xi(\mathit{dests})$ is created containing only pairs $(\xi(\texttt{rip}),*)$ with $\xi(\texttt{rip}) \in \texttt{vD}(\xi(\texttt{rt}))$.

**Pro. 1, Line 31; Pro. 3, Line 13; Pro. 4, Lines 16, 32; Pro. 5, Line 20:** Four lines before the $\texttt{precs}$ is used, the set $\xi(\mathit{dests})$ is created containing only pairs $(\xi(\texttt{rip}),*)$ with $\xi(\texttt{rip}) \in \texttt{vD}(\xi(\texttt{rt}))$.

**Pro. 3, Line 20:** Line 18 states that $\xi(\texttt{dip}) \in \texttt{iD}(\xi(\texttt{rt})) \subseteq \texttt{kD}(\xi(\texttt{rt}))$.

**Pro. 6, Line 6:** Similar to Pro. 1, Line 30; the set $\xi(\texttt{dests})$ is created under the assumption $\xi(\texttt{rip}) \in \texttt{vD}(\xi(\texttt{rt}))$ in Line 2. □

**Proposition 7.21** In our specification of AODV, the function $\texttt{update}$ is only used when it is defined.

*Proof.* $\texttt{update}(rt,r)$ is defined only under the assumptions $\pi_4(r)=\texttt{val}$, $\pi_2(r)=0 \Leftrightarrow \pi_3(r)=\texttt{unk}$ and $\pi_3(r)=\texttt{unk} \Rightarrow \pi_5(r)=1$. In Pro. 1, Lines 10, 14 and 18, the entry $\xi(\texttt{sip},0,\texttt{unk},\texttt{val},1,\texttt{sip},\emptyset)$ is used as second argument, which obviously satisfies the assumptions. The function is used at four other locations:

**Pro. 4, Line 4:** Here, the entry $\xi(\texttt{oip},\texttt{osn},\texttt{kno},\texttt{val},\texttt{hops}+1,\texttt{sip},\emptyset)$ is used as $r$ to update the routing table. This entry fulfils $\pi_4(r)=\texttt{val}$. Since $\pi_3(r)=\texttt{kno}$, it remains to show that $\pi_2(r)=\xi(\texttt{osn}) \geq 1$. The sequence number $\xi(\texttt{osn})$ stems, through Line 8 of Pro. 1, from an incoming RREQ message and is not changed within Pro. 4. Hence, by Invariant (11), $\xi(\texttt{osn}) \geq 1$.

**Pro. 5, Lines 1, 2, 26:** The update is similar to the one of Pro. 4, Line 4. The only difference is that the information stems from an incoming RREP message and that a routing table entry to $\xi(\texttt{dip})$ (instead of $\xi(\texttt{oip})$) is established. Therefore, the proof is similar to the one of Pro. 4, Line 4; instead of Invariant (11) we use Invariant (12). □

**Proposition 7.22** In our specification of AODV, the function $\texttt{addpreRT}$ is only used when it is defined.

*Proof.* It suffices to check that for any call $\texttt{addpreRT}(rt,dip,*)$ the destination has an entry in the routing table, i.e., $dip \in \texttt{kD}(rt)$.

**Pro. 4, Line 22:** Line 20 shows that $\xi(\texttt{dip}) \in \texttt{vD}(\xi(\texttt{rt})) \subseteq \texttt{kD}(\xi(\texttt{rt}))$.

**Pro. 4, Line 23:** In Line 4 an entry to $\xi(\texttt{oip})$ is updated. In case there was no entry before, it is inserted; hence we know $\xi(\texttt{oip}) \in \texttt{kD}(\xi(\texttt{rt}))$.

**Pro. 5, Line 11:** Similar to the previous case: Line 2 updates a routing entry to $\xi(\texttt{dip})$.

**Pro. 5, Line 12:** Line 2 updates the routing table entry with destination $\xi(\texttt{dip})$. By Line 1 it is known that the entry $\xi(\texttt{dip},\texttt{dsn},\texttt{kno},\texttt{val},\texttt{hops}+1,\texttt{sip},\emptyset)$ is inserted; hence $\texttt{nhop}(\xi(\texttt{rt}),\xi(\texttt{dip})) = \xi(\texttt{sip})$. A routing table entry for $\xi(\texttt{sip})$ exists by Line 14 of Pro. 1. □

**Proposition 7.23** In our specification of AODV, the functions $\texttt{head}$ and $\texttt{tail}$ are only used when they are defined.

*Proof.* These functions are defined if the list given as argument is non-empty.

**Pro. 1, Line 22:** The function $\texttt{head}$ tries to return the first element of $\sigma_{queue}(\xi(\texttt{store}),\xi(\texttt{dip}))$, which is, by Line 21 ($\xi(\texttt{dip}) \in \texttt{qD}(\xi(\texttt{store}))$) and (1), not empty.

**Pro. 7, Line 6:** Here, the functions work on the list $\xi(\texttt{msgs})$; Line 3 shows that $\xi(\texttt{msgs}) \neq []$. □

**Proposition 7.24** In our specification of AODV, the function $\texttt{drop}$ is only used when it is defined.

*Proof.* The function $\texttt{drop}$ is only used in Pro. 1, Line 24. It tries to delete the oldest packet queued for destination $\xi(\texttt{dip})$; the function is defined if at least one packet for $\xi(\texttt{dip})$ is stored in $\xi(\texttt{store})$—this is guaranteed by Line 21, which states $\xi(\texttt{dip}) \in \texttt{qD}(\xi(\texttt{store}))$, and (1). □

**Proposition 7.25** In our specification of AODV, the function $\sigma_{p\text{-}flag}$ is only used within formulas. □

The function is called only in Pro. 1 in Line 33 using $\sigma_{p\text{-}flag}(\xi(\texttt{store}),\xi(\texttt{dip}))$. Again, if one would use lazy evaluation for conjunction, then $\sigma_{p\text{-}flag}$ is used where it is defined.

### 7.5 The Quality of Routing Table Entries

In this section we define a total preorder $\sqsubseteq_{dip}$ on routing table entries for a given destination *dip*. Entries are ordered by the *quality* of the information they provide. This order will be defined in such a way that (a) the quality of a node's routing table entry for *dip* will only increase over time, and (b) the quality of valid routing table entries along a route to *dip* strictly increases every hop (at least prior to reaching *dip*). This order allows us to prove *loop freedom* of AODV in the next section.

A main ingredient in the definition of the quality preorder is the sequence number of a routing table entry. A higher sequence number denotes fresher information. However, it generally is not the case that along a route to *dip* found by AODV the sequence numbers are only increasing. This is since AODV increases the sequence number of an entry at an intermediate node when invalidating it. To "compensate" for that we introduce the concept of a *net sequence number*. It is defined by a function $\mathtt{nsqn}:\mathtt{R}\to\mathtt{SQN}$

$$\mathtt{nsqn}(r) \;:=\; \begin{cases} \pi_2(r) & \text{if } \pi_4(r)=\mathtt{val} \;\vee\; \pi_2(r)=0 \\ \pi_2(r)-1 & \text{otherwise .} \end{cases}$$

For $n\in\mathbb{N}$ define $n \mathbin{\dot{-}} 1 := \max(n{-}1,0)$, so that $\mathtt{inc}(n)\mathbin{\dot{-}}1=n$. Then $\mathtt{nsqn}(r)=\pi_2(r)\mathbin{\dot{-}}1$ if $\pi_4(r)=\mathtt{inv}$.

To model increase in quality, we define $\sqsubseteq_{dip}$ by first comparing the net sequence numbers of two entries—a larger net sequence number denotes fresher and higher quality information. In case the net sequence numbers are equal, we decide on their hop counts—the entry with the least hop count is the best. This yields the following lexicographical order:

Assume two routing table entries $r,r'\in\mathtt{R}$ with $\pi_1(r)=\pi_1(r)=dip$. Then

$$r \sqsubseteq_{dip} r' \;:\Leftrightarrow\; \mathtt{nsqn}(r)<\mathtt{nsqn}(r') \;\vee\; \big(\mathtt{nsqn}(r)=\mathtt{nsqn}(r') \;\wedge\; \pi_5(r)\geq\pi_5(r')\big) \;.$$

To reason about AODV, net sequence numbers and the quality preorder is lifted to routing tables. As for $\mathtt{sqn}$ we define a total function to determine net sequence numbers.

$$\begin{aligned} \mathtt{nsqn}:\mathtt{RT}\times\mathtt{IP} \;&\to\; \mathtt{SQN} \\ \mathtt{nsqn}(rt,dip) \;&:=\; \begin{cases} \mathtt{nsqn}(\sigma_{route}(rt,dip)) & \text{if } \sigma_{route}(rt,dip) \text{ is defined} \\ 0 & \text{otherwise} \end{cases} \\ &=\; \begin{cases} \mathtt{sqn}(rt,dip) & \text{if } \mathtt{flag}(rt,dip)=\mathtt{val} \;\vee\; \mathtt{sqn}(rt,dip)=0 \\ \mathtt{sqn}(rt,dip)-1 & \text{otherwise .} \end{cases} \end{aligned}$$

If two routing tables *rt* and *rt'* have a routing table entry to *dip*, i.e., $dip\in\mathtt{kD}(rt)\cap\mathtt{kD}(rt')$, the preorder can be lifted as well.

$$\begin{aligned} rt \sqsubseteq_{dip} rt' \;:\Leftrightarrow&\; \sigma_{route}(rt,dip) \sqsubseteq_{dip} \sigma_{route}(rt',dip) \\ \Leftrightarrow&\; \mathtt{nsqn}(rt,dip)<\mathtt{nsqn}(rt',dip) \;\vee \\ &\; \big(\mathtt{nsqn}(rt,dip)=\mathtt{nsqn}(rt',dip) \;\wedge\; \mathtt{dhops}(rt,dip)\geq\mathtt{dhops}(rt',dip)\big) \end{aligned}$$

For all destinations $dip\in\mathbf{IP}$, the relation $\sqsubseteq_{dip}$ on routing tables with an entry for *dip* is total preorder. The equivalence relation induced by $\sqsubseteq_{dip}$ is denoted by $\approx_{dip}$.

As with $\mathtt{sqn}$, we shorten $\mathtt{nsqn}$: $\mathtt{nsqn}^{ip}_{N}(dip) := \mathtt{nsqn}(\xi^{ip}_{N}(\mathtt{rt}),dip)$. Note that

$$\mathtt{sqn}^{ip}_{N}(dip)\mathbin{\dot{-}}1 \leq \mathtt{nsqn}^{ip}_{N}(dip) \leq \mathtt{sqn}^{ip}_{N}(dip) \;. \tag{16}$$

After setting up this notion of quality, we now show that routing tables, when modified by AODV, never decrease their quality.

**Proposition 7.26** Assume a routing table $rt \in \mathtt{RT}$ with $dip \in \mathtt{kD}(rt)$.

(a) An update of $rt$ can only increase the quality of the routing table. That is, for all routes $r$ such that $\mathtt{update}(rt,r)$ is defined (i.e., $\pi_4(r) = \mathtt{val}$, $\pi_2(r) = 0 \Leftrightarrow \pi_3(r) = \mathtt{unk}$ and $\pi_3(r) = \mathtt{unk} \Rightarrow \pi_5(r) = 1$) we have

$$rt \sqsubseteq_{dip} \mathtt{update}(rt,r)\ . \tag{17}$$

(b) An invalidate on $rt$ does not change the quality of the routing table if, for each $(rip,rsn) \in dests$, $rt$ has a valid entry for $rip$, and

- $rsn$ is the by one incremented sequence number from the routing table, or
- both $rsn$ and the sequence number in the routing table are 0.

That is, for all partial functions $dests$ (subsets of $\mathtt{IP} \times \mathtt{SQN}$)

$$\begin{aligned} &\big((rip,rsn) \in dests \ \Rightarrow\ rip \in \mathtt{vD}(rt)\ \wedge\ rsn = \mathtt{inc}(\mathtt{sqn}(rt,rip))\big)\\ \Rightarrow\quad & rt \approx_{dip} \mathtt{invalidate}(rt,dests)\ . \end{aligned} \tag{18}$$

(c) If precursors are added to an entry of $rt$, the quality of the routing table does not change. That is, for all $dip \in \mathbf{IP}$ and sets of precursors $npre \in \mathscr{P}(\mathbf{IP})$,

$$rt \approx_{dip} \mathtt{addpreRT}(rt,dip,npre)\ . \tag{19}$$

*Proof.* For the proof we denote the routing table after the update by $rt'$.

(a) By assumption, there is an entry $(dip,dsn_{rt},*,f_{rt},hops_{rt},*,*)$ for $dip$ in $rt$. In case $\pi_1(r) \neq dip$ the quality of the routing table w.r.t. $dip$ stays the same, since the entry for $dip$ is not changed.

We first assume that $r := (dip,0,\mathtt{unk},\mathtt{val},1,*,*)$. This means that the Clause 5 in the definition of update is used. The updated routing table entry to $dip$ has the form $(dip,dsn_{rt},\mathtt{unk},\mathtt{val},1,*,*)$. So

$$\mathtt{nsqn}(rt,dip) \leq \mathtt{sqn}(rt,dip) = dsn_{rt} = \mathtt{nsqn}(rt',dip)\ ,\ \text{and}$$
$$\mathtt{dhops}(rt,dip) = hops_{rt} \geq 1 = \mathtt{dhops}(rt',dip)\ .$$

The first inequality holds by (16); the penultimate step by Invariant (4).

Next, we assume that the sequence number is known and therefore the route used for the update has the form $r = (dip,dsn,\mathtt{kno},\mathtt{val},hops,*,*)$ with $dsn \geq 1$. After the performed update the routing entry for $dip$ either has the form $(dip,dsn_{rt},*,f_{rt},hops_{rt},*,*)$ or $(dip,dsn,\mathtt{kno},\mathtt{val},hops,*,*)$. In the former case the invariant is trivially preserved; in the latter, we know, by definition of update, that either (i) $dsn_{rt} < dsn$, (ii) $dsn_{rt} = dsn \wedge hops_{rt} > hops$, or (iii) $dsn_{rt} = dsn \wedge f_{rt} = \mathtt{inv}$ holds. We complete the proof of the invariant by a case distinction.

**(i) holds:** First, $\mathtt{nsqn}(rt,dip) \leq dsn_{rt} < dsn = \mathtt{sqn}(rt',dip) = \mathtt{nsqn}(rt',dip)$. Since $dsn_{rt}$ is strictly smaller than $\mathtt{nsqn}(rt',dip)$, there is nothing more to prove.

**(iii) holds:** We have $\mathtt{nsqn}(rt,dip) = dsn_{rt} \mathbin{\dot{-}} 1 < dsn = \mathtt{sqn}(rt',dip) = \mathtt{nsqn}(rt',dip)$. The inequality holds since either $dsn_{rt} \mathbin{\dot{-}} 1 = 0 < 1 \leq dsn$ or $dsn_{rt} \mathbin{\dot{-}} 1 = dsn_{rt} - 1 < dsn_{rt} = dsn$.

**(ii) holds but (iii) does not:** Then $f_{rt} = \mathtt{val}$. In this case the update does not change the net sequence number for $dip$:

$$\mathtt{nsqn}(rt,dip) = dsn_{rt} = dsn = \mathtt{nsqn}(rt',dip)\ .$$

By (ii), the hop count decreases:

$$\mathtt{dhops}(rt,dip) = hops_{rt} > hops = \mathtt{dhops}(rt',dip)\ .$$

(b) Assume that invalidate modifies an entry of the form $(rip,dsn,*,flag,*,*,*)$. Let $(rip,rsn) \in dests$; then $flag = \mathtt{val}$ and the update results in the entry $(rip,\mathtt{inc}(dsn),*,\mathtt{inv},*,*,*)$. By definition of net sequence numbers,

$$\mathtt{nsqn}(rt,rip) = \mathtt{sqn}(rt,rip) = dsn = \mathtt{inc}(dsn) \mathbin{\dot{-}} 1 = \mathtt{nsqn}(rt',rip)\,.$$

Since the hop count is not changed by invalidate, we also have $\mathtt{dhops}(rt,rip) = \mathtt{dhops}(rt',rip)$, and hence $rt \approx_{dip} \mathtt{invalidate}(rt,dests)$.

(c) The function $\texttt{addpreRT}$ only modifies a set of precursors; it does not change the sequence number, the validity, the flag, nor the hop count of any entry of the routing table *rt*. □

We can apply this result to obtain the following theorem.

**Theorem 7.27** In AODV, the quality of routing tables can only be increased, never decreased.
Assume $N \xrightarrow{\ell} N'$ and $ip, dip \in \mathbf{IP}$. If $dip \in \texttt{kD}_N^{ip}$, then $dip \in \texttt{kD}_{N'}^{ip}$ and

$$\xi_N^{ip}(\texttt{rt}) \sqsubseteq_{dip} \xi_{N'}^{ip}(\texttt{rt}) \ .$$

*Proof.* If $dip \in \texttt{kD}_N^{ip}$, then $dip \in \texttt{kD}_{N'}^{ip}$ follows by Proposition 7.4. To show $\xi_N^{ip}(\texttt{rt}) \sqsubseteq_{dip} \xi_{N'}^{ip}(\texttt{rt})$, by Remark 7.3 and Proposition 7.26(a) and (c) it suffices to check all calls of $\texttt{invalidate}$.

**Pro. 1, Line 28; Pro. 3, Line 10; Pro. 4, Lines 13, 29; Pro. 5, Line 17:**
By construction of $\texttt{dests}$ (immediately before the invalidation call) $(rip, rsn) \in \xi_N^{ip}(\texttt{dests}) \Rightarrow rip \in \texttt{vD}(\xi_N^{ip}(\texttt{rt})) \wedge rsn = \texttt{inc}(\texttt{sqn}(\xi_N^{ip}(\texttt{rt}), rip))$ and hence, by Proposition 7.26(b), $\xi_N^{ip}(\texttt{rt}) \approx_{dip} \texttt{invalidate}(\xi_N^{ip}(\texttt{rt}), \xi_N^{ip}(\texttt{dests})) = \xi_{N'}^{ip}(\texttt{rt})$.

**Pro. 6, Line 3:** Assume that $\texttt{invalidate}$ modifies an entry of the form $(rip, dsn, *, flag, *, *, *)$. Let $(rip, rsn) \in dests$; then the update results in the entry $(rip, rsn, *, \texttt{inv}, *, *, *)$. Moreover, by Line 2 of Pro. 6, $flag = \texttt{val}$. By definition of net sequence numbers,

$$\texttt{nsqn}(\xi_N^{ip}(\texttt{rt}), rip) = \texttt{sqn}(\xi_N^{ip}(\texttt{rt}), rip) \leq rsn \stackrel{\bullet}{-} 1 = \texttt{nsqn}(\xi_{N'}^{ip}(\texttt{rt}), rip) \, .$$

The second step holds, since, by Line 2, $\texttt{sqn}(\xi_{N_2}^{ip}(\texttt{rt}), rip) < rsn$. Since the hop count is not changed by $\texttt{invalidate}$, we also have $\texttt{dhops}(\xi_N^{ip}(\texttt{rt}), rip) = \texttt{dhops}(\xi_{N'}^{ip}(\texttt{rt}), rip)$, and therefore $\xi_N^{ip}(\texttt{rt}) \sqsubseteq_{dip} \xi_{N'}^{ip}(\texttt{rt})$. □

Theorem 7.27 states in particular that if $N \xrightarrow{\ell} N'$ then $\texttt{nsqn}_N^{ip}(dip) \leq \texttt{nsqn}_{N'}^{ip}(dip)$.

**Proposition 7.28** If, in a reachable network expression $N$, a node $ip \in \mathbf{IP}$ has a routing table entry to *dip*, then also the next hop *nhip* towards *dip*, if not *dip* itself, has a routing table entry to *dip*, and the net sequence number of the latter entry is at least as large as that of the former.

$$dip \in \texttt{kD}_N^{ip} \ \wedge \ nhip \neq dip \ \Rightarrow \ dip \in \texttt{kD}_N^{nhip} \ \wedge \ \texttt{nsqn}_N^{ip}(dip) \leq \texttt{nsqn}_N^{nhip}(dip) \ , \tag{20}$$

where $nhip := \texttt{nhop}_N^{ip}(dip)$ is the IP address of the next hop.

*Proof.* As before, we first check the initial states of our transition system and then check all locations in Processes 1–7 where a routing table might be changed. For an initial network expression, the invariant holds since all routing tables are empty.

A modification of $\xi_N^{nhip}(\texttt{rt})$ is harmless, as it can only increase $\texttt{kD}_N^{nhip}$ (cf. Proposition 7.4) as well as $\texttt{nsqn}_N^{nhip}(dip)$ (cf. Theorem 7.27).

Adding precursors to $\xi_N^{ip}(\texttt{rt})$ does not harm since the invariant does not depend on precursors. It remains to examine all calls of $\texttt{update}$ and $\texttt{invalidate}$ to $\xi_N^{ip}(\texttt{rt})$. Without loss of generality we restrict attention to those applications of $\texttt{update}$ or $\texttt{invalidate}$ that actually modify the entry for *dip*, beyond its precursors; if $\texttt{update}$ only adds some precursors in the routing table, the invariant—which is assumed to hold before—is maintained. If $\texttt{invalidate}$ occurs, the next hop *nhip* is not changed. Since the invariant has to hold before the execution, it follows that $dip \in \texttt{kD}_N^{nhip}$ also holds after execution.

**Pro. 1, Lines 10, 14, 18:** The entry $\xi(\texttt{sip}, 0, \texttt{unk}, \texttt{val}, 1, \texttt{sip}, \emptyset)$ is used for the update; its destination is $dip := \xi(\texttt{sip})$. Since $dip = \xi(\texttt{sip}) = \texttt{nhop}_N^{ip}(\xi(\texttt{sip})) = \texttt{nhop}_N^{ip}(dip) = nhip$, the antecedent of the invariant to be proven is not satisfied.

**Pro. 1, Line 28; Pro. 3, Line 10; Pro. 4, Lines 13, 29; Pro. 5, Line 17:**
In each of these cases, the precondition of (18) is satisfied by the executions of the line immediately before the call of `invalidate` (Pro. 1, Line 27, Pro. 3, Line 9; Pro. 4, Lines 12, 28; Pro. 5, Line 16). Thus, the quality of the routing table w.r.t. *dip*, and thereby the net sequence number of the routing table entry for *dip*, remains unchanged. Therefore the invariant is maintained.

**Pro. 4, Line 4:** We assume that the entry $\xi(\mathtt{oip},\mathtt{osn},\mathtt{kno},\mathtt{val},\mathtt{hops}+1,\mathtt{sip},*)$ is inserted into $\xi(\mathtt{rt})$. So $dip := \xi(\mathtt{oip})$, $nhip := \xi(\mathtt{sip})$, $\mathtt{nsqn}_N^{ip}(dip) := \xi(\mathtt{osn})$ and $\mathtt{dhops}_N^{ip}(dip) := \xi(\mathtt{hops})+1$. This information is distilled from a received route request message (cf. Lines 1 and 8 of Pro. 1). By Proposition 7.1 this message was sent before, say in state $N^\dagger$; by Proposition 7.8 the sender of this message is $\xi(\mathtt{sip})$.

By Invariant (13), with $ip_c := \xi(\mathtt{sip}) = nhip$, $oip_c := \xi(\mathtt{oip}) = dip$, $osn_c := \xi(\mathtt{osn})$ and $hops_c := \xi(\mathtt{hops})$, and using that $ip_c = nhip \neq dip = oip_c$, we get that $dip \in \mathtt{kD}_{N^\dagger}^{nhip}$ and

$$\begin{aligned}\mathtt{sqn}_{N^\dagger}^{nhip}(dip) \;&=\; \mathtt{sqn}_{N^\dagger}^{ip_c}(oip_c) \;>\; osn_c \;=\; \xi(\mathtt{osn})\ \text{, or}\\ \mathtt{sqn}_{N^\dagger}^{nhip}(dip) \;&=\; \xi(\mathtt{osn}) \;\wedge\; \mathtt{flag}_{N^\dagger}^{nhip}(dip) \;=\; \mathtt{val}\ .\end{aligned}$$

We first assume that the first line holds. Then, by Theorem 7.27 and (16),

$$\mathtt{nsqn}_N^{nhip}(dip) \geq \mathtt{nsqn}_{N^\dagger}^{nhip}(dip) \geq \mathtt{sqn}_{N^\dagger}^{nhip}(dip) \mathbin{\dot{-}} 1 \geq \xi(\mathtt{osn}) = \mathtt{nsqn}_N^{ip}(dip)\ .$$

We now assume the second line to be valid. From this we conclude

$$\mathtt{nsqn}_N^{nhip}(dip) \geq \mathtt{nsqn}_{N^\dagger}^{nhip}(dip) = \mathtt{sqn}_{N^\dagger}^{nhip}(dip) = \xi(\mathtt{osn}) = \mathtt{nsqn}_N^{ip}(dip).$$

**Pro. 5, Line 2:** The update is similar to the one of Pro. 4, Line 4. The only difference is that the information stems from an incoming RREP message and that a routing table entry to $\xi(\mathtt{dip})$ (instead of $\xi(\mathtt{oip})$) is established. Therefore, the proof is similar to the one of Pro. 4, Line 4; instead of Invariant (13) we use Invariant (14).

**Pro. 6, Line 3:** Let $N_3$ and $N$ be the network expressions right before and right after executing Pro. 6, Line 3. The entry for destination *dip* can be affected only if $(dip,dsn) \in \xi_{N_2}^{ip}(\mathtt{dests})$ for some $dsn \in \mathtt{SQN}$. In that case, by Line 2, $(dip,dsn) \in \xi_{N_2}^{ip}(\mathtt{dests})$, $dip \in \mathtt{vD}_{N_2}^{ip}$, and $\mathtt{nhop}_{N_2}^{ip}(dip) = \xi_{N_2}^{ip}(\mathtt{sip})$. By definition of `invalidate`, $\mathtt{sqn}_N^{ip}(dip) = dsn$ and $\mathtt{flag}_N^{ip}(dip) = \mathtt{inv}$, so

$$\mathtt{nsqn}_N^{ip}(dip) = \mathtt{sqn}_N^{ip}(dip) \mathbin{\dot{-}} 1 = dsn \mathbin{\dot{-}} 1\ .$$

Hence we need to show that $dsn \mathbin{\dot{-}} 1 \leq \mathtt{nsqn}_N^{nhip}(dip)$.

The values $\xi_{N_2}^{ip}(\mathtt{dests})$ and $\xi_{N_2}^{ip}(\mathtt{sip})$ stem from a received route error message (cf. Lines 1 and 16 of Pro. 1). By Proposition 7.1(a), a transition labelled $R\colon$***cast**$(\mathtt{rerr}(dests_c\,,ip_c))$ with $dests_c := \xi_{N_2}^{ip}(\mathtt{dests})$ and $ip_c := \xi_{N_2}^{ip}(\mathtt{sip})$ must have occurred before, say in state $N^\dagger$. By Proposition 7.8, the node casting this message is $ip_c = \xi_{N_2}^{ip}(\mathtt{sip}) = \mathtt{nhop}_{N_2}^{ip}(dip) = \mathtt{nhop}_N^{ip}(dip) = nhip$. The penultimate equation holds since the next hop to *dip* is not changed during the execution of Pro. 6.

By Proposition 7.15 we have $dip \in \mathtt{iD}_{N^\dagger}^{nhip}$ and $dsn \leq \mathtt{sqn}(\xi_{N^\dagger}^{nhip}(\mathtt{rt}),dip)$. Hence

$$\mathtt{nsqn}_N^{nhip}(dip) \geq \mathtt{nsqn}_{N^\dagger}^{nhip}(dip) = \mathtt{nsqn}(\xi_{N^\dagger}^{nhip}(\mathtt{rt}),dip) = \mathtt{sqn}(\xi_{N^\dagger}^{nhip}(\mathtt{rt}),dip) \mathbin{\dot{-}} 1 \geq dsn \mathbin{\dot{-}} 1\ ,$$

where the first inequality follows by Theorem 7.27. $\square$

To prove loop freedom we will show that on any route established by AODV the quality of routing tables increases when going from one node to the next hop. Here, the preorder is not sufficient, since we need a strict increase in quality. Therefore, on routing tables *rt* and $rt'$ that both have an entry to *dip*, i.e., $dip \in \mathtt{kD}(rt) \cap \mathtt{kD}(rt')$, we define a relation $\sqsubset_{dip}$ by

$$rt \sqsubset_{dip} rt' \;:\Leftrightarrow\; rt \sqsubseteq_{dip} rt' \;\wedge\; rt \not\approx_{dip} rt' \;.$$

**Corollary 7.29** The relation $\sqsubset_{dip}$ is irreflexive and transitive.

**Theorem 7.30** The quality of the routing table entries for a destination *dip* is strictly increasing along a route towards *dip*, until it reaches either *dip* or a node with an invalid routing table entry to *dip*.

$$dip \in \mathtt{vD}^{ip}_{N} \cap \mathtt{vD}^{nhip}_{N} \;\wedge\; nhip \neq dip \;\Rightarrow\; \xi^{ip}_{N}(\mathtt{rt}) \sqsubset_{dip} \xi^{nhip}_{N}(\mathtt{rt}) \;, \tag{21}$$

where $N$ is a reachable network expression and $nhip := \mathtt{nhop}^{ip}_{N}(dip)$ is the IP address of the next hop.

*Proof.* As before, we first check the initial states of our transition system and then check all locations in Processes 1–7 where a routing table might be changed. For an initial network expression, the invariant holds since all routing tables are empty. Adding precursors to $\xi^{ip}_{N}(\mathtt{rt})$ or $\xi^{nhip}_{N}(\mathtt{rt})$ does not affect the invariant, since the invariant does not depend on precursors, so it suffices to examine all modifications to $\xi^{ip}_{N}(\mathtt{rt})$ or $\xi^{nhip}_{N}(\mathtt{rt})$ using `update` or `invalidate`. Moreover, without loss of generality we restrict attention to those applications of `update` or `invalidate` that actually modify the entry for *dip*, beyond its precursors; if `update` only adds some precursors in the routing table, the invariant—which is assumed to hold before—is maintained.

Applications of `invalidate` to either $\xi^{ip}_{N}(\mathtt{rt})$ or $\xi^{nhip}_{N}(\mathtt{rt})$ lead to a network state in which the antecedent of (21) is not satisfied. Now consider an application of `update` to $\xi^{nhip}_{N}(\mathtt{rt})$. We restrict attention to the case that the antecedent of (21) is satisfied right after the update, so that right before the update we have $dip \in \mathtt{vD}^{ip}_{N} \wedge nhip \neq dip$. In the special case that $\mathtt{sqn}^{nhip}_{N}(dip) = 0$ right before the update, we have $\mathtt{nsqn}^{nhip}_{N}(dip) = 0$ and thus $\mathtt{nsqn}^{ip}_{N}(dip) = 0$ by Invariant (20). Since $\mathtt{flag}^{ip}_{N}(dip) = \mathtt{val}$, this implies $\mathtt{sqn}^{ip}_{N}(dip) = 0$. By Proposition 7.12(d) we have $nhip = dip$, contradicting our assumptions. It follows that right before the update $\mathtt{sqn}^{nhip}_{N}(dip) > 0$, and hence $\mathtt{nsqn}^{nhip}_{N}(dip) < \mathtt{sqn}^{nhip}_{N}(dip)$.

An application of `update` to $\xi^{nhip}_{N}(\mathtt{rt})$ that changes $\mathtt{flag}^{nhip}_{N}(dip)$ from `inv` to `val` cannot decrease the sequence number of the entry to *dip* and hence strictly increases its net sequence number. Before the `update` we had $\mathtt{nsqn}^{ip}_{N}(dip) \leq \mathtt{nsqn}^{nhip}_{N}(dip)$ by Invariant (20), so afterwards we must have $\mathtt{nsqn}^{ip}_{N}(dip) < \mathtt{nsqn}^{nhip}_{N}(dip)$, and hence $\xi^{ip}_{N}(\mathtt{rt}) \sqsubset_{dip} \xi^{nhip}_{N}(\mathtt{rt})$. An `update` to $\xi^{nhip}_{N}(\mathtt{rt})$ that maintains $\mathtt{flag}^{nhip}_{N}(dip) = \mathtt{val}$ can only increase the quality of the entry to *dip* (cf. Theorem 7.27), and hence maintains Invariant (21).

It remains to examine the `update`s to $\xi^{ip}_{N}(\mathtt{rt})$.

**Pro. 1, Lines 10, 14, 18:** The entry $\xi(\mathtt{sip}, 0, \mathtt{unk}, \mathtt{val}, 1, \mathtt{sip}, \emptyset)$ is used for the update; its destination is $dip := \xi(\mathtt{sip})$. Since $dip = \mathtt{nhop}^{ip}_{N}(dip) = nhip$, the antecedent of the invariant to be proven is not satisfied.

**Pro. 4, Line 4:** We assume that the entry $\xi(\mathtt{oip}, \mathtt{osn}, \mathtt{kno}, \mathtt{val}, \mathtt{hops}+1, \mathtt{sip}, *)$ is inserted into $\xi(\mathtt{rt})$. So $dip := \xi(\mathtt{oip})$, $nhip := \xi(\mathtt{sip})$, $\mathtt{nsqn}^{ip}_{N}(dip) := \xi(\mathtt{osn})$ and $\mathtt{dhops}^{ip}_{N}(dip) := \xi(\mathtt{hops}) + 1$. This information is distilled from a received route request message (cf. Lines 1 and 8 of Pro. 1). By Proposition 7.1 this message was sent before, say in state $N^{\dagger}$; by Proposition 7.8 the sender of this message is $\xi(\mathtt{sip})$.

By Invariant (13), with $ip_c := \xi(\mathtt{sip}) = nhip$, $oip_c := \xi(\mathtt{oip}) = dip$, $osn_c := \xi(\mathtt{osn})$ and $hops_c := \xi(\mathtt{hops})$, and using that $ip_c = nhip \neq dip = oip_c$, we get that

$$\mathtt{sqn}^{nhip}_{N^{\dagger}}(dip) \;=\; \mathtt{sqn}^{ip_c}_{N^{\dagger}}(oip_c) \;>\; osn_c \;=\; \xi(\mathtt{osn}) \;, \text{ or}$$

$$\mathtt{sqn}^{nhip}_{N^{\dagger}}(dip) \;=\; \xi(\mathtt{osn}) \;\wedge\; \mathtt{dhops}^{nhip}_{N^{\dagger}}(dip) \leq \xi(\mathtt{hops}) \;\wedge\; \mathtt{flag}^{nhip}_{N^{\dagger}}(dip) \;=\; \mathtt{val} \;.$$

We first assume that the first line holds. Then, by the assumption $dip \in \mathtt{vD}(\xi_N^{nhip}(\mathtt{rt}))$, the definition of net sequence numbers, and Proposition 7.6,

$$\mathtt{nsqn}_N^{nhip}(dip) = \mathtt{sqn}_N^{nhip}(dip) \geq \mathtt{sqn}_{N^\dagger}^{nhip}(dip) > \xi(\mathtt{osn}) = \mathtt{nsqn}_N^{ip}(dip)\ .$$

and hence $\xi_N^{ip}(\mathtt{rt}) \sqsubset_{dip} \xi_N^{nhip}(\mathtt{rt})$.

We now assume the second line to be valid. From this we conclude

$$\mathtt{nsqn}_{N^\dagger}^{nhip}(dip) = \mathtt{sqn}_{N^\dagger}^{nhip}(dip) = \xi(\mathtt{osn}) = \mathtt{nsqn}_N^{ip}(dip)\ .$$

Moreover, $\quad \mathtt{dhops}_{N^\dagger}^{nhip}(dip) \ \leq\ \xi(\mathtt{hops}) \ <\ \xi(\mathtt{hops})+1 \ =\ \mathtt{dhops}_N^{ip}(dip)\ .$

Hence $\xi_N^{ip}(\mathtt{rt}) \sqsubset_{dip} \xi_{N^\dagger}^{nhip}(\mathtt{rt})$. Together with Theorem 7.27 and the transitivity of $\sqsubseteq_{dip}$ this yields $\xi_N^{ip}(\mathtt{rt}) \sqsubset_{dip} \xi_N^{nhip}(\mathtt{rt})$.

**Pro. 5, Line 2:** The update is similar to the one of Pro. 4, Line 4. The only difference is that the information stems from an incoming RREP message and that a routing table entry to $\xi(\mathtt{dip})$ (instead of $\xi(\mathtt{oip})$) is established. Therefore, the proof is similar to the one of Pro. 4, Line 4; instead of Invariant (13) we use Invariant (14). □

## 7.6 Loop Freedom

The "naïve" notion of loop freedom is a term that informally means that "a packet never goes round in cycles without (at some point) being delivered". This dynamic definition is not only hard to formalise, it is also too restrictive a requirement for AODV. There are situations where packets are sent in cycles, but which are not considered harmful. This can happen when the topology keeps changing.

#### Loops within a Topology that Changes Forever

The following example shows that data packets can travel in cycles without being delivered. However, it is our belief that this example is not a loop that should be avoided by a good routing protocol.

The example consists of a "core" network built up by the nodes *s*, *a* and *b*, which form a ring topology. All links between these three nodes are stable. Node *d* is part of the network and keeps moving around the core such that it is always connected to only one of the nodes at a time; see Figure 4. In the initial state *d* is connected to *a* and node *s* wants to send a data packet to *d*.

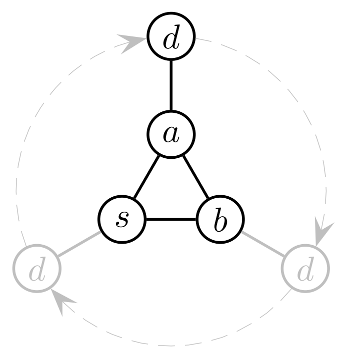



Figure 4: Loop considered harmless

Since *s* does not have a routing table entry to *d*, it generates and broadcasts a RREQ message, which is received by *d* via node *a* (Figure 5 (a)).[35] In Figure 5(b), *d* sends a RREP message back to *s* (via *a*).

Since *s* now has a routing table entry for *d*, it sends the data packet to *a*—the next hop on the route to *d* (Figure 5(c)). In the meantime, node *d* has moved away from node *a*, and is now connected to node *b*. In Figure 5(d), node *a* detects the link break (e.g. while trying to send the data packet from node *s* to node *d*), and proceeds to do a local repair.[36] The data packet is buffered while waiting for the local repair

[35] The "snapshots" in this figure are slightly different from the ones presented before; in Figures 2 and 3 (as well as in 6), each snapshot presents the system in a state after an AODV control message or data packet has been received and already partly handled (e.g., the routing tables are updated). Here, the subfigures describe the system when each message has either been handled completely or has been received and stored in the buffer, but *not* yet handled.

[36] Even though we do not model the local repair feature, we use it here to illustrate scenarios where data packets can travel in cycles. It is easy to modify the presented example into one without local repair; however the modified example would require error handling and hence would be longer.

process to complete. To repair the link break, node *a* generates a new RREQ message, which is received by node *d* via node *b*.

In Figure 5(e), node *d* sends a RREP message back to node *a* (via node *b*), thus enabling node *a* to repair its routing table entry to node *d*.[37]

With a valid entry in its routing table for node *d*, node *a* can now send the buffered data packet to node *b*—the next hop on the route towards node *d* (Figure 5(f)). If node *d* now moves away from node *b* and into the transmission range of node *s*, the events of Parts (d)–(f) will repeat. This will continue as long as the destination node *d* keeps moving "around" nodes *s*, *a* and *b*. The data packet will then travel through a loop *a*–*b*–*s*–*a*. Though this is a loop, it is not undesirable behaviour since the data packet is always travelling on the shortest path towards node *d*; it is due to the movement of node *d* that the data packet is never delivered. □

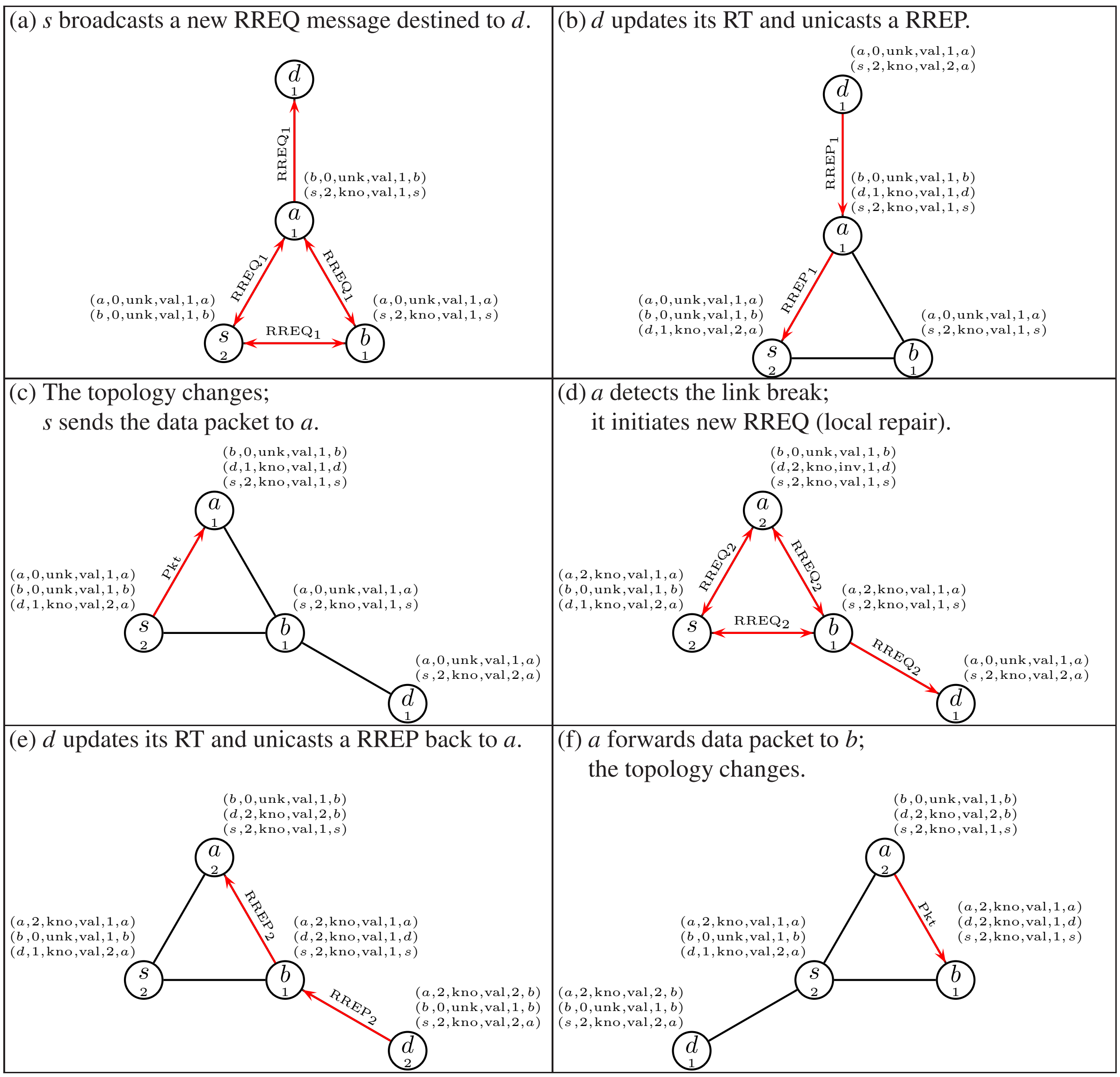



Figure 5: A "dynamic loop"

[37] We simplify the description of the local repair process here. Further details are available in the RFC [79].

Due to this dynamic behaviour, the sense of loop freedom is much better captured by a static invariant, saying that at any given time the collective routing tables of the nodes do not admit a loop. Such a requirement does not rule out the dynamic loop exemplified above. However, in situations where the topology remains stable sufficiently long it does guarantee that packets will not keep going around in cycles. In the above example the packet would actually be delivered as soon as the topology stops changing—it does not matter when.

To this end we define the *routing graph* of network expression $N$ with respect to destination $dip$ by $\mathscr{R}_N(dip) := (\mathbf{IP}, E)$, where all nodes of the network form the set of vertices and there is an arc $(ip, ip') \in E$ iff $ip \neq dip$ and $(dip\,,*\,,*\,,\texttt{val}\,,*\,,ip'\,,*) \in \xi_N^{ip}(\texttt{rt})$.

An arc in a routing graph states that $ip'$ is the next hop on a valid route to $dip$ known by $ip$; a path in a routing graph describes a route towards $dip$ discovered by AODV. We say that a network expression $N$ is *loop free* if the corresponding routing graphs $\mathscr{R}_N(dip)$ are loop free, for all $dip \in \mathbf{IP}$. A routing protocol, such as AODV, is *loop free* iff all reachable network expressions are loop free.

Using this definition of a routing graph, Theorem 7.30 states that along a path towards a destination $dip$ in the routing graph of a reachable network expression $N$, until it reaches either $dip$ or a node with an invalided routing table entry to dip, the quality of the routing table entries for $dip$ is strictly increasing. From this, we can immediately conclude

**Theorem 7.31** The specification of AODV given in Section 6 is loop free.

*Proof.* If there were a loop in a routing graph $\mathscr{R}_N(dip)$, then for any edge $(ip, nhip)$ on that loop one has $\xi_N^{ip}(\texttt{rt}) \sqsubset_{dip} \xi_N^{nhip}(\texttt{rt})$, by Theorem 7.30. Thus, by transitivity of $\sqsubset_{dip}$, one has $\xi_N^{ip}(\texttt{rt}) \sqsubset_{dip} \xi_N^{ip}(\texttt{rt})$, which contradicts the irreflexivity of $\sqsubset_{dip}$ (cf. Corollary 7.29). □

According to Theorem 7.31 any route to a destination $dip$ established by AODV—i.e. a path in $\mathscr{R}_N(dip)$—ends after finitely many hops. There are three possible ways in which it could end:

(1) by reaching the destination,

(2) by reaching a node with an invalid entry to $dip$, or

(3) by reaching a node without any entry to $dip$.

(1) is what AODV attempts to accomplish, whereas (2) is an unavoidable due to link breaks in a dynamic topology. It follows directly from Proposition 7.28 that (3) can never occur.

### 7.7 Route Correctness

The creation of a routing table entry at node $ip$ for destination $dip$ is no guarantee that a route from $ip$ to $dip$ actually exists. The entry is created based on information gathered from messages received in the past, and at any time link breaks may occur. The best one could require of a protocol like AODV is that routing table entries are based on information that was valid at some point in the past. This is the essence of what we call *route correctness*.

We define a *history* of an AODV-like protocol as a sequence $H = N_0 N_1 \ldots N_k$ of network expressions, where $N_0$ is an initial state of the protocol, and for $1 \leq i \leq k$ there is a transition $N_{i-1} \xrightarrow{\ell} N_i$; we call $H$ a history *of* the state $N_k$. The *connectivity graph* of a history $H$ is $\mathscr{C}_H := (\mathbf{IP}, E)$, where the nodes of the network form the set of vertices and there is an arc $(ip, ip') \in E$ iff $ip' \in R_{N_i}^{ip}$ for some $0 \leq i \leq k$, i.e. if at some point during that history node $ip'$ was in transmission range of $ip$. A protocol satisfies the property *route correctness* if for every history $H$ of a reachable state $N$ and for every routing table entry $(dip\,,*\,,*\,,*\,,hops\,,nhip\,,*) \in \xi_N^{ip}(\texttt{rt})$ there is a path $ip \to nhip \to \cdots \to dip$ in $\mathscr{C}_H$ from $ip$ to $dip$ with $hops$ hops and (if $hops > 0$) next hop $nhip$.[38]

[38] A path with 0 hops consists of a single node only.

**Theorem 7.32** Let $H$ be a history of a network state $N$.

(a) For each routing table entry $(dip,*,*,*,hops,nhip,*)\in\xi_N^{ip}(\texttt{rt})$ there is a path $ip\rightarrow nhip\rightarrow\cdots\rightarrow dip$ in $\mathscr{C}_H$ from $ip$ to $dip$ with $hops$ hops and (if $hops>0$) next hop $nhip$.

(b) For each route request sent in state $N$ there is a corresponding path in the connectivity graph of $H$.

$$\begin{array}{ll} & N \xrightarrow{R:\textbf{*cast}(\texttt{rreq}(hops_c,*,*,*,*,oip_c,*,ip_c))}_{ip} N' \\ \Rightarrow & \text{there is a path } ip_c\rightarrow\cdots\rightarrow oip_c \text{ in } \mathscr{C}_H \text{ from } ip_c \text{ to } oip_c \text{ with } hops_c \text{ hops} \end{array} \tag{22}$$

(c) For each route reply sent in state $N$ there is a corresponding path in the connectivity graph of $H$.

$$\begin{array}{ll} & N \xrightarrow{R:\textbf{*cast}(\texttt{rrep}(hops_c,dip_c,*,*,ip_c))}_{ip} N' \\ \Rightarrow & \text{there is a path } ip_c\rightarrow\cdots\rightarrow dip_c \text{ in } \mathscr{C}_H \text{ from } ip_c \text{ to } dip_c \text{ with } hops_c \text{ hops} \end{array} \tag{23}$$

*Proof.* In the course of running the protocol, the set of edges $E$ in the connectivity graph $\mathscr{C}_H$ only increases, so the properties are invariants. We prove them by simultaneous induction.

(a) In an initial state the invariant is satisfied because the routing tables are empty. Since routing table entries can never be removed, and the functions `addpreRT` and `invalidate` do not affect *hops* and *nhip*, it suffices to check all application calls of `update`. In each case, if the update does not change the routing table entry beyond its precursors (the last clause of `update`), the invariant is trivially preserved; hence we examine the cases that an update actually occurs.

**Pro. 1, Lines 10, 14, 18:** The update changes the entry into $\xi(\texttt{sip},*,\texttt{unk},\texttt{val},1,\texttt{sip},*)$; hence $hops=1$ and $nhip=dip:=\xi(\texttt{sip})$. The value $\xi(\texttt{sip})$ stems through Lines 8, 12 or 16 of Pro. 1 from an incoming AODV control message. By Proposition 7.1 this message was sent before, say in state $N^\dagger$; by Proposition 7.8 the sender of this message is $\xi(\texttt{sip})=nhip$. Since in state $N^\dagger$ the message must have reached the queue of incoming messages of node $ip$, it must be that $ip\in R_{N^\dagger}^{nhip}$. In our formalisation of AWN the connectivity graph is always symmetric: $nhip\in R_{N^\dagger}^{ip}$ iff $ip\in R_{N^\dagger}^{nhip}$. It follows that $(ip,nhip)\in E$, so there is a 1-hop path in $\mathscr{C}_H$ from $ip$ to $dip$.

**Pro. 4, Line 4:** Here $dip:=\xi(\texttt{oip})$, $hops:=\xi(\texttt{hops})+1$ and $nhip:=\xi(\texttt{sip})$. These values stem from an incoming RREQ message, which must have been sent beforehand, say in state $N^\dagger$. As in the previous case we obtain $(ip,nhip)\in E$. By Invariant (22), with $oip_c:=\xi(\texttt{oip})=dip$, $hops_c:=\xi(\texttt{hops})$ and $ip_c:=\xi(\texttt{sip})=nhip$, there is a path $nhip\rightarrow\cdots\rightarrow dip$ in $\mathscr{C}_H$ from $ip_c$ to $oip_c$ with $hops_c$ hops. It follows that there is a path $ip\rightarrow nhip\rightarrow\cdots\rightarrow dip$ in $\mathscr{C}_H$ from $ip$ to $dip$ with $hops$ hops and next hop $nhip$.

**Pro. 5, Line 2:** Here $dip:=\xi(\texttt{dip})$, $hops:=\xi(\texttt{hops})+1$ and $nhip:=\xi(\texttt{sip})$. The reasoning is exactly as in the previous case, except that we deal with an incoming RREP message and use Invariant (23).

(b) We check all occasions where a route request is sent.

**Pro. 1, Line 39:** A new route request is initiated with $ip_c=oip_c:=\xi(\texttt{ip})=ip$ and $hops_c:=0$. Indeed there is a path in $\mathscr{C}_H$ from $ip_c$ to $oip_c$ with 0 hops.

**Pro. 4, Line 36:** The broadcast message has the form

$$\xi(\texttt{rreq}(\texttt{hops}+1,\texttt{rreqid},\texttt{dip},\max(\texttt{sqn}(\texttt{rt},\texttt{dip}),\texttt{dsn}),\texttt{dsk},\texttt{oip},\texttt{osn},\texttt{ip}))\ .$$

Hence $hops_c:=\xi(\texttt{hops})+1$, $oip_c:=\xi(\texttt{oip})$ and $ip_c:=\xi(\texttt{ip})=ip$. The values $\xi(\texttt{hops})$ and $\xi(\texttt{oip})$ stem through Line 8 of Pro. 1 from an incoming RREQ message of the form

$$\xi(\texttt{rreq}(\texttt{hops},\texttt{rreqid},\texttt{dip},\texttt{dsn},\texttt{dsk},\texttt{oip},\texttt{osn},\texttt{sip}))\ .$$

By Proposition 7.1 this message was sent before, say in state $N^{\dagger}$; by Proposition 7.8 the sender of this message is $sip := \xi(\mathtt{sip})$. By induction, using Invariant (22), there is a path $sip \to \cdots \to oip_c$ in $\mathscr{C}_{H^{\dagger}} \subseteq \mathscr{C}_H$ from $sip$ to $oip_c$ with $\xi(\mathtt{hops})$ hops. It remains to show that there is a 1-hop path from $ip$ to $sip$. In state $N^{\dagger}$ the message sent by $sip$ must have reached the queue of incoming messages of node $ip$, and therefore $ip$ was in transmission range of $sip$, i.e., $ip \in R^{sip}_{N^{\dagger}}$. Since the connectivity graph of AWN is always symmetric (cf. Tables 3 and 4, and explanation on Page 15), $ip \in R^{sip}_{N^{\dagger}}$ holds as well. Hence it follows that $(ip, sip) \in E$.

(c) We check all occasions where a route reply is sent.

**Pro. 4, Line 10:** A new route reply with $hops_c := 0$ and $ip_c := \xi(\mathtt{ip}) = ip$ is initiated. Moreover, by Line 7, $dip_c := \xi(\mathtt{dip}) = \xi(\mathtt{ip}) = ip$. Thus there is a path in $\mathscr{C}_H$ from $ip_c$ to $dip_c$ with 0 hops.

**Pro. 4, Line 25:** We have $dip_c := \xi(\mathtt{dip})$, $hops_c := \mathtt{dhops}^{ip}_{N}(dip_c)$ and $ip_c := \xi(\mathtt{ip}) = ip$. By Line 20 there is a routing table entry $(dip_c, *, *, *, hops_c, *, *) \in \xi^{ip}_{N}(\mathtt{rt})$. Hence by Invariant (a), which we may assume to hold when using simultaneous induction, there is a path $ip \to \cdots \to dip_c$ in $\mathscr{C}_H$ from $ip = ip_c$ to $dip_c$ with $hops_c$ hops.

**Pro. 5, Line 13:** The RREP message has the form $\xi(\mathtt{rrep}(\mathtt{hops}+1, \mathtt{dip}, \mathtt{dsn}, \mathtt{oip}, \mathtt{ip}))$ and the proof goes exactly as for Pro. 4, Line 36 of Part (b), by using $dip_c := \xi(\mathtt{dip})$ instead of $oip_c := \xi(\mathtt{oip})$, and an incoming RREP message instead of an incoming RREQ message. □

Theorem 7.32(a) says that the AODV protocol is route correct. For the proof it is essential that we use the version of AWN were a node $ip'$ is in the range of node $ip$, meaning that $ip'$ can receive messages sent by $ip$, if and only if $ip$ is in the range of $ip'$. If AWN is modified so as to allow asymmetric connectivity graphs, as indicated in Section 4.3, it is trivial to construct a 2-node counterexample to route correctness.

A stronger concept of route correctness requires that for each $(dip, *, *, *, hops, nhip, *) \in \xi^{ip}_{N}(\mathtt{rt})$

- either $hops = 0$ and $dip = ip$,
- or $hops = 1$ and $dip = nhip$ and there is a $N^{\dagger}$ in $H$ such that $nhip \in R^{ip}_{N^{\dagger}}$,
- or $hops > 1$ and there is a $N^{\dagger}$ in $H$ with $nhip \in R^{ip}_{N^{\dagger}}$ and $(dip, *, *, \mathtt{val}, hops-1, *, *) \in \xi^{nhip}_{N^{\dagger}}(\mathtt{rt})$.

It turns out that this stronger form of route correctness does not hold for AODV.

## 7.8 Further Properties

We conclude this section by proving a few more properties of AODV; these will be used later in the paper and/or shed some light on how AODV operates.

### 7.8.1 Queues

**Proposition 7.33** A node $ip \in \mathbf{IP}$ never queues data packets intended for itself.

$$ip \notin \mathtt{qD}(\xi^{ip}_{N}(\mathtt{store})) \tag{24}$$

*Proof.* We first show the claim for the initial states; afterwards we go through our specification (step by step) and look at all locations where the store of an arbitrary node $ip \in \mathbf{IP}$ can be changed.

In an initial network expression all sets of queued data are empty. There is only one place where a new destination is added to `store`, namely Pro. 2, Line 4. Here, $\xi(\mathtt{dip})$ is added as new queued destination. However, Line 3 shows that $\xi(\mathtt{dip}) \neq \xi(\mathtt{ip})$. □

### 7.8.2 Route Requests and RREQ IDs

A transition $N \xrightarrow{R:*\mathbf{cast}(\mathtt{rreq}(*,rreqid,dip,*,*,oip,osn,*))} N'$ that stems from Pro. 1, Line 39 marks the initiation of a *new* route request. Each such transition that stems from Pro. 4, Line 36, which is the only alternative, marks the *forwarding* of a route request. In this case, the variables `rreqid`, `dip`, `oip` and `osn`, which supply the values *rreqid*, *dip*, *oip* and *osn*, get these values in Pro. 1, Line 8; nowhere else is the value of these variables set or changed. Hence the values mentioned are copied directly from another RREQ message, read in Pro. 1, Line 1. By Proposition 7.1(a), this message has to be sent before; and this is the message that is forwarded. Now a *route request* can be defined as an equivalence class of route request messages (transitions in our operational semantics), namely by considering a forwarded RREQ message to belong to the same route request as the message being forwarded.

**Proposition 7.34** A route request is uniquely determined by the pair $(oip,rreqid)$ of the originator IP address and its route request identifier.

*Proof.* As argued above, each forwarded RREQ message carries the same pair $(oip,rreqid)$ as the message being forwarded. It remains to show that each new route request is initiated with a different pair $(oip,rreqid)$.

The broadcast message id is determined by the function `nrreqid`. At the initial state the function `nrreqid` will return 1, since $\mathtt{rreqs}(\mathtt{ip})$ is empty. If a new id—determined by the function `nrreqid`—is used by a node *ip*, the id is also added to ${\xi^{ip}}_N(\mathtt{rreqs})$ (Pro. 1, Line 38). By Proposition 7.5, this id will never be deleted from $\xi(\mathtt{rreqs})$. Therefore, whenever the function `nrreqid` is called afterwards by the same node, the return value will be strictly higher. In fact it will be increased by 1 each time a new request is sent. It follows that for each route request the pair $(oip,rreqid)$ is unique. □

This pair $(oip,rreqid)$ is stored in the local variables `rreqs` maintained by each node that encounters the route request.

The following proposition paves the way for the conclusion that the role of the component *rreqid* in route request messages could just as well be taken over by the existing component *osn* of these messages.

**Proposition 7.35**

(*a*) A node's sequence number is greater than its last used RREQ id, i.e.,

$$\xi_N^{ip}(\mathtt{sn}) > \mathtt{rreqid}_N^{ip} \,,$$

where $\mathtt{rreqid}_N^{ip} := \max\{n \mid (ip,n) \in \xi_N^{ip}(\mathtt{rreqs})\}$and the maximum of the empty set is defined to be 0.

(*b*) A route request is uniquely determined by the combination of *osn* and *oip*.

*Proof.*

(*a*) In the initial state $\xi_N^{ip}(\mathtt{sn}) = 1$ and $\mathtt{rreqid}_N^{ip} = 0$. Both numbers are increased by 1 if a route request is initiated; the `sn` is increased first. $\mathtt{rreqid}_N^{ip}$ is not changed elsewhere; however, when generating a route reply $\xi_N^{ip}(\mathtt{sn})$ might be increased (cf. Pro. 4, Line 8).

(*b*) When a route request is initiated, the value of the component *osn* in the initial RREQ message equals the (newly incremented) current value of `sn` maintained by the initiating node, just like the component *rreqid* in the initial RREQ message equals the (newly incremented) current value of $\mathtt{rreqid}_N^{ip}$ of the initiating node. Now the statement follows since the value of `sn` is increased whenever a route request is initiated and *osn* and *oip* are passed on unchanged when forwarding a route request, just like *rreqid* and *oip*. □

The following proposition states three properties about sending a route request.

**Proposition 7.36**

(a) If a route request is sent by a node $ip_c \in \mathbf{IP}$, the sender has stored the unique pair of the originator's IP address and the request id.

$$N \xrightarrow{R:\textbf{*cast}(\texttt{rreq}(*,rreqid_c,*,*,*,oip_c,*,ip_c))}_{ip} N' \;\Rightarrow\; (oip_c, rreqid_c) \in \xi_N^{ip_c}(\texttt{rreqs}) \tag{25}$$

(b) If a route request is sent, the originator has stored the unique pair of the originator's IP address and the request id.

$$N \xrightarrow{R:\textbf{*cast}(\texttt{rreq}(*,rreqid_c,*,*,*,oip_c,*,*))}_{ip} N' \;\Rightarrow\; (oip_c, rreqid_c) \in \xi_N^{oip_c}(\texttt{rreqs}) \tag{26}$$

(c) The sequence number of an originator appearing in a route request can never be greater than the originator's own sequence number.

$$N \xrightarrow{R:\textbf{*cast}(\texttt{rreq}(*,*,*,*,*,oip_c,osn_c,*))}_{ip} N' \;\Rightarrow\; osn_c \leq \xi_N^{oip_c}(\texttt{sn}) \tag{27}$$

*Proof.* We have to check that the consequent holds whenever a route request is sent. In all the processes there are only two locations where this happens, namely Pro. 1, Line 39 and Pro. 4, Line 36.

(a) **Pro. 1, Line 39:** A request with content $\xi(*,\texttt{rreqid},*,*,*,\texttt{ip},*,\texttt{ip})$ is sent. So $ip_c := \xi(\texttt{ip})$, $oip_c := \xi(\texttt{ip})$ and $rreqid_c := \xi(\texttt{rreqid})$. Hence, using (3), $\xi_N^{ip_c} = \xi_N^{ip} = \xi$. Right before broadcasting the request, $(\xi(\texttt{ip}), \xi(\texttt{rreqid}))$ is added to the set $\xi(\texttt{rreqs})$.

**Pro. 4, Line 36:** The information $(\xi(\texttt{oip}), \xi(\texttt{rreqid}))$ is added to $\xi(\texttt{rreqs})$ at Line 5. Moreover, the set of handled requests $\xi(\texttt{rreqs})$ as well as the values of $\texttt{oip}$ and $\texttt{rreqid}$ do not change between Line 5 and 36. Again $(oip_c, rreqid_c) = (\xi(\texttt{oip}), \xi(\texttt{rreqid})) \in \xi(\texttt{rreqs}) = \xi_N^{ip_c}(\texttt{rreqs})$.

(b) **Pro. 1, Line 39:** A request with content $\xi(*,\texttt{rreqid},*,*,*,\texttt{ip},*,\texttt{ip})$ is sent. So $oip_c := \xi(\texttt{ip})$ and hence, by (3), $\xi_N^{oip_c} = \xi$. Moreover, $rreqid_c := \xi(\texttt{rreqid})$. Right before broadcasting the request, the pair $(\xi(\texttt{ip}), \xi(\texttt{rreqid}))$ is added to the set $\xi(\texttt{rreqs})$.

**Pro. 4, Line 36:** A request with content $\xi(*,\texttt{rreqid},*,*,*,\texttt{oip},*,*)$ is sent. The values of the variables $\texttt{rreqid}$ and $\texttt{oip}$ do not change in Pro. 4; they stem, through Line 8 of Pro. 1, from an incoming RREQ message (Pro. 1, Line 1). Now the claim follows immediately from the fact the each RREQ message received, has been sent by some node (Proposition 7.1(a)), and induction on reachability.

(c) **Pro. 1, Line 39:** The sender is the originator, so $oip_c := \xi(\texttt{ip}) = ip$ and $osn_c := \xi(\texttt{sn})$. By (3), $\xi_N^{oip_c} = \xi$, which immediately implies $osn_c := \xi_N^{oip_c}(\texttt{sn})$.

**Pro. 4, Line 36:** Here $oip_c := \xi(\texttt{oip})$ and $osn_c := \xi(\texttt{osn})$. The values of the variables $\texttt{oip}$ and $\texttt{osn}$ do not change in Pro. 4; they stem from Line 8 of Pro. 1. By Proposition 7.1(a), a transition labelled $R:\textbf{*cast}(\texttt{rreq}(*,*,*,*,*,oip_c,osn_c,*))$ must have occurred before, say in state $N^\dagger$. Thus, by induction and Proposition 7.2, $osn_c \leq \xi_{N^\dagger}^{oip_c}(\texttt{sn}) \leq \xi_N^{oip_c}(\texttt{sn})$. □

### 7.8.3 Routing Table Entries

**Proposition 7.37**

(a) The sequence number of a destination appearing in a route reply can never be greater than the destination's own sequence number.

$$N \xrightarrow{R:\textbf{*cast}(\texttt{rrep}(*,dip_c,dsn_c,*,*))}_{ip} N' \;\Rightarrow\; dsn_c \leq \xi_N^{dip_c}(\texttt{sn}) \tag{28}$$

(b) A known destination sequence number of a valid routing table entry can never be greater than the destination's own sequence number.

$$(dip, dsn, \mathtt{kno}, \mathtt{val}, *, *, *) \in \xi_N^{ip}(\mathtt{rt}) \Rightarrow dsn \leq \xi_N^{dip}(\mathtt{sn}) \tag{29}$$

*Proof.* We apply simultaneous induction to prove these invariants.

(a) We have to check that the consequent holds whenever a route reply is sent.

**Pro. 4, Line 10:** A route reply with sequence number $dsn_c := \xi_N^{ip}(\mathtt{sn})$ is initiated. Moreover, by Line 7, $dip_c := \xi_N^{ip}(\mathtt{dip}) = \xi_N^{ip}(\mathtt{ip}) = ip$. So $dsn_c = \xi_N^{dip_c}(\mathtt{sn})$.

**Pro. 4, Line 25:** A route reply with $dip_c := \xi_N^{ip}(\mathtt{dip})$ and $dsn_c := \xi_N^{ip}(\mathtt{sqn}(\mathtt{rt},\mathtt{dip})) = \mathtt{sqn}_N^{ip}(dip_c)$ is initiated. By Line 20 $dsn_c$ is a known sequence number, stemming from a valid entry for $dip_c$ in the routing table of $ip$. Hence by Invariant (29) $dsn_c = \mathtt{sqn}_N^{ip}(dip_c) \leq \xi_N^{dip_c}(\mathtt{sn})$.

**Pro. 5, Line 13:** The RREP message has the form $\xi_N^{ip}(\mathtt{rrep}(\mathtt{hops}+1,\mathtt{dip},\mathtt{dsn},\mathtt{oip},\mathtt{ip}))$. Hence $dip_c := \xi_N^{ip}(\mathtt{dip})$ and $dsn_c := \xi_N^{ip}(\mathtt{dsn})$. The values of the variables dip and dsn do not change in Pro. 5; they stem, through Line 12 of Pro. 1, from an incoming RREP message (Pro. 1, Line 1). By Proposition 7.1 this message was sent before, say by node *sip* in state $N^\dagger$. By induction we have $dsn_c \leq \xi_{N^\dagger}^{dip_c}(\mathtt{sn}) \leq \xi_N^{dip_c}(\mathtt{sn})$, where the latter inequality is by Proposition 7.2.

(b) We have to examine all application calls of `update`—entries resulting from a call of `invalidate` are not valid. Moreover, without loss of generality we restrict attention to those applications of `update` that actually modify the entry for *dip*, beyond its precursors; if `update` only adds some precursors in the routing table, the invariant—which is assumed to hold before—is maintained.

**Pro. 1, Lines 10, 14, 18:** These calls yield entries with `unknown` destination sequence numbers.

**Pro. 4, Line 4:** Here $dip := \xi(\mathtt{oip})$ and $dsn := \xi(\mathtt{osn})$. These values stem from an incoming RREQ message, which must have been sent beforehand, say in state $N^\dagger$. By Invariant (27), with $oip_c := \xi(\mathtt{oip}) = dip$ and $osn_c := \xi(\mathtt{osn}) = dsn$ we have $dsn \leq \xi_{N^\dagger}^{dip}(\mathtt{sn}) \leq \xi_N^{dip}(\mathtt{sn})$, where the latter inequality is by Proposition 7.2.

**Pro. 5, Line 2:** Here $dip := \xi(\mathtt{dip})$ and $dsn := \xi(\mathtt{dsn})$. These values stem from an incoming RREP message, which must have been sent beforehand, say in state $N^\dagger$. By Invariant (28), with $dip_c := \xi(\mathtt{dip}) = dip$ and $dsn_c := \xi(\mathtt{dsn}) = dsn$ we have $dsn \leq \xi_{N^\dagger}^{dip}(\mathtt{sn}) \leq \xi_N^{dip}(\mathtt{sn})$. □

**Proposition 7.38** Whenever *ip*'s routing table contains an entry with next hop *nhip*, it also contains an entry for *nhip*.

$$(*, *, *, *, *, nhip, *) \in \xi_N^{ip}(\mathtt{rt}) \Rightarrow (nhip, *, *, *, *, *, *) \in \xi_N^{ip}(\mathtt{rt}) \tag{30}$$

*Proof.* As usual we only consider function calls of `update` and assume that the update changes the routing table.

**Pro. 1, Lines 10, 14 and 18:** 1-hop connections are inserted into the routing table. By Invariant (9), the new entry has the form $(nhip, *, *, *, *, nhip, *)$. Therefore *ip* has an entry for *nhip*.

**Pro. 4, Line 4:** We assume that the entry $\xi(\mathtt{oip},\mathtt{osn},\mathtt{kno},\mathtt{val},\mathtt{hops}+1,\mathtt{sip},*)$ is inserted into $\xi(\mathtt{rt})$. So, $nhip := \xi(\mathtt{sip})$. This information is distilled from a received route request message (cf. Lines 1 and 8 of Pro. 1). Right after receiving the message, a route to $\xi(\mathtt{sip})$ is created or updated (Line 10 of Pro. 1); hence an entry for the next hop exists.

**Pro. 5, Line 2:** The update is similar to the one of Pro. 4, Line 4. The only difference is that the information stems from an incoming RREP message and that a routing table entry to $\xi(\mathtt{dip})$ (instead of $\xi(\mathtt{oip})$) is established. □

# 8 Interpreting the IETF RFC 3561 Specification

It is our belief that, up to the abstractions discussed in Section 3, the specification presented in the previous sections reflects precisely the intention and the meaning of the IETF RFC [79]. However, when formalising the AODV routing protocol, we came across some ambiguities, contradictions and unspecified behaviour in the RFC. This is also reflected by the fact that different implementations of AODV behave differently, although they all follow the lines of the RFC. Of course a specification "needs to be reasonably implementation independent"[39] and can leave some decisions to the software engineer; however it is our belief that any specification should be clear and unambiguous enough to guarantee the same behaviour when given to different developers. As we will show, this is not the case for AODV.

In this section, we discuss and formalise many of the problematic behaviours found, as well as their possible resolutions. An *interpretation* of the RFC is given by the allocation of a resolution to each of the ambiguities, contradictions and unspecified behaviours. Each reading, implementation, or formal analysis of AODV must pertain to one of its interpretations. The formal specification of AODV presented in Sections 5 and 6 constitutes one interpretation; the inventory of ambiguities and contradictions is formalised in Section 8.2 by specifying each resolution of each of the ambiguities and contradictions as a modification of this formal specification, typically involving a rewrite of a few lines of code only. We also show which interpretations give rise to routing loops or other unacceptable behaviour. Beforehand, in Section 8.1, we show how a decrease in the destination sequence number in a routing table entry generally gives rise to unacceptable protocol behaviour; later on we use this analysis to reject some of the possible interpretations of the RFC. After we have presented the ambiguities and their consequences, in Section 8.3 we briefly discuss five of the most popular implementations of AODV and demonstrate that the anomalies we discovered are not only theoretically driven, but *do* occur in practice. In particular, we show that three implementations can yield routing loops.

## 8.1 Decreasing Destination Sequence Numbers

In the RFC it is stated that a sequence number is

> "A monotonically increasing number maintained by each originating node."
> [79, Sect. 3]

Based on this, it is tempting to assume that also any destination sequence number within a routing table entry should be increased monotonically. In fact this is also stated in the RFC: The sequence number for a particular destination

> "is updated whenever a node receives new (i.e., not stale) information about the sequence number from RREQ, RREP, or RERR messages that may be received related to that destination. [...] In order to ascertain that information about a destination is not stale, the node compares its current numerical value for the sequence number with that obtained from the incoming AODV message. [...] If the result of subtracting the currently stored sequence number from the value of the incoming sequence number is less than zero, then the information related to that destination in the AODV message MUST be discarded, since that information is stale compared to the node's currently stored information." [79, Sect. 6.1]

This long-winded description simply says that all information distilled from any AODV control message that has a smaller sequence number for the destination under consideration, MUST be discarded. AODV should never decrease any destination sequence number, since this could create loops. We illustrate this by Figure 6.

[39] http://www.ietf.org/iesg/statement/pseudocode-guidelines.html

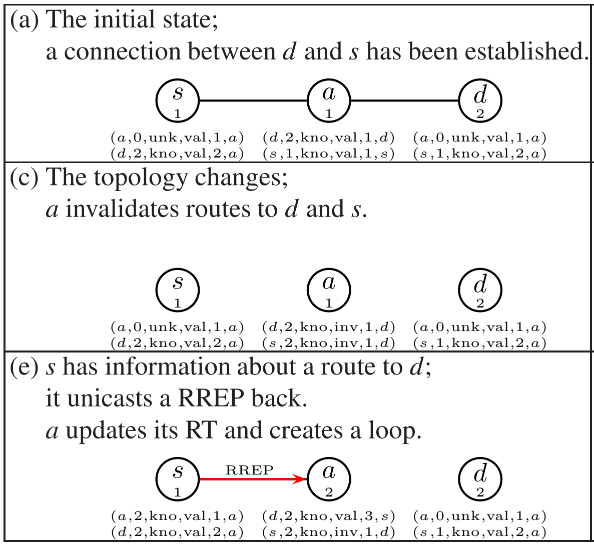



Figure 6: Creating a loop when sequence numbers are decreased

Assume a linear topology with three nodes. In the past, node *d* sent a request to establish a route to *s*. This RREQ message was answered by a RREP message of node *s*. After the route has been established, the network is in the state of Figure 6(a). In Part (b) we assume that the sequence number of the routing table entry to *d* of *a*'s routing table is decreased. Due to topology changes, node *a* then looses connection to all neighbours and invalidates its routing table entries (Part (c)). In particular, it increments all sequence numbers of the routing table and sets the status flags to `inv`. A possible error message sent by node *a* is not received by any other node. After the link between *a* and *s* has appeared again, node *a* wants to re-establish a route to *d*; it broadcasts a new RREQ message (Part (d)). The AODV control message generated is $\texttt{rreq}(0, \mathit{rreqid}, d, 2, \texttt{kno}, a, 2, a)$, where *rreqid* is the unique id of the message. Since node *s* has information about *d*, which is fresh enough, it generates the RREP message $\texttt{rrep}(2, d, 2, a, s)$ (Part (e)). Finally node *a* receives the reply and establishes a route to *d* via *a*. A loop has been created.

Further on, we will discuss how sequence numbers might be decreased when following the RFC literally or interpreting the RFC in a wrong way.

## 8.2 Interpreting the RFC

In the following we discuss some ambiguities in the RFC, each giving rise to up to 6 interpretations of AODV. To resolve ambiguities, we often looked into real implementation, such as AODV-UU [2], Kernel AODV [1] and AODV-UCSB [13] to determine the intended version of AODV. Additionally, we tried to derive unwanted behaviour from some of the possible interpretations.

### 8.2.1 Updating Routing Table Entries

One of the crucial aspects of AODV is the maintenance of routing tables. In this subsection we consider the update of routing table entries with new information. In our specification we used the function `update` to specify the desired behaviour. Unfortunately, the RFC specification only gives hints how to update routing table entries; an exact and precise definition is missing.

**Ambiguity 1: Updating the Unknown Sequence Number in Response to a Route Reply**

If a node receives a RREP message, it might have to update its routing table:

> "`the existing entry is updated only in the following circumstances:`
> `(i) the sequence number in the routing table is marked as invalid`[40]
> `[...]`" [79, Sect. 6.7]

In the same section it is also stated which actions occur if a route is updated:

> "`- the route is marked as active`[41] `[(val)],`
> `- the destination sequence number is marked as valid [(kno)],`
> `- the next hop in the route entry is assigned to be the node from which the RREP is received, [...]`
> `- the hop count is set to the value of the New Hop Count` [obtained by incrementing "`the hop count value in the RREP by one, to account for the new hop through the intermediate node`"],
> `[...]`
> `- and the [new] destination sequence number is the Destination Sequence Number in the RREP message.`" [79, Sect. 6.7]

To model this fragment of the RFC accurately, we define another update function, which adds a case to the original definition:

$$\texttt{update}^{\texttt{RREP}} : \texttt{RT} \times \texttt{R} \rightharpoonup \texttt{RT}$$

$$\texttt{update}^{\texttt{RREP}}(rt\,,r) := \begin{cases} nrt \cup \{nr\} & \text{if } \pi_1(r) \in \texttt{kD}(rt) \wedge \texttt{sqnf}(rt\,,r) = \texttt{unk} \\ \texttt{update}(rt\,,r) & \text{otherwise ,} \end{cases}$$

where, as in the definition of `update`, $nrt := rt - \{\sigma_{route}(rt\,,\pi_1(r))\}$ is the routing table without the current entry in the routing table for the destination of $r$ and $nr := \texttt{addpre}(r\,,\pi_7(\sigma_{route}(rt\,,\pi_1(r))))$ is identical to $r$ except that the precursors from the original entry are added. This function is now used in the process for RREP handling instead of `update`. In particular, Lines 1, 2 and 26 have to be changed in Pro. 5; all other processes (Pro. 1–Pro. 4 and Pro. 6, 7) remain unchanged and use the original version of `update`.

Using this fragment of the RFC, a sequence number of a routing table entry could be decreased. For example, an entry $(d,2,\texttt{unk},\texttt{val},1,d,*)$ is replaced by $(d,1,\texttt{kno},\texttt{val},n{+}1,a,*)$ if the reply has the form $\texttt{rrep}(n\,,d\,,1\,,*\,,a)$.[42] As indicated in Section 8.1, this in turn can create routing loops. This updating mechanism is in contradiction to the quote from [79, Sect. 6.1] in Section 8.1. In view of the undesirability of routing loops, the only way to resolve this contradiction is by ignoring (i) in [79, Sect. 6.7], the statement quoted at the beginning of this paragraph.

**Ambiguity 2: Updating with the Unknown Sequence Number**

Above we have discussed the update mechanism if a routing table entry with an unknown sequence number has to be updated. But what happens if the incoming AODV message carries an unknown number? This occurs regularly: whenever a node receives a forwarded AODV control message from a 1-hop neighbour (i.e., the neighbour is not the originator of the message), it creates a new or updates an existing routing table entry to that neighbour (cf. Lines 10, 14, 18 of Pro. 1). For example,

---

[40] The RFC [79] uses the term "invalid" in relation to sequence numbers as synonym for "unknown". We use "unknown" ($\texttt{unk} \in \texttt{K}$) only, in order to avoid confusion with the validity of the routing table entry in which the sequence number occurs ($\texttt{val},\texttt{inv} \in \texttt{F}$).

[41] The RFC uses the term "active" in relations to routes—actually referring to routing table entries—as a synonym for "valid".

[42] To see that this can actually happen, consider a variant of the example of Figure 3 in Section 2 in which node $s$ starts out with a routing table entry $(d,2,\texttt{kno},\texttt{inv},1,d,*)$, which may have resulted from a previous RREQ-RREP cycle, initiated by $s$, followed by an invalidation after the link between $s$ and $d$ broke down. Then in Figure 3(e) this entry is updated to $(d,2,\texttt{unk},\texttt{val},1,d,*)$, and in Figure 3(h) node $d$ sends a RREP message of the form $\texttt{rrep}(0\,,d\,,1\,,s\,,d)$.

> “[w]hen a node receives a RREQ, it first creates or updates a route to the previous hop without a valid sequence number” [79, Sect. 6.5]

In case a new routing table entry is created, the sequence number is set to zero and the sequence-number-status flag is set to unk to signify that the sequence number corresponding to the neighbour is unknown. But, what happens if the routing table entry $(a,2,\mathtt{kno},\mathtt{val},2,b,\emptyset)$ of node $d$ is updated by $(a,0,\mathtt{unk},\mathtt{val},1,a,\emptyset)$ as a consequence of the incoming RREQ message $\mathtt{rreq}(1,rreqid,x,7,\mathtt{kno},s,2,a)$, sent by node $a$? This situation is sketched in Figure 7.[43]

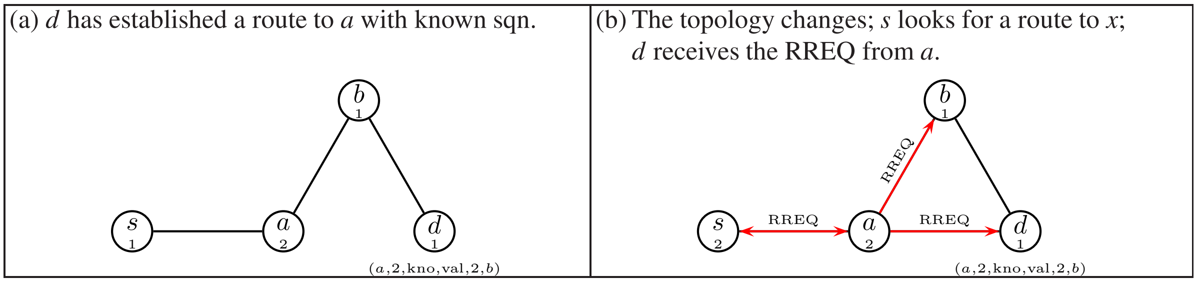



Figure 7: Updating routing table entries with the unknown sequence number

Following the RFC the routing table has to be updated. Unfortunately, it is not stated how the update is done. There are four reasonable updates—we call them (2a), (2b), (2c) and (2d) to label them as resolutions of Ambiguity 2:

(2a) $(a\,,2\,,\mathtt{kno}\,,\mathtt{val}\,,2\,,b\,,\emptyset)$: no update occurs (more precisely, only an update of the lifetime of the routing table entry happens; this is not modelled in this paper). To formalise this resolution, one skips the fifth option (out of 6) in the definition of update in Section 5.5.2: With this modification all our proofs in Section 7 remain valid, which yields loop freedom and route correctness of this alternative interpretation of AODV. It can be argued that the RFC rules out this resolution by including “or updates” in the quote above.

(2b) $(a\,,0\,,\mathtt{unk}\,,\mathtt{val}\,,1\,,a\,,\emptyset)$: all information is taken from the incoming AODV control message. To formalise this resolution, one changes the definition of update by replacing $nr'$ by $nr$. Since this can decrease sequence numbers, routing loops might occur. Hence this update must not be used.

(2c) $(a,2,\mathtt{unk},\mathtt{val},1,a,\emptyset)$: the information from the routing table and from the incoming AODV control message is merged, by taking only the destination sequence number from the existing routing table entry and all other information from the AODV control message; as usual the sets of precursors are combined. This is how our specification works. As we have shown in Section 7, no loops can occur. Moreover, node $d$ establishes an optimal route to $a$. In case $d$’s routing table would contain the tuple $(a,1,\mathtt{kno},\mathtt{val},1,a)$, the sequence-number-status flag would also be set to unk—this might be surprising, but it is consistent with the RFC.

(2d) $(a\,,2\,,\mathtt{kno}\,,\mathtt{val}\,,1\,,a\,,\emptyset)$: the information from the routing table and from the incoming AODV control message is merged, by taking the destination sequence number and the sequence-number-status flag from the existing routing table entry and all other information from the AODV control message; as usual the sets of precursors are combined. To formalise this resolution, one takes $nr' := (dip_{nr}\,,\pi_2(s)\,,\pi_3(s)\,,flag_{nr}\,,hops_{nr}\,,nhip_{nr}\,,pre_{nr})$ in the definition of update in Section 5.5.2. In the case where $\mathtt{sqn}(\mathtt{rt}\,,\pi_1(r)) = \pi_2(r)$ the routes $nr$ and $nr'$ are not equal anymore and hence the function is not well defined. To achieve well-definedness, we create mutual exclusive cases by

[43] Only the routing table entry under consideration is depicted.

using the fourth and fifth clause only if $\pi_3(r) = \texttt{kno}$. With this modification all results of Section 7, except for Proposition 7.37,[44] remain valid, with the same proofs, which yields loop freedom and route correctness of this alternative interpretation.

One could also mix Resolution (2a) with (2c) or (2d), for instance by applying (2c) for updates in response to a RREQ message and (2a) for updates in response to a RREP or RERR message. This could be justified by the location of the quote above in Sect. 6.5 of the RFC, which deals with processing RREQ messages only. Furthermore, as a variant of (2c) one could skip the update of Line 14 of Pro. 1 in the special case that $\xi(\texttt{sip}) = \xi(\texttt{dip})$, since in that case a sequence number for the previous hop is known. Also for these variants, which we will not further elaborate here, the proofs of Section 7 (with the exception of Proposition 7.37) remain valid, and loop freedom and route correctness hold.

When taking Resolutions (2a) or (2d), it is easy to check that for any routing table entry $r$ we always have $\pi_3(r) = \texttt{unk} \Leftrightarrow \pi_2(r) = 0$. As a consequence, the sequence-number-status flag is redundant, and can be omitted from the specification altogether.[45] This is the way in which AODV-UU [2] is implemented: it skips the sequence-number-status flag and follows (2d). Since Resolution (2b) can lead to loops, and (2a) and (2d) do not make proper use of sequence-number-status flags, we assume that Resolution (2c) is in line with the intention of the RFC. In Section 10.3 we will discuss the relative merits of the Resolutions (2a), (2c) and (2d) and propose an improvement.

**Ambiguity 3: More Inconclusive Evidence on Dealing with the Unknown Sequence Number**
Section 6.2 of the RFC describes under which circumstances an update occurs.

> “The route is only updated if the new sequence number is either
> (i) higher than the destination sequence number in the route table, or
> (ii) the sequence numbers are equal, but the hop count (of the new information) plus one, is smaller than the existing hop count in the routing table, or
> (iii) the sequence number is unknown.” [79, Sect. 6.2]

Part (iii) is ambiguous. The most plausible reading appears to be that “the sequence number” refers to the new sequence number, i.e., the one provided by an incoming AODV control message triggering a potential update of the node’s routing table. This reading is incompatible with (2a) above, and thus supports only Resolutions (2b), (2c) and (2d). An alternative reading is that it refers to the sequence number in the routing table, meaning that the corresponding sequence-number-status flag has the value unk. This reading of (iii) is consistent with the quote from Section 6.7 above, and leads to routing loops in the same way.[46] The remaining possibility is that Part (iii) refers to the sequence number in the routing table, but only deals with the case that that number is truly unknown, i.e. has the value 0. This reading is consistent with Resolution (2a) above. However, it implies that the routing table may not be updated if the existing entry has a known sequence number whereas the route distilled from the incoming information does not. This is in contradiction the quote from Sect. 6.5 in the RFC above. It is for this reason that we take the first reading of (iii) as our default.

An IETF Internet draft—published after the RFC—rephrases the above statement as follows:

> “A route is only updated if one of the following conditions is met:
> [...]
> (iv) the sequence number in the routing table is unknown.” [78, Sect. 6.2]

[44] The proof of Proposition 7.37 breaks down on the case Pro. 1, Lines 10, 14, 18.

[45] In Pro. 4, Line 34, “$\texttt{sqnf}(\texttt{rt},\texttt{dip}) = \texttt{unk}$” should then be replaced by “$\texttt{sqn}(\texttt{rt},\texttt{dip}) = 0$”, and likewise, in Line 20, “$\texttt{sqnf}(\texttt{rt},\texttt{dip}) = \texttt{kno}$” by “$\texttt{sqn}(\texttt{rt},\texttt{dip}) \neq 0$”.

[46] It can be formalised by using $\texttt{update}^{\texttt{RREP}}$ instead of update in all process Pro. 1–Pro. 5, and furthermore skipping the fifth option in the definition of update in Section 5.5.2.

Since in [78] the sequence-number-status flag has been dropped, the only "unknown" sequence number left is 0, so this quote takes the third reading above. We do not know, however, whether this is meant to be a clarification of [79], or a proposal to change it.

**Ambiguity 4: Updating Invalid Routes**

Another closely related question that arose during formalising AODV is whether an invalid route should be updated in all cases. For example, should an entry $(a,3,\texttt{kno},\texttt{inv},4,b,\emptyset)$ of a routing table be overwritten by $(a,1,\texttt{kno},\texttt{val},2,c,\emptyset)$? Of course this should not be allowed: if an invalid routing table entry were to be replaced by *any* valid entry—even with smaller sequence number—the protocol would not be loop free.

This time, the RFC [79] confirms this assumption:

> "`Whenever any fresh enough (i.e., containing a sequence number at least equal to the recorded sequence number) routing information for an affected destination is received by a node that has marked that route table entry as invalid, the node SHOULD update its route table information according to the information contained in the update.`" [79, Sect. 6.1]

However, it is somewhat less clear what should be done in case the sequence numbers are equal. For example, should an entry $(a,3,\texttt{kno},\texttt{inv},2,b,\emptyset)$ of a routing table be overwritten by $(a,3,\texttt{kno},\texttt{val},4,c,\emptyset)$? According to the quote from Sect. 6.1 above the answer is yes, but according the preceding quote from Sect. 6.2 of the RFC, the answer is no. Our formalisation follows Sect. 6.1 in this regard. To formalise the alternative, one skips the fourth option in the definition of `update` in Section 5.5.2. This contradiction needs to be resolved in favour of Sect. 6.1: none of the two options gives rise to routing loops, but the alternative interpretation would result in a severely handicapped version of AODV, in which many broken routes will never be repaired. We illustrate this by the following example.

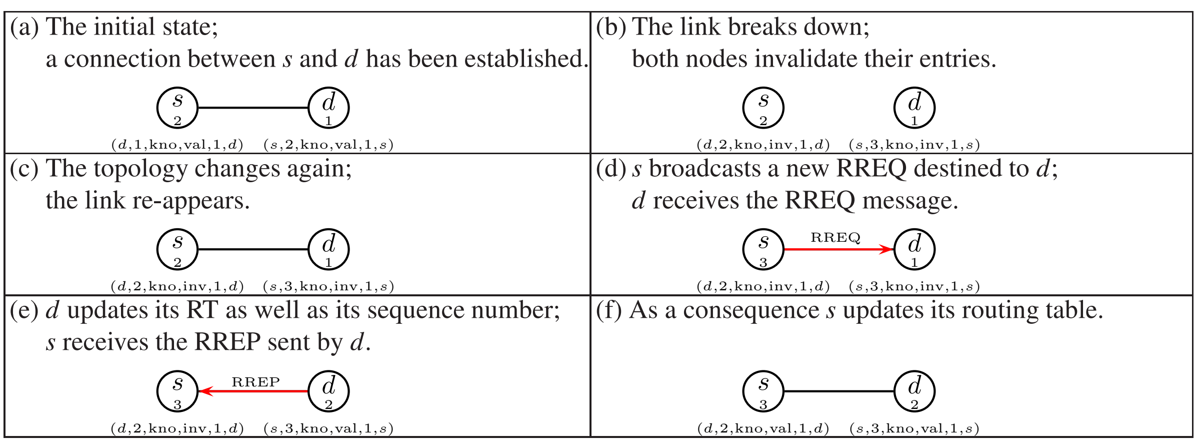



Figure 8: Invalidating a route

We assume a network with two nodes only. Node $s$ has already sent out a route request destined for $d$ and received a route reply message (Figure 8(a)). Due to mobility the link between the nodes breaks. After nodes $s$ and $d$ have invalidated their routing table entries to each other (Figure 8(b)), the link becomes available again. Node $s$ initiates a new route request destined to $d$ (for instance because $s$ wants to send another data-packet to $d$). As usual, node $d$ receives the request (Part (d)), and, depending on which version of AODV we follow, may update its routing table. Figures 8(e) and (f) depict the standard and non-handicapped version of AODV where first node $d$ updates its routing table with a valid routing table entry and sends a reply back to $s$. Then $s$ receives the reply and also updates its routing table. In the handicapped version of AODV neither node $s$ nor $d$ will update their routing tables—the messages would be send around without any actual update being performed. At the end of the RREQ-RREP cycle the network would be in the same state as depicted in Figure 8(c). Only if node $s$ initiates yet another route request—and therefore increases its own sequence number to 4, the resulting routing table of $d$ will contain a valid entry with destination $s$—$s$ would still end up with an invalid entry for $d$. As long as node $d$ does not increase its own sequence number (e.g., due to the initiation of a route request), node $s$ cannot re-establish a valid route.

#### 8.2.2 Self-Entries in Routing Tables

In any practical implementation, when a node sends a message to itself, the message will be delivered to the corresponding application on the local node without ever involving a routing protocol and therefore without being "seen" by AODV or any other routing protocol. Hence it ought not matter if any node using AODV creates a routing table entry to itself. However, as we will show later, there are situations where these *self-entries* yield loops.

**Ambiguity 5: (Dis)Allowing Self-Entries**

In AODV, when a node receives a RREP message, it creates a routing table entry for the destination node if such an entry does not already exist [79, Sect 6.7]. If the destination node happens to be the processing node itself, this leads to the creation of a self-entry. The RFC does mention self-entries explicitly; it only refers to them at one location:

> "`A node may change the sequence number in the routing table entry of a destination only if:`
>
> `- it is itself the destination node [...]`" [79, Sect. 6.1]

This points at least to the possibility of having self-entries. On the other hand, some implementations, such as AODV-UU, disallow self-entries. For our specification (Sections 5 and 6) we have chosen to allow (arbitrary) self-entries since the RFC does *not* prohibits their occurrence. We will refer to this resolution of Ambiguity 5 as (5a).

Looking at our specification, self-entries can only occur using the function update. More precisely, we show the following.

**Proposition 8.1** There is only one location where self-entries can be established, namely Pro. 5, Line 2.

*Proof.* No self-entries are established at the initialisation of the protocol (cf. Section 6.7). Hence we have to consider merely all occurrences of update:

**Pro. 1, Lines 10, 14, 18:** By Corollary 7.9 we have $\xi(\mathtt{sip}) \neq ip$ and therefore no self-entry can be written in the routing table.

**Pro. 4, Line 4:** An entry with destination $\xi(\mathtt{oip})$ is updated/inserted. The value $\xi(\mathtt{oip})$ stems from a received RREQ message (cf. Lines 1 and 8 of Pro. 1). A self-entry can only be established if $\xi(\mathtt{oip}) = ip$. In that case, by Invariant (26), $\xi((\mathtt{oip},\mathtt{rreqid})) \in \xi_{N^\dagger}^{ip}(\mathtt{rreqs})$, where $N^\dagger$ is the network expression at the time when the RREQ message was sent. (Here we use Proposition 7.1.) By Proposition 7.5, we obtain $\xi((\mathtt{oip},\mathtt{rreqid})) \in \xi_{N_3}^{ip}(\mathtt{rreqs})$ and therefore Line 3 evaluates to false and the update is never performed.

**Pro. 5, Line 2:** Self-entries can occur; an example is given below. □

We now show that self-entries can occur in our specifications. The presented example (Figure 9) is not the smallest possible one; however, later on it will serve as a basis for showing how routing loops can occur. The example consists of six nodes, where 5 of them form a circular topology, including one link—between the nodes $d$ and $s$—that is is unstable and unreliable. This link will disappear and re-appear several times in the example.

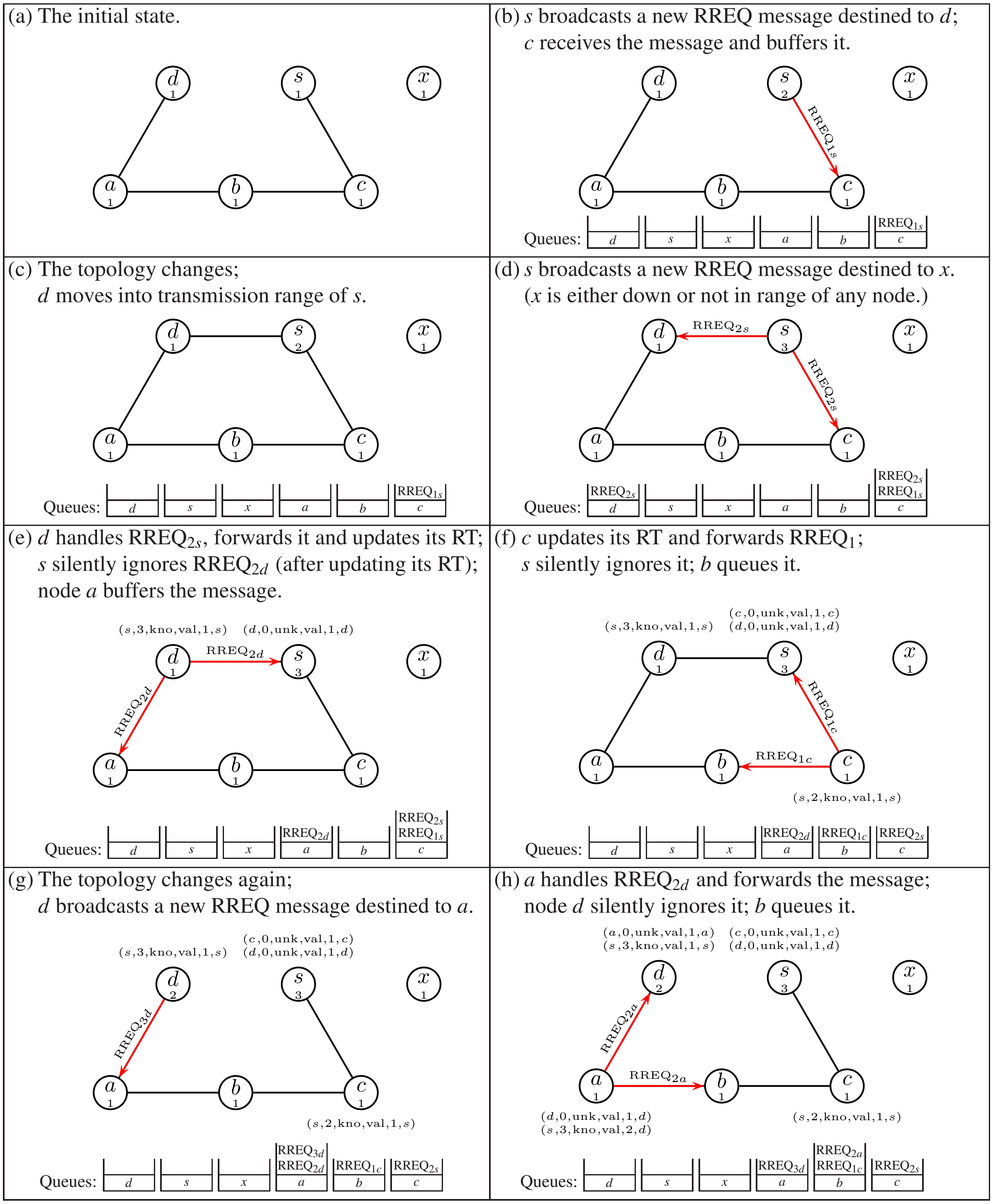



Figure 9: Self-entries

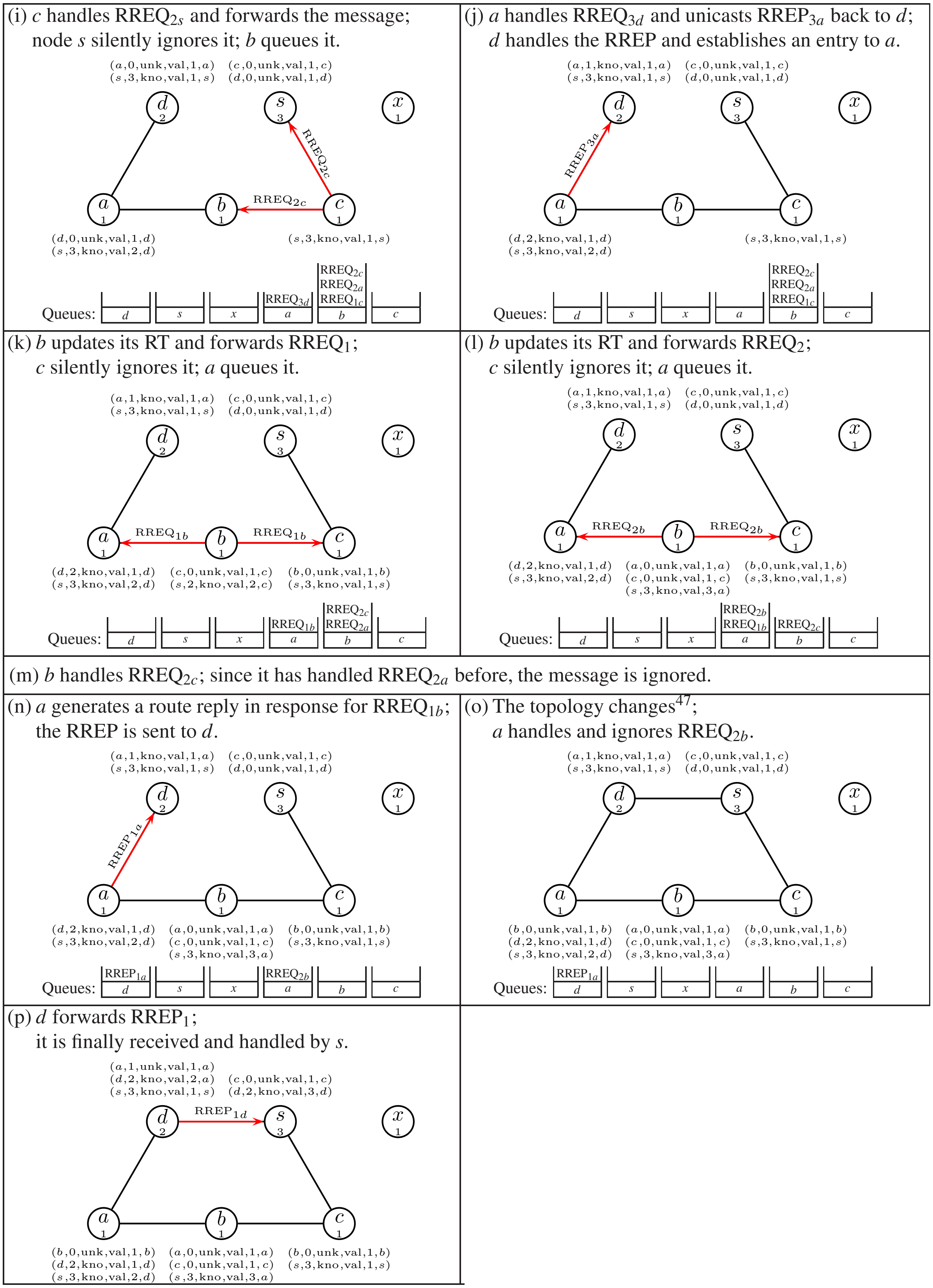



Figure 9 (cont'd): Self-entries

First, node $s$ broadcasts a new route request message $\mathrm{RREQ}_{1s}$ destined for node $d$—the second index $s$ only indicates the sender of the message; a message $\mathrm{RREQ}_{1c}$ belongs to the same route discovery process. After the route discovery has been initiated, $d$ moves into transmission range of $s$ (Figure 9 (c)). Next, node $s$ sends a second route request; this time it is destined for node $x$. In Figure 9(e), node $d$ handles the request destined to $x$; since it has not seen this route request before and it is not the destination, the request is forwarded. Node $s$ receives the forwarded message, updates its routing table and silently ignores $\mathrm{RREQ}_{2d}$ afterwards; node $a$ receives $\mathrm{RREQ}_{2d}$ and stores it into its message buffer. In Figure 9(f) node $c$ handles $\mathrm{RREQ}_{1s}$, which is the first message in its message queue; it inserts an entry with destination $s$ in its routing table and forwards $\mathrm{RREQ}_1$. The link between nodes $d$ and $s$ disappears in Part (g). Moreover a third route discovery search is initiated: this time node $d$ is looking for a route to node $a$. Node $a$ is the only node who receives the broadcast message and queues it in its buffer—it cannot handle $\mathrm{RREQ}_{3d}$ immediately since its message buffer queue is not empty. The topmost element is $\mathrm{RREQ}_{2d}$, which has been received earlier in Part (e). Node $a$ now handles this request;it creates entries for $d$ and $s$ in its routing table and forwards the message (Figure 9(h)). In Part (i), node $c$ forwards the second request. The message $\mathrm{RREQ}_{2c}$ is received by nodes $b$ and $s$; $s$ immediately handles the incoming message, updates its routing table and then silently ignores it. After that, node $a$ empties its message queue and handles the third request, initiated by $d$: since $a$ is the destination, it creates a route reply and unicasts it towards $d$. After a single hop, node $d$, the originator, receives the reply and establishes a routing table entry to $a$ (cf. Figure 9(j)). All but the message buffer of node $b$ are now empty; the queue of $b$ contains the messages $\mathrm{RREQ}_{1c}$, $\mathrm{RREQ}_{2a}$ and $\mathrm{RREQ}_{2c}$. In Part (k), $b$ handles the topmost message ($\mathrm{RREQ}_{1c}$). As usual, it updates its routing table and forwards the message to its neighbours $a$ and $c$; node $a$ buffers the message, whereas $c$ silently ignores it. In Part (l) the same procedure happens again—but this time node $b$ handles $\mathrm{RREQ}_2$ instead of $\mathrm{RREQ}_1$. The last message stored by node $b$ is $\mathrm{RREQ}_{2c}$, which is handled now. Since $b$ has handled a message belonging to the second route discovery search before, this message is ignored (Part (m)). The only node that has messages stored in its buffer is now node $a$. The first message in its queue is $\mathrm{RREQ}_{1b}$, a route request sent by its originator $s$ in Figure 9(b) and destined for node $d$. Since node $a$ has an entry for $d$ in its routing table ($(d,2,\texttt{kno},\mathit{val},1,d)$), it generates a route reply using $\texttt{rrep}(1,d,2,s,a)$. $\mathrm{RREP}_{1a}$ has to be sent to the next hop on the route to $s$, which stored in the routing table of $a$: this is node $d$, the destination of the original route request. In Part (o) the topology changes and the message queue of $a$ is emptied. Node $a$ handles $\mathrm{RREQ}_{2b}$ and silently ignores it since, in Part (h), the node has already handled the second request. In the last step, node $d$ handles the reply generated by node $a$, updates its routing table and forwards the message to $s$, which establishes a route to $d$. When updating the routing table, node $d$ creates a self-entry since $\mathrm{RREP}_{1a}$ unicasts information about a route to $d$.

Later on, in Section 8.2.3, we continue this example to show that the combination of allowing self-entries and literally following the RFC when invalidating routing table entries yields loops.

By Proposition 8.1 only Pro. 5 has to be changed to disallow self-entries. There are two possibilities to accomplish this:

(5b) If a node receives a route reply and would create a self-entry, it silently ignores the message and continues with the main process `AODV`. This resolution is implemented in Pro. 8. A disadvantage of this process is that more replies are lost. In the above example (Figure 9), node $s$ would never receive a reply as a consequence of the very first request sent. Nevertheless, this resolution appears closest in spirit to the RFC, which lists “`a forward route has been created or updated`” as a precondition for forwarding the RREP. The invariants of Section 7 remain valid, with the very same proofs.

[47] The link between $d$ and $s$ can appear at any time between Parts (h) and (o).

**Process 8** RREP handling (Resolution (5b))

```
RREP(hops,dip,dsn,oip,sip,ip,rt,sn,rreqs,store) ≝
 [ dip ≠ ip ∧ rt ≠ update(rt,(dip,dsn,kno,val,hops+1,sip,∅)) ]       /* the routing table has to be updated */
     ...       /* Lines 2–25 of Pro. 5 */
 + [ dip = ip ∨ rt = update(rt,(dip,dsn,kno,val,hops+1,sip,∅)) ]     /* the routing table is not updated */
     AODV(ip,sn,rt,rreqs,store)
```

(5c) The alternative is that the node who would create a self-entry does forward the message without updating the its routing table. (Pro. 9). This resolution bears the risk that the main-invariant (Invariant (21)) is violated, since information is forwarded by a node without updating the node's routing table. However, all invariants established in Section 7 still hold, with the very same proofs—except that the proof of Invariant (14) requires one extra case, which is trivial since $ip_c = dip_c$.

**Process 9** RREP handling (Resolution (5c))

```
RREP(hops,dip,dsn,oip,sip,ip,rt,sn,rreqs,store) ≝
 [ dip ≠ ip ∧ rt ≠ update(rt,(dip,dsn,kno,val,hops+1,sip,∅)) ]       /* the routing table has to be updated */
     ...       /* Lines 2–25 of Pro. 5 */
 + [ dip = ip ∧ rt ≠ update(rt,(dip,dsn,kno,val,hops+1,sip,∅)) ]     /* update would yield a self-entry */
     /* skip all routing table updates */
     ...       /* Lines 3–9 of Pro. 5 */
     ...       /* Lines 13–25 of Pro. 5 */
 + [ rt = update(rt,(dip,dsn,kno,val,hops+1,sip,∅)) ]     /* the routing table is not updated */
     AODV(ip,sn,rt,rreqs,store)
```

Both resolutions by themselves do not yield weird or unwanted behaviour.

### Ambiguity 6: Storing the Own Sequence Number

> "`AODV depends on each node in the network to own and maintain its destination sequence number to guarantee the loop freedom of all routes towards that node.`" [79, Sect. 6.1]

The RFC does not specify how own sequence numbers should be stored. Since the own sequence numbers are never mentioned in combination with destination sequence numbers that are stored in routing tables, it is reasonable to assume that the own sequence number should be stored in a separate data structure. However, there are implementations (e.g. Kernel AODV) that maintain a node's own sequence number in the node's routing table. Of course, just storing a variable does not cause routing loops itself; but since the way of maintenance influences other design decisions, we list this ambiguity.

The resolution where the own sequence number is stored in a separate variable—Resolution (6a)—has been modelled in the specification presented in Sections 5 and 6. The other resolution stores the own sequence number of node $s$ as a self-entry in $s$'s routing table, i.e., as an entry with destination $s$. All other components of that routing table entry are more or less artificial, so the self-entry could for instance have the form $(s, \mathtt{sn}, \mathtt{kno}, \mathtt{val}, 0, s, *)$, where `sn` is the maintained sequence number.

A variant of AODV in which a node's own sequence number is stored in its routing table—Resolution (6b)—is obtained by adapting the specification of Section 6 as follows:

(i) The argument `sn` of the processes `AODV`, `PKT`, `RREQ`, `RREP` and `RERR` is dropped.

(ii) In Pro. 1, Line 39 and Pro. 4, Line 10, the occurrence of `sn` as argument of a **broadcast** or **unicast** is replaced by `sqn(rt,ip)`.

(iii) In the initial state each node *ip* has a routing table containing exactly one optimal self-entry:

$$(ip, 1, \mathtt{kno}, \mathtt{val}, 0, ip, \emptyset) \in \xi(\mathtt{rt}) \wedge |\xi(\mathtt{rt})| = 1.$$

(iv) In Pro. 1, Line 35 the assignment $[\![\mathtt{sn} := \mathtt{inc}(\mathtt{sn})]\!]$, incrementing the node's own sequence number, is replaced by

$$[\![\mathtt{rt} := \mathtt{update}(\mathtt{rt}, (\mathtt{ip}, \mathtt{inc}(\mathtt{sqn}(\mathtt{rt}, \mathtt{ip})), \mathtt{kno}, \mathtt{val}, 0, \mathtt{ip}, \emptyset))]\!].$$

(v) In Pro. 4, Line 8 the assignment $[\![\mathtt{sn} := \max(\mathtt{sn}, \mathtt{dsn})]\!]$, updating $\mathtt{ip}$'s own sequence number, is replaced by

$$[\![\mathtt{rt} := \mathtt{update}(\mathtt{rt}, (\mathtt{ip}, \max(\mathtt{sqn}(\mathtt{rt}, \mathtt{ip}), \mathtt{dsn}), \mathtt{kno}, \mathtt{val}, 0, \mathtt{ip}, \emptyset))]\!].$$

**Theorem 8.2** The interpretation of AODV that follows Resolution (6b) and in all other ways our default specification of Sections 5 is loop free and route correct.

*Proof.* All invariants established in Section 7 and their proofs remain valid, with $\mathtt{sqn}(\mathtt{rt}, \mathtt{ip})$ substituted for all occurrences of $\mathtt{sn}$ (in Proposition 7.2, the proof of Proposition 7.13, and Propositions 7.35, 7.36(c) and 7.37 and their proofs), and with the following modifications:

- Proposition 7.2 now follows from Proposition 7.6 and an inspection of the initial state.
- Proposition 7.10 now holds for non–self-entries only:

$$(dip, *, *, *, hops, *, *) \in \xi_N^{ip}(\mathtt{rt}) \wedge dip \neq ip \;\Rightarrow\; hops \geq 1$$

  The claim holds for the initial state, since there are only self-entries in the routing tables. Furthermore there are two more calls of `update` to be checked, but they all deal with self-entries.
- In the proof of Proposition 7.11(b), when calling Proposition 7.10, we need to check that $dip \neq ip$. This follows by Pro. 4, Line 18.
- In the proof of Proposition 7.12 two more calls of `update` have to be checked, all trivial.
- In the proof of Proposition 7.21 two more calls of `update` have to be checked. The case Pro. 1, Line 35 is trivial; the case Pro. 4, Line 8 uses Proposition 7.2.
- Proposition 7.26(a) now holds for updates with non–self-entries only. The reason is that its proof depends on Proposition 7.10—this is in fact the only other place where we use Proposition 7.10.
- In the proof of Theorem 7.27 we now have to check the updates with self-entries explicitly, since they are no longer covered by Proposition 7.26(a)—this is the only use of Proposition 7.26(a). There are two of them (both introduced above), and none of them can decrease the quality of routing tables.
- In the proofs of Proposition 7.28 and Theorem 7.30 two more calls of `update` have to be checked. If any of those calls actually modifies the entry for *dip*, beyond its precursors, then in the resulting routing table $nhip = dip = ip$, and the precondition of the proposition or theorem is not met.
- In the proofs of Theorem 7.32(a) and Propositions 7.37(b) and 7.38 two more calls of `update` have to be checked, all trivial. □

Theorem 8.2 remains valid for any of the Resolutions (2a, 3c), (2c, 3a) or (2d, 3a), in combination with (5a)–(5c) and with (6b), for the modifications in the proofs of Section 7 induced by these resolutions are orthogonal.

**Corollary 8.3** Assume any interpretation of AODV that uses Resolution (6b) in combination with one of the Resolutions (2a, 3c), (2c, 3a) or (2d, 3a) and any of the Resolutions (5a)–(5c); in all other ways it follows our default specification of Sections 5 and 6. This interpretation is loop free and route correct. □

The following proposition shows that under Resolution (6b), unless combined with (2d) and (5a), the own sequence number of a node is stored in an *optimal* self-entry—with hop count 0—and that such a self-entry cannot be invalided, nor overwritten by data from an incoming message (as this would result in a non-optimal self-entry—cf. Proposition 7.10 and its proof).

**Proposition 8.4** Assume an interpretation of AODV of the kind described in Corollary 8.3, except that it does not use a combination of Resolutions (2d) and (5a).

(a) Each node *ip* maintains an optimal self-entry at all times, i.e., a valid entry for *ip* with hop count 0 and *ip* as next hop.

$$(ip,*,\mathtt{kno},\mathtt{val},0,ip,*) \in \xi_N^{ip}(\mathtt{rt}) \tag{31}$$

(b) Only self-entries of *ip* have hop count 0, and only self-entries of *ip* have *ip* itself as the next hop. In other words, if $(dip,*,*,*,hops,nhip,*) \in \xi_N^{ip}(\mathtt{rt})$ then

$$dip = ip \;\Leftrightarrow\; hops = 0 \;\Leftrightarrow\; nhip = ip \tag{32}$$

*Proof.* We prove both invariants by simultaneous induction. In the initial state each routing table contains exactly one entry (see (iii) above), which is a self-entry satisfying $dip = ip$, $hops = 0$ and $nhip = ip$. By Remark 7.3 it suffices to look at the application calls of `update` and `invalidate`. If an update does not change the routing table entry (the last clause of `update`), both invariants are trivially preserved; hence we only examine the cases that an update actually occurs.

(a) **Pro. 1, Line 35; Pro. 4, Line 8:** In these (new) cases the update yields a self-entry of the required form.

**Pro. 1, Lines 10, 14, 18:** By Corollary 7.9 the update does not result in a self-entry.

**Pro. 4, Line 4:** As in the proof of Proposition 8.1 we conclude that $dip = \xi_N^{ip}(\mathtt{oip}) \neq ip$, i.e. the inserted entry is not a self-entry.

**Pro. 5, Line 2:** The update has the form $\xi_N^{ip}(\mathtt{update}(\mathtt{rt}\,,(\mathtt{dip},\mathtt{dsn},\mathtt{kno},\mathtt{val},\mathtt{hops}+1,\mathtt{sip},\emptyset)))$. Assume, towards a contradiction, that it results in a self-entry, i.e. that $\xi_N^{ip}(\mathtt{dip}) = ip$. The values $\xi_N^{ip}(\mathtt{dip})$ and $\xi_N^{ip}(\mathtt{dsn})$ stem through Line 12 of Pro. 1 from a received RREP message, which by Proposition 7.1 was sent before, say in state $N^\dagger$. By Invariant (28), with $dsn_c := \xi_N^{ip}(\mathtt{dsn})$ and $dip_c := \xi_N^{ip}(\mathtt{dip}) = dip$, we have

$$\xi_N^{ip}(\mathtt{dsn}) = dsn_c \leq \xi_{N^\dagger}^{dip_c}(\mathtt{sqn}(\mathtt{rt}\,,\mathtt{ip})) \leq \xi_N^{dip_c}(\mathtt{sqn}(\mathtt{rt}\,,\mathtt{ip})) = \xi_N^{ip}(\mathtt{sqn}(\mathtt{rt}\,,\mathtt{ip}))\;,$$

the third step by Proposition 7.2. Since Invariant (31) holds before the `update`, $\mathtt{dhops}_N^{ip}(ip) = 0$ and $\mathtt{flag}_N^{ip}(ip) = \mathtt{val}$. Hence, by the definition of `update`, no update actually occurs.
Thus, the update does not result in a self-entry.

**Pro. 1, Line 28:** By construction of `dests` in Line 27, for any $(rip,rsn) \in \xi_N^{ip}(\mathtt{dests})$ we have that $\mathtt{nhop}_{N_{27}}^{ip}(rip) = \mathtt{nhop}_{N_{27}}^{ip}(dip)$, where $dip := \xi_{N_{27}}^{ip}(\mathtt{dip}) = \xi_{N_{22}}^{ip}(\mathtt{dip})$. By Line 21 and Invariant (24) $dip \neq ip$. Hence by Invariant (32), which holds at Line 27, $\mathtt{nhop}_{N_{27}}^{ip}(dip) \neq ip$. Thus $\mathtt{nhop}_{N_{27}}^{ip}(rip) \neq ip$ and by Invariant (32) $rip \neq ip$. It follows that Line 28 will never invalidate a self-entry.

**Pro. 3, Line 10:** The proof is similar to the previous case, except that $dip \neq ip$ follows from Line 3.

**Pro. 4, Lines 13, 29:** The proof is again the same, but with $\xi_N^{ip}(\mathtt{oip})$ taking the role of *dip* and $\xi_N^{ip}(\mathtt{dip})$. That $\xi_N^{ip}(\mathtt{oip}) \neq ip$ follows as in the case Pro. 4, Line 4 of the proof of Proposition 8.1.

**Pro. 5, Line 17:** This follows as in the previous case, except that $\xi_N^{ip}(\mathtt{oip}) \neq ip$ follows from Line 7.

**Pro. 6, Line 3:** The proof is like the previous ones, but this time with $\mathtt{nhop}_{N_2}^{ip}(rip) = \xi_{N_2}^{ip}(\mathtt{sip}) \neq ip$ following from Corollary 7.9. $\square$

(b) The function `invalidate` neither changes the destination, the hop count nor the next hop; hence the invariant is preserved under function calls of `invalidate`. Moreover, Invariant (31) already shows $dip = ip \Rightarrow hops = 0$ and $dip = ip \Rightarrow nhip = ip$.

**Pro. 1, Line 35; Pro. 4, Line 8:** The update yields a self-entry satisfying $dip = ip$, $hops = 0$ and $nhip = ip$.

**Pro. 1, Lines 10, 14, 18:** By Corollary 7.9 the update does not result in a self-entry, and indeed $hops := 1$ and $nhip := \xi_N^{ip}(\mathtt{sip}) \neq ip$.

**Pro. 4, Line 4; Pro. 5, Line 2:** As observed under (a) above, the inserted entry can not be a self-entry. Moreover $hops := \xi_N^{ip}(\mathtt{hops})+1 > 0$ and $nhip := \xi_N^{ip}(\mathtt{sip}) \neq ip$ by Corollary 7.9.

The above proof uses Invariant (28) (Proposition 7.37), which is not available under Resolution (2d). However, when using Resolutions (5b) or (5c), the call to (28) can be avoided, because Line 1 of Pro. 8 or 9 guarantees that the update of Pro. 5, Line 2 does not yield a self-entry. $\square$

When using Resolution (6b) in combination with either Resolution (2a) or (2c), the choice of one of the Resolutions (5a), (5b) or (5c) doesn't make any difference in protocol behaviour, as the optimal self-entry prevents "accidental" non-optimal self-entries to be written in the routing table.

### 8.2.3 Invalidating Routing Table Entries

We have seen that decreasing a sequence number of a routing table entry yields potential loops. A similar effect occurs if the routing table entry is invalidated, but the sequence number is not incremented.

Of course, invalidating routing table entries is closely related to route error message generation.

```
"A node initiates processing for a RERR message in three situations:
  (i) if it detects a link break for the next hop of an active [(val)] route
      in its routing table while transmitting data [...], or
 (ii) if it gets a data packet destined to a node for which it does not have
      an active route [...], or
(iii) if it receives a RERR from a neighbor for one or more active routes."
```

[79, Sect. 6.11]

Before the error message is transmitted, the routing table has to be updated:

```
"1. The destination sequence number of this routing entry, if it exists and
    is valid, is incremented for cases (i) and (ii) above, and copied from
    the incoming RERR in case (iii) above.
 2. The entry is invalidated by marking the route entry as invalid
 [...]"
```

[79, Sect. 6.11]

**Ambiguity 7: Invalidating Entries in Response to a Link Break or Unroutable Data Packet**

Part 2. of the above quotation is clear, whereas Part 1. is ambiguous: Where does "it" refer to? Does the destination sequence number have to exist and be valid (`kno`) or is it the routing table entry that should exist and be valid (`val`)?

From a linguistic point of view, it is more likely that "it" refers to the destination sequence number: first, the sequence number is the first noun and subject in the sentence; second, the pronoun "this" already indicates that a routing entry must exist; hence the condition of existence would be superfluous. Following this resolution, the routing table entry $(d, 1, \mathtt{kno}, \mathtt{val}, 1, d)$ would be updated to $(d, 2, \mathtt{kno}, \mathtt{inv}, 1, d)$, but $(d,1,\texttt{unk},\texttt{val},1,d)$ would yield $(d,1,\texttt{unk},\texttt{inv},1,d)$. A formalisation of this resolution could be obtained by changing Line 27 of Pro. 1 into

$$\begin{aligned}[\![\texttt{dests}:=\{(\texttt{rip},\texttt{inc}(\texttt{sqn}(\texttt{rt},\texttt{rip})))\,|\,\texttt{rip}\in\texttt{vD}(\texttt{rt})\wedge\texttt{nhop}(\texttt{rt},\texttt{rip})=\texttt{nhop}(\texttt{rt},\texttt{dip})\wedge\texttt{sqnf}(\texttt{rt},\texttt{dip})=\texttt{kno}\}\\ \cup\{(\texttt{rip},\texttt{sqn}(\texttt{rt},\texttt{rip}))\,|\,\texttt{rip}\in\texttt{vD}(\texttt{rt})\wedge\texttt{nhop}(\texttt{rt},\texttt{rip})=\texttt{nhop}(\texttt{rt},\texttt{dip})\wedge\texttt{sqnf}(\texttt{rt},\texttt{dip})=\texttt{unk}\}]\!]\,.\end{aligned}$$

Here, sequence numbers of known routing table entries are incremented, whereas sequence numbers of unknown entries remain the same. Both kinds of sequence numbers are used later on to invalidate routing table entries and for further error handling. Similar changes need to be made for Pro. 3, Line 9; Pro. 4, Line 12, 28 and Pro. 5, Lines 16. With this interpretation AODV is able to create routing loops; Figure 10 shows an example.

Part (a) shows a network, in which the node $s$ has already established a route to $d$. This was done by a single route discover process (cf. the first example of Section 2.2). Next, node $b$ tries to establish a route to $a$. To that end, it initiates and broadcasts a route request; the RREQ message is forwarded by $d$ (Part (b)). Node $a$ receives the message and updates the routing table entry $(d,1,\texttt{kno},\texttt{val},1,d)$ to $(d,1,\texttt{unk},\texttt{val},1,d)$, following Resolution (2c) of Section 8.2.1 (Updating with the Unknown Sequence Number). After the route has been established, the topology changes and all links to $a$ break down; the node itself notices that the link to $d$ is down, invalidates the route, and sends a RERR message, but its RERR message is not received by any node (Part (d)). Invalidating the routes uses the assumption that the routing table entry $(d,1,\texttt{unk},\texttt{val},1,d)$ is updated to $(d,1,\texttt{unk},\texttt{inv},1,d)$. In Part (e), $a$ reconnects to $s$ and a new link between $s$ and $d$ occurs. Last, node $a$ tries to reestablish a route to $d$, broadcasts a request with destination sequence number 1 and immediately receives an answer by $s$. Now, the routing table of $a$ contains an entry to $d$ with next hop $s$, and $s$ has a routing table entry to $d$ with next hop $a$. A packet which is sent to node $d$ by either of these two nodes would circulate in a loop forever.

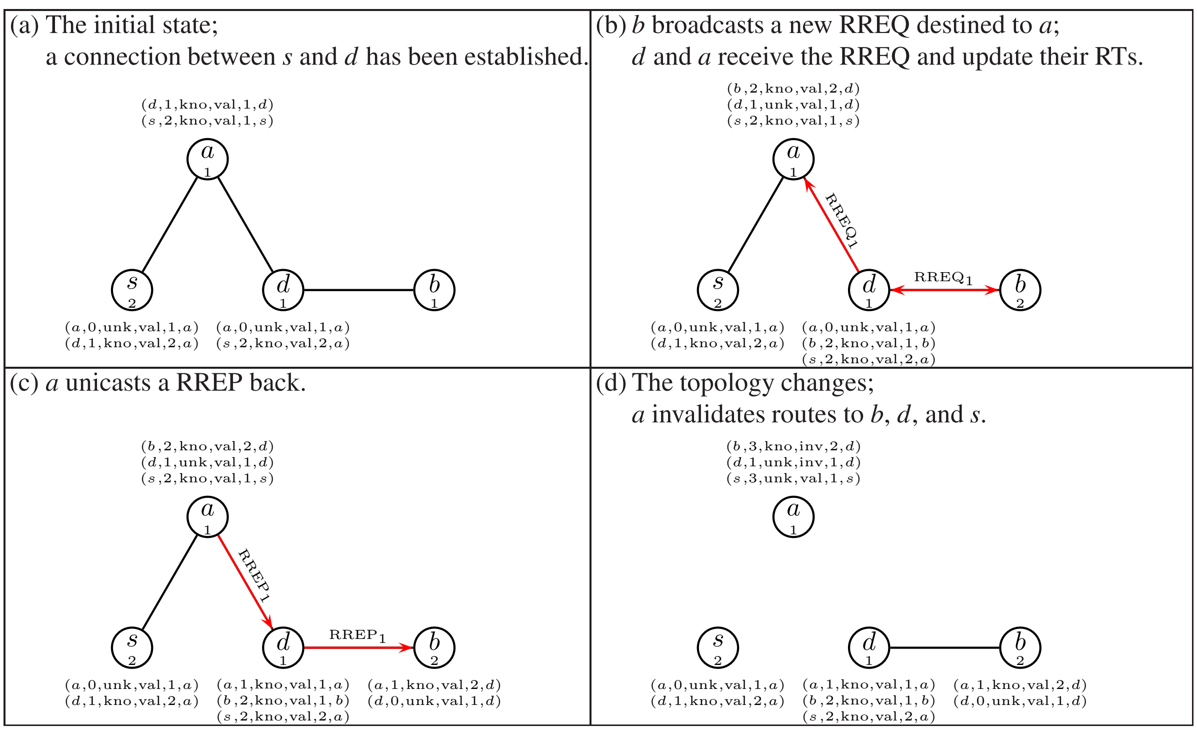



Figure 10: Creating a loop by not incrementing unknown sequence numbers

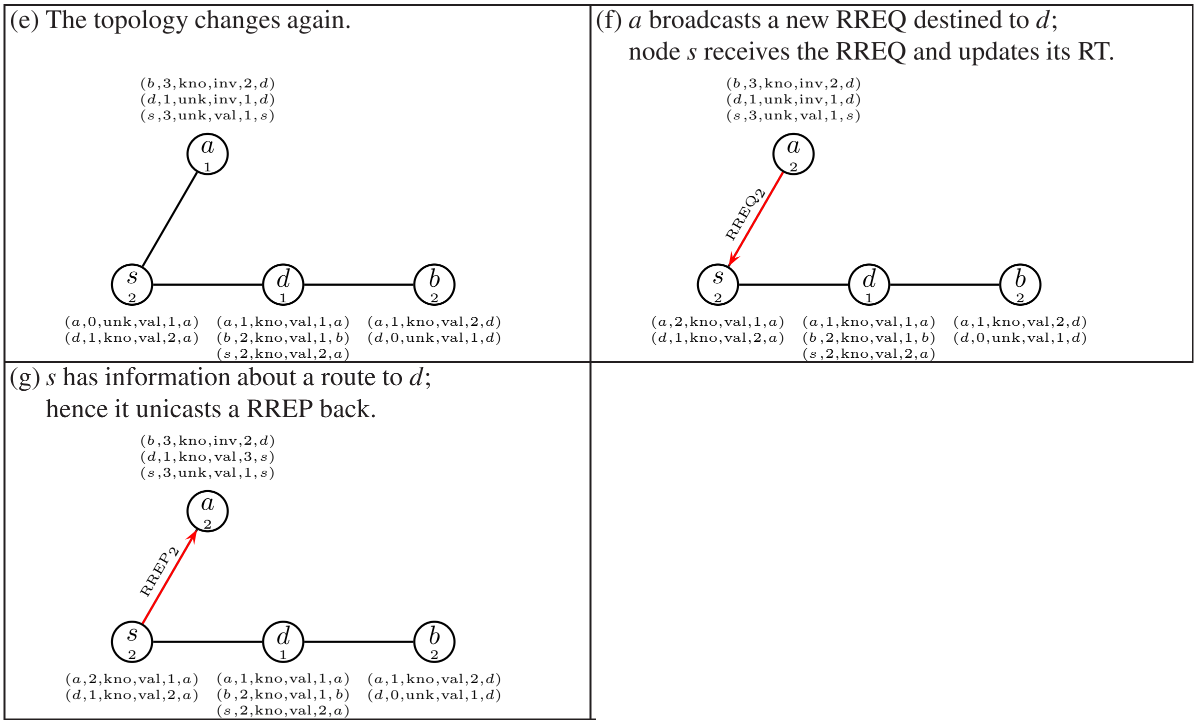



Figure 10 (cont'd): Creating a loop by not incrementing unknown sequence numbers

In sum, the only acceptable reading of Part 1. above is the one where "it" refers to "routing entry": before a RERR message is sent, the destination sequence number of a routing table entry is incremented, if such an entry exists and is valid. This is the interpretation formalised in Section 6.

**Ambiguity 8: Invalidating Entries in Response to a Route Error Message**

The part "`and copied from the incoming RERR in case (iii) above`" of the quote given on Page 75 (from Sect. 6.11 of the RFC) is unambiguous. It describes the replacement of an existing destination sequence number in a routing table entry with another one, which may be strictly smaller. This literal interpretation gives rise to a version of AODV without the requirement $\mathtt{sqn}(\mathtt{rt},\mathtt{rip}) < \mathtt{rsn}$ in Pro. 6, Line 2 (cf. Resolution (8a) below). However, replacing a sequence number with a strictly smaller one contradicts the quote from Sect. 6.1 of the RFC displayed in Section 8.1. To make the process of invalidation consistent with Sect. 6.1 of the RFC, one could use Resolutions (8b) or (8c) instead. Resolution (8b), which strictly follows Sect. 6.1, aborts the invalidation attempt if the destination sequence number provided by the incoming RERR message is smaller than the one already in the routing table. Resolution (8c), on the other hand, still invalidates in these circumstances, but prevents a decrease in the destination sequence number by taking the maximum of the stored and the incoming number.

(8a) Follow Section 6.11 of the RFC, in defiance of 6.1, i.e., *always* invalidate the routing table entry, and copy the destination sequence number from the error message to the corresponding entry in the routing table.[48] This is formalised by skipping the requirement $\mathtt{sqn}(\mathtt{rt},\mathtt{rip}) < \mathtt{rsn}$ in Pro. 6, Line 2.

(8b) Follow Section 6.11 only where it does not contradict 6.1, i.e., invalidate the routing table entry and copy the destination sequence number *only* if this does not give rise to a decrease of the des-

[48] It could be argued that this is not a reasonable interpretation of the RFC, since Section 6.1 should have priority over 6.11. However, this priority is not explicitly stated.

tination sequence number in the routing table. This if formalised by replacing the requirement by $\mathtt{sqn}(\mathtt{rt},\mathtt{rip}) \leq \mathtt{rsn}$.

(8c) Always invalidate the routing table entry (skip the requirement completely), but use a version of the function `invalidate` of Section 5.5.3 that uses $\max(\pi_2(r), rsn)$ instead of $rsn$, thereby updating the destination sequence number in the routing table to the maximum of its old value and the value contributed by the incoming RERR message.

We now show that in combination with allowing self-entries (cf. Section 8.2.2) each of these resolutions gives rise to routing loops. Figure 11 continues the example of Figure 9, and is valid for any of them.

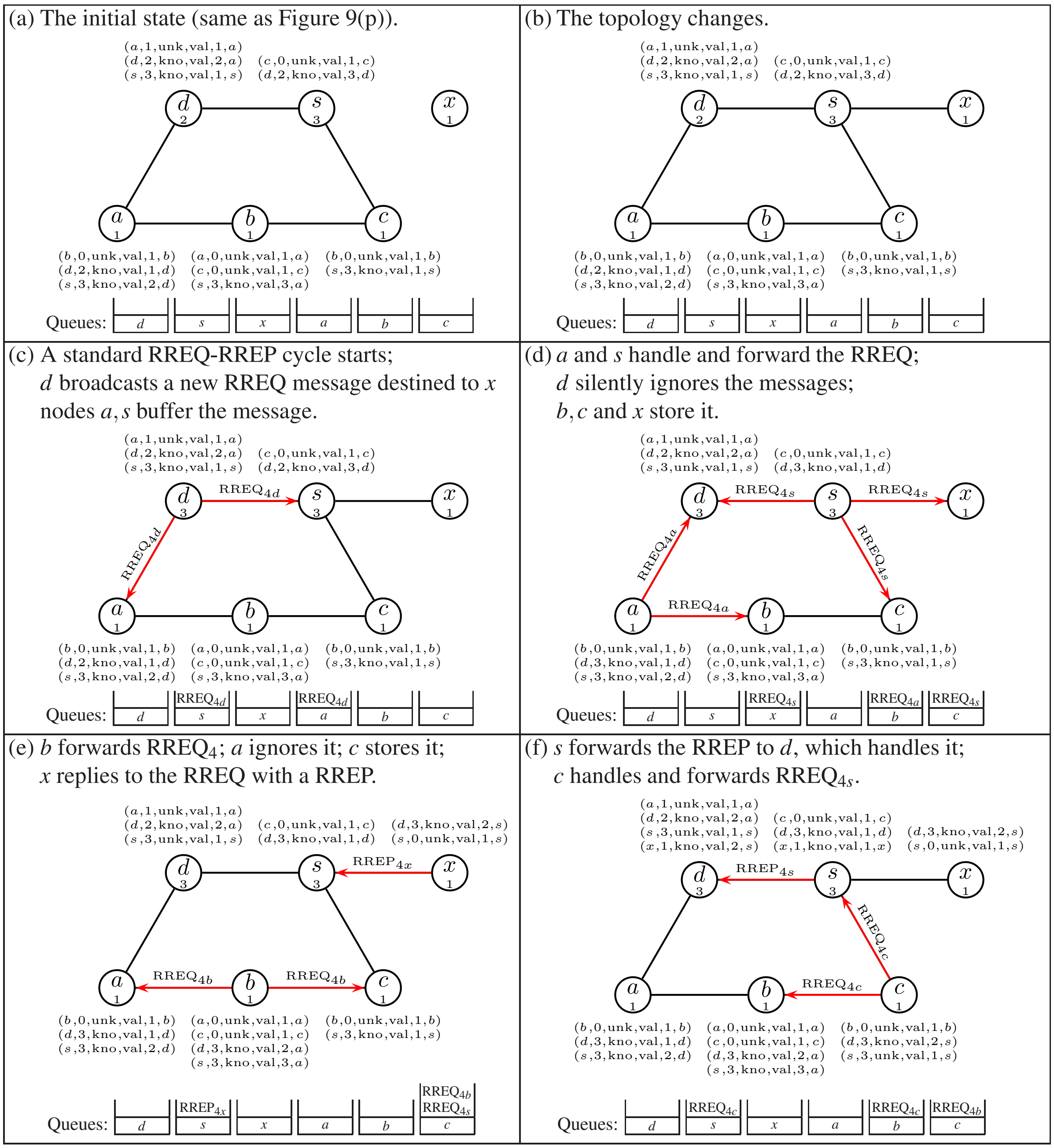



Figure 11: Combination of self-entries and an inappropriate `invalidate` yields loops

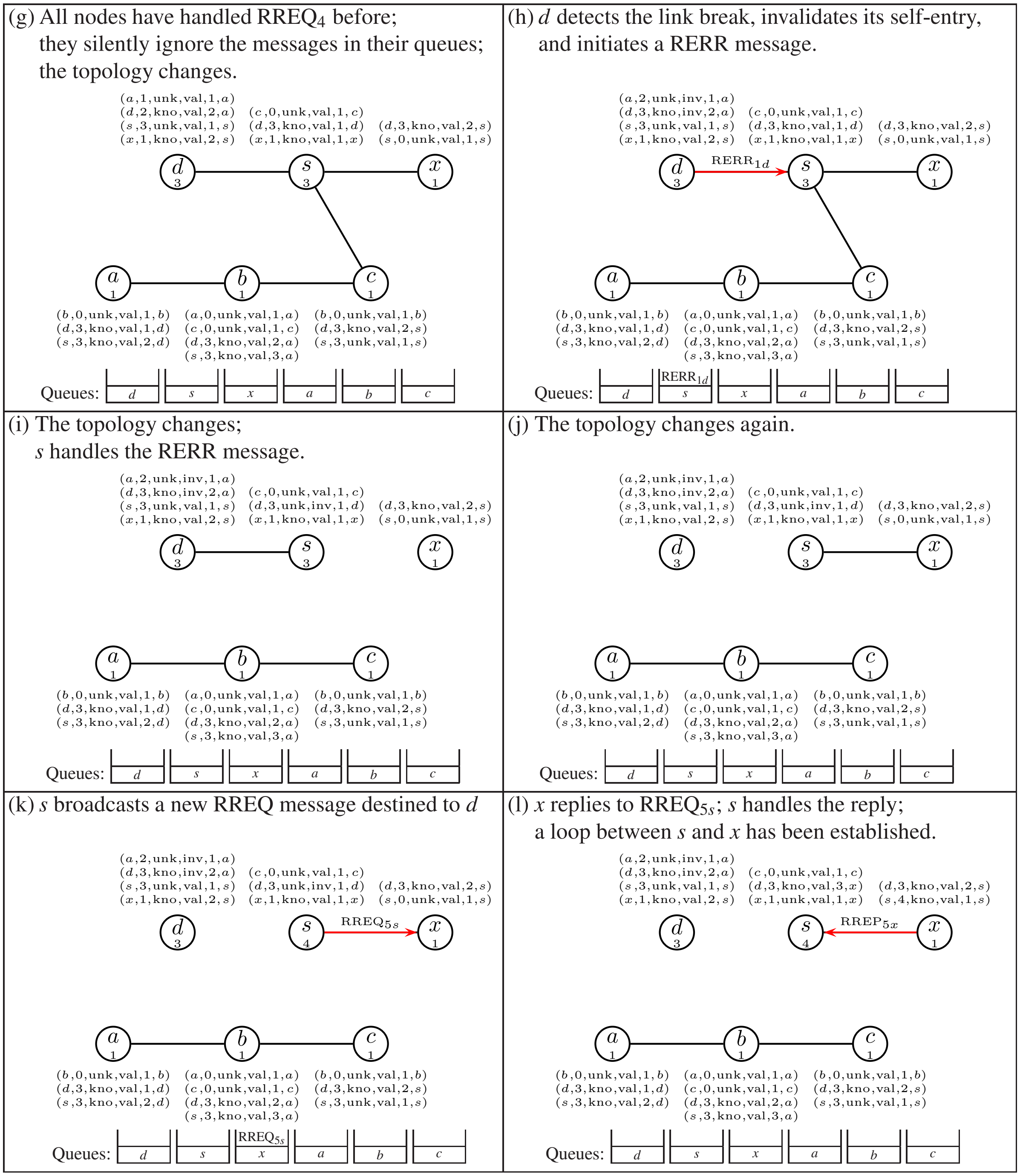



Figure 11 (cont'd): Combination of self-entries and an inappropriate `invalidate` yields loops

At the initial state (Part (a)), the message queues of all nodes are empty. A couple of routes have been found and many routing table entries are already set up. Node *d* has among standard entries also a self-entry, a valid entry to itself with sequence number and hop count 2. The example continues with node *x* moving into the transmission range of *s*, followed by a standard RREQ-RREP cycle (Figure 11(b–g)) .

In Part (c), *d* initiates a new route request for *x*. The generated message is received by nodes *a* and *s*; both nodes create reverse routes to *d* and forward the request (Part (d)). Node *x* now handles the forwarded request and since this node is the intended destination it generates and unicasts a route reply.

Meanwhile the broadcast request still flows around in the network. In Part (e), node $b$ forwards it; in (f) the message is handled by node $c$. Here the route reply is also unicast back from $s$ to $d$. The RREQ-RREP cycle ends with silently ignoring all remaining route request messages of all message queues; this is due to the fact that all nodes have already handled the request sent out by node $d$ in Part (d).

The last part of the example, which finally creates a routing loop, starts with a topology change in Figure 11(g). Node (d) detects the link break and invalidates its link to $a$. It also invalidates its self-entry since the next hop on its recorded route to $d$ is node $a$. (At this point all treatments of the invalidation procedure contemplated in this section agree.) As a consequence of the link break node $d$ also casts a route error message to $s$. Due to unreliable links (for example due to node $s$ moving around), the error message cannot be sent forward; hence *only* node $s$ invalidates its entry for $d$. Before the entry is invalidated it is, by Line 18 of Pro. 1, updated to $(d,3,\texttt{unk},\texttt{val},1,d)$. For the invalidation we assume either of the Resolutions (8a), (8b) or (8c)—according to each the routing table entry of $s$ to $d$ *is* invalidated and the destination sequence number 3 remains unchanged. In Part (j) node $s$ moves into transmission range of $x$. We assume that it wants to send a data packet to $d$. Since its routing table entry for $d$ has been invalidated, a new route request is sent out in Part (k). The destination sequence number in this control message is set to 3. The message is received by node $x$, which immediately initiates a route reply since its routing table contains a valid entry to $d$ with a sequence number that is large enough. After node $s$ receives this message, a routing loop between $s$ and $x$ for destination $d$ has been established.

The problem in this example is that a routing table entry in invalidated without increasing the destination sequence number.

As the above example shows, none of the above variants should be used in combination with non-optimal self-entries; thus either non-optimal self-entries should be forbidden, or one should reject all plausible interpretations of the invalidation process that are consistent with the combination of Sections 6.1 and 6.11 of the RFC. However, the preceding quote from Sect. 6.2 of the RFC suggests the interpretation proposed in Sections 5 and 6—Resolution (8f) below. Here we invalidate the routing table entry and copy the destination sequence number only if this gives rise to an *increase* of the destination sequence number in the routing table. This if formalised by the requirement $\texttt{sqn}(\texttt{rt},\texttt{rip}) < \texttt{rsn}$ in Pro. 6, Line 2. Another solution, Resolution (8d), is to still invalidate in this circumstances, but guarantee an increase in the destination sequence number in the routing table by taking the maximum of its incremented old value and the value stemming from the incoming RERR message. Finally, Resolution (8e) is a combination of (8b) and (8d).

(8d) Always invalidate the entry (skip the requirement $\texttt{sqn}(\texttt{rt},\texttt{rip}) < \texttt{rsn}$), but use a version of the function `invalidate` of Section 5.5.3 that uses $\max(\texttt{inc}(\pi_2(r)), \mathit{rsn})$ instead of $\mathit{rsn}$.

(8e) Invalidate the routing table entry only if $\texttt{sqn}(\texttt{rt},\texttt{rip}) \leq \texttt{rsn}$ and update the destination sequence number to $\max(\texttt{inc}(\pi_2(r)), \mathit{rsn})$. [49]

(8f) Invalidate the routing table entry only if $\texttt{sqn}(\texttt{rt},\texttt{rip}) < \texttt{rsn}$.[50]

In Sections 5 and 6 we have shown that our default specification of AODV, implementing Resolution (8f), is loop free and route correct. We now show that the same holds when using Resolutions (8d) or (8e) instead. In fact, all invariants established in Section 7 and their proofs remain valid, with the following modifications.

- The proof of Proposition 7.6 simplifies, because not even the (modified) function `invalidate` can decrease a sequence number.

[49] The variant that invalidates only if $\texttt{sqn}(\texttt{rt},\texttt{rip}) \leq \texttt{rsn}$ and updates to $\max(\pi_2(r), \texttt{rsn})$ needs no separate consideration, since it is equivalent to Resolution (8b).

[50] Here, it does not matter whether we update to $\texttt{rsn}$, $\max(\pi_2(r), \texttt{rsn})$ or to $\max(\texttt{inc}(\pi_2(r)), \texttt{rsn})$; they are all equivalent.

- In Proposition 7.15 the requirement $rsn_c = \mathtt{sqn}^{ip}_{N}(rip_c)$ is weakened to $rsn_c \leq \mathtt{sqn}^{ip}_{N}(rip_c)$. This change is harmless, since Proposition 7.15 is applied in the proof of Proposition 7.28 only (at the end), where the weakened version is used anyway.

  The first case in the proof of Proposition 7.15 is adapted to:

  **Pro. 1, Line 32:** The set $dests_c$ is constructed in Line 31 as a subset of $\xi^{ip}_{N_{31}}(\mathtt{dests}) = \xi^{ip}_{N_{28}}(\mathtt{dests})$. For each $(rip_c, rsn_c) \in \xi^{ip}_{N_{28}}(\mathtt{dests})$ one has $rip_c = \xi^{ip}_{N_{27}}(\mathtt{rip}) \in \mathtt{vD}^{ip}_{N_{27}}$. Then in Line 28, using the modified function $\mathtt{invalidate}$, $\mathtt{flag}(\xi(\mathtt{rt}), rip_c)$ becomes $\mathtt{inv}$ and $\mathtt{sqn}(\xi(\mathtt{rt}), rip_c)$ becomes $\max(\mathtt{inc}(\mathtt{sqn}(\xi^{ip}_{N_{27}}(\mathtt{rt}), rip_c)), rsn_c)$. Thus we obtain $rip_c \in \mathtt{iD}^{ip}_{N}$ and $\mathtt{sqn}^{ip}_{N}(rip_c) \geq rsn_c$.

- The proof of Theorem 7.27 simplifies, because the (modified) function `invalidate` can never decrease the quality of routing tables.

- The last case in the proof of Proposition 7.28 is adapted to:

  **Pro. 6, Line 3:** Let $N_3$ and $N$ be the network expressions right before and right after executing Pro. 6, Line 3. The entry for destination $dip$ can be affected only if $(dip, dsn) \in \xi^{ip}_{N_2}(\mathtt{dests})$ for some $dsn \in \mathtt{SQN}$. In that case, by Line 2, $(dip, dsn) \in \xi^{ip}_{N_2}(\mathtt{dests})$, $dip \in \mathtt{vD}^{ip}_{N_2}$, and $\mathtt{nhop}^{ip}_{N_2}(dip) = \xi^{ip}_{N_2}(\mathtt{sip})$. By the modified definition of `invalidate`,

  $$\mathtt{sqn}^{ip}_{N}(dip) = \max(\mathtt{inc}(\mathtt{sqn}^{ip}_{N_3}(dip)), dsn) \qquad \text{and} \qquad \mathtt{flag}^{ip}_{N}(dip) = \mathtt{inv}, \qquad \text{so}$$

  $$\begin{aligned} \mathtt{nsqn}^{ip}_{N}(dip) &= \mathtt{sqn}^{ip}_{N}(dip) \mathbin{\dot{-}} 1 \\ &= \max(\mathtt{inc}(\mathtt{sqn}^{ip}_{N_3}(dip)) \mathbin{\dot{-}} 1, dsn \mathbin{\dot{-}} 1) \\ &= \max(\mathtt{sqn}^{ip}_{N_3}(dip), dsn \mathbin{\dot{-}} 1)\ . \end{aligned}$$

  Hence we need to show that (i) $\mathtt{sqn}^{ip}_{N_3}(dip) \leq \mathtt{nsqn}^{nhip}_{N}(dip)$ and (ii) $dsn \mathbin{\dot{-}} 1 \leq \mathtt{nsqn}^{nhip}_{N}(dip)$.

  (i) Since $dip \in \mathtt{vD}^{ip}_{N_2} = \mathtt{vD}^{ip}_{N_3}$, we have

  $$\mathtt{sqn}^{ip}_{N_3}(dip) = \mathtt{nsqn}^{ip}_{N_3}(dip) \leq \mathtt{nsqn}^{nhip}_{N_3}(dip) = \mathtt{nsqn}^{nhip}_{N}(dip)$$

  The inequality holds since the invariant is valid right before executing Line 3.

  (ii) This case goes exactly as the corresponding case in Section 7. □

When forbidding non-optimal self-entries—either by choosing one of the Resolutions (5b) or (5c) of AODV proposed on Page 72, or by storing the own sequence number in an optimal self-entry as described in the previous section—all Variants (8a)–(8f) of the invalidation process described in this section behave exactly the same. Hence all are loop free and route correct. This follows by the following invariants, which are established not for our default specification of AODV, but for either of the resolutions without non-optimal self-entries, still following (8f) above.

**Proposition 8.5** Assume an interpretation of AODV that takes one of the Resolutions (2a, 3c), (2c, 3a) or (2d, 3a) in combination with (8f) and any resolution of Ambiguities 5 and 6, but not (5a) and (6a) at the same time, and not (2d) with (5a) and (6b); in all other ways it follows our default specification of Sections 5 and 6.

(1) Whenever Line 2 of Pro. 6 is executed by node $ip$ in state $N$ we have $\mathtt{sqn}^{ip}_{N}(rip) < rsn$ for all $(rip, rsn) \in \xi^{ip}_{N}(\mathtt{dests})$ with $rip \in \mathtt{vD}^{ip}_{N}$ and $\mathtt{nhop}^{ip}_{N}(rip) = \xi^{ip}_{N}(\mathtt{sip})$.

(2) Whenever node $ip$ makes a call $\mathtt{invalidate}(\mathtt{rt},\mathtt{dests})$ in state $N$, then $\max(\mathtt{inc}(\mathtt{sqn}^{ip}_{N}(rip)), rsn) = \max(\mathtt{sqn}^{ip}_{N}(rip), rsn) = rsn$ for all $(rip, rsn) \in \xi^{ip}_{N}(\mathtt{dests})$.

*Proof.*

(1) Suppose Line 2 of Pro. 6 is executed in state $N$, and let $(rip,rsn) \in \xi_N^{ip}(\texttt{dests})$ with $rip \in \texttt{vD}_N^{ip}$ and $nhip := \texttt{nhop}_N^{ip}(rip) = \xi_N^{ip}(\texttt{sip})$. The values $\xi_N^{ip}(\texttt{dests})$ and $\xi_N^{ip}(\texttt{sip})$ stem from a received route error message (cf. Lines 1 and 16 of Pro. 1). By Proposition 7.1(a), a transition labelled $R\!:\!\textbf{*cast}(\texttt{rerr}(dests_c\,,ip_c))$ with $dests_c := \xi_N^{ip}(\texttt{dests})$ and $ip_c := \xi_N^{ip}(\texttt{sip})$ must have occurred before, say in state $N^\dagger$. By Proposition 7.8, the node casting this message is $ip_c = \xi_N^{ip}(\texttt{sip}) = nhip$. By Invariant (15) we have $rip \in \texttt{iD}_{N^\dagger}^{nhip}$ and $rsn = \texttt{sqn}_{N^\dagger}^{nhip}(rip)$. Since (invalid) self-entries cannot occur,[51] it follows that $nhip \neq rip$.

Since $rip \in \texttt{vD}_N^{ip}$, the last function call prior to state $N$ that created or updated this valid routing table entry of node $ip$, apart from an update of the precursors only, must have been a call $\texttt{update}(*, (rip,rsn^\star,*,*,*,sip^\star,*))$, where one of the first five clauses in the definition of update was applied. Let $N^\star$ be the state in which this call was made. Then $sip^\star = nhip$. We consider all possibilities for this call.

**Pro. 1, Lines 10, 14, 18:** The entry $\xi_{N^\star}^{ip}(\texttt{sip},0,\texttt{unk},\texttt{val},1,\texttt{sip},\emptyset)$ is used for the update; its next hop is $\xi_{N^\star}^{ip}(\texttt{sip}) = sip^\star = nhip$ and its destination $\xi_{N^\star}^{ip}(\texttt{sip}) = rip$. This contradicts the conclusion that $nhip \neq rip$, and thus these cases cannot apply.

**Prop. 4, Line 4:** The update has the form $\xi_{N^\star}^{ip}(\texttt{update}(\texttt{rt},(\texttt{oip},\texttt{osn},\texttt{kno},\texttt{val},\texttt{hops}+1,\texttt{sip},\emptyset)))$. Hence one of the first four clauses in the definition of update was used, with $\xi_{N^\star}^{ip}(\texttt{oip}) = rip$, $\xi_{N^\star}^{ip}(\texttt{osn}) = \texttt{sqn}_N^{ip}(rip)$ and $\xi_{N^\star}^{ip}(\texttt{sip}) = sip^\star = nhip$. The values $\xi_{N^\star}^{ip}(\texttt{oip})$, $\xi_{N^\star}^{ip}(\texttt{osn})$ and $\xi_{N^\star}^{ip}(\texttt{sip})$ stem from a received RREQ message (cf. Lines 1 and 8 of Pro. 1). By Proposition 7.1(a), a transition $R\!:\!\textbf{*cast}(\texttt{rreq}(*\,,*\,,*\,,*\,,*\,,\xi_{N^\star}^{ip}(\texttt{oip})\,,\xi_{N^\star}^{ip}(\texttt{osn})\,,\xi_{N^\star}^{ip}(\texttt{sip})))$ must have occurred before, say in state $N^\ddagger$. By Proposition 7.8 the node casting this message is $\xi_{N^\star}^{ip}(\texttt{sip}) = nhip$. By Invariant (13), using that $nhip \neq rip$, we have $\xi_{N^\star}^{ip}(\texttt{osn}) < \texttt{sqn}_{N^\ddagger}^{nhip}(rip)$ or $\xi_{N^\star}^{ip}(\texttt{osn}) = \texttt{sqn}_{N^\ddagger}^{nhip}(rip)$ and $\texttt{flag}_{N^\ddagger}^{nhip}(rip) = \texttt{val}$. In either case $\xi_{N^\star}^{ip}(\texttt{osn}) \leq \texttt{nsqn}_{N^\ddagger}^{nhip}(rip)$. Since node $ip$ handled the incoming RREQ message prior to the above-mentioned RERR message, the RREQ message was entered earlier in the FIFO queue of node $ip$ and hence transmitted earlier by node $nhip$. So $N^\ddagger$ is prior to $N^\dagger$. We obtain

$$\texttt{sqn}_N^{ip}(rip) = \xi_{N^\star}^{ip}(\texttt{osn}) \leq \texttt{nsqn}_{N^\ddagger}^{nhip}(rip) \leq \texttt{nsqn}_{N^\dagger}^{nhip}(rip) < \texttt{sqn}_{N^\dagger}^{nhip}(rip) = rsn\ ,$$

where the first, second and last step have been established before; the third uses Theorem 7.27, and the penultimate step follows from the definition of net sequence numbers and

$$\texttt{sqn}_{N^\dagger}^{nhip}(rip) \geq \texttt{sqn}_{N^\ddagger}^{nhip}(rip) \geq \xi_{N^\star}^{ip}(\texttt{osn}) \geq 1\ ,$$

which follows from Proposition 7.6 and Invariant (11).

**Prop. 5, Line 2:** The proof is similar to the one of Pro. 4, Line 4, the main difference being that the information stems from an incoming RREP message; instead of oip and osn we use dip and dsn, and instead of Invariants (13) and (11) we use Invariants (14) and (12).

(2) We check all calls of invalidate.

**Pro. 1, Line 28; Pro. 3, Line 10; Pro. 4, Lines 13, 29; Pro. 5, Line 17:**
By construction of dests (right before the invalidation call) if $(rip,rsn) \in \xi_N^{ip}(\texttt{dests})$ then $rsn = \texttt{inc}(\texttt{sqn}_N^{ip}(rip))$.

**Pro. 6, Line 3:** Immediately from (1). □

[51] When using Resolution (6b), but not in combination with (5a) and (2d), invalid self-entries cannot occur by Invariant (31) in combination with Proposition 7.16(a); otherwise under Resolutions (5b) or (5c) self-entries cannot occur at all.

By this proposition, Resolutions (8a)–(8f) behave the same if non-optimal self-entries are forbidden. Hence, by using Corollary 8.3, we obtain the following result.

**Corollary 8.6** Assume an interpretation of AODV that takes one of the Resolutions (2a, 3c), (2c, 3a) or (2d, 3a) in combination with any resolution of Ambiguities 5, 6 and 8, but not (5a) and (6a) at the same time and not (2d) with (5a) and (6b); in all other ways it follows our default specification of Sections 5 and 6. This interpretation is loop free and route correct. □

### 8.2.4 Further Ambiguities

**Ambiguity 9: Packet Handling for Unknown Destinations**
In rare situations, often caused by node reboots, it may be possible that a node receives a data packet from another node for a destination for which it has no entry in its routing table at all. Such a situation cannot occur in our specification—this is a direct consequence of Proposition 7.28. Nevertheless, since our specification given in Section 6 is intended to model *all* possible scenarios which might occur, we have to decide which rules AODV should follow. The RFC states that an error message should be generated `if [a node] gets a data packet destined to a node for which it does not have an active route` [79, Sect. 6.11]. It also states that the sequence number for the unreachable destination, to be listed in the error message, should be taken from the routing table and that the `neighboring node(s) that should receive the RERR are all those that belong to a precursor list of at least one of the unreachable destination(s).` In this case neither the sequence number nor the list of precursors are available. There are two possible solutions:

(9a) no error message is generated, since no information is available in the routing table—in Section 6, we follow that approach (Pro. 3, Lines 21–22);

(9b) the error message is broadcast and the sequence number is set to unknown (0)—formalised in Pro. 10. This resolution makes sense only when using Resolutions (8c) or (8d) of the invalidation process: Resolutions (8b), (8e) and (8f) would systematically ignore the broadcasted error message—so that there is no point in sending it—whereas with Resolution (8a) this obviously leads to a decrease in destination sequence numbers and routing loops.

**Process 10** Routine for packet handling (Resolution (9b))

$\texttt{PKT}(\texttt{data},\texttt{dip},\texttt{oip},\texttt{ip},\texttt{sn},\texttt{rt},\texttt{rreqs},\texttt{store}) \stackrel{def}{=}$

```
...      /* Lines 1–20 of Pro. 3 */
+ [ dip ∉ iD(rt) ]      /* route not in rt */
   broadcast(rerr({(dip, 0)}, ip)) . AODV(ip, sn, rt, rreqs, store)
...      /* Lines 23–24 of Pro. 3 */
```

In Section 7 we have shown that Resolution (9a) is loop free. Resolution (9b) is loop free as well, since all invariants of Section 7 and their proofs remain valid, with the following modifications:

- Proposition 7.15 (Invariant (15)) has to be weakened: it holds only for pairs $(rip_c, rsn_c)$ with $rsn_c > 0$. In the adapted proof there is an extra case to consider, but there $rsn_c = 0$. This proposition is only used in the proof of Proposition 7.28; we show below that the weaker form is sufficient.
- In the proof of Proposition 7.16(c) there is an extra case to consider, which is trivial.
- In the last case of the proof of Proposition 7.28, when using Invariant (15), there is an extra case to consider, namely that $dsn = 0$. In that case surely $dsn \dot{-} 1 = 0 \leq \texttt{nsqn}_N^{nhip}(dip)$, which we needed to establish.

The proof of Proposition 8.5 is no longer valid when using Resolution (9b) of the packet handling for unknown destinations, since it uses a version of Invariant (15) that no longer holds. This indicates that Resolutions (8a), (8b) and (8c) of the invalidation process are not necessarily compatible with Resolution (9b), even if non-optimal self-entries are forbidden.[52] On the other hand, it can be argued that in Resolution (9a) the originator node that initiated the sending of the data packet might send more packets, which increases network traffic without delivering the data.

**Ambiguity 10: Setting the Own Sequence Number when Generating a RREP Message**
In the RFC, the way in which a destination of a route request updates its own sequence number before initiating a route reply is described in two ways:

> "`Immediately before a destination node originates a RREP in response to a RREQ, it MUST update its own sequence number to the maximum of its current sequence number and the destination sequence number in the RREQ packet.`"
> [79, Sect. 6.1]

> "`If the generating node is the destination itself, it MUST increment its own sequence number by one if the sequence number in the RREQ packet is equal to that incremented value. Otherwise, the destination does not change its sequence number before generating the RREP message.`" [79, Sect. 6.6.1]

In most cases these two descriptions yield the same result (because the destination sequence number in the RREQ message is usually not more than 1 larger than the destination's own sequence number). However, this is not guaranteed.

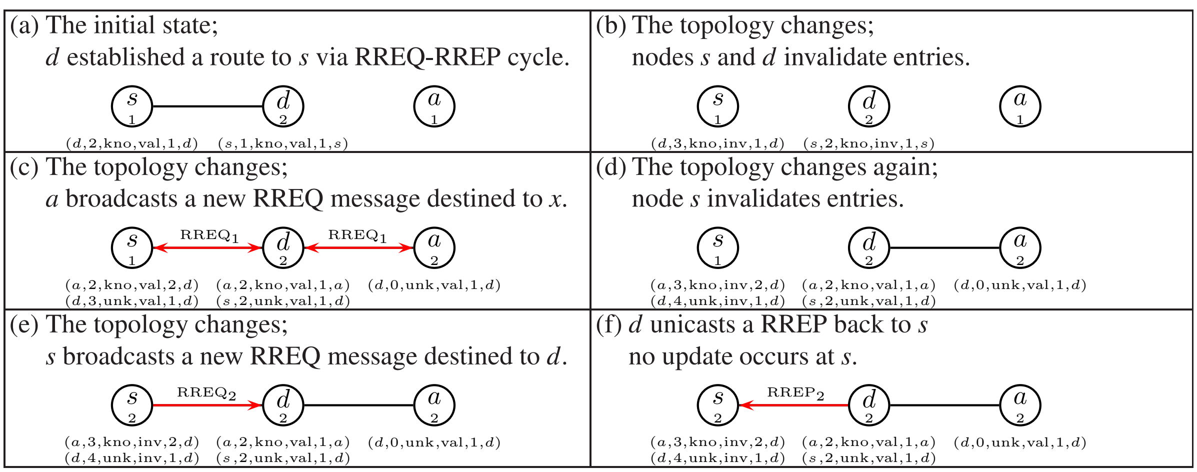



Figure 12: Failing to update the own sequence number before issuing a route reply

This is illustrated in Figure 12. In the initial state node *s* has a route to *d*, with destination sequence number (2) equal to *d*'s own sequence number; this is default behaviour of AODV. Due to a link break between *s* and *d*, node *s* increments its destination sequence number for *d* when invalidating the entry (Figure 12(b)). Afterwards, in Figure 12(c), the link comes up again, and when *d* forwards a RREQ message (from another node *a*, destined to an arbitrary node *x* that is not in the vicinity) to its neighbour *s*, node *s* validates its 1-hop route to *d*, without changing its destination sequence number. These events (link break – invalidation – link coming back up) are repeated at least once (Part (d)), resulting in a destination sequence number for *d* at node *s* that is at least 2 higher than *d*'s own sequence number. Now,

[52] It turns out that Resolutions (8b) and (8c) are compatible with (9b) after all; we skip the proof of this claim.

when $s$ searches for a route to $d$ (Figure 12(e)), $d$ will not update its own sequence number when sending a route reply to $s$, so the route reply will have outdated information (a too low sequence number) from the perspective of $s$, and thus will be ignored by $s$. No matter how often $s$ sends a new route request to $d$, it will never receive an answer that is good enough to restore its routing table entry to $d$.

**Process 11** RREQ handling (Resolution (10b))

```
RREQ(hops,rreqid,dip,dsn,dsk,oip,osn,sip,ip,sn,rt,rreqs,store) ≝
  ...       /* Lines 1–5 of Pro. 4 */
  (
    [ dip = ip ∧ inc(sn) = dsn ]      /* this node is the destination and the sequence number has to be updated */
      ⟦sn := inc(sn)⟧      /* update the sqn of ip */
      ...       /* Lines 9–17 of Pro. 4 */
    + [ dip = ip ∧ inc(sn) ≠ dsn ]      /* this node is the destination and the sequence number need no update */
      ...       /* Lines 9–17 of Pro. 4 */
    + [ dip ≠ ip ]      /* this node is not the destination node */
      ...       /* Lines 19–38 of Pro. 4 */
  )
```

In our specification we resolved this contradiction by following Sect. 6.1 of the RFC, in defiance of Sect. 6.6.1. The alternative is obtained by modifying the RREQ handling process as indicated in Pro. 11. As the above example shows, this alternative leads to a severely handicapped version of AODV, in which certain routes (in the example the one from $s$ to $d$) can not be established.[53]

### 8.2.5 Further Assumptions

During the creation of our specification (cf. Section 6), we did not only come along some ambiguities, we also found some unspecified cases—we were forced to specify these situations on our own.

#### Recording and Invalidating the Truly Unknown Sequence Number

When creating a routing table entry to a new destination—not already present in the routing table—for which no destination sequence number is known (i.e. in response to an AODV control message from a neighbour; following Lines 10, 14 and 18 of Pro. 1), the RFC does not stipulate how to fill in the destination-sequence-number field in the new entry. It does say

> “The sequence number is either determined from the information contained in the control packet, or else the valid sequence number field is set to false.”
>
> [79, Sect. 6.1]

Accordingly, the sequence-number-status flag in the entry is set to unk, but that does not tell what to fill in for the destination sequence number itself. Here, following the implementation AODV-UU [2], we use the special value 0, indicating a *truly unknown destination sequence number*. As this value does not represent a regular sequence number, we do not increment it when invalidating the entry.

#### Packet Handling

Even though not specified in the RFC, our model of AODV includes a mechanism for handling data packets—this is necessary to trigger any AODV activity. A data packet injected at a node $s$ by a client of

[53] On the IETF MANET mailing list (http://www.ietf.org/mail-archive/web/manet/current/msg02589.html) I. Chakeres proposes a third resolution of this ambiguity, namely “Immediately before a destination node issues a route reply in response to a RREQ, it MUST update its own sequence number to the maximum of its current sequence number and the destination sequence number in the RREQ packet plus one (1).” As this is not a possible reading of the RFC, it ought to be construed as proposal for improvement of AODV.

the protocol (normally the application layer in the protocol stack) for delivery at a destination $d$ towards which $s$ has no valid route, is inserted in a queue of packets for $d$ maintained by node $s$. In case there is no queue for $d$ yet, such a queue is created and a route discovery process is initiated, by means of a new route request. As long as that process is pending, no new route request should be issued when new packets for $d$ arrive; for it could be that packets for $d$ are injected by the application layer at a high rate, and sending a fresh route request for each of them would flood the protocol with useless RREQ messages. For this reason we await the route reply corresponding to the request, or anything else that creates a route to $d$. Afterwards packets to $d$ can be send, and the queue is emptied out. In case the route to $d$ is invalidated before the queue is empty, it is appropriate to initiate a new route discovery process, by generating a fresh route request. To this end we created the "request-required" flag, one for each queue, that is set when the route to the destination is invalidated, and unset when a new route request has been issued. The only sensible way we see to omit such a flag would be to use the non-existence of a queue of data packets for $d$ as the trigger to initiate a route request when a data packet for $d$ is posted at node $s$. But for that to work one would have to drop the entire queue of packets waiting for transmission towards $d$ when the route to $d$ is invalidated, just as packets are dropped when an intermediate node on the path towards $d$ loses its connection to the next hop.

**Receiving a RREP Message**
When an (intermediate) node receives a RREP message destined for a node $s$, it might happen that the node has an invalid routing table entry for $s$ only. The RFC does not consider this situation; however, this case *can* occur and must be specified. For our specification we decided that under these circumstances the AODV control message is lost and no error message is generated.

## 8.3 Implementations

To show that the ambiguities we found in the RFC and the associated problems are not only theoretically driven, but *do* occur in practice, we analyse five different open source implementations of AODV:

- *AODV-UU* [2] is an RFC compliant implementation of AODV, developed at Uppsala University. `http://aodvuu.sourceforge.net/`
- *Kernel AODV* [1] is developed at NIST and is another RFC compliant implementation of AODV. `http://w3.antd.nist.gov/wctg/aodv_kernel/`
- *AODV-UIUC* [57] (University of Illinois at Urbana-Champaign) is an implementation that is based on an early draft (version 10) of AODV. `http://sourceforge.net/projects/aslib/`
- *AODV-UCSB* [13] (University of California, Santa-Barbara) is another implementation based on an early draft (version 6). `http://moment.cs.ucsb.edu/AODV/aodv-ucsb-0.1b.tar.gz`
- *AODV-ns2* is an AODV implementation in the ns2 network simulator [76], originally developed by the CMU Monarch project and improved upon later by S. Das and E. Belding-Royer (the authors of the AODV RFC [79]). It is based on an early draft (version 8) of AODV. It is frequently used by academic and industry researchers to simulate AODV. `http://ns2.sourcearchive.com/documentation/2.35~RC4-1/aodv_8cc-source.html`

Even though the latter three implementations of AODV are not RFC compliant, they *do* capture the main aspects of the AODV protocol, as specified in the RFC [79]. As we have shown in the previous section, implementing the AODV protocol based on the RFC specification does not necessarily guarantee loop freedom. Therefore, we look at these five concrete AODV implementations to determine whether any of them is susceptible to routing loops. AODV-UU, Kernel AODV and AODV-UIUC maintain an invalidation procedure that conforms to Resolution (8a), whereas AODV-UCSB and AODV-ns2 follow

Resolution (8b). Since both resolutions give rise to routing loops when used in combination with non-optimal self-entries, we examine the code of these implementations to see if routing loops such as the one described in Figure 11 occur. The results of this analysis are summarised in Table 7.

| Implementation | Analysis |
|---|---|
| AODV-UU [2] | Loop free, since self-entries are explicitly excluded. |
| Kernel AODV [1] | Loop free, due to optimal self-entries. |
| AODV-UIUC [57] | Yields routing loops, through sequence number reset. |
| AODV-UCSB [13] | Yields routing loops, through sequence number reset. |
| AODV-ns2 | Yields routing loops, since it implements Resolution (8b) of the invalidation procedure presented in Section 8.2.3 and does allow self-entries. |

Table 7: *Analysis of AODV implementations*

In AODV-UU, self-entries are never created because a check is always performed on an incoming RREP message to make sure that the destination IP address is not the same as the node's own IP address, just as in Resolution (5b). By Corollary 8.6, this interpretation of the RFC is loop free.

In Kernel AODV, an optimal self-entry is always maintained by every node in the network, just as in Resolution (6b). By Corollary 8.6, this interpretation of the RFC is loop free.

Both AODV-UIUC and AODV-UCSB allow non-optimal self-entries to occur in nodes (Resolution (5a) of Ambiguity 5). These are generated based on information contained in received RREP messages. While self-entries are allowed, the processing of RERR messages in AODV-UIUC and AODV-UCSB does not adhere to the RFC specification (or even the draft versions that these implementation are based upon). Due to this non-adherence, we are unable to re-create the routing loop example of Figure 11. However, if both AODV-UIUC and AODV-UCSB were to strictly follow the RFC specification with respect to the RERR processing, loops would have been created.

Even though the routing loop example of Figure 11 could not be recreated in AODV-UIUC or AODV-UCSB, both implementations allow a decrease of destination sequence numbers in routing table entries to occur, by following Resolution (2b).[54] This gives rise to routing loops in the way described in Section 8.1.

In AODV-ns2, self-entries are allowed to occur in nodes. Unlike AODV-UIUC and AODV-UCSB, the processing of RERR messages follows the RFC specification. However, whenever a node generates a RREQ message, sequence numbers are incremented by two instead of by one as specified in the RFC. We have modified the AODV-ns2 code such that sequence numbers are incremented by one whenever a node generates a RREQ message, and are able to replicate the routing loop example presented in [39][55] in the ns2 simulator, with the results showing the existence of a routing loop between nodes $s$ and $x$. However, even if the code remains unchanged and sequence numbers are incremented by two, AODV-ns2 can still yield loops; the example is very similar to the one presented and only varies in subtle details.

In sum, we discovered not only that three out of five AODV implementations can produce routing loops, but also that there are essential differences between the various implementations in various aspects of protocol behaviour. This is due to different interpretations of the RFC.

[54] AODV-ns2 follows Resolution (2a), whereas AODV-UU follows (2d). Kernel AODV is not compliant with the RFC in this matter and operates differently.

[55] The example in [39] is a simplification of the one in Figure 11, but is based on the interpretation of AODV without the sequence-number-status flag, following Resolution (2d). The example of Figure 11 itself works equally well in the presence of that flag.

## 8.4 Summary

The following table summarises the ambiguities we discovered, as well as their consequences. The resolutions coloured red lead to unacceptable protocol behaviour, such as routing loops. The white and green resolutions are all acceptable readings of the RFC; the green ones have been chosen in our default specification of Sections 5 and 6. The section numbers refer to the RFC [79].

| | | |
|---|---|---|
| **Updating Routing Table Entries** | | |
| **1. Updating the Unknown Sequence Number in Response to a Route Reply** | | |
| 1a. | the destination sequence number (DSN) is copied from the RREP message (Sect 6.7) | decrement of sequence numbers and loops |
| 1b. | routing table is not updated when the information that it has is "fresher" (Sect. 6.1) | does not cause loops; used in our specification |
| **2. Updating with the Unknown Sequence Number (Sect. 6.5)** | | |
| 2a. | no update occurs | does not cause loops, but opportunity to improve routes is missed |
| 2b. | overwrite any routing table entry by an update with an unknown DSN | decrement of sequence numbers and loops |
| 2c. | use the new entry with the old DSN | does not cause loops; used in our specification |
| 2d. | use the new entry with the old DSN and DSN-flag | does not cause loops |
| **3. More Inconclusive Evidence on Dealing with the Unknown Sequence Number (Sect. 6.2)** | | |
| 3a. | update when *incoming* sequence number is unknown | supports Interpretations 2b or 2c above; used in our specification |
| 3b. | update when *existing* sequence number is *marked as* unknown | decrement of sequence numbers and loops; implies 1a and 2a |
| 3c. | update when no *existing* sequence number is known | supports Interpretation 2a above |
| **4. Updating Invalid Routes** | | |
| 4a. | update an invalid route when the new route has the same sequence number (Sect. 6.1) | does not cause loops; used in our specification |
| 4b. | do not update an invalid route when the new route has the same sequence number (Sect. 6.2) | results in handicapped version of AODV, in which many broken routes will never be repaired. |
| **Self-Entries in Routing Tables** | | |
| **5. (Dis)Allowing Self-Entries** | | |
| 5a. | allow (arbitrary) self-entries | loop free if used with appropriate `invalidate`; used in our specification |
| 5b. | disallow (non-optimal) self-entries; if self-entries would be created, ignore message | does not cause loops |
| 5c. | disallow (non-optimal) self-entries; if self-entries would be created, forward message | does not cause loops |
| **6. Storing the Own Sequence Number** | | |
| 6a. | store sequence number as separate value | does not cause loops; used in our specification |
| 6b. | store sequence number inside routing table | does not cause loops |
| **Invalidating Routing Table Entries** | | |
| **7. Invalidating Entries in Response to a Link Break or Unroutable Data Packet (Sect. 6.11)** | | |
| 7a. | "it" refers to routing table entry | does not cause loops; used in our specification |
| 7b. | "it" refers to DSN | loops |

| 8. Invalidating Entries in Response to a Route Error Message | | |
|---|---|---|
| 8a. | copy DSN from RERR message (Sect. 6.11) | decrement of sequence numbers and loops (when allowing self-entries (Interpretation 5a)) |
| 8b. | no action if the DSN in the routing table is larger than the one in the RERR mess. (Sect. 6.1 & 6.11) | loops (when allowing self-entries) |
| 8c. | take the maximum of the DSN of the routing table and the one from the RERR message | loops (when allowing self-entries) |
| 8d. | take the maximum of the increased DSN of the routing table and the one from the RERR mess. | does not cause loops |
| 8e. | combine 8b and 8d | does not cause loops |
| 8f. | only invalidate if the DSN in the routing table is smaller than the one from the RERR message | does not cause loops; used in our specification |
| **Further Ambiguities** | | |
| **9. Packet Handling for Unknown Destinations (Sect. 6.11)** | | |
| 9a. | do nothing | the sender is not informed and keeps sending; used in our specification |
| 9b. | broadcast RERR message with unknown DSN | loop free if used with adequate `invalidate` |
| **10. Setting the Own Sequence Number when Generating a RREP Message** | | |
| 10a. | taking max (Sect. 6.1) | used in our specification |
| 10b. | taking the "conditional increment" (Sect. 6.6.1) | loss of RREP message |

Table 8: *Different interpretations and consequences of ambiguities in the RFC*

The above classification of ambiguities and their resolutions can be used to calculate the number of possible readings of the RFC. The table shows that the resolution for Ambiguity 3 is uniquely determined by the choice of resolutions for Ambiguities 1 and 2; except for the case of taking (1a) in combination with (2a); here Resolutions (3b) and (3c) are possible. Hence Ambiguity 3 only adds one new variant. In sum we have $[(2\times 4)+1]\times 2\times 3\times 2\times 2\times 6\times 2\times 2 = 5184$ possible interpretations of the AODV RFC. Only $\big([(1\times 3)+0]\times 1\times [(3\times 2\times 1\times 3\times 2)+(5\times 1\times 5)]\times 1\big) - 5 = 178$ are loop free and without major flaws. (Here the first "5" refers to all resolutions of Ambiguities 5 and 6 except for the combination of (5a) and (6a); the second "5" refers to the first 3 resolutions of Ambiguity 8 and both resolutions of Ambiguity 9, except for the combination of (8a) and (9b); and the last "5" deducts the combinations of (6b) with (2d), (5a) and one of the second "5").

All these ambiguities, missing details and misinterpretations of the RFC show that the specification of a reasonably rich protocol such as AODV cannot be described by simple (English) text; is *has to be done* using formal methods in a precise way.

## 9 Formalising Temporal Properties of Routing Protocols

Our formalism enables verification of correctness properties. While some properties, such as loop freedom and route correctness, are invariants on routing tables, others require reasoning about the temporal order of transitions. Here we use Linear-time Temporal Logic (LTL) [86] to specify and discuss two of such properties, namely *route discovery* and *packet delivery*.

Let us briefly recapitulate the syntax and semantics of LTL. The logic is built from a set of *atomic propositions*. Such propositions stand for facts that may hold at some point (in some state) during a protocol run. An example is "two nodes are connected in the (current) topology".

LTL formulas are interpreted on paths in a transition system, where each state is labelled with the atomic propositions that hold in that state. A *path* is an alternating sequence of states and transitions, starting from a state and either being infinite or ending in a state, such that each transition in the sequence goes from the state before to the state after it. An atomic proposition $p$ holds on a path $\pi$ if $p$ holds in the first state of $\pi$.

LTL [86] uses the temporal operators $\mathbf{G}$ and $\mathbf{F}$. The formulas $\mathbf{G}\phi$ and $\mathbf{F}\phi$ mean that $\phi$ holds *globally* in all states on a path, and *eventually* in some state, respectively. Here a formula $\phi$ is deemed to *hold in a state on a path* $\pi$ iff it holds for the remainder of $\pi$ when starting from that state. In later work on LTL, two more temporal operators were added—the *next-state* and the *until* operator; these will not be needed here. LTL formulas can be combined by the logical connectives *conjunction* $\wedge$, *disjunction* $\vee$, *implication* $\Rightarrow$ and *negation* $\neg$. An LTL formula holds for a transition system iff it holds for all *complete* paths in the system starting from an initial state. A path is complete iff it leaves no transitions undone without a good reason; in the original work on temporal logic [86] the complete paths are exactly the infinite ones, but in Section 9.1 we will propose a different concept of completeness (cf. Definition 9.1).

Below we will apply LTL to the transition system $\mathscr{T}$ generated by the structural operational semantics of AWN from an arbitrary AWN specification, and from our specification of AODV in particular. Here we use two kinds of atomic propositions. The first kind are predicates on the states (or network expressions) $N$ that are fully determined by the (local) values of all variables maintained by the nodes in the network, as well as by the current topology, i.e. by $\xi_N^{ip}$, $\zeta_N^{ip}$ and $R_N^{ip}$ for all $ip \in \mathbf{IP}$. The second kind are predicates on transitions $N \xrightarrow{\ell} N'$ that are fully determined either by the label $\ell$ of the transition, or by transition-labels appearing in the derivation from the structural operational semantics of AWN of a $\tau$-transition—compare the $R\!:\!\textbf{*cast}(m)$-transitions in Section 7.1.

To incorporate the transition-based atomic propositions into the framework of temporal logic, we perform a translation of the transition-labelled transition system $\mathscr{T}$ into a state-labelled transition system $\mathscr{S}$, and apply LTL to the latter. A suitable translation, proposed in [19], introduces new states halfway the existing transitions, thereby splitting a transition $\ell$ into $\ell;\tau$, and attaches transition labels, or predicates evaluated on transitions, to the new mid-way states. Since we also have state-based atomic propositions, we furthermore declare any atomic proposition that holds in state $N'$ to also hold for the new state midway a transition $N \xrightarrow{\ell} N'$.

Below we use LTL to formalise properties that say that whenever a precondition $\phi^{pre}$ holds in a reachable state, the system will eventually reach a state satisfying the postcondition $\phi^{post}$. Such a property is called an *eventuality property* in [86]; it is formalised by the LTL formula

$$\mathbf{G}\big(\phi^{pre} \;\Rightarrow\; \mathbf{F}\phi^{post}\big)\ . \tag{33}$$

However, sometimes we want to guarantee such a property only when a side condition $\psi$ keeps being satisfied from the state where $\phi^{pre}$ holds until $\phi^{post}$ finally holds. There are three ways to formalise this:

$$\mathbf{G}\big((\phi^{pre} \wedge \mathbf{G}\psi) \Rightarrow \mathbf{F}\phi^{post}\big) \qquad \mathbf{G}\big((\phi^{pre} \wedge \psi\mathbf{W}\phi^{post}) \Rightarrow \mathbf{F}\phi^{post}\big) \qquad \mathbf{G}\big(\phi^{pre} \Rightarrow \mathbf{F}(\phi^{post} \vee \neg\psi)\big)\ . \tag{34}$$

The first formula is derived from (33) by adding to the precondition $\phi^{pre}$ the requirement that $\psi$ is valid as well, and remains valid ever after. If that precondition is not satisfied, nothing is required about $\phi^{post}$. One might argue that this precondition is too strong: it requires the side condition to be valid forever, even after $\phi^{post}$ has occurred. The second formula addresses this issue by weakening the precondition $\phi^{pre} \wedge \mathbf{G}\psi$. It uses a binary temporal operator $\mathbf{W}$—the *weak until* operator—that can be expressed in terms of $\mathbf{G}$ and the (strong) until operator. The meaning of an expression $\psi\mathbf{W}\phi$ is that either $\psi$ holds forever, or at some point $\phi$ holds and until then $\psi$ holds. In other words, $\psi$ holds until we reach a state where $\phi$ holds, or forever if the latter never happens.

Although the precondition of the second formula is weaker than the one of the first, as a whole the two formulas are equivalent: they are satisfied by all runs of the system, except those for which

– at some point $\phi^{pre}$ holds,
– and from that point onwards $\psi$ remains valid,
– yet never a state occurs satisfying $\phi^{post}$.

Both formulas are also equivalent to the third formula in (34). It can be understood to say that once $\phi^{pre}$ holds, we will eventually reach a state where $\phi^{post}$ holds, except that we are off the hook (in the sense that nothing further is required) when (prior to that) we reach a state where $\psi$ fails to hold. It is this last form that we will use further on.

## 9.1 Progress, Justness and Fairness

In Sections 9.2 and 9.3, we will formalise properties that say that under certain conditions some desired activity will eventually happen, or some desired state will eventually be reached. As a particularly simple instance of this, consider the transition systems in Figures 13(a)–(c), where the double-circled state satisfies a desired property $\phi$. The formula $\mathbf{G}(a \Rightarrow \mathbf{F}\phi)$ says that once the action $a$ occurs, eventually we will reach a state where $\phi$ holds. In this section we investigate reasons why this formula might not hold, and formulate assumptions that guarantee that it does.

**Progress.** The first thing that can go wrong is that the process in Figure 13(a) performs $a$, thereby reaching the state $s$, and subsequently remains in the state $s$ without ever performing the internal action $\tau$ that leads to the desired state $t$, satisfying $\phi$. If there is the possibility of remaining in a state even when there are enabled internal actions, no useful temporal property about processes will ever be guaranteed. We therefore make an assumption that rules out this type of behaviour.

*A process in a state that admits an internal transition $\tau$ will eventually perform a transition.* $(P_1)$

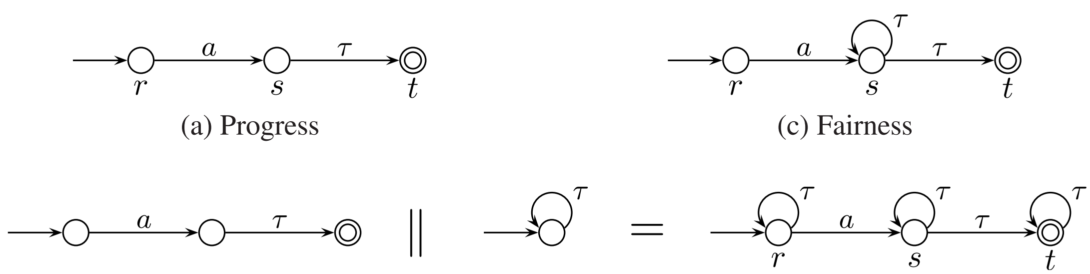
$(P_1)$ is called a *progress* property. It guarantees that the process depicted in Figure 13(a) satisfies the LTL formula $\mathbf{G}(a \Rightarrow \mathbf{F}\phi)$. We do not always assume progress when only external transitions are possible.[56] For instance, the process of Figure 13(a), when in its initial state $r$, will not necessarily perform the $a$-transition, and hence need not satisfy the formula $\mathbf{F}\phi$. The reason is that external transitions could be synchronisations with the environment, and the environment may not be ready to synchronise. This can happen for instance when $a$ is the action $\mathbf{receive}(m)$. However, for our applications it makes sense to distinguish two kinds of external transitions: those whose execution requires cooperation from the environment in which the process runs, and those who do not. The latter kind could be called *output transitions*. As far as progress properties go, output transitions can be treated just like internal transitions:

*A process in a state that admits an output transition will eventually perform a transition.* $(P_2)$

Whether a transition is an output transition is completely determined by its label; hence we also speak of *output actions*. In case $a$ is an output action, which can happen independent of the environment, the formula $\mathbf{F}\phi$ does hold for the process of Figure 13(a).

We formalise $(P_1)$ and $(P_2)$ through a suitable definition of a complete path. In early work on temporal logic, formulas were interpreted on Kripke structures: transition systems with unlabelled transitions, subject to the condition of *totality*, saying that each state admits at least one outgoing transition. In this context, the complete paths are defined to be all infinite paths of the transition system. When giving up totality, it is customary to deem complete also those paths that end in a state from which no further transitions are possible [19]. Here we go a step further, and (for now) define a path to be *complete* iff it is either infinite or ends in a state from which no further *internal or output* transitions are possible. This definition exactly captures the progress properties $(P_1)$ and $(P_2)$ proposed above. (Dropping all progress properties amounts to defining *each* path to be complete.) Below we will restrict the notion of a complete path to also capture a forthcoming justness property.

[56] A transition is external iff it is not internal, i.e. iff its label is different from $\tau$.




(a) Progress

(c) Fairness

(b) Justness

Figure 13: Progress, Justness and Fairness

It remains to be determined which transitions generated by the structural operational semantics of AWN should be classified as output transitions. In the transition system for (encapsulated) network expressions generated by the rules of Table 4, only five types of transition labels occur: **connect**$(ip, ip')$, **disconnect**$(ip, ip')$, $ip$:**newpkt**$(d, dip)$, $ip$:**deliver**$(d)$ and $\tau$. These are all actions to be considered, since we regard (LTL-)properties on network expressions only. The actions **connect**$(ip, ip')$, **disconnect**$(ip, ip')$ and $ip$:**newpkt**$(d, dip)$ are entirely triggered by the environment of the network, and thus cannot be classified as output actions. Transitions labelled $\tau$ are internal. For transitions labelled $ip$:**deliver**$(d)$ two points of view are possible. It could be that the action $ip$:**deliver**$(d)$ is seen as attempt of the network to synchronise with its client in delivering a message; the synchronisation will then happen only when both the network and the client are ready to engage in this activity. A possible scenario would be that Pro. 3 gets stuck in Line 2 because the client is not ready for such a synchronisation (the same happens in Pro. 2, Line 2). This interpretation of our formalisation of AODV would give rise to deadlock possibilities that violate useful properties we would like the protocol to have, such as the forthcoming *route discovery* and *packet delivery* properties. We therefore take the opposite point of view by classifying $ip$:**deliver**$(d)$ as an output action. Hereby we disallow a deadlock when attempting a **deliver**-action, since the environment of the network cannot prevent delivery of data packets. As a consequence, finite complete paths of AODV can end only in states $N$ where all message queues are empty, all nodes $ip$ are are either in their initial state or about to call the process AODV,[57] and for all destinations $dip$ for which $ip$ has a (non-empty) queue of data packets we have $dip \notin \mathtt{vD}_N^{ip}$ and $\sigma_{p\text{-}flag}(\xi_N^{ip}(\mathtt{store}), dip) = \mathtt{no\text{-}req}$. This follows since our specification of AODV is input-enabled, is non-blocking, and avoids livelocks.

In the remainder of this paper we will only use LTL-formulas to check (encapsulated) network expressions. However, when defining output transitions also on partial networks, parallel processes and sequential processes, it is easy to carry over our mechanism to arbitrary expressions of AWN. On the level of partial network expressions $R$:***cast**$(m)$ counts as an output action, as its occurrence cannot be prevented by other nodes in the network. Similarly, on the level of sequential and parallel processes **broadcast**$(m)$, **groupcast**$(D, m)$, **unicast**$(dip, m)$, $\neg$**unicast**$(dip, m)$ and **deliver**$(d)$ are output actions, but **send**$(m)$ is not, for it requires synchronisation with **receive**$(m)$. The remaining actions (**arrive**$(m)$, **receive**$(m)$) are not considered output actions.

**Justness.** Now suppose we have two concurrent systems that work independently in parallel, such as two completely disconnected nodes in our network. One of them is modelled by the transition system of Figure 13(a), and the other is doing internal transitions in perpetuity. The parallel composition is depicted on the left-hand side of Figure 13(b). According to our structural operational semantics, the overall transition system resulting from this parallel composition is the one depicted on the right. In this

[57]More precisely these positions are at the beginning of Pro. 1, Line 25, Pro. 4, Lines 2, 26, 37, Pro. 5, Lines 6, 14, 23, 27, and in the middle of Lines 32, 39 (Pro. 1), 2, 4 (Pro. 2), 2, 7, 14, 20, 22 (Pro. 3), 10, 17, 33 (Pro. 4), 21 (Pro. 5), 8 (Pro. 6).

transition system, the LTL formula $\mathbf{G}(a \Rightarrow \mathbf{F}\phi)$ is not valid, because, after performing the action $a$, the process may do an infinite sequence of internal transitions that stem from the other component in the parallel composition, instead of the transition to the desired success state. Yet the formula $\mathbf{G}(a \Rightarrow \mathbf{F}\phi)$ does hold intuitively, because no amount of internal activity in the remote node should prevent our own node from making progress. That this formula does not hold can be seen as a pitfall stemming from the use of interleaving semantics. The intended behaviour of the process is captured by the following *justness* property:[58]

> *A component in a parallel composition in a state that admits an internal or output transition will eventually perform a transition.* $(J)$

Progress can be seen as a special case of justness, obtained by regarding a system as a parallel composition of one component only. We will formalise the justness requirement $(J)$ by fine-tuning our definition of a complete path.

Any path $\pi^{[]}$ starting from an AWN network expression $[M]$ is derived through the structural operational semantics of Table 4 from a path $\pi^{\|}$ starting from the partial network expression $M$. All states occurring in $\pi^{[]}$ have the form $[M']$ for some partial network expression $M'$, and in $\pi^{\|}$ such a state is replaced by $M'$. Moreover, some transition labels $\tau$ in $\pi^{[]}$ are replaced by $R\!:\!\textbf{*cast}(m)$ in $\pi^{\|}$, and transition labels $ip\!:\!\textbf{newpkt}(d,dip)$ are replaced by $\{ip\}\neg K\!:\!\textbf{arrive}(\texttt{newpkt}(d,dip))$. To indicate the relationship between $\pi^{[]}$ and $\pi^{\|}$ we write $\pi^{[]} = [\pi^{\|}]$. It might be that $\pi^{\|}$ is not uniquely determined by $\pi^{[]}$; if this happens, the partial network expression $M$ admits different paths that upon encapsulating become indistinguishable.

In the same way, any path $\pi^{\|}$ starting from a partial network expression $M$ that happens to be a parallel composition of $n$ node expressions derives through the structural operational semantics of Table 4 from $n$ paths $\pi_1^{::},\dots,\pi_n^{::}$ starting from each of these node expressions. In this case we write $\pi^{\|} = \pi_1^{::}\|\cdots\|\pi_n^{::}$. Here it could be that $\pi^{\|}$ is infinite, yet some (but not all) of the $\pi_i^{::}$ are finite. As before, it might be that the $\pi_i^{::}$ are not uniquely determined by $\pi^{\|}$.

Zooming in further, any path $\pi^{::}$ starting from a node expression $ip:P:R$ derives through the structural operational semantics of Table 3 from a path $\pi^{\langle\!\langle}$ starting from the parallel process expression $P$. As transitions labelled $\textbf{connect}(ip,ip')$ or $\textbf{disconnect}(ip,ip')$ occurring in $\pi^{[]}$, $\pi^{\|}$ and $\pi^{::}$ do not occur in $\pi^{\langle\!\langle}$, it can be that $\pi^{\langle\!\langle}$ is finite even though $\pi^{::}$ is infinite. We write $\pi^{::} = ip:\pi^{\langle\!\langle}:*$ (without filling in the $R$, since it may change when following $\pi^{::}$).

Finally, any path $\pi^{\langle\!\langle}$ of a parallel process expression $P$ that is the parallel composition of $m$ sequential process expressions derives through the structural operational semantics of Table 2 from $m$ paths $\pi_1,\dots,\pi_m$ starting from each of these sequential process expressions. In this case we write $\pi^{\langle\!\langle} = \pi_1 \langle\!\langle \cdots \langle\!\langle \pi_m$. Again it may happen that $\pi^{\langle\!\langle}$ is infinite, yet some (but not all) of the $\pi_i$ are finite.

**Definition 9.1** A path starting from any AWN expression (i.e. a sequential or parallel process expression, a node expression or (partial) network expression) *ends prematurely* if it is finite and from its last state an internal or output transitions is possible.

- A path $\pi_i$ starting from a sequential process expression is *complete* if it does not end prematurely—hence is infinite or ends in a state from which no further internal or output transitions are possible.
- A path $\pi^{\langle\!\langle}$ starting from a parallel process expression is *complete* if it does not end prematurely and can be written as $\pi_1 \langle\!\langle \cdots \langle\!\langle \pi_m$ where each of the $\pi_i$ is complete.
- A path $\pi^{::}$ starting from a node expression is *complete* if it does not end prematurely and can be written as $ip:\pi^{\langle\!\langle}:*$ where $\pi^{\langle\!\langle}$ is complete.

[58] In the literature *justness* is often used as a synonym for *weak fairness*, defined on Page 95—see, e.g., [64]. In this paper we introduce a different concept of justness: fairness is a property of schedulers that repeatedly choose between several tasks, whereas justness is a property of parallel-composed transition systems, guaranteeing progress of all components.

- A path $\pi^{\|}$ starting from a partial network expression is *complete* if it does not end prematurely and can be written as $\pi_1^{::}\|\cdots\|\pi_n^{::}$ where each of the $\pi_i^{::}$ is complete.
- A path $\pi^{[]}$ starting from a network expression is *complete* if it does not end prematurely and can be written as $[\pi^{\|}]$ where $\pi^{\|}$ is complete.

Note that if $\pi^{[]} = [\pi^{\|}]$ and $\pi^{[]}$ ends prematurely, then also $\pi^{\|}$ ends prematurely. This holds because any internal or output action enabled in the last state of $\pi^{[]}$ must stem from an internal or output action enabled in the last state of $\pi^{\|}$. For this reason the requirement "it does not end prematurely" is redundant in the above definition of complete path starting from a network expression. For the same reason this requirement is redundant in the definition of a complete path for node expressions or partial network expressions, but not in the definition for parallel process expressions. The reason for including this requirement in each part of the definition above, is to establish a general pattern that ought to lift smoothly to languages other than AWN.

This definition of a complete path captures our (progress and) justness requirement, and ensures that the formula $\mathbf{G}(a \Rightarrow \mathbf{F}\phi)$ holds for the process of Figure 13(b). For example, the infinite path $\pi$ starting from $r$ that after the $a$-transitions keeps looping through the $\tau$-loop at $s$ can only be derived as $\pi_1\|\pi_2$, where $\pi_1$ is a finite path ending right after the $a$-transitions. Since $\pi_1$ fails to be complete (because its ends prematurely, by its end state admitting a $\tau$-transition), $\pi$ is defined to be incomplete as well, and hence does not count when searching for a complete path that fails to satisfy the formula.

**Fairness.** With the justness requirement $(J)$[59] embedded in our semantics of LTL, the processes of Figure 13(a)–(b) satisfy the formula $\mathbf{G}(a \Rightarrow \mathbf{F}\phi)$. Yet, the process of Figure 13(c) does not satisfy this formula. The reason is that in state $s$ a choice is made between two internal transitions. One leads to the desired state satisfying $\phi$, whereas the other gives the process a chance to make the decision again. This can go wrong in exactly one way, namely if the $\tau$-loop is chosen each and every time.

For some applications it is warranted to make a *global fairness assumption*, saying that in verifications we may simply assume our processes to eventually escape from a loop such as in Figure 13(c) and do the right thing. A process-algebraic verification approach based on such an assumption is described in [3]. Moreover, a global fairness assumption is incorporated in the weak bisimulation semantics employed in [71].

An alternative approach, which we follow here, is to explicitly declare certain choices to be fair, while leaving open the possibility that others are not. To see which choices come into question, we search for all occurrences of the choice operator $+$ in our AODV specification in Processes 1–7. A nondeterministic choice occurs in Lines 21 and 33 of Pro. 1 and in Lines 3 and 8 of Pro. 7. All other occurrences of the $+$-operator are of the form $[\varphi_1]p + \ldots + [\varphi_n]q$ where the guards $\varphi_i$ are mutually exclusive; these are deterministic choices, where in any reachable state at most one of the alternatives is enabled.

Considering Lines 1, 21 and 33 of Pro. 1, the process AODV running on a node in a network can be seen as a scheduler that needs to schedule three kinds of tasks. Lines 1–20 deal with handling an incoming message. This task is enabled when there is a message in the message queue of that node. Lines 21–32 deal with sending a data packet towards a destination *dip*. This task is enabled when there is a queued data packet for destination *dip*, i.e. $dip \in \mathtt{qD}(\xi(\mathtt{store}))$, and moreover a valid route to *dip* exists, i.e. $dip \in \mathtt{vD}(\xi(\mathtt{rt}))$. As data queues for multiple destinations *dip* may have formed, each time when sending a data packet is scheduled a choice is made which destination to serve. Finally, Lines 33–39 deal with the initiation of a route discovery process for destination *dip*. It is enabled when the guard of Line 33 evaluates to `true`. No matter which of these tasks is chosen, the chosen instance always

[59] Remember that $(J)$ implies the progress requirements $(P_1)$ and $(P_2)$.

terminates in a finite amount of time,[60] after which the AODV-scheduler needs to make another choice.

For each of these tasks we postulate a *weak fairness* property. It requires that if this task, from some point onwards, is perpetually enabled, it will eventually be scheduled. A weak fairness property is expressed in LTL as the requirement $\mathbf{G}(\mathbf{G}\psi \Rightarrow \mathbf{F}\phi)$; here $\psi$ is the condition that states that the task is enabled, whereas $\phi$ states that it is being executed.[61] The property says that if the condition $\psi$ holds uninterruptedly from some time point onwards, then eventually $\phi$ will hold. This is the first formula of (34) with $\phi^{pre} = \mathtt{true}$ and $\phi^{post} = \phi$. Hence a logically equivalent formula is $\mathbf{GF}(\phi \vee \neg\psi)$. Another equivalent formula expressing weak fairness is $\mathbf{FG}\psi \Rightarrow \mathbf{GF}\phi$. It says that if, from some point onwards, a task is perpetually enabled, it will be scheduled infinitely often.[62]

Sometimes a *strong fairness* property is needed, saying that if a task is enabled infinitely often,[63] but allowing interruptions during which it is not enabled, it will eventually be scheduled. Such a property is expressed in LTL as $\mathbf{G}(\mathbf{GF}\psi \Rightarrow \mathbf{F}\phi)$,[64] or equivalently $\mathbf{GF}\psi \Rightarrow \mathbf{GF}\phi$. We do not need strong fairness properties in this paper.

Our first fairness property ($\mathrm{F}_1$) requires that if the guard of Pro. 1, Line 21 evaluates to `true` from some state onwards, for a particular value of *dip*, then eventually Line 21 (or equivalently Line 22 or 23) will be executed, for that value of *dip*. Naturally, such a property needs to be required for each node *ip* in the network, and for each possible destination *dip*. Later, we will formulate a *packet delivery* property, saying that under certain circumstances a data packet will surely be delivered to its destination. Without the fairness property ($\mathrm{F}_1$) there is no hope on such a property being satisfied by AODV. It could be that a node *ip* with a valid route to *dip* has a queued data packet for *dip*, but will never send it, because it is constantly busy processing messages—that is, executing Line 1 instead of Line 21. Alternatively, it could be that the node has a constant supply of data packets for another destination $dip'$, and always chooses to send a packet to $dip'$ instead of to *dip*.

Fairness property ($\mathrm{F}_1$) can be formalised as an instance of the template $\mathbf{G}(\mathbf{G}\psi \Rightarrow \mathbf{F}\phi)$ by taking $\psi$ to be the formula that says that the guard in Line 21 is satisfied, and $\phi$ a formula that holds after Line 21 has been executed. We take $\psi$ to be the atomic proposition $dip \in \mathtt{qD}^{ip} \cap \mathtt{vD}^{ip}$, which we define to hold for state $N$ iff $dip \in \mathtt{qD}(\xi_N^{ip}(\mathtt{store})) \cap \mathtt{vD}_N^{ip}$. Other atomic propositions used below are defined along the same lines. In order to formulate $\phi$ we use the atomic proposition $\mathbf{unicast}(*,\mathtt{pkt}(*,dip,ip))$, which is defined to hold when node *ip* tries to unicast a data packet with destination *dip*. Thus we require, for all $ip, dip \in \mathbf{IP}$, that

$$\mathbf{G}\big(\mathbf{G}(dip \in \mathtt{qD}^{ip} \cap \mathtt{vD}^{ip}) \Rightarrow \mathbf{F}\big(\mathbf{unicast}(*,\mathtt{pkt}(*,dip,ip))\big)\big). \qquad (\mathrm{F}_1)$$

($\mathrm{F}_1$) says that whenever the node *ip* perpetually has queued packets for the destination *dip* as well as a valid route to *dip*, it will eventually forward a data packet originating from *ip* towards *dip*—i.e. Line 23 will be executed. In classifying this property as a weak fairness property, we count a task as enabled when its guard is valid, notwithstanding that the task cannot be started during the time AODV is working on a competing task.

Our second fairness property ($\mathrm{F}_2$) demands fairness for the task starting with Line 33 of Pro. 1. We require, for all $ip, dip \in \mathbf{IP}$, that

$$\mathbf{G}\big(\mathbf{G}(dip \in \mathtt{qD}^{ip} - \mathtt{vD}^{ip} \wedge \sigma_{p\text{-}flag}^{ip}(dip) = \mathtt{req}) \Rightarrow \mathbf{F}\big(\mathbf{broadcast}(\mathtt{rreq}(*,*,dip,*,*,ip,*,ip))\big)\big). \quad (\mathrm{F}_2)$$

[60] Here we use that each of these tasks consists of finitely many actions, of which only the initial one could be blocking. The task of handling an incoming message could fail to terminate if the message received is not of the form specified in any of the guards of Lines 4, 6, 8, 12 or 16; in this case a deadlock would occur in Line 3. However, using Proposition 7.1(a), this will never happen, as all messages sent have the required form.

[61] These properties were introduced and formalised in LTL in [28] under the name “responsiveness to insistence”. They were deemed “the minimal fairness requirement” for any scheduler.

[62] or is scheduled in the final state of the system. This possibility needs to be added because, unlike in [86, 28], we allow complete paths to be finite.

[63] or in the final state of the system

[64] These properties were introduced and formalised in LTL in [28] under the name “responsiveness to persistence”.

($F_2$) says that whenever *ip* perpetually has queued packets for *dip* but no valid route to *dip*, and the request-required flag at *ip* for destination *dip* is set to `req`, indicating that a new route discovery process needs to be initiated, then node *ip* does issue a request for a route from *ip* to *dip*—so Line 39 will be executed.

We do not formalise a fairness property saying that Line 1 of Pro. 1 will be executed eventually. Since the **receive**-action of Line 1 of Pro. 1 has to synchronise with the **send**-action in Line 6 of Pro. 7 it suffices to formalise a fairness property for `QMSG`.

Process `QMSG` can be understood as scheduling two tasks: (1) store an incoming message at the end of the message queue, and (2) pop the top message from the queue and send it to `AODV` for handling. The reason that (1) occurs twice in the specification (Lines 1–2 as well as 7–8) is that we require our node to be input enabled, meaning that (1) must be possible in every state.

Our third and last fairness property ($F_3$) guards against starvation of task (2). It says that if the guard of Line 3 of Pro. 7 evaluates to `true` from some state onwards, then eventually Line 6 of Pro. 7 will be executed. In order to formulate this property we use the atomic propositions $\mathtt{msgs}^{ip} \neq []$, which holds in state $N$ iff $\xi_N^{ip}(\mathtt{msgs}) \neq []$, and $ip:\mathbf{send}(*)$, saying that the process `QMSG` running on node *ip* performs a **send**-action. We need to explicitly annotate this activity with the name of node *ip*, as—unlike for **unicast** and **broadcast**—this information cannot be derived from the message being sent. We require, for all $ip \in \mathbf{IP}$, that

$$\mathbf{G}\big(\mathbf{G}(\mathtt{msgs}^{ip} \neq []) \ \Rightarrow\ \mathbf{F}\big(ip:\mathbf{send}(*)\big)\big). \tag{$F_3$}$$

($F_3$) says that whenever node *ip* perpetually has a non-empty queue of incoming messages, eventually one of these messages will be handled. Just as for the first task of the process `AODV`, there is no need to specify a fairness property for task (1): our justness property forbids any component from stopping when it can do a ***cast**-action, and our structural operational semantics requires each component within transmission range of a component doing a ***cast** to receive the transmitted message.

To say that a run of AODV is fair amounts to requiring the corresponding complete path to satisfy properties ($F_1$)–($F_3$) for all values of *ip* and *dip*. In order to require fairness for all runs of AODV we augment the specification of AODV with a fairness component. Henceforth, our specification of AODV consists of two parts: (A) the AWN specification of Section 6, which by the operational semantics of AWN generates a labelled transition system $L$, and (B) a *fairness specification*, consisting of a collection of LTL formulas. The latter narrows down the complete paths in $L$ to the ones that satisfy those formulas.[65]

[65] Formally, we require the labelled transition system $L$ and the fairness specification to be consistent with each other. By this we mean that one cannot reach a state in $L$ from where, given a sufficiently uncooperative environment, it is impossible to satisfy the fairness specification—in other words [58], 'the automaton can never "paint itself into a corner."' In [58] this requirement is called *machine closure*, and demands that any finite path in $L$, starting from an initial state, can be extended to a path satisfying the fairness specification. Since we deal with a reactive system here, we need a more complicated consistency requirement, taking into account all possibilities of the environment to allow or block transitions that are not fully controlled by the specified system itself. This requirement can best be explained in terms of a two player game between a *scheduler* and the *environment*.

Define a *run* of $L$ as a path that starts from an initial state. Thus a *finite run* is an alternating sequence of states and transitions, starting from an initial state and ending in a state, such that each transition in the sequence goes from the state before to the state after it. Moreover, a *complete run* is a finite or infinite path starting from an initial state. The game begins with any finite run $\pi$ of $L$, chosen by the environment. In each turn, first the environment selects a set $\mathit{next}(p)$ of transitions starting in the last state $N$ of $\pi$; this set has to include all internal and output transitions starting from $N$, but can also include further transitions starting in $N$. If $\mathit{next}(p)$ is empty, the game ends; otherwise the scheduler selects a transition from this set, which is, together with its ending state, appended to $\pi$, and a new turn starts with the prolonged finite run. The *result* of the game is the finite run in which the game ends, or—if it does not—the infinite run that arises as the limit of all finite runs encountered during the game. So the result of the game always is a complete run. The game is *won* by the scheduler iff the result satisfies the fairness specification. Now $L$ is *consistent* with a fairness specification iff there exists a winning strategy for the scheduler.

Our AODV specification and our fairness properties ($F_1$)–($F_3$) are constructed in such a way that they are consistent.

There are many ways in which we could alter our AWN specification of AODV so as to ensure that ($F_1$)–($F_3$) are satisfied and thus need not be required as an extra part of our specification. For example, Pro. 1 could be modified in a way such that the three different activities (Lines 1–20, Lines 21–32 and Lines 33–39) are prioritised. The process could first initiate all route discovery processes, then handle all queued data packets (for which a valid route is known) and finally handle a fixed number of received messages (less if there are not enough messages in the queue). After the messages have been handled, the modified process would loop back and start initiating route discovery processes again. However, for the purpose of protocol specification we do not want to commit to any particular method of ensuring fairness. Therefore we state fairness as an extra requirement without telling how it should be implemented.

When we later claim that an LTL formula $\phi$ holds for AODV, as specified by (A) and (B) together, this is equivalent to the claim that $\psi \Rightarrow \phi$ holds for AODV as specified by (A) alone, where $\psi$ is the conjunction of all LTL formulas that make up the fairness specification (B).

## 9.2 Route Discovery

An important property that every routing protocol ought to satisfy is that if a route discovery process is initiated in a state where the source is connected to the destination and during this process no (relevant) link breaks, then the source will eventually discover a route to the destination. In case of AODV a route discovery process is initiated when a route request is issued. So for any pair of IP addresses $oip, dip \in \mathbf{IP}$ the following should hold:

$$\mathbf{G}\left(\begin{array}{l}\big(\mathbf{connected}^*(oip,dip)\ \wedge\ \mathbf{broadcast}(\mathtt{rreq}(*,*,dip,*,*,oip,*,oip))\big)\\ \qquad\qquad\qquad \Rightarrow\ \mathbf{F}\big(dip \in \mathtt{vD}^{oip}\ \vee\ \mathbf{disconnect}(*,*)\big)\end{array}\right).$$

Here, the predicate $\mathbf{connected}^*(oip,dip)$ holds in state $N$ iff there exist nodes $ip_0,\dots,ip_n$ such that $ip_0 = oip$, $ip_n = dip$ and $ip_i \in R_N^{ip_{i-1}}$ for $i = 1,\dots,n$. The latter condition describes the fact that $ip_i$ is in range of $ip_{i-1}$.[66] All other predicates follow the description of Page 90: $\mathbf{broadcast}(\mathtt{rreq}(*,*,dip,*,*,oip,*,oip))$ models that node $oip$ issues a request for a route from $oip$ to $dip$; the predicate $dip \in \mathtt{vD}^{oip}$ holds in state $N$ iff $dip \in \mathtt{vD}_N^{oip}$, i.e. $oip$ has found a valid route to $dip$, and $\mathbf{disconnect}(*,*)$ is the action of disconnecting any two nodes. By means of the last disjunct, the property does not require a route to be found once any link in the network breaks.[67]

The following theorem might be a surprise.

**Theorem 9.2** AODV does not satisfy the property route discovery.

We show this by an example (Figure 14). In particular, we show that a route reply may be dropped. This problem has been raised before, back in Oct 2004.[68] We discuss modifications of AODV to solve this problem in Section 10.2. Figure 14 shows a network consisting of 3 nodes in a linear topology. Two nodes ($a$ and $s$) are both searching for a route to destination $d$.[69] First, node $a$ broadcasts a route request, $\text{RREQ}_1$ (Figure 14(b)). As usual all recipients update their routing tables. Since node $s$ still has no information about $d$, it also initiates a route request, $\text{RREQ}_2$. After $a$ has forwarded that request (Figure 14(c)), $d$ initiates a route reply as a consequence of $\text{RREQ}_1$. When node $a$ receives this reply, it updates its own routing table (Figure 14(d)). Finally, node $d$ reacts on the second route request received ($\text{RREQ}_2$) and sends yet another route reply. Node $a$ receives $\text{RREP}_2$, but does *not* forward it. This is

[66] Since the connectivity graph of AWN is always symmetric, this condition suffices to guarantee that both the RREQ message and the RREP message reach their destinations.

[67] Here $\neg\mathbf{disconnect}(*,*)$ is the side condition $\psi$ of (34).

[68] `http://www.ietf.org/mail-archive/web/manet/current/msg05702.html` shows the same shortcoming using a 4-node linear topology.

[69] In [48] we present a version of this example in a non-linear 4-node topology with symmetry between the two nodes that search for a route to $d$.

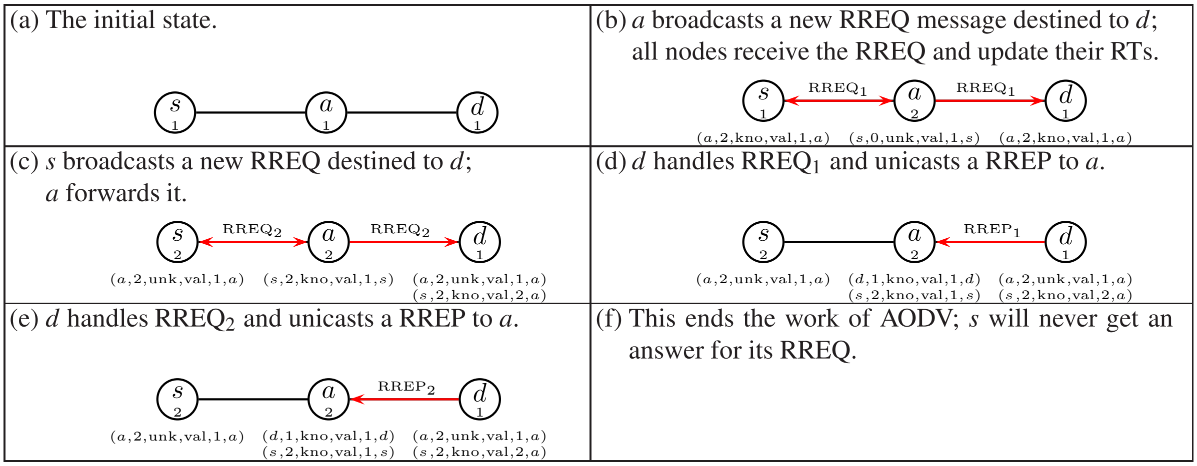



Figure 14: Route discovery fails

because $\mathrm{RREP}_2$ does not contain any fresher information about destination $d$, in comparison with the information in node $a$'s existing routing table entry for $d$. As a result, $\mathrm{RREP}_2$ is dropped at node $a$, and node $s$ never receives a route reply for its route request. Looking at our model (Process 5), the node does not forward a request since Line 1 evaluates to false whereas Line 26 evaluates to true. □

At first glance, it seems that this behaviour can be fixed by a repeated route request. If node $s$ would initiate and broadcast another route request, node $a$ would receive it and generate a route reply immediately. The AODV RFC specifies that a node can broadcast another route request if it has not received a route reply within a pre-defined time. However, a repeated route request does not guarantee the receipt of a route reply. It is easy to construct an example similar to Figure 14 where, instead of a linear topology with 3 nodes, we use a linear topology with $n+2$ nodes, where $n$ is the maximum number of repeated route requests.

But the situation is even worse. Even in a 4-node topology an infinite stream of repeated route requests cannot guarantee route discovery. Figure 15 illustrates this fact.

In the initial state, node $a$ has established a route to $d$ via a standard RREQ-RREP cycle, initiated by $a$. Subsequently, in Part (b), node $b$ searches for a route to $x$ (an arbitrary node that is not connected to any of the nodes we consider). After $d$ forwards the RREQ message destined for $x$, node $a$ creates a valid route to $d$ with an unknown sequence number that equals $d$'s own sequence number.[70] Now $s$ initiates

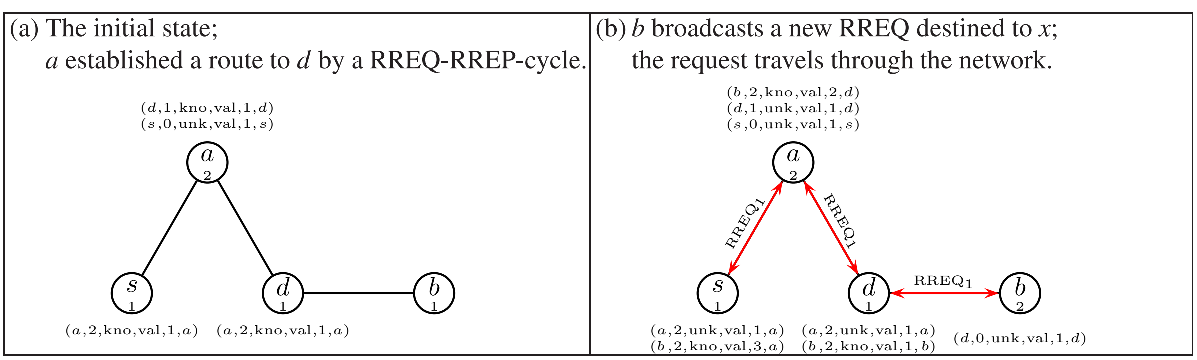



Figure 15: Route discovery also fails with repeated request resending

[70] This examples hinges on our choice of Resolution (2c) of Ambiguity 2. Taking Resolutions (2a) or (2d) would avoid this problem; another solution would be following the suggestion of I. Chakares in Footnote 29 on Page 35. We will propose a more thorough solution, that also tackles the problem of Figure 14, in Section 10.2.

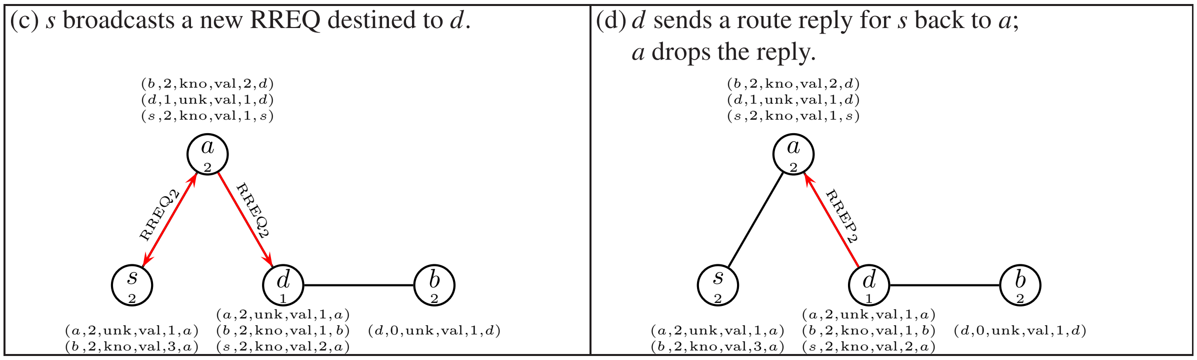



Figure 15 (cont'd): Route discovery also fails with repeated request resending

a route request, searching for a route to $d$. Since node $a$ does not have a known sequence number for $d$ it may not generate an intermediate route reply (Pro. 4, Line 20 evaluate to `false`). Hence it forwards the route request (Part (c)), and node $d$ answers with a RREP message (Part (d)). However, node $a$ will not update its routing table entry for $d$, because it already has an entry with the same sequence number and the same hop count (Line 1 of Pro. 5 evaluates to false whereas Line 26 evaluates to true). As a consequence, $a$ does not forward the route reply to $s$, and $s$ will not create a route to $d$. Repeating the route request by $s$ will not help, as the same events will be repeated.

Both counterexamples show a failure in forwarding a route reply back to the originator of the route discovery process. This travelling back can be seen as the second step of a route discovery process. The first step consists of the route request travelling from the originator to either the destination or to a node that has a valid route to the destination (with known sequence number) in its routing table. The following property states that this step always succeeds: whenever a route request is issued in a state where the source is connected to the destination and subsequently no link break occurs, then some node will eventually send a route reply back towards the source.

$$\mathbf{G}\left(\begin{array}{l}\big(\mathbf{connected}^*(oip,dip) \;\wedge\; \mathbf{broadcast}(\mathtt{rreq}(*,*,dip,*,*,oip,*,oip))\big)\\ \qquad \Rightarrow\; \mathbf{F}\big(\mathbf{unicast}(\mathtt{rrep}(*,dip,*,oip,*),*) \;\vee\; \mathbf{disconnect}(*,*)\big)\end{array}\right).$$

This property does hold for AODV. Namely, Pro. 4 is structured in such a way that upon receipt of a RREQ message, either a matching RREP is sent or the RREQ is forwarded. So if a route reply is never generated, then the route request floods the network and reaches all nodes connected to the originator of the request, which by assumption includes the destination—this would cause a RREP to be sent.

### 9.3 Packet Delivery

The property of *packet delivery* says that if a client injects a packet, it will eventually be delivered to the destination. However, in a WMN it is not guaranteed that this property holds, since nodes can get disconnected, e.g., due to node mobility. A useful formulation has to be weaker. A higher-layer communication protocol should guarantee packet delivery only if an end-to-end route exists long enough. More precisely, such a protocol should guarantee delivery of a packet injected by a client at node $oip$ with destination $dip$, when $oip$ is connected to $dip$ and afterwards no link in the network is disconnected. This means that for all $oip,dip \in \mathbf{IP}$, and any data packet $dp \in \mathtt{DATA}$, the following should hold:

$$\mathbf{G}\left(\begin{array}{l}\big(\mathbf{connected}^*(oip,dip) \;\wedge\; oip:\mathbf{newpkt}(dp,dip)\big)\\ \qquad \Rightarrow\; \mathbf{F}\big(dip:\mathbf{deliver}(dp) \;\vee\; \mathbf{disconnect}(*,*)\big)\end{array}\right). \tag{PD$_1$}$$

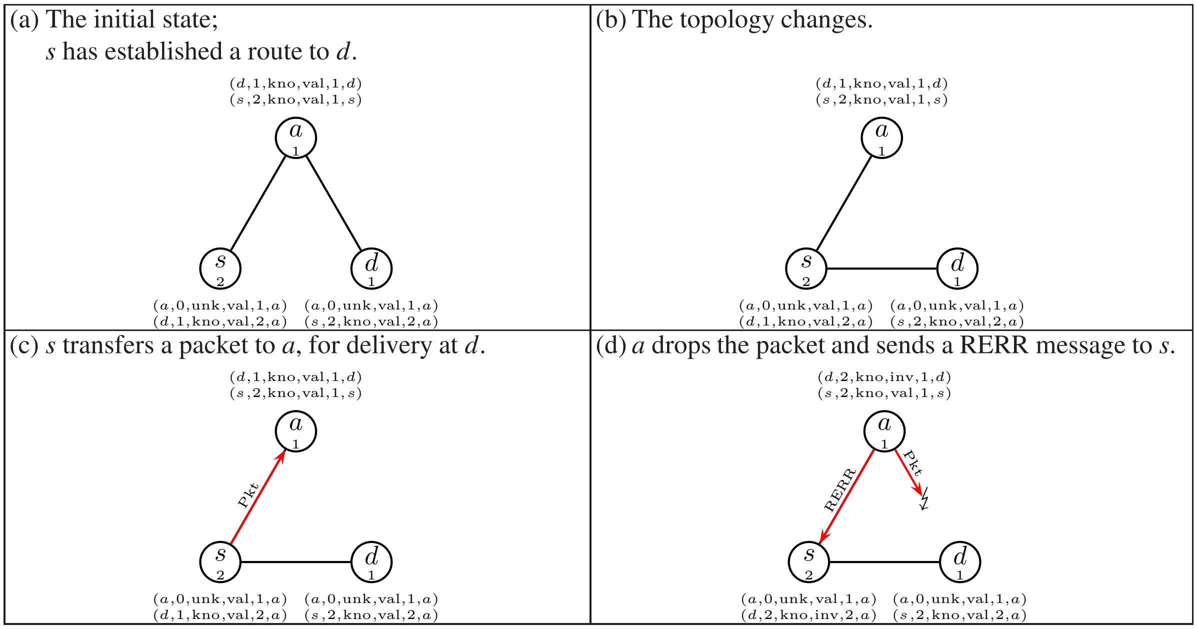



Figure 16: Packet delivery property $\text{PD}_1$ fails

Here $oip\!:\!\textbf{newpkt}(dp,dip)$ models injection of a new data packet $dp$ at $oip$, and $dip\!:\!\textbf{deliver}(dp)$ that the destination receives it. This formulation of packet delivery does not specify any particular route, but merely requires that $dp$ will eventually be delivered. The property does not require a packet to arrive once any link in the network breaks down.

For a routing protocol like AODV, this form of packet delivery is a much too strong requirement. The example of Figure 16 shows why it does not hold.

In the initial state node $s$ has, through a standard RREQ-RREP cycle, established a route to $d$. Afterwards, the link between $a$ and $d$ breaks, and a new link between $s$ and $d$ is established. Subsequently, say in state $S$, the application layer injects a data packet $dp$ destined for $d$ at node $s$. Based on the information in its routing table, $s$ transfers the packet to $a$. However, the packet is dropped by $a$ when $a$ fails to forward the packet to $d$. To be precise, the reachable state $S$ satisfies $\textbf{connected}^*(s,d) \wedge s:\textbf{newpkt}(dp,d)$ but there is a path from $S$ that does not feature any state with $d:\textbf{deliver}(dp)$ or $\textbf{disconnect}(*,*)$.

This failure of ($\text{PD}_1$) is normal behaviour of a routing protocol. A higher layer in the network stack (e.g. the transport or the application layer) may use an acknowledgement and retransmission protocol on top of its use of a routing protocol, and this combination might guarantee ($\text{PD}_1$). For the routing protocol itself, it suffices that a packet will eventually be delivered if the client (higher-layer protocol) injects the same data packet again and again, until the packet has reached the destination. This gives rise to the following weaker form of packet delivery:

$$\mathbf{G}\left(\begin{array}{c}\big(\textbf{connected}^*(oip,dip) \wedge oip:\textbf{newpkt}(dp,dip)\big)\\ \Rightarrow\ \mathbf{F}\big(dip:\textbf{deliver}(dp) \vee \textbf{disconnect}(*,*) \vee \neg\mathbf{F}\big(oip:\textbf{newpkt}(dp,dip)\big)\big)\end{array}\right). \qquad (\text{PD}_2)$$

This is the property ($\text{PD}_1$), but under the side condition $\psi = \mathbf{F}\big(oip:\textbf{newpkt}(dp,dip)\big)$ that is required to hold after the initial injection of the data packet and until the packet is delivered—see (34). This side condition says that one will keep injecting copies of the same data packet, i.e. every state for which $\psi$ holds is followed by one where such a packet is injected. In ($\text{PD}_2$), the clause $oip:\textbf{newpkt}(dp,dip)$ in the precondition is redundant, as it is implied by the side condition $\psi$. Moreover, by the equivalence of

(34), ($PD_2$) can also be formulated as

$$\mathbf{G}\left(\begin{array}{c}\big(\mathbf{connected}^*(oip,dip)\ \wedge\ \mathbf{GF}\big(oip:\mathbf{newpkt}(dp,dip)\big)\big)\\ \Rightarrow\ \mathbf{F}\big(dip:\mathbf{deliver}(dp)\ \vee\ \mathbf{disconnect}(*,*)\big)\end{array}\right).$$

Here, $\mathbf{GF}\big(oip:\mathbf{newpkt}(dp,dip)\big)$ states that the injection of the data packet *dp* at node *oip* will be repeated infinitely often.[71] If during that time no two nodes get disconnected, the packet will eventually be delivered at its destination *dip*.

Continuing the example of Figure 16, in Part (d), node *a* sends a route error message to *s*, as a result of which *s* invalidates its routing table entry for *d*. If now a new data packet destined for *d* is injected at *s*, node *s* initiates a new route discovery process and finds the 1-hop connection. As a result of this, the packet will be delivered at *d*, as required by ($PD_2$).

($PD_2$) appears to be a reasonable packet delivery property for a routing protocol like AODV. Yet, it is still too strong for our purposes. A failure of ($PD_2$) can occur easily in the following scenario: node *oip* has a packet for node *dip*, and initiates a route discovery process by issuing a route request, while setting the request-required flag for the route towards *dip* to `no-req`. The route request reaches *dip*, but the corresponding route reply is lost on the way back to *oip*, due to a link break. From that moment onwards the topology remains stable and a route from *oip* to *dip* exists. We may even assume that it would be found if only *oip* does a second route request. However, such a second route request will never happen because the request-required flag keeps having the value `no-req` in perpetuity.

This failure of ($PD_2$) is a flaw of our model rather than of AODV. A more realistic model would specify that the request-required flag cannot keep the value `no-req` forever. After a timeout, either the flag should revert to `req`, so that a new route request will be made, or the entire queue of data packets destined to *dip* will be dropped, so that a newly injected packet will start a fresh queue, which is initialised with a request-required flag `req`. Such modelling requires timing primitives; however, since we abstract from timing issues, we did not build such a feature into our packet handling routine.

To compensate for this omission, we add a precondition to the packet delivery property, namely that if *oip* perpetually has queued packets for *dip* but no valid route to *dip*, then eventually the request-required flag at *oip* for destination *dip* will be set to `req`:

$$\mathbf{G}\big(\mathbf{G}(dip \in \mathtt{qD}^{oip} - \mathtt{vD}^{oip})\ \Rightarrow\ \mathbf{F}(\sigma^{oip}_{p\text{-}flag}(dip) = \mathtt{req})\big)$$

Adding this precondition to ($PD_2$) yields ($PD_3$), our final *packet delivery* property:

$$\begin{array}{l}\mathbf{G}\big(\mathbf{G}(dip \in \mathtt{qD}^{oip} - \mathtt{vD}^{oip})\ \Rightarrow\ \mathbf{F}\big(\sigma^{oip}_{p\text{-}flag}(dip) = \mathtt{req}\big)\\ \Rightarrow\ \mathbf{G}\left(\begin{array}{c}\mathbf{connected}^*(oip,dip)\\ \Rightarrow\ \mathbf{F}\big(dip:\mathbf{deliver}(dp)\ \vee\ \mathbf{disconnect}(*,*)\ \vee\ \neg\mathbf{F}\big(oip:\mathbf{newpkt}(dp,dip)\big)\big)\end{array}\right).\end{array} \tag{$PD_3$}$$

This property ought to be satisfied by a protocol like AODV. Nevertheless,

**Theorem 9.3** AODV does not satisfy the property packet delivery.

Figure 15 presents an example where an infinite stream of repeated route request does not result in route discovery, let alone in packet delivery.

Figure 17 shows yet another counterexample against packet delivery, this time when the route discovery property is satisfied. Initially, node *d* requests a route to *b* (Figure 17(a)). As a result, *a* creates a routing table entry for *d*, with an empty set of precursors.[72] In Part (b), the reply is sent from node *b*

[71] Due to the existence of finite complete paths, the formula $\mathbf{GF}\big(oip:\mathbf{newpkt}(dp,dip)\big)$ also holds for complete paths whose final state satisfies $oip:\mathbf{newpkt}(dp,dip)$. However, in our specification of AODV such complete paths do not occur.

[72] In fact, this example all lists of precursors are empty.

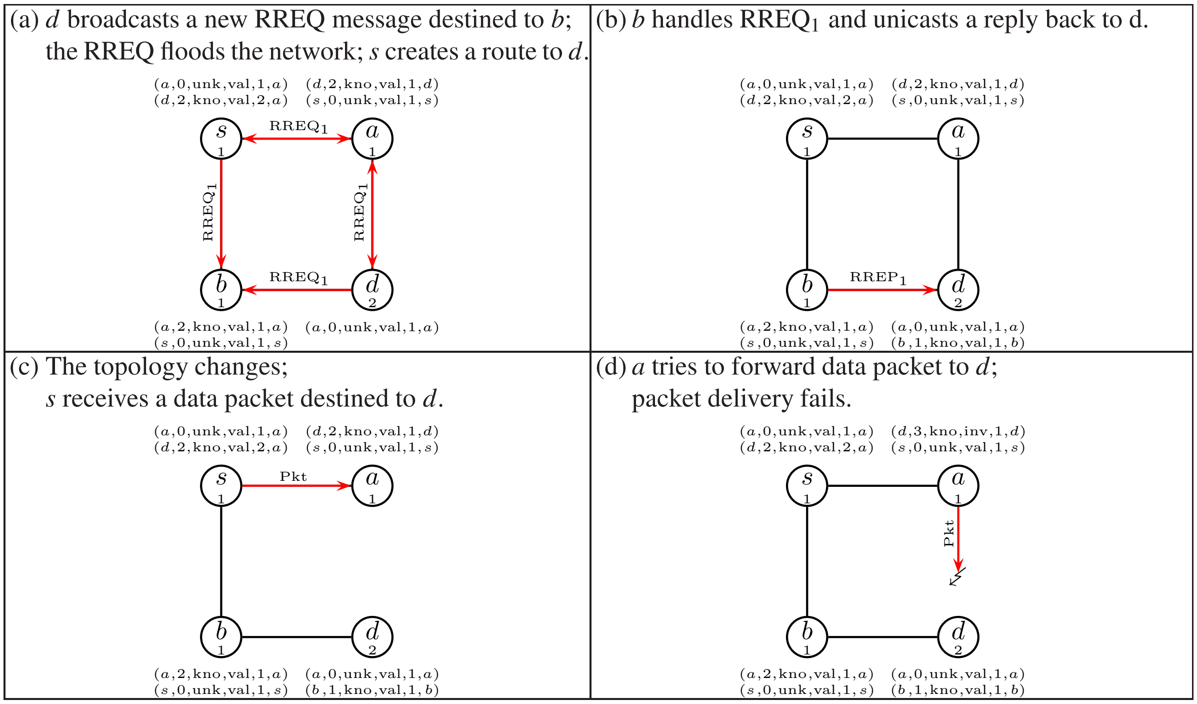



Figure 17: Precursor maintenance limits packet delivery ratio

to node $d$. Afterwards, in Part (c), the link between $a$ and $d$ breaks. From here on the topology remains stable, and **connected**$^*(s,d)$ holds. In Part (c) the application layer injects a packet at $s$ for delivery at $d$. Since $s$ already has a routing table entry for $d$, no new route request needs to be initiated, and the packet can be sent right away. Unfortunately, the packet is dropped when $a$ fails to forward it to $d$. Node $a$ invalidates its entry, but has no precursors for the route to $d$ to send an error message to.[73] As a consequence, $s$ will not learn about the broken link, and all subsequent packets travelling from $s$ to $d$ will be dropped at $a$ (Pro. 3, Lines 15–20).

## 10 Analysing AODV—Problems and Improvements

In this section we point at shortcomings of the AODV protocol and discuss possible solutions. The solutions are again modelled in our process algebra. This makes it easy to ensure that the presented improvements are unambiguous and still satisfy the invariants discussed in the Section 7. In particular we show that all variants of AODV presented in the remainder of this section are loop free and satisfy the route correctness property.

More precisely we propose five changes to the AODV protocol.

In Section 10.1 we show that the route request identifier (RREQ ID) is redundant and can be dropped from the specification of AODV without changing the behaviour of the protocol in any way. This is a small improvement, but reduces the size of message headers.

In Sections 10.2–10.4 we address three deficiencies of AODV that each cause a failure of the packet delivery property discussed in Section 9. The first two deal with failures of the route discovery property, which is a necessary precondition to ensure packet delivery.

[73]The same behaviour occurs when node $a$ detects the link break earlier, for instance by using Hello messages.

In Section 10.2 we discuss a known problem of AODV, namely that a node fails to forward a RREP message that does not contain new information. This leads to a failure of route discovery because the information can be new to the nodes to which the message ought to be forwarded.

In Section 10.3 we discuss failures of route discovery that depend on the convention for routing table updates in response to an AODV control message from a neighbour (cf. Ambiguity 2) and analyse conventions that are not prone to such failures.

In Section 10.4 we show how error messages may fail to reach nodes that need to be informed of a link break. This may cause a failure of packet delivery even when route discovery is guaranteed. This problem can be solved by always broadcasting error messages.

Finally, in Section 10.5, we show that AODV inadvertently establishes sub-optimal routes, i.e., even when there is a shorter route towards a destination, AODV will use (much) longer paths to send packets. This problem can be avoided by modifying the process RREQ for handling message requests.

## 10.1 Skipping the RREQ ID

AODV does not need the route request identifier. This number, in combination with the IP address of the originator, is used to identify every RREQ message in a unique way. However, we have shown that the combination of the originator's IP address and its sequence number is just as suited to uniquely determine the route request to which the message belongs (cf. Proposition 7.35(b)). Hence, the route request identifier field is not required. This can then reduce the size of the RREQ message.

In detail, the following changes have to be made:

- The set RREQID (including the variable rreqid) and the function nrreqid are skipped.
- The variable rreqs is now of type $\mathscr{P}(\mathtt{IP}\times\mathtt{SQN})$.
- The function rreq to generate route requests has now the type
$$\mathtt{rreq}:\mathbb{N}\times\mathtt{IP}\times\mathtt{SQN}\times\mathtt{K}\times\mathtt{IP}\times\mathtt{SQN}\times\mathtt{IP}\to\mathtt{MSG}\,.$$
All the parameters are the same, except that the request identifier is left out.
- The modified basic routine (Pro. 1) is given by Pro. 12.

**Process 12** The basic routine, not using rreqid

```
AODV(ip,sn,rt,rreqs,store) ≝
  ...    /* Lines 1–7 of Pro. 1 */
  + [ msg = rreq(hops,dip,dsn,dsk,oip,osn,sip) ]      /* RREQ */
    /* update the route to sip in rt */
    [[rt := update(rt,(sip,0,unk,val,1,sip,∅))]]      /* 0 is used since no sequence number is known */
    RREQ(hops,dip,dsn,dsk,oip,osn,sip,ip,sn,rt,rreqs,store)
  ...    /* Lines 12–35 of Pro. 1 */
    /* update rreqs by adding (ip,sn) */
    [[rreqs := rreqs ∪ {(ip,sn)}]]
    broadcast(rreq(0,dip,sqn(rt,dip),sqnf(rt,dip),ip,sn,ip)) . AODV(ip,sn,rt,rreqs,store)
```

- In Pro. 4, the occurrences of rreqid in Lines "0" and 36 are dropped; all other occurrences (Lines 1, 3 and 5) are replaced by osn.

The statements and proofs of Sections 7 and 8 are all valid, but need the following modifications.

- Whenever the function rreq is used, the second parameter (rreqid) has to be dropped.
- Propositions 7.34 and 7.35(a) use the variable rreqid; they can be dropped. The statement that a route request is uniquely determined by the pair (oip,osn), the replacement of Proposition 7.34, is already stated and proven in Proposition 7.35(b).

- The statement of Invariant (25) in Proposition 7.36 changes into

$$N \xrightarrow{R:*\textbf{cast}(\texttt{rreq}(*,*,*,*,oip_c,osn_c,ip_c))}_{ip} N' \Rightarrow (oip_c, osn_c) \in \xi_N^{ip_c}(\texttt{rreqs}) \tag{35}$$

  and likewise for Invariant (26). In the proof, "content $\xi(*,\texttt{rreqid},*,*,*,\texttt{ip},*,\texttt{ip})$" changes into "content $\xi(*,*,*,*,\texttt{ip},\texttt{osn},\texttt{ip})$". All other occurrences of "rreqid" change into "osn", and "$rreqid_c$" into "$osn_c$".

- In the proof of Proposition 8.1 "rreqid" changes into "osn".

### 10.2 Forwarding the Route Reply

In AODV's route discovery process, a RREP message from the destination node is unicast back along a route towards the originator of the RREQ message. Every intermediate node on the selected route will process the RREP message and, in most cases, forward it towards the originator node. However, there is a possibility that the RREP message is discarded at an intermediate node, which results in the originator node not receiving a reply. The discarding of the RREP message is due to the RFC specification of AODV [79] stating that an intermediate node only forwards the RREP message if it is not the originator node *and* it has created or updated a routing table entry to the destination node described in the RREP message:

> "If the current node is not the node indicated by the Originator IP Address in the RREP message AND a forward route has been created or updated as described above, the node consults its route table entry for the originating node to determine the next hop for the RREP packet, and then forwards the RREP towards the originator using the information in that route table entry." [79, Sect. 6.7]

The latter requirement means that if a valid routing table entry to the destination node already exists, and is not updated when processing the RREP message, then the intermediate node will not forward the message. In Section 9 we have illustrated this problem with two examples (Figures 14 and 15), also showing that this leads to a failure of route discovery.

A solution to this problem is to require intermediate nodes to forward *all* RREP messages that they receive. In the example presented in Figure 14, the intermediate node $a$ will forward $\text{RREP}_2$, after $\text{RREP}_2$ was received in Part (e). As a result, node $s$ will establish a route to $d$. Likewise, in Figure 15(d), node $a$ will forward $\text{RREP}_2$ and again $s$ will establish a route to $d$.

To implement this behaviour one can simply drop the Lines 1 and 26–27 of Pro. 5 (RREP handling), keeping Lines 2–25 only.

This solution guarantees the forwarding of the RREP message. However, it might be the case that outdated information is forwarded and, as a consequence, non-optimal information is stored in the routing tables. This is shown by the example presented in Figure 18.

The example assumes a linear topology with 5 nodes. In Part (b), node $s$ receives a data packet destined to node $d$; it initiates a route discovery process. The request is forwarded by nodes $a$, $b$ and $c$ until it reaches the destination $d$. Node $d$ then generates a route reply and unicasts the message to $c$ (Part (c)). After the RREP message is (successfully) sent, Figure 18(d), a link between $a$ and $d$ is established and node $d$ broadcasts a new RREQ message, destined to $a$. This message is received by nodes $a$ and $c$. In principle node $c$ would later forward the request; however, this forwarding and the subsequent actions do not add anything to the example and therefore we drop this bit.

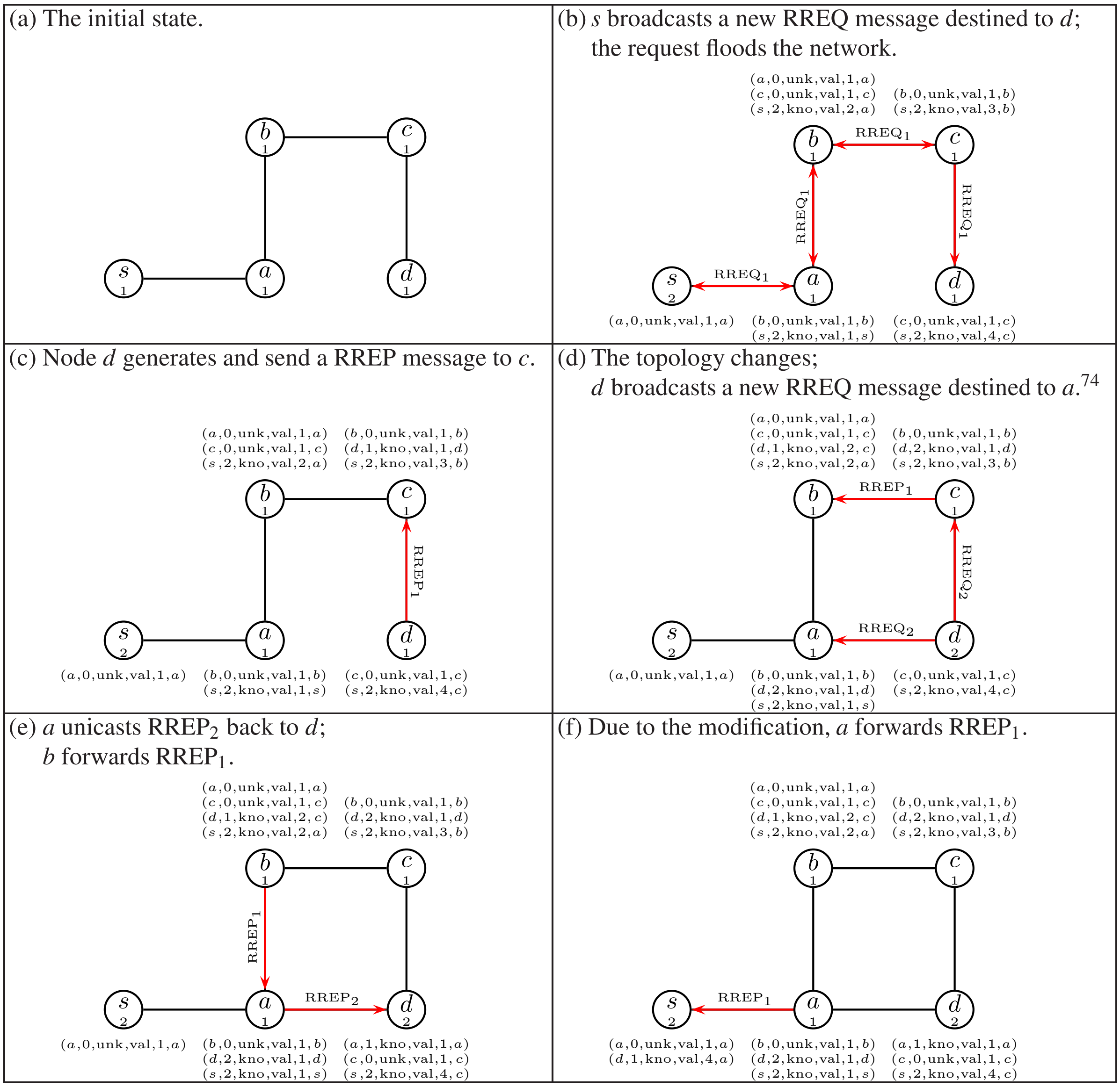



Figure 18: Always forwarding RREP messages

After $a$ has initiated a route reply as a consequence of $\text{RREQ}_2$, which is sent back to $d$, it receives $\text{RREP}_1$ from $b$—the reply generated by node $d$ and destined to $s$. In the original version of AODV, as presented in Sections 5 and 6, the reply would be dropped, since $a$ does not update its routing table. In the modified version, $a$ creates a message by $\texttt{rrep}(3,d,1,s,a)$, which is sent to node $s$ ($\texttt{nhop}(\texttt{rt},\texttt{oip})$). Note, that $a$ does not update its own routing table. As a consequence of this message, node $s$ updates its routing table and creates an entry to $d$ with sequence number 1 and hop count 4 (Part (f)).

Although this information is not incorrect, it is outdated. Any data packet sent from $s$ to $d$ would be forwarded to $a$ and then immediately to the destination, thanks to node $a$ having fresher information (in its routing table the sequence number belonging to $d$ is 2). As a general rule, it makes sense to use the newest available information on the route to the destination node: if an intermediate node's routing table contains an entry for the destination node that is valid and fresher than that in the received RREP

[74]The message $\text{RREQ}_2$ is also sent to node $c$. Since it does not change the example, we suppress this message.

message, the intermediate node ought to update the contents of the RREP message to reflect this. To achieve this one can replace Line 13 of Pro. 5 by

$$\textbf{unicast}(\texttt{nhop}(\texttt{rt},\texttt{oip}),\texttt{rrep}(\texttt{dhops}(\texttt{rt},\texttt{dip}),\texttt{dip},\texttt{sqn}(\texttt{rt},\texttt{dip}),\texttt{oip},\texttt{ip}))\ .$$

In case the received reply contained fresher information, the routing table was already updated. The full modified RREP handling is shown in Pro. 13. Note that Lines 8 and 21 (Lines 9 and 22 in the original process) are also changed. The reason for this change is that information should only be forwarded when the intermediate node has a *valid* route to the destination of the route discovery process. Assume for example the situation given in Figure 18(d). As before, node $a$ sends $\text{RREP}_2$; but just before $\text{RREP}_1$ is handled by $a$, the unreliable link between $a$ and $d$ breaks and $a$ invalidates its routing table for $d$, i.e., it changes into $(d,3,\texttt{kno},\texttt{inv},1,d)$. Under such circumstances a route reply should not be forward, since any data packet reaching the intermediate node (in the example $a$) would be dropped.

All invariants presented in Sections 7 and 8 remain valid. However, a few proofs need adaptation.

- In Proposition 7.11(b), the case dealing with Pro. 5 now reads as follows:

  **Pro. 13, Line 12:**[75] The message has the form $\texttt{rrep}(\xi(\texttt{dhops}(\texttt{rt},\texttt{dip})),*,*,*,*)$. By Proposition 7.10 $\xi(\texttt{dhops}(\texttt{rt},\texttt{dip}))>0$, so the antecedent does not hold.

**Process 13** RREP handling (Forwarding the Route Reply)

```
RREP(hops,dip,dsn,oip,sip,ip,sn,rt,rreqs,store) =def
1.  [[rt := update(rt,(dip,dsn,kno,val,hops+1,sip,∅))]]
2.  (
3.    [ oip = ip ]      /* this node is the originator of the corresponding RREQ */
4.      /* a packet may now be sent; this is done in the process AODV */
5.      AODV(ip,sn,rt,rreqs,store)
6.    + [ oip ≠ ip ]      /* this node is not the originator; forward RREP */
7.      (
8.        [ oip ∈ vD(rt) ∧ dip ∈ vD(rt) ]      /* valid route to oip and to dip */
9.          /* add next hop towards oip as precursor and forward the route reply */
10.         [[rt := addpreRT(rt,dip,{nhop(rt,oip)})]]
11.         [[rt := addpreRT(rt,nhop(rt,dip),{nhop(rt,oip)})]]
12.         unicast(nhop(rt,oip),rrep(dhops(rt,dip),dip,sqn(rt,dip),oip,ip)) .
13.           AODV(ip,sn,rt,rreqs,store)
14.         ▶ /* If the transmission is unsuccessful, a RERR message is generated */
15.           [[dests := {(rip,inc(sqn(rt,rip)))|rip ∈ vD(rt) ∧ nhop(rt,rip) = nhop(rt,oip)}]]
16.           [[rt := invalidate(rt,dests)]]
17.           [[store := setRRF(store,dests)]]
18.           [[pre := ⋃{precs(rt,rip)|(rip,*) ∈ dests}]]
19.           [[dests := {(rip,rsn)|(rip,rsn) ∈ dests ∧ precs(rt,rip) ≠ ∅}]]
20.           groupcast(pre,rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
21.       + [ oip ∉ vD(rt) ∨ dip ∉ vD(rt) ]      /* no valid route to oip or to dip */
22.         AODV(ip,sn,rt,rreqs,store)
23.     )
24. )
```

- In Proposition 7.13(b), the case dealing with Pro. 5 now reads as follows:

  **Pro. 13, Line 12:** Here, $dsn_c := \xi(\texttt{sqn}(\texttt{rt},\texttt{dip}))$. The last routing table update happened in Line 1. The update uses $\xi(\texttt{dsn})$, which stems, through Line 12 of Pro. 1, from an incoming RREP message (Pro. 1, Line 1). For this incoming RREP message the invariant holds, i.e. $\xi(\texttt{dsn})\geq 1$. By Proposition 7.6, the sequence number is increased monotonically, and hence $dsn_c := \xi(\texttt{sqn}(\texttt{rt},\texttt{dip}))\geq\xi(\texttt{dsn})\geq 1$.

[75] This line corresponds to Pro. 5, Line 13 in the original specification.

- The case of Proposition 7.14(b) dealing with RREP handling now becomes

  **Pro. 13, Line 12:** The message has the form $\xi(\mathtt{rrep}(\mathtt{dhops}(\mathtt{rt},\mathtt{dip}),\mathtt{dip},\mathtt{sqn}(\mathtt{rt},\mathtt{dip}),\mathtt{oip},\mathtt{ip}))$. Hence $hops_c := \xi(\mathtt{dhops}(\mathtt{rt},\mathtt{dip}))$, $dip_c := \xi(\mathtt{dip})$, $dsn_c := \xi(\mathtt{sqn}(\mathtt{rt},\mathtt{dip}))$, $ip_c := \xi(\mathtt{ip}) = ip$ and $\xi_N^{ip_c} = \xi$. Line 1 guarantees that $dip_c = \xi(\mathtt{dip}) \in \mathtt{kD}_N^{ip_c}$. Since the sequence number and the hop count are taken from the routing table, we get immediately
  $$\begin{aligned}\mathtt{sqn}_N^{ip_c}(dip_c) &= \mathtt{sqn}(\xi(\mathtt{rt}),\xi(\mathtt{dip})) &&= dsn_c\\ \mathtt{dhops}_N^{ip_c}(dip_c) &= \mathtt{dhops}(\xi(\mathtt{rt}),\xi(\mathtt{dip})) &&= hops_c\,.\end{aligned}$$
  With exception of its precursors, which are irrelevant here, the routing table does not change between Lines 8 and 12. So, by Line 8, $dip_c = \xi(\mathtt{dip}) \in \mathtt{vD}(\xi(\mathtt{rt}))$ and therefore
  $$\mathtt{flag}_N^{ip_c}(dip_c) \;=\; \mathtt{flag}(\xi(\mathtt{rt}),\xi(\mathtt{dip})) \;=\; \mathtt{val}\,.$$

- The 7th case of the proof of Proposition 7.19 is changed to

  **Pro. 13, Line 11:** By Line 8 $\xi(\mathtt{oip}) \in \mathtt{vD}(\xi(\mathtt{rt}))$ and $\xi(\mathtt{dip}) \in \mathtt{vD}(\xi(\mathtt{rt}))$.

- In Proposition 7.18, the following case needs to be added:

  **Pro. 13, Line 12:** By Line 8 $\xi(\mathtt{dip}) \in \mathtt{vD}(\xi(\mathtt{rt}))$.

- In the proof of Proposition 7.21, the cases for Pro. 5, Lines 1 and 26 are skipped.

- In Proposition 7.22, the last case changes into

  **Pro. 13, Line 11:** By Line 8, $\xi(\mathtt{dip}) \in \mathtt{vD}(\xi(\mathtt{rt})) \subseteq \mathtt{kD}(\xi(\mathtt{rt}))$, so a routing table entry for $\xi(\mathtt{dip})$ exists. Using Proposition 7.38, this implies that $\mathtt{nhop}(\xi(\mathtt{dip}),\xi(\mathtt{rt})) \in \mathtt{kD}(\xi(\mathtt{rt}))$.

- In Theorem 7.32(c), the case dealing with Pro. 5 becomes

  **Pro. 13, Line 12:** The proof is the same as for Pro. 4, Line 25.

- The last case of Proposition 7.37(a) is changed to

  **Pro. 13, Line 12:** A route reply with $dip_c := \xi_N^{ip}(\mathtt{dip})$ and $dsn_c := \xi_N^{ip}(\mathtt{sqn}(\mathtt{rt},\mathtt{dip})) = \mathtt{sqn}_N^{ip}(dip_c)$ is initiated. By Invariant (29) $dsn_c = \mathtt{sqn}_N^{ip}(dip_c) \leq \xi_N^{dip_c}(\mathtt{sn})$.

Surely, always forwarding (unicasting) replies increases the number of messages in the network. However, as illustrated by the examples of Figures 14 and 15, the policy to always forward the route reply significantly increases the probability of a route discovery process being successful. As a consequence, the probability of the originator re-issuing a route request to establish a route is much smaller. Such a re-sending would yield another broadcast cycle, which, with respect to network load, is much more expensive than the extra unicast of RREP messages.

## 10.3 Updating with the Unknown Sequence Number

In this section we evaluate the resolutions of Ambiguity 2 of Section 8. We have already discarded Resolution (2b), as it leads to routing loops. The alternatives, (2a), (2c) and (2d), have been shown to satisfy the loop freedom and route correctness property.

A disadvantage of Resolution (2a) is that it misses opportunities to improve the routes between two neighbouring nodes. It can lead to situations in which a node $s$ knows that node $d$ can be reached using $a$ as next hop, but at the same time does not know that there is a valid 1-hop route to $a$ itself: assume the topology given in Figure 19. The link between the nodes $s$ and $a$ is unreliable—messages sent via this link might get lost and the neighbouring nodes might detect that this link is broken. Let us further assume that $s$ has established a route to $a$; the corre-

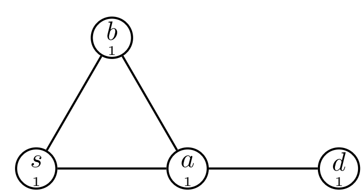



Figure 19: 4-node topology missing route optimisation opportunity

sponding routing table entry might be $(a,1,\texttt{kno},\texttt{val},2,b)$ (the RREQ message from $s$ to $a$ got lost) or $(a,2,\texttt{kno},\texttt{inv},1,a)$ (a 1-hop connection was established, but the link broke down). Next, node $d$ searches for a route to $s$. The generated RREQ message is received by $a$ and forwarded to $s$. Node $s$ creates a routing table entry to $d$ ($(d,2,\texttt{kno},\texttt{val},2,a)$) and tries to update its entry to $a$. However, by use of Resolution (2a), neither of the above mentioned entries would be changed.

This strongly gives the impression that information is not used in an optimal way.

Resolutions (2c) and (2d) do not suffer from this drawback. However, they have their own problems. Resolution (2d) gives rise to non-optimal routes, as illustrated in Figure 20. In the initial state (Figure 20(a)), a route between $a$ and $d$ is established through a standard RREQ-RREP cycle. Then, in Part (b)), the connection between $a$ and $d$ breaks down. $a$ and $d$ detect the link break and invalidate their routing table entries for each other, thereby increasing the destination sequence numbers. Subsequently, the connection between $a$ and $d$ comes back up, and node $b$ (connected to $d$) initiates a route request for a node $x$, which is not to be found in the vicinity (Figure 20(c)).

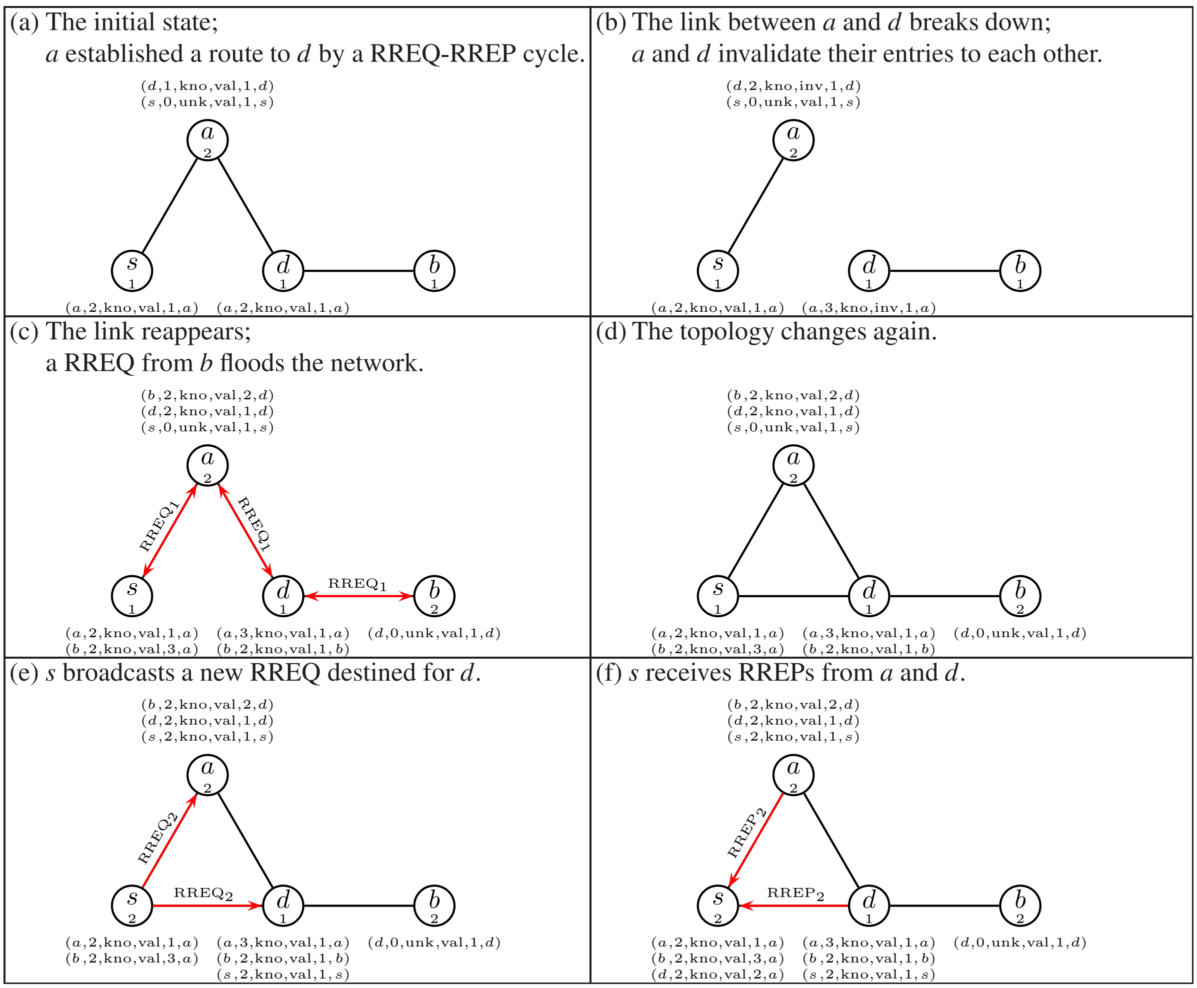



Figure 20: Resolution (2d) gives rise to non–optimal routes

As a consequence, when $a$ receives the forwarded RREQ message from $d$, it validates its routing table entry for $d$, the destination sequence number being higher than $d$'s own sequence number. In Parts (d) and (e), a direct link between $s$ and $d$ appears, and $s$ searches for a route to $d$. Its RREQ message is answered both by $a$, which knows a route to $d$, and by $d$ itself (Figure 20(f)). Regardless which of the two RREP messages arrives first, $s$ establishes a route to $d$ of length 2 via $a$, since the RREP message from $a$ carries a higher destination sequence number for $d$ than the RREP message from $d$ itself. This anomaly pleads against the use of Resolution (2d).

Although Resolution (2c) seems to be the intention of the RFC (cf. Ambiguity 2), it gives rise to route discovery failures as illustrated in Figure 15. This situation is so common, and the lack of route discovery is such a severe problem, that for the original AODV Resolution (2c) can be judged worse than (2a) and (2d), and should not be used. The problem is a combination of the use of Resolution (2c) and AODV's failure to forward route replies. Once the latter problem is satisfactory addressed, for instance by following our proposal in Section 10.2, the problem of Figure 15 is solved, and Resolution (2c) is back in the race. Nevertheless, the following example shows a remaining problem, that pertains to both Resolutions (2c) and (2d). (The sequence-number-status flags in the routing table entries of Figure 21 conform to Resolution (2c)—however, they play no role in this example.)

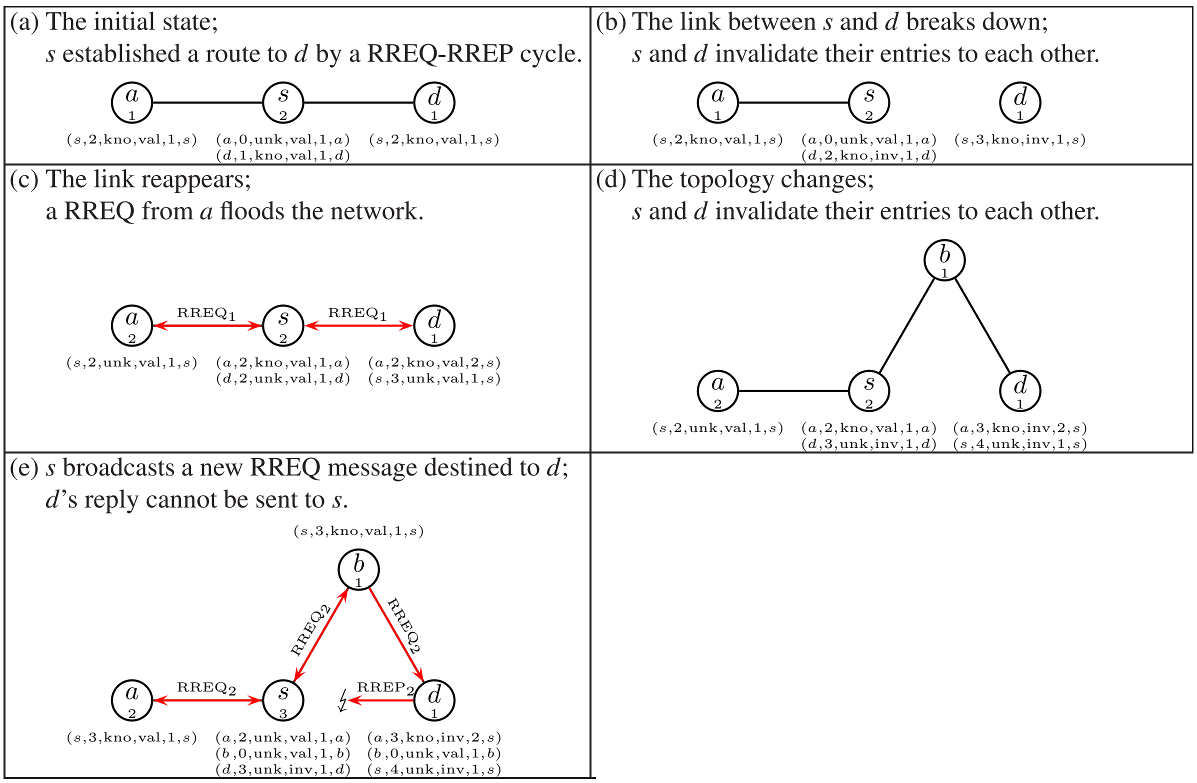



Figure 21: Failure in generating a route reply (Resolutions (2c) and (2d))

In the initial state (Figure 21(a)), a route between $s$ and $d$ is established through a standard RREQ-RREP cycle. Then, in Part (b), the connection between $s$ and $d$ breaks down. $d$ detects the link break and invalidates its routing table entry for $s$, thereby increasing the destination sequence number. In Figure 21(c), the connection between $s$ and $d$ comes back up, and $a$ initiates a route request for a node $x$ (which is not to be found in the vicinity). As a consequence, when $d$ receives the forwarded route request from $s$, it validates its routing table entry for $s$, the destination sequence number being higher than $s$'s own sequence number. In Figure 21(d), the connection breaks down and the entry becomes again invalid. The destination sequence number of the entry is now 2 higher than $s$'s own sequence number. Moreover, a node $b$ appears in the network, and gets connected to $s$ and $d$. From this point onwards the topology remains stable and the predicate **connected**$^*(s,d)$ (cf. Page 97) holds. In Part (e), $s$ searches for a route to $d$. Even though this increases $s$'s own sequence number, it is still smaller than the destination sequence number for $s$ at $d$. When the route request reaches $d$ (via $b$), $d$ tries to update its own routing table entry for $s$. However, $d$ already has an invalid entry for $s$ with a higher sequence number. As a result, no update occurs and the route from $d$ to $s$ remains invalid. Therefore $s$ does not get a reply.

Since each of the Resolutions (2a-d) turned out to have serious disadvantages, we now propose an alternative—Resolution (2e)—that does not share these disadvantages. The intuition is that when a node changes an invalid route into a valid one, while keeping the sequence number from the routing table (as done in Resolutions (2c-d)), it needs to undo the increment of the sequence number performed upon invalidation of the route. This involves decrementing destination sequence numbers, a practice that goes strongly against the spirit of the RFC. Nevertheless, since the net sequence number stays the same, we are able to show that all our invariants are maintained, which constitutes a formal proof of loop freedom and route correctness. So in this special case decrementing destination sequence numbers turns out to be harmless.

Resolution (2e) is a variant of Resolution (2d), defined through a modification in the definition of update. The 5th clause ($nrt \cup \{nr'\}$ if $\pi_1(r) \in \mathtt{kD}(rt) \wedge \pi_3(r) = \mathtt{unk}$) is split into two parts (depending on the validity of the route).

$$\mathtt{update}(rt,r) := \begin{cases} rt \cup \{r\} & \text{if } \pi_1(r) \not\in \mathtt{kD}(rt) \\ nrt \cup \{nr\} & \text{if } \pi_1(r) \in \mathtt{kD}(rt) \wedge \mathtt{sqn}(rt,\pi_1(r)) < \pi_2(r) \\ nrt \cup \{nr\} & \text{if } \pi_1(r) \in \mathtt{kD}(rt) \wedge \mathtt{sqn}(rt,\pi_1(r)) = \pi_2(r) \wedge \mathtt{dhops}(rt,\pi_1(r)) > \pi_5(r) \\ & \qquad\qquad\qquad\quad \wedge\, \pi_3(r) = \mathtt{kno} \\ nrt \cup \{nr\} & \text{if } \pi_1(r) \in \mathtt{kD}(rt) \wedge \mathtt{sqn}(rt,\pi_1(r)) = \pi_2(r) \wedge \mathtt{flag}(rt,\pi_1(r)) = \mathtt{inv} \\ & \qquad\qquad\qquad\quad \wedge\, \pi_3(r) = \mathtt{kno} \\ nrt \cup \{nr''\} & \text{if } \pi_1(r) \in \mathtt{kD}(rt) \wedge \pi_3(r) = \mathtt{unk} \wedge \mathtt{flag}(rt,\pi_1(r)) = \mathtt{val} \\ nrt \cup \{nr'''\} & \text{if } \pi_1(r) \in \mathtt{kD}(rt) \wedge \pi_3(r) = \mathtt{unk} \wedge \mathtt{flag}(rt,\pi_1(r)) = \mathtt{inv} \\ nrt \cup \{ns\} & \text{otherwise}\ , \end{cases}$$

where (in the terminology of Section 5.5.2) $nr'' := (\mathtt{dip}_{nr}\,,\pi_2(s)\,,\pi_3(s)\,,\mathtt{flag}_{nr}\,,\mathtt{hops}_{nr}\,,\mathtt{nhip}_{nr}\,,\mathtt{pre}_{nr})$ and $nr''' := (\mathtt{dip}_{nr}\,,\pi_2(s) \dot{-} 1\,,\pi_3(s)\,,\mathtt{flag}_{nr}\,,\mathtt{hops}_{nr}\,,\mathtt{nhip}_{nr}\,,\mathtt{pre}_{nr})$. We illustrate the behaviour of this modification using a similar example as in the Section about Ambiguity 2: as a consequence of the incoming RREQ message $\mathtt{rreq}(1\,,rreqid\,,x\,,7\,,\mathtt{kno}\,,s\,,2\,,a)$ the routing table entry $(a,2,\mathtt{kno},\mathtt{val},2,b,\emptyset)$ of node $d$ is now updated to $(a,2,\mathtt{kno},\mathtt{val},1,a,\emptyset)$—the same behaviour as in Resolution (2d)—but the entry $(a,2,\mathtt{kno},\mathtt{inv},2,b,\emptyset)$ is updated to $(a,1,\mathtt{kno},\mathtt{val},1,a,\emptyset)$.

Any of the interpretations and variants of AODV using Resolution (2c)—our default resolution of Ambiguity 2—that have been shown loop free in this paper, remain loop free when using Resolution (2e) instead—the invariants, proofs and proof modifications of Section 7–10.2 remain valid, with the following modifications:

- Proposition 7.6 is reformulated as:

  In each node's routing table, the *net* sequence number for a given destination increases monotonically. That is, for $ip, dip \in \mathbf{IP}$, if $N \xrightarrow{\ell} N'$ then $\mathtt{nsqn}^{ip}_{N}(dip) \leq \mathtt{nsqn}^{ip}_{N'}(dip)$.

  For the proof, note that the modified update cannot decrease a net sequence number, so again the only function that can decrease a net sequence number is invalidate. When invalidating routing table entries using the function invalidate(rt,dests), sequence numbers are copied from dests to the corresponding entry in rt. It is sufficient to show that for all $(rip,rsn) \in \xi^{ip}_{N}(\mathtt{dests})$ $\mathtt{sqn}^{ip}_{N}(rip) \leq rsn \dot{-} 1$, as all other sequence numbers in routing table entries remain unchanged.

  **Pro. 1, Line 28; Pro. 3, Line 10; Pro. 4, Lines 13, 29; Pro. 5, Line 17:**

  The set dests is constructed immediately before the invalidation procedure. For $(rip,rsn) \in \xi^{ip}_{N}(\mathtt{dests})$, we have $\mathtt{sqn}^{ip}_{N}(rip) = \mathtt{inc}(\mathtt{sqn}^{ip}_{N}(rip)) \dot{-} 1 = rsn \dot{-} 1$.

**Pro. 6, Line 3:** When constructing `dests` in Line 2, the condition $\xi_{N_2}^{ip}(\texttt{sqn}(\texttt{rt},\texttt{rip})) < \xi_{N_2}^{ip}(\texttt{rsn})$ is taken into account, which immediately yields the claim for $(rip,rsn) \in \xi_N^{ip}(\texttt{dests})$.

- We also need a weakened version of the old Proposition 7.6:

  An application of `invalidate` never decreases a sequence number in a routing table. (36)

  The proof is contained in proof of the old Proposition 7.6 and does not rely on the (modified) function `update`.

- The reference to Proposition 7.6 in the proof of Proposition 7.12 is replace by a reference to (36).
- In the beginning of the proof of Proposition 7.26(a) the inequality

$$\texttt{nsqn}(rt,dip) \leq \texttt{sqn}(rt,dip) = dsn_{rt} = \texttt{nsqn}(rt',dip)$$

  turns into an equality $\texttt{nsqn}(rt,dip) = dsn_{rt} = \texttt{nsqn}(rt',dip)$,
  and follows, by the new definition of `update`, without the step that references (16).

- To adapt the proof of Theorem 7.30 we need some new auxiliary invariants. The first states that the sequence number of an invalid routing table entry can never be 1.

$$dip \in \texttt{iD}_N^{ip} \;\Rightarrow\; \texttt{sqn}_N^{ip}(dip) \neq 1\ . \tag{37}$$

  *Proof.* Invalid routing table entries only arise by applications of `invalidate` on valid routing table entries; furthermore, only calls of `invalidate` can change the sequence number of an invalid routing table entry while keeping the route invalid. Hence it suffices to check all calls of `invalidate`. An application $\xi_N^{ip}(\texttt{invalidate}(\texttt{rt},\texttt{dests}))$ invalidates the routing table entry to *rip* and changes its sequence number into *rsn* for any pair $(rip,rsn) \in \xi_N^{ip}(\texttt{dests})$.

  **Pro. 1, Line 28; Pro. 3, Line 10; Pro. 4, Lines 13, 29; Pro. 5, Line 17:**
  By construction of `dests` (immediately before the invalidation call) $(rip,rsn) \in \xi_N^{ip}(\texttt{dests})$ implies $rsn = \texttt{inc}(\texttt{sqn}(\xi_N^{ip}(\texttt{rt}),rip))$. By definition of `inc` we have $\texttt{inc}(n) \neq 1$, for $n \in \mathbb{N}$.

  **Pro. 6, Line 3:** Let $(rip,rsn) \in \xi_{N_3}^{ip}(dests)$; then $(rip,rsn) \in \xi_{N_2}^{ip}(dests)$, and $\xi_{N_2}^{ip}(dests)$ stems from a received RERR message that must have been sent beforehand, say by a node $ip_c$ in state $N^\dagger$. By Proposition 7.15, $rip \in \texttt{iD}_{N^\dagger}^{ip_c}$ and $rsn = \texttt{sqn}_{N^\dagger}^{ip}(rip)$. So by induction on reachability, $rip \neq 1$. □

  As an immediate corollary of this invariant we obtain that

$$dip \in \texttt{iD}_N^{ip} \;\Rightarrow\; \texttt{inc}(\texttt{nsqn}_N^{ip}(dip)) = \texttt{sqn}_N^{ip}(dip)\ . \tag{38}$$

- Define the *upgraded sequence number* of destination *dip* at node *ip* by

$$\texttt{usqn}_N^{ip}(dip) = \begin{cases} \texttt{inc}(\texttt{sqn}_N^{ip}(dip)) & \text{if } dip \in \texttt{vD}_N^{ip} \wedge \texttt{dhops}_N^{ip}(dip) = 1 \\ \texttt{sqn}_N^{ip}(dip) & \text{otherwise.} \end{cases}$$

  By this definition we immediately get the following inequation.

$$\texttt{usqn}_N^{dip}(dip) \geq \texttt{sqn}_N^{dip}(dip) \geq \texttt{usqn}_N^{dip}(dip) \mathbin{\dot{-}} 1 \tag{39}$$

  After Theorem 7.27 has been established, we obtain the following invariant, saying that in each routing table, the upgraded sequence number for any given destination increases monotonically: for $ip, dip \in \mathbf{IP}$ and a reachable network expression $N$,

$$N \xrightarrow{\ell} N' \;\Rightarrow\; \texttt{usqn}_N^{ip}(dip) \leq \texttt{usqn}_{N'}^{ip}(dip)\ . \tag{40}$$

*Proof.* We distinguish four cases.

(i) Neither $dip \in \mathtt{vD}^{ip}_{N} \wedge \mathtt{dhops}^{ip}_{N}(dip) = 1$ nor $dip \in \mathtt{vD}^{ip}_{N'} \wedge \mathtt{dhops}^{ip}_{N'}(dip) = 1$ holds. Then $\mathtt{usqn}^{ip}_{N}(dip) = \mathtt{sqn}^{ip}_{N}(dip) \leq \mathtt{sqn}^{ip}_{N'}(dip) = \mathtt{usqn}^{ip}_{N'}(dip)$, where the inequality follows just as in the proof of Proposition 7.6, taking into account that the modified 5th clause of `update` cannot apply, as, using Proposition 7.21, it would result in a routing table entry with $dip \in \mathtt{vD}^{ip}_{N'} \wedge \mathtt{dhops}^{ip}_{N'}(dip) = 1$.

(ii) Both $dip \in \mathtt{vD}^{ip}_{N} \wedge \mathtt{dhops}^{ip}_{N}(dip) = 1$ and $dip \in \mathtt{vD}^{ip}_{N'} \wedge \mathtt{dhops}^{ip}_{N'}(dip) = 1$ hold. Then

$$\mathtt{usqn}^{ip}_{N}(dip) = \mathtt{inc}(\mathtt{sqn}^{ip}_{N}(dip)) = \mathtt{inc}(\mathtt{nsqn}^{ip}_{N}(dip)) \leq \\ \mathtt{inc}(\mathtt{nsqn}^{ip}_{N'}(dip)) = \mathtt{inc}(\mathtt{sqn}^{ip}_{N'}(dip)) = \mathtt{usqn}^{ip}_{N'}(dip)\ ,$$

where the inequality follows by Theorem 7.27.

(iii) $dip \in \mathtt{vD}^{ip}_{N'} \wedge \mathtt{dhops}^{ip}_{N'}(dip) = 1$ holds, but $dip \in \mathtt{vD}^{ip}_{N} \wedge \mathtt{dhops}^{ip}_{N}(dip) = 1$ does not. Then

$$\mathtt{usqn}^{ip}_{N}(dip) = \mathtt{sqn}^{ip}_{N}(dip) \leq \mathtt{inc}(\mathtt{nsqn}^{ip}_{N}(dip)) \leq \\ \mathtt{inc}(\mathtt{nsqn}^{ip}_{N'}(dip)) = \mathtt{inc}(\mathtt{sqn}^{ip}_{N'}(dip)) = \mathtt{usqn}^{ip}_{N'}(dip)\ ,$$

where the first inequality is by (38) in case that $dip \in \mathtt{iD}^{ip}_{N}$ and by $\mathtt{sqn}^{ip}_{N}(dip) = \mathtt{nsqn}^{ip}_{N}(dip)$ otherwise; the second inequality follows by Theorem 7.27.

(iv) $dip \in \mathtt{vD}^{ip}_{N} \wedge \mathtt{dhops}^{ip}_{N}(dip) = 1$ holds, but $dip \in \mathtt{vD}^{ip}_{N'} \wedge \mathtt{dhops}^{ip}_{N'}(dip) = 1$ does not. We consider two subcases.

  – $dip \notin \mathtt{vD}^{ip}_{N'}$, which is equivalent to $dip \in \mathtt{iD}^{ip}_{N'} \vee \mathtt{dip} \notin \mathtt{kD}^{dip}_{N'}$. By Proposition 7.4, $\mathtt{dip} \notin \mathtt{kD}^{dip}_{N'}$ is not possible, hence $dip \in \mathtt{iD}^{ip}_{N'}$. Then, again using Theorem 7.27 and (38),

$$\mathtt{usqn}^{ip}_{N}(dip) = \mathtt{inc}(\mathtt{sqn}^{ip}_{N}(dip)) = \mathtt{inc}(\mathtt{nsqn}^{ip}_{N}(dip)) \leq \\ \mathtt{inc}(\mathtt{nsqn}^{ip}_{N'}(dip)) = \mathtt{sqn}^{ip}_{N'}(dip) = \mathtt{usqn}^{ip}_{N'}(dip)\ .$$

  – $dip \in \mathtt{vD}^{ip}_{N'}$ and $\mathtt{dhops}^{ip}_{N'}(dip) \neq 1$. Then, by Proposition 7.10, $\mathtt{dhops}^{ip}_{N'}(dip) \geq 2 > 1 = \mathtt{dhops}^{ip}_{N}(dip)$. As $N$ changes into $N'$, by Theorem 7.27 the quality of the route to $dip$ cannot decrease: $\xi^{ip}_{N}(\mathtt{rt}) \sqsubseteq_{dip} \xi^{ip}_{N'}(\mathtt{rt})$. Yet the hop count strictly increases, so the net sequence number must strictly increase as well: $\mathtt{nsqn}^{ip}_{N}(dip) < \mathtt{nsqn}^{ip}_{N'}(dip)$. From this we get

$$\mathtt{usqn}^{ip}_{N}(dip) = \mathtt{inc}(\mathtt{sqn}^{ip}_{N}(dip)) = \mathtt{inc}(\mathtt{nsqn}^{ip}_{N}(dip)) \leq \\ \mathtt{nsqn}^{ip}_{N'}(dip) = \mathtt{sqn}^{ip}_{N'}(dip) = \mathtt{usqn}^{ip}_{N'}(dip)\ .$$

□

- In the proof of Theorem 7.30, the case of Pro. 4, Line 4, where we "assume that the first line holds", we may no longer appeal to Proposition 7.6. Instead we consider two sub cases.

  – First, let $\mathtt{dhops}^{nhip}_{N}(dip) = 1$.
  Since $nhip \neq dip$ we have, by Proposition 7.12(c), $\mathtt{dhops}^{ip}_{N}(dip) \neq 1$ and hence, by Proposition 7.10, $\mathtt{dhops}^{ip}_{N}(dip) \geq 2 > 1 = \mathtt{dhops}^{nhip}_{N}(dip)$. Hence to conclude that $\xi^{ip}_{N}(\mathtt{rt}) \sqsubset_{dip} \xi^{nhip}_{N}(\mathtt{rt})$, it suffices to show that $\mathtt{nsqn}^{nhip}_{N}(dip) \geq \mathtt{nsqn}^{ip}_{N}(dip)$. Using the Equations (39) (twice) and (40), we get

$$\mathtt{nsqn}^{nhip}_{N}(dip) = \mathtt{sqn}^{nhip}_{N}(dip) \geq \mathtt{usqn}^{nhip}_{N}(dip) \stackrel{\bullet}{-} 1 \geq \\ \mathtt{usqn}^{nhip}_{N^\dagger}(dip) \stackrel{\bullet}{-} 1 \geq \mathtt{sqn}^{nhip}_{N^\dagger}(dip) \stackrel{\bullet}{-} 1 \geq \xi(\mathtt{osn}) = \mathtt{nsqn}^{ip}_{N}(dip)\ ,$$

  where the last inequality follows from $\mathtt{sqn}^{nhip}_{N^\dagger}(dip) > \xi(\mathtt{osn})$, which holds in the circumstances considered (cf. Page 53).

– If $\mathtt{dhops}^{nhip}_{N}(dip) \neq 1$ the net sequence number is strictly increased:

$$\mathtt{nsqn}^{nhip}_{N}(dip) = \mathtt{sqn}^{nhip}_{N}(dip) = \mathtt{usqn}^{nhip}_{N}(dip) \geq \\ \mathtt{usqn}^{nhip}_{N^\dagger}(dip) \geq \mathtt{sqn}^{nhip}_{N^\dagger}(dip) > \xi(\mathtt{osn}) = \mathtt{nsqn}^{ip}_{N}(dip)\ .$$

(As before for we use Equations (40), (39) and $\mathtt{sqn}^{nhip}_{N^\dagger}(dip) > \xi(\mathtt{osn})$ for the proof of this inequality.) Hence $\xi^{ip}_{N}(\mathtt{rt}) \sqsubset_{dip} \xi^{nhip}_{N}(\mathtt{rt})$.

- The proof of Proposition 7.37(b) needs to be modified; to this end we strengthen the statement as in (b) below, and establish (c) by simultaneous induction.

(b) The net destination sequence number of a routing table entry can never be greater than the destination's own sequence number.

$$\mathtt{nsqn}^{ip}_{N}(dip) \leq \xi^{dip}_{N}(\mathtt{sn}) \tag{41}$$

(c) The sequence number of a destination appearing in a route error message can never be more than 1 greater than the destination's own sequence number.

$$N \xrightarrow{R:^*\mathbf{cast}(\mathtt{rerr}(dests_c,ip_c))}_{ip} N' \;\wedge\; (rip_c, rsn_c) \in dests_c \;\Rightarrow\; rsn_c \mathbin{\dot{-}} 1 \leq \xi^{rip_c}_{N}(\mathtt{sn}) \tag{42}$$

*Proof.*

(b) The statement holds in the initial states. By Proposition 7.2, any update of $\xi^{dip}_{N}(\mathtt{sn})$ is harmless. Hence we have to examine all application calls of `update` and `invalidate`, restricting attention to those calls that actually modify the entry for *dip*, beyond its precursors.

**Pro. 1, Lines 10, 14, 18:** With Resolution 2(e) these calls maintain $\mathtt{nsqn}^{dip}_{N}(dip)$.

**Pro. 4, Line 4; Pro. 5, Line 2:** These updates yield a valid routing table entry with a known sequence number. The proof is unchanged from the one of Proposition 7.37(b).

**Pro. 1, Line 28; Pro. 3, Line 10; Pro. 4, Lines 13, 29; Pro. 5, Line 17:** By construction of `dests` (immediately before the invalidation call) $(rip, rsn) \in \xi^{ip}_{N}(\mathtt{dests})$ implies $rsn = \mathtt{inc}(\mathtt{sqn}(\xi^{ip}_{N}(\mathtt{rt}), rip))$. Hence the call maintains $\mathtt{nsqn}^{dip}_{N}(dip)$.

**Pro. 6, Line 3:** Let $(rip, rsn) \in \xi^{ip}_{N_3}(dests)$; then $(rip, rsn) \in \xi^{ip}_{N_2}(dests)$, and $\xi^{ip}_{N_2}(dests)$ stems from a received RERR message that must have been sent beforehand, say by a node $ip_c$ in state $N^\dagger$. By Invariant (42) we have $\mathtt{nsqn}^{ip}_{N}(rip) = rsn \mathbin{\dot{-}} 1 \leq \xi^{rip}_{N^\dagger}(\mathtt{sn}) \leq \xi^{rip}_{N}(\mathtt{sn})$.

(c) Immediately from Proposition 7.15 and Invariant (41). Here the weakened form of Proposition 7.15 proposed in the proof modifications for Resolutions (8d–e) is sufficient. The same holds for the weakened form of Proposition 7.15 proposed in the proof modifications for Resolution (9b). ☐

In case Resolution (2e) is chosen, the sequence-number-status flag becomes redundant and can be skipped, just as for Resolutions (2a) and (2d)—see Footnote 45. Moreover, with Resolution (2e) it would make sense to record the net sequence number in routing table entries rather than the sequence number, because only the net sequence number is monotonically increasing. This means that sequence numbers of routing table entries are not incremented upon invalidation, but instead a node that may initiate a route reply bases its actions on the incremented value of the destination sequence number in the received RREQ message.

### 10.4 From Groupcast to Broadcast

> “For each valid route maintained by a node as a routing table entry, the node also maintains a list of precursors that may be forwarding packets on this route. These precursors will receive notifications from the node in the event of detection of the loss of the next hop link.” [79, Sect. 2]

This notification is modelled by means of a groupcast mechanism. It sends error messages pertaining to certain routes to the precursors collected for those routes only. The idea is to reduce the number of messages received and handled. However, precursor lists are incomplete. They are updated only when a RREP message is sent (Lines 22, 23 of Pro. 4 and Lines 11, 12 of Pro. 5). The following example, showing a standard RREQ-RREP cycle, illustrates that all nodes not handling a route reply have no information about precursors; even those nodes that handle the RREP message may have insufficient information. As a consequence, some nodes are not informed of a link break and will use a broken route; hence packets are lost.

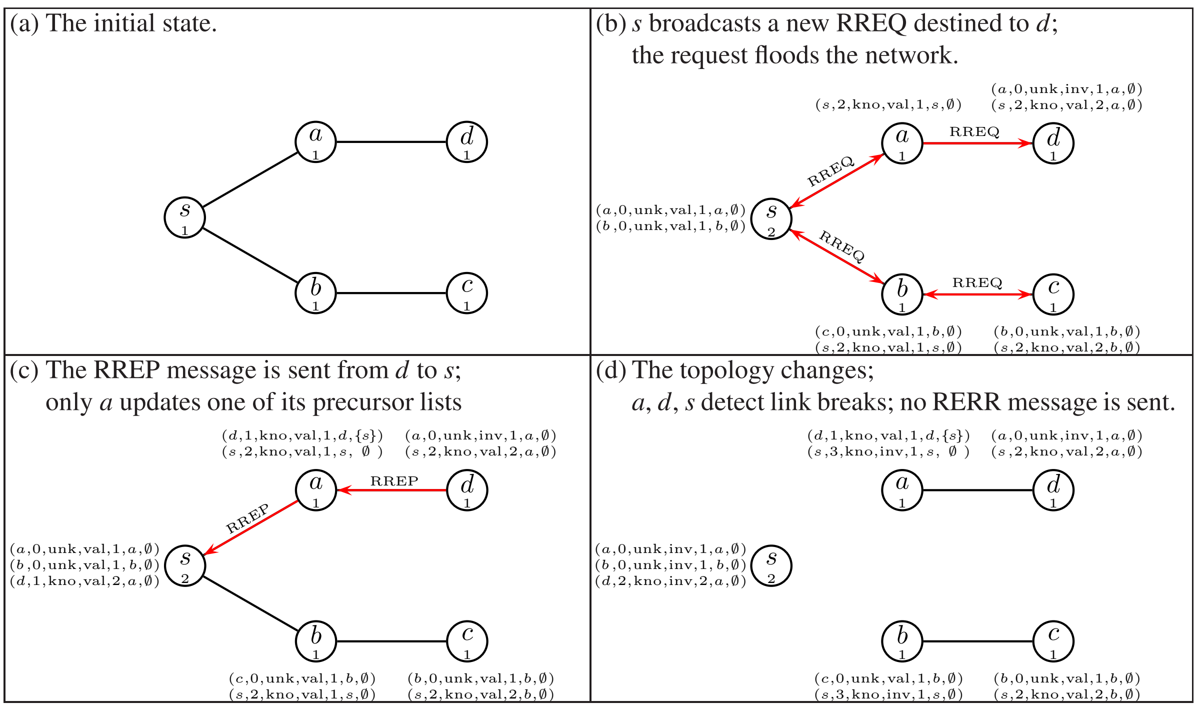



Figure 22: Precursor lists are incomplete[76]

The example is a standard RREQ-RREP cycle. Within the network given in Figure 22(a), a data packet is inserted at node $s$, destined for $d$. Consequently, $s$ issues a route discovery process. In Part (b) the RREQ message floods the network. While handling RREQ messages no (non-empty) precursor list is set or changed. In fact, it is not possible to detect the precursors for a route to the originator of the route request when handling and forwarding a RREQ message; the necessary information is not available. Thus, whenever a link break is detected during a route request process, no RERR message is sent, except when a node has information from previous control messages. In Figure 22(c), the reply is sent from node $d$ to $s$. When node $a$ forwards the RREP message, its adds $s$ to its list of precursors of the route to $d$ (Pro. 5, Line 11). However, it fails to add $d$ as a precursor of the route to $s$. In Part (d),

[76]This is the only example where precursor lists are shown; they are the last component of an entry.

the links between nodes $s$, $a$ and $b$ break down. Although nodes $a$, $b$ and $s$ detect the link break, they do not send error messages—so nodes $c$ and $d$ will not be informed about the broken routes. If these nodes receive packets for $s$ they will keep sending them via $a$ or $b$, without ever learning that none of those packets ever reaches $s$. In detail, when node $b$ (or $a$) receives a data packet for $s$ from $c$ (or $d$) it drops the packet (Pro. 3, Line 16) and composes a error message reporting the broken link to $s$ (Pro. 3, Line 20). However, this error message is send to the list of precursors for its route to $s$, which in our example is still empty. A variant of this example that constitutes a counterexample to the packet delivery property of Section 9 was already presented in Figure 17.

As already remarked in Section 6.4, the failure of node $a$ to add $d$ as a precursor of its route to $s$ can be remedied by the addition of the line $[\![\mathtt{rt} := \mathtt{addpreRT}(\mathtt{rt}, \mathtt{oip}, \{\mathtt{nhop}(\mathtt{rt}, \mathtt{dip})\})]\!]$ to Pro. 5, right after Line 12. One can even go a step further and also add the line

$$[\![\mathtt{rt} := \mathtt{addpreRT}(\mathtt{rt}, \mathtt{nhop}(\mathtt{rt}, \mathtt{oip}), \{\mathtt{nhop}(\mathtt{rt}, \mathtt{dip})\})]\!],$$

which is the equivalent to Pro. 5, Line 12. However, the problem at node $b$ cannot be fixed using precursors.

A possible solution is to abandon precursors and to replace every groupcast by a broadcast. At first glance this strategy seems to need more bandwidth, but this is not the case. Sending error messages to a set of precursors is implemented at the link layer by broadcasting the message anyway; a node receiving such a message then checks the header to determine whether it is one of the intended recipients. Instead of analysing the header only, a node can just as well read the message and decide whether the information contained in the message is of use. To be more precise: an error message is useful for a node if the node has established a route to one of the nodes listed in the message, and the next hop to a listed node is the sender of the error message. In case a node finds useful information inside the message, it should update its routing table and distribute another error message. This is exactly what happens in the route error process (Pro. 6).

In the specification given in Sections 5 and 6, the last entry of a routing table entry can be dropped; yielding small adaptations in functions and function calls (for example Line 10 of Pro. 1 should be $[\![rt := \mathtt{update}(\mathtt{rt}, (\mathtt{sip}, 0, \mathtt{unk}, \mathtt{val}, 1, \mathtt{sip}))]\!]$). Next to these small adaptations, the following changes need to be implemented:

**Pro. 1, Line 32; Pro. 3, Line 14; Pro. 4, Lines 17, 33; Pro. 5, Line 21:**
The command **groupcast**$(\mathtt{pre}, \mathtt{rerr}(\mathtt{dests}, \mathtt{ip}))$ is replaced by **broadcast**$(\mathtt{rerr}(\mathtt{dests}, \mathtt{ip}))$.

**Pro. 1, Lines 30, 31; Pro. 3, Lines 12, 13; Pro. 4, Lines 15, 16, 22, 23, 31, 32;**
**Pro. 5, Lines 11, 12, 19, 20:** These lines are dropped without replacement.

**Pro. 3, Line 20:** The command **groupcast**$(\mathtt{precs}(\mathtt{rt}, \mathtt{dip}), \mathtt{rerr}(\{(\mathtt{dip}, \mathtt{sqn}(\mathtt{rt}, \mathtt{dip}))\}, \mathtt{ip}))$ is replaced by **broadcast**$(\mathtt{rerr}(\{(\mathtt{dip}, \mathtt{sqn}(\mathtt{rt}, \mathtt{dip}))\}, \mathtt{ip}))$.

**Pro. 6, Lines 6–8:** The error forwarding is replaced by

```
(
  [ dests ≠ ∅ ]        /* the RERR needs to be forwarded */
    broadcast(rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
  + [ dests = ∅ ]      /* no valid route via broken link */
    AODV(ip,sn,rt,rreqs,store)
)
```

All invariants and statements of Sections 7 and 8 remain valid; the necessary proof adaptations are marginal and straightforward.

### 10.5 Forwarding the Route Request

In AODV's route discovery process, a destination node (or an intermediate node with an active route to the destination) will generate a RREP message in response to a received RREQ message. The RREQ message is then dropped and not forwarded. This termination of the route discovery process at the destination can lead to other nodes inadvertently creating non-optimal routes to the source node [72], where route optimality is defined in terms of a metric, for example hop count. In [72] it is shown that during the route discovery process in AODV, the only nodes that generally discover optimal routes to the source and destination nodes are those lying on the selected route between the source node and the destination node (or the intermediate node) generating the reply. All other network nodes receiving the RREQ message (in particular those located "downstream" of the destination node) may inadvertently be creating non-optimal routes to the source node due to the unavoidable receipt of RREQ messages over other routes. These "poorly selected paths have significantly higher routing-metric costs and their duration can extend to minute time scales" [72].

We illustrate this by the example in Figure 23. There, node $s$ wants to find a route to node $d$. It generates and broadcasts a RREQ message that is received by its neighbour nodes $d$ and $b$ (Figure 23(a)). Since node $d$ is the destination, it responds with a RREP message; the received RREQ message is not forwarded. On the other hand, $b$ continues to forward its received RREQ message, which eventually arrives at $a$ (Part (b)). At node $a$, a routing table entry is created for the source $s$, with a hop count of six. This is clearly not optimal, as $a$ is only two hops away from $s$. Due to the discarding of the RREQ message at node $d$, node $a$ is prevented from discovering its optimal route to $s$, via node $d$. In a next step, the RREQ message would also reach $d$ via $a$, but this message is then silently ignored by $d$.

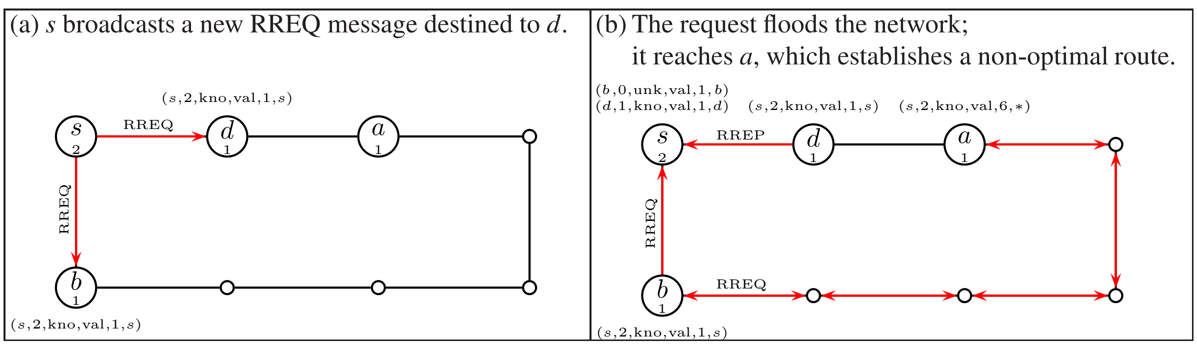



Figure 23: Non-optimal route selection

A possible modification to solve this problem is to allow the destination node to continue to forward the RREQ message. This will then enable node $a$ in Figure 23 to discover its optimal route to $s$. A route request is only stopped if it has been handled before. The forwarded RREQ message from the destination node needs to be modified to include a Boolean flag `handled`[77] that indicates a RREP message has already been generated and sent in response to the former message. In case the flag is set to `true`, it prevents other nodes (with valid route to the destination) from sending a RREP message in response to their reception of the forwarded RREQ message.

The entire specification of this variant differs only in eight lines from the original. Pro. 1 needs only slight adaptations. First the newly introduced flag needs to be introduced in Lines 8 and 11; these lines now read `[msg = rreq(hops,rreqid,dip,dsn,dsk,oip,osn,sip,handled)]` and `RREQ(hops,rreqid,dip,dsn,dsk,oip,osn,sip,handled,ip,sn,rt,rreqs,store)`, respectively. The broadcast

[77] The AODV RFC provides a field `Reserved` as part of a RREQ message [79, Sect. 5.1], which is more or less designed to cater for such extensions.

in Line 39 needs also be equipped with the flag. Since the route request is initiated, the flag is set to `false`:

$$\mathbf{broadcast}(\mathtt{rreq}(\mathtt{hops}+1,\mathtt{rreqid},\mathtt{dip},\mathtt{dsn},\mathtt{dsk},\mathtt{oip},\mathtt{osn},\mathtt{ip},\mathtt{false}))$$

All other changes happen in the process `RREQ`. The new process `RREQ` is given in Process 14.

**Process 14** The modified RREQ handling

```
RREQ(hops,rreqid,dip,dsn,dsk,oip,osn,sip,handled , ip,sn,rt,rreqs,store) ≝
1.  [ (oip,rreqid) ∈ rreqs ]      /* the RREQ has been received previously */
2.     AODV(ip,sn,rt,rreqs,store)      /* silently ignore RREQ, i.e. do nothing */
3.  + [ (oip,rreqid) ∉ rreqs ]      /* the RREQ is new to this node */
4.     ⟦rt := update(rt,(oip,osn,kno,val,hops+1,sip,∅))⟧      /* update the route to oip in rt */
5.     ⟦rreqs := rreqs ∪ {(oip,rreqid)}⟧      /* update rreqs by adding (oip,rreqid) */
6.     (
7.        [ handled = false ]      /* the request has not yet been handled */
8.           (
9.              [ dip = ip ]      /* this node is the destination node */
10.                ⟦sn := max(sn,dsn)⟧      /* update the sqn of ip */
11.                /* unicast a RREP towards oip of the RREQ and forward the request */
12.                unicast(nhop(rt,oip),rrep(0,dip,sn,oip,ip)) .
13.                   broadcast(rreq(hops+1,rreqid,dip,dsn,dsk,oip,osn,ip,true)) .
14.                   AODV(ip,sn,rt,rreqs,store)
15.                ▶ /* If the transmission is unsuccessful, a RERR message is generated */
16.                   ⟦dests := {(rip,inc(sqn(rt,rip)))|rip ∈ vD(rt) ∧ nhop(rt,rip) = nhop(rt,oip)}⟧
17.                   ⟦rt := invalidate(rt,dests)⟧
18.                   ⟦store := setRRF(store,dests)⟧
19.                   ⟦pre := ⋃{precs(rt,rip)|(rip,∗) ∈ dests}⟧
20.                   ⟦dests := {(rip,rsn)|(rip,rsn) ∈ dests ∧ precs(rt,rip) ≠ ∅}⟧
21.                   groupcast(pre,rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
22.             + [ dip ≠ ip ]      /* this node is not the destination node */
23.                (
24.                   [dip ∈ vD(rt) ∧ dsn ≤ sqn(rt,dip) ∧ sqnf(rt,dip) = kno]      /* fresh enough valid route to dip */
25.                      /* update rt by adding precursors */
26.                      ⟦rt := addpreRT(rt,dip,{sip})⟧
27.                      ⟦rt := addpreRT(rt,oip,{nhop(rt,dip)})⟧
28.                      /* unicast a RREP towards the oip of the RREQ and forward the request */
29.                      unicast(nhop(rt,oip),rrep(dhops(rt,dip),dip,sqn(rt,dip),oip,ip)) .
30.                         broadcast(rreq(hops+1,rreqid,dip,dsn,dsk,oip,osn,ip,true)) .
31.                         AODV(ip,sn,rt,rreqs,store)
32.                      ▶ /* If the transmission is unsuccessful, a RERR message is generated */
33.                         ⟦dests := {(rip,inc(sqn(rt,rip)))|rip ∈ vD(rt) ∧ nhop(rt,rip) = nhop(rt,oip)}⟧
34.                         ⟦rt := invalidate(rt,dests)⟧
35.                         ⟦store := setRRF(store,dests)⟧
36.                         ⟦pre := ⋃{precs(rt,rip)|(rip,∗) ∈ dests}⟧
37.                         ⟦dests := {(rip,rsn)|(rip,rsn) ∈ dests ∧ precs(rt,rip) ≠ ∅}⟧
38.                         groupcast(pre,rerr(dests,ip)) . AODV(ip,sn,rt,rreqs,store)
39.                   + [ dip ∉ vD(rt) ∨ sqn(rt,dip) < dsn ∨ sqnf(rt,dip) = unk ]      /* no fresh enough valid route */
40.                      /* no further update of rt */
41.                      broadcast(rreq(hops+1,rreqid,dip,max(sqn(rt,dip),dsn),dsk,oip,osn,ip,false)) .
42.                      AODV(ip,sn,rt,rreqs,store)
43.                )
44.          )
45.       + [ handled = true ]      /* the request has been answered before */
46.          /* the request is just forwarded (the RREQ was not handled before) */
47.          broadcast(rreq(hops+1,rreqid,dip,dsn,dsk,oip,osn,ip,true)) . AODV(ip,sn,rt,rreqs,store)
48.    )
```

The changes introduce the new flag and a case distinction based on that (Lines 7 and 45), as well as three new broadcasts (Lines 13, 30 and 47). For example, after initiating a route reply at the destination (Pro. 14, Line 12), the route request message is forwarded. In case the **unicast** of Line 12 is not successful (Lines 15–21), no forwarding is necessary, since it does not make sense to establish a link back to the originator of the RREQ message—the intermediate node just detected that this link is broken.

The proofs of Sections 7 and 8 are still valid, but need, as usual, some modifications. The newly introduced flag does not have any effect on the proofs—except that some line numbers change and the additional flag is added to all calls of `rreq`. The only real modification is that in Propositions 7.11(a), 7.13(a), 7.14(a) and 7.36(a)–(c), as well as in Theorem 7.32(b), three new **broadcast**-actions need to be examined. However, these cases are identical to the case Pro 4, Line 36 (corresponding to Line 41 of Pro. 14).

If this modification is applied to the example presented earlier, *a* now establishes an optimal route. The example is illustrated in detail in Figure 24. As before node *s* issues a route discovery in Figure 24(a). The RREQ message is received by *b* and *d*. Following the standard behaviour of AODV, node *b* forwards the RREQ and the destination *d* unicasts a RREP message back to *s*. Additionally, node *d* also broadcasts the modified request RREQ$_m$ (Part (b)). It is received by *s* and *a*. Since *s* initiated the request, it silently ignores the modified RREQ message; node *a* establishes an optimal route. Subsequently, both the original and the modified request are sent through the network (Figure 24(c)). The flooding is terminated as soon as every node has handled one of the RREQ messages—both have the same route request identifier. In the example, the node in the lower right corner receives both RREQ messages, forwards one of them (here RREQ) and silently ignores the other.

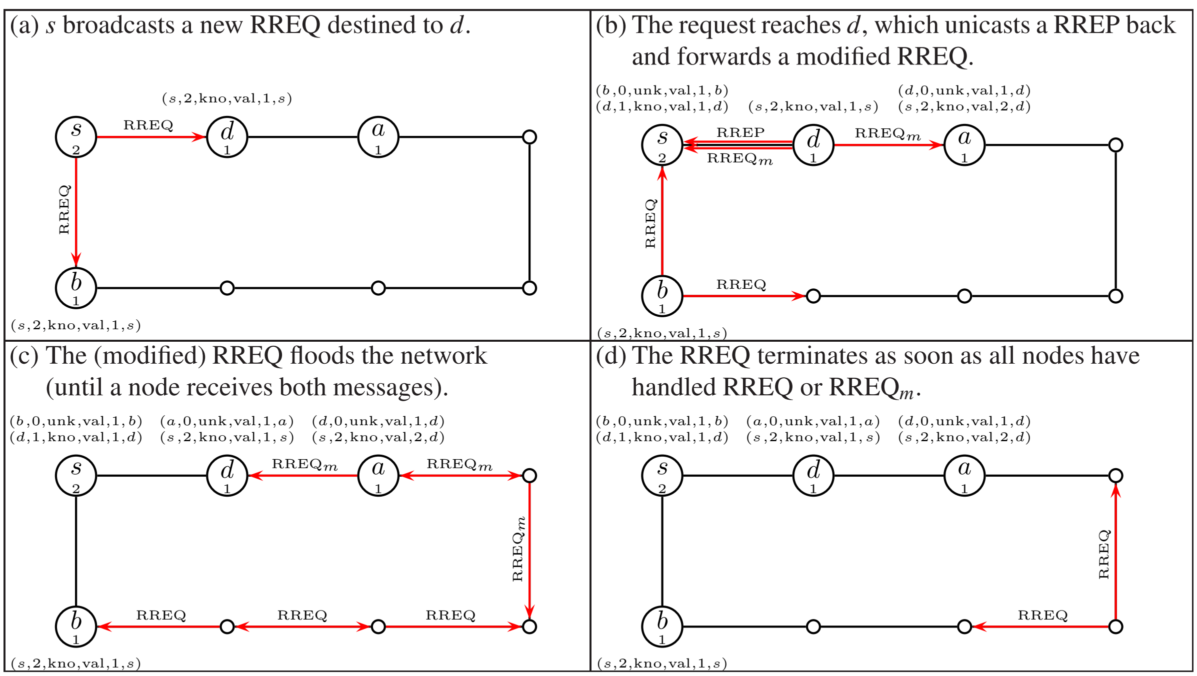



Figure 24: Forwarding route requests

An intermediate node answering the route request on behalf of the destination will also forward the RREQ message. The destination will receive the modified message and establish a route to the originator. By this, a bidirectional route between the source and the destination is established.

This finishes our list of improvements. All presented improvements are "orthogonal", i.e., they can be combined without problems; the properties of Sections 7 and 8 remain valid. In case new shortcomings are found, our specification as well as the proofs can easily be changed, as illustrated in this section.

# 11 Related Work

## 11.1 Process Algebras for Wireless Mesh Networks

Several process algebras modelling broadcast communication have been proposed before: the Calculus of Broadcasting Systems (CBS) [87, 88], the $b\pi$-calculus [22], CBS# [74], the Calculus of Wireless Systems (CWS) [69], the Calculus of Mobile Ad Hoc Networks (CMAN) [40], the Calculus for Mobile Ad Hoc Networks (CMN) [66], the $\omega$-calculus [94], rooted branching process theory (RBPT) [34], $bA\pi$ [42] and the broadcast psi-calculi [9]. The latter eight of these were specifically designed to model MANETs. However, we believe that none of these process calculi provides all features needed to fully model routing protocols such as AODV, namely data handling, (conditional) unicast and (local) broadcast. Moreover, all these process algebras lack the feature of guaranteed receipt of message. Due to this, it is not possible to analyse properties such as route discovery. We will elaborate on this in the following.

**Modelling Broadcast Communication** All these languages, as well as ours, feature a form of broadcast communication between nodes in a network, in which a single message emitted by one node can be received by multiple other nodes. In terms of operational semantics, this is captured by rules like

$$\frac{M \xrightarrow{\textbf{broadcast}(m)} M' \quad N \xrightarrow{\textbf{receive}(m)} N'}{M\|N \xrightarrow{\textbf{broadcast}(m)} M'\|N'} \qquad\qquad \frac{M \xrightarrow{\textbf{receive}(m)} M' \quad N \xrightarrow{\textbf{broadcast}(m)} N'}{M\|N \xrightarrow{\textbf{broadcast}(m)} M'\|N'}$$

that stem from [87] and can be found in the operational semantics of each of these languages, except for $bA\pi$. In such a rule the broadcast action in the conclusion is simply inherited from the broadcasting argument of the parallel composition, so that it remains available for the parallel composition with another receiver. In order to guarantee associativity of the parallel composition, i.e.

$$\big(\textbf{broadcast}(m).P\|\textbf{receive}(m).Q\big)\|\textbf{receive}(m).R = \textbf{broadcast}(m).P\|\big(\textbf{receive}(m).Q\|\textbf{receive}(m).R\big)$$

one also needs a rule like

$$\frac{M \xrightarrow{\textbf{receive}(m)} M' \quad N \xrightarrow{\textbf{receive}(m)} N'}{M\|N \xrightarrow{\textbf{receive}(m)} M'\|N'}\ .$$[78]

**Lossy Broadcast versus Enforced Synchronisation without Blocking** The languages CMAN, CMN, RBPT, the $\omega$-calculus, $bA\pi$ and the broadcast psi-calculi model *lossy* communication, which allows, as a nondeterministic possibility, any node to miss a message of another node, even when the two nodes are within transmission range. The corresponding operational rules are

$$\frac{M \xrightarrow{\textbf{broadcast}(m)} M'}{M\|N \xrightarrow{\textbf{broadcast}(m)} M'\|N} \qquad\qquad \frac{N \xrightarrow{\textbf{broadcast}(m)} N'}{M\|N \xrightarrow{\textbf{broadcast}(m)} M\|N'}\ .$$

[78] CMN lacks such a rule, resulting in a non-associative parallel composition. This in turn leads to a failure of [66, Lemma 3.2], saying that structural congruence "respects transitions", i.e. is a strong bisimulation. This mistake is propagated in [31]. The $\omega$-calculus lacks this rule as well, but there associativity of the parallel composition is enforced by closing the transition relation under structural congruence. Nevertheless, the same problem returns in the form of a failure of [94, Theorem 9], stating that strong bisimilarity is a congruence. Namely, $\mathbf{r}(x).nil:\{g\} \mid \mathbf{r}(x).nil:\{g\}$, the parallel composition of two connected nodes, each doing a receive action, is bisimilar to $\mathbf{r}(x).\mathbf{r}(x).nil:\{g\}$, a single node doing two receive actions; yet in the context $\mathbf{b}(u).nil:\{g\} \mid \_$, involving a fully connected node broadcasting a value $u$, only the former can do a broadcast transition to a process that can do no further receive actions.

In such a language it impossible to formulate valid properties of modelled protocols like "if there is a path from *ip* to *dip*, and the topology does not change, then a packet for *dip* submitted at *ip* will eventually be delivered at *dip*" (cf. Section 9.3). Namely, there is never a guarantee that any message arrives.

In the operational semantics, the only alternative to the lossy rules above appears to be enforced synchronisation of a broadcast action of one component in a parallel composition with some (in)activity of the other. This approach is followed in CBS, $b\pi$, CBS# and CWS, as well as in AWN. In CBS, $b\pi$, CBS# and CWS, and in the optional augmentation of AWN presented in Section 4.5, any node within transmission range *must* receive a message $m$ sent to it, provided the node is *ready* to receive it, i.e., in a state that admits a transition $\textbf{receive}(m)$. This proviso makes all these calculi *non-blocking*, meaning that no sender can be delayed in transmitting a message simply because one of the potential recipients is not ready to receive it. The default version of AWN (Section 4.3) lacks this proviso and hence does allow blocking. However, in applications of our language we model nodes in such a way that any message can be received at any time (cf. Section 6.6). Nodes with this property are called *input enabled*, a concept introduced in the work on *IO-automata* [61]. For such applications our models are non-blocking.

In CBS [87], actions $\textbf{discard}(m)$ are used to model situations where a process cannot receive a message $m$. The definitions are such that $N \xrightarrow{\textbf{discard}(m)} N'$ only if $N' = N$—so the discard actions do not correspond with any state-change—and $N \xrightarrow{\textbf{discard}(m)} N'$ if an only if $N \overset{\textbf{receive}(m)}{\not\longrightarrow}$. Now the rules for broadcast are augmented by

$$\frac{M \xrightarrow{\textbf{receive}(m)} M' \quad N \xrightarrow{\textbf{discard}(m)} N'}{M\|N \xrightarrow{\textbf{receive}(m)} M'\|N'} \qquad \frac{M \xrightarrow{\textbf{discard}(m)} M' \quad N \xrightarrow{\textbf{discard}(m)} N'}{M\|N \xrightarrow{\textbf{discard}(m)} M'\|N'} \qquad \frac{M \xrightarrow{\textbf{discard}(m)} M' \quad N \xrightarrow{\textbf{receive}(m)} N'}{M\|N \xrightarrow{\textbf{receive}(m)} M'\|N'}$$

$$\frac{M \xrightarrow{\textbf{broadcast}(m)} M' \quad N \xrightarrow{\textbf{discard}(m)} N'}{M\|N \xrightarrow{\textbf{broadcast}(m)} M'\|N'} \qquad \frac{M \xrightarrow{\textbf{discard}(m)} M' \quad N \xrightarrow{\textbf{broadcast}(m)} N'}{M\|N \xrightarrow{\textbf{broadcast}(m)} M'\|N'}\ .$$

This way, in a parallel composition $M\|N$, a broadcast action of one component can never be blocked by the other component; it synchronises either with a receive or a discard of the other component, depending on whether the other component is ready to receive a message or not. In $b\pi$ and CBS# the same approach is followed, except that in CBS# messages are annotated with their sender (i.e. read $m$ as a sender-message pair in all rules above). This way, one can say that a node $N$ (in a certain state) can receive a message from one sender, but not from another.

At the expense of the use of negative premises, it is possible to eliminate the discard action, and replace a premise $N \xrightarrow{\textbf{discard}(m)} N'$ by $N \overset{\textbf{receive}(m)}{\not\longrightarrow}$ (cf. Section 4.5). Another variant of the same idea, applied in [88] and CWS [69], is to simply replace the discard transitions $N \xrightarrow{\textbf{discard}(m)} N$ by $N \xrightarrow{\textbf{receive}(m)} N$. Thus, whenever a node $N$ is unable to receive a particular message from a particular other node, its operational semantics introduces a discard transition $N \xrightarrow{\textbf{receive}(m)} N$ that essentially allows the message to be received and completely ignored.

**Local Broadcast with Arbitrary Dynamic Topologies** CBS models global broadcast communication, where all processes (or nodes) are able to receive any broadcast message. The language $b\pi$ allows processes to join *groups*, and receive all messages sent by members of that group. In order to join a group a process needs to have knowledge of the name of this group. This appears less suitable for the specification of wireless mesh networks, where nodes may receive messages from unknown other nodes as soon as they move into transmission range. The other languages allow arbitrary network topologies, and feature a local broadcast, which can be received only by nodes within transmission range of the sender. CWS deals with static topologies, whereas in CBS#, CMAN, CMN, the $\omega$-calculus, RBPT, $bA\pi$ and the broadcast psi-calculus, as in our approach, the topology is subject to arbitrary changes.

**Guaranteed Receipt of Messages Broadcast within Transmission Range** The syntax of CBS# and CWS does not permit the construction of meaningful nodes that are always ready to receive a message. Hence our model is the first that assumes that any message *is* received by a potential recipient within range. It is this feature that allows us to evaluate whether a protocol satisfies the *packet delivery* property of Section 9.3. *Any* routing protocol formalised in any of the other formalisms would automatically fail to satisfy such a property.

**Modelling Connectivity** To model connectivity of nodes in the current topology, our node expressions have the form $ip : P : R$, where $P$ is a process running on the node, $ip$ is the node's address, and $R$ is the current transmission range, given as the set of addresses of nodes that can receive messages sent by this node. Changes in the transmission range occur through **connect** and **disconnect** actions, which can occur at any time (cf. Section 4.3). This follows CMAN [40], where our $l{:}p{:}\sigma$ is denoted as $\lfloor p\rfloor_l^\sigma$, with $l$ being a *location*, which plays the role of the node's address. In CWS nodes have the form $n[P]^c_{l,r}$, where $n$ is the node address (our $ip$), $c$ denotes the broadcast channel (e.g. a frequency) to which the node is tuned, $l$ is the physical location and $r \in \mathbb{R}$ the *radius* or transmission range of the node. A global function $d$ is postulated that given two locations $l$ and $l'$ returns the distance $d(l,l')$ between them; comparing this value with the radius of a node at $l$ determines whether broadcast messages from that node reach a node at $l'$. In comparison with CWS our formalism could be said to use only one possible channel. CMN uses the same syntax as CWS, except that the channel is replaced by a *mobility tag* $\mu$, telling whether the node is mobile or stationary. In the latter case, the physical location $l$ of the node is subject to chance. The $\omega$-calculus has node expressions of the form $P : G$, where $P$ is a process and $G$ the set of *groups* the node belongs to. Each group is a clique in the graph expressing the network topology, and two nodes can communicate iff they belong to a common group. Contrary to these approaches, in CBS#, RBPT and the broadcast $\psi$-calculi node expressions do not contain connectivity information. Instead, connectivity is modelled in the semantics only, by labelling transitions with (information about) the topologies that support them. In CBS# node expressions have the form $n[P,S]$ where $n$ is the *location* or identifier of a node, $P$ a process and $S$ the node's memory, storing values that could have been received. RBPT node expressions simply have the form $[\![P]\!]_l$, denoting a process $P$ at the location $l$. Broadcast psi differs from the above calculi in that it makes no distinction between processes and node (or network) expressions. Consequently, nodes are not equipped with an address and connectivity cannot be expressed as a relation between nodes. Instead it is expressed as a relation between channel expressions occurring in processes. In $bA\pi$ nodes have the form $\lfloor p\rfloor_l$, as in CMAN, but without any connectivity information. The operational semantics differs from those of the other calculi, in that a broadcast action results in messages sitting as separate components in the parallel composition among the nodes. Connectivities of the form $\{l \mapsto m\}$, saying that node $l$ can receive message send by node $m$, also occur as separate entities in this parallel composition, and can react with messages to guide them in appropriate directions.

The above comparison between the various formalisms is summarised in Table 9, of which the last three columns are largely taken from [34]. The sixth column tells whether connectivity information is stored in the syntax of node or network expressions, or whether it appears in the structural operation semantics of the language only. The last column indicates whether the formalism assumes the connectivity relation between nodes to be symmetric. In this regard there are two versions of AWN; in [26] the default version is asymmetric, whereas here, in view of the application to AODV, we made the symmetric version the default.

**Operational Semantics of Local Broadcast with Enforced Synchronisation** Whereas CBS# and CWS enrich receive actions of messages with their senders—to indicate that a message can be received from one sender but not from another, based on the topology—in our operational semantics this adminis-

| Process algebra | Message loss | Type of broadcast | | Connectivity model | | |
|---|---|---|---|---|---|---|
| CBS [88] '91 | enforced synchr. | global broadcast | | | | symmetric |
| $b\pi$ [22] '99 | enforced synchr. | subscription-based broadcast | | | | symmetric |
| CBS# [74] '06 | enforced synchr. | local bc. | dynamic top. | $n[P,S]$ | op. sem. | symmetric |
| CWS [69] '06 | enforced synchr. | local bc. | static topology | $n[P]^c_{l,r}$ | node | symmetric |
| CMAN [40] '07 | lossy broadcast | local bc. | dynamic top. | $\lfloor p \rfloor^\sigma_l$ | node | symmetric |
| CMN [66] '07 | lossy broadcast | local bc. | dynamic top. | $n[P]^\mu_{l,r}$ | node | symmetric |
| $\omega$ [94] '07 | lossy broadcast | local bc. | dynamic top. | $P:G$ | node | symmetric |
| RBPT [34] '08 | lossy broadcast | local bc. | dynamic top. | $[\![P]\!]_l$ | op. sem. | asymmetric |
| $bA\pi$ [42] '09 | lossy broadcast | local bc. | dynamic top. | $\lfloor p \rfloor_l$ | network | asymmetric |
| $b\psi$ [9] '11 | lossy broadcast | local bc. | dynamic top. | $P$ | op. sem. | asymmetric |
| AWN here '11 | enforced synchr. with guar. receipt | local bc. | dynamic top. | $ip$:$P$:$R$ | node | asym./sym. |

Table 9: Process algebras modelling broadcast communication

trative burden is shifted to the broadcast actions—they are annotated with the range of possible receivers. This enables us to model groupcast and unicast actions, which are not treated in CBS# and CWS, in the same way as broadcast actions. However, the price to be payed for this convenience is that our actions **arrive**$(m)$, which are synchronisations of (non)receive actions of multiple components, need to be annotated with the locations of all these components. Moreover, this set of locations is partitioned into the ones that are in and out of transmission range of the message $m$. It does not appear possible to model our groupcast in the style of CBS# and CWS.

**Conditional Unicast** Our novel *conditional unicast* operator chooses a continuation process dependent on whether the message can be delivered. This operator is essential for the correct formalisation of AODV and other network protocols. In practice such an operator may be implemented by means of an acknowledgement mechanism; however, this is typically done at the link layer, from which the AODV specification [79], and hence our formalism, abstracts. One could formalise a conditional unicast as a standard unicast in the scope of a priority operator [16]; however, our operator allows an operational semantics within the de Simone format. Of the other process algebras of Table 9, only the $\omega$-calculus, $bA\pi$ and the broadcast psi-calculi model unicast at all, next to broadcast; they do not have anything comparable to the conditional unicast.

**Data Structures** Although our treatment of data structures follows the classical approach of universal algebra, and is in the spirit of formalisms like $\mu$CRL [46], we have not seen a process algebra that freely mixes in imperative programming constructs like variable assignment. Yet this helps to properly capture AODV and other routing protocols. This mixture should make the syntax of AWN on the level of sequential processes easy to read for anybody who has some experience in programming, thus making it easier to implement protocol specifications written in AWN.

**Other Process Algebras for WMNs** In [31] CMN is extended with mechanisms for unicast and multicast/groupcast communication; the paper focuses on power-consumption issues. Process calculi in the same spirit as the ones above, but focusing on security aspects and trust, appear in [43, 68]. Probabilistic and stochastic calculi for WMNs, based on similar design principles as the process algebras discussed above, are proposed in [97, 38, 29, 60, 98, 10, 11, 30]. An extended and improved version of CWS appears in [59]. Extensions of CWS with time appear in [67, 62, 60, 12, 102]; these process algebras focus on the MAC-layer rather than the network layer of the TCP/IP reference model. In [41] a variant of CMAN is proposed that limits mobility. A variant of CMAN that incorporates another mobility model appears in [44]. In [35] the process algebra RBPT is enriched with specifications of sets of topologies into Computed Network Theory (CNT). This facilitates the equational axiomatisation of RBPT. In [36] RBPT and CNT are extended with encapsulation and abstraction operators; a simple abstraction of AODV has been shown to be loop free in this framework by means of equational reasoning [37].

## 11.2 Modelling, Verifying and Analysing AODV and Related Protocols

Our complete formalisation of AODV, presented here, has grown from elaborating a partial formalisation of AODV in [94]. The features of our process algebra were largely determined by what we needed to enable a complete and accurate formalisation of this protocol. The same formalism has been used to model the Dynamic MANET On-demand (DYMO) Routing Protocol (also known as AODVv2) [21]. By this we did not only derive an unambiguous specification for draft-version 22 (as we did for the RFC of AODV); we were also able to verify that some of the problems discovered for AODV have been addressed and solved. However, we showed that other limitations still exist, e.g., the establishment of non-optimal routes (cf. Section 10.5). We conjecture that AWN is also applicable to a wide range of other wireless protocols, such as the Dynamic Source Routing (DSR) protocol [56], the Lightweight Underlay Network Ad-hoc Routing (LUNAR) protocol [100, 101], the Optimized Link State Routing (OSLR) protocol [15] or the Better Approach To Mobile Adhoc Networking (B.A.T.M.A.N.) [75]. The specification and the correctness of the latter three, however, rely heavily on timing aspects; hence an AWN-extension with time appears necessary (see also Section 12).

**Test-bed Experiments and Simulation** While process algebra can be used to formally model and verify the correctness of network routing protocols, test-bed experiments and simulations are complementary tools that can be used to quantitatively evaluate the performance of the protocols. While test-bed experiments are able to capture the full complex characteristics of the wireless medium and its effect on the network routing protocols [63, 84], network simulators [76, 92] offer the ease and flexibility of evaluating and comparing the performance of different routing protocols in a large-scale network of hundreds of nodes, coupled with the added advantage of being able to repeat and reproduce the experiments [18, 80, 54].

**Loop Freedom** Loop freedom is a crucial property of network protocols, commonly claimed to hold for AODV [79]. Merlin and Segall [65] were amongst the first to use sequence numbers to guarantee loop freedom of a routing protocol. We have shown that several *interpretations* of AODV—consistent ways to revolve the ambiguities in the RFC—fail to be loop free, while proving loop freedom of others.

A preliminary draft of AODV has been shown to be not loop free by Bhargavan et al. in [6]. Their counterexamples to loop freedom have to do with timing issues: the premature deletion of invalid routes, and a too quick restart of a node after a reboot. Since then, AODV has changed to such a degree that these examples do not apply to the current version [79]. However, similar examples, claimed to apply to the current version, are reported in [33, 90]. All these papers propose repairs that avoid these loops through better timing policies. In contrast, the routing loops documented in [39] as well as in Section 8 of this paper are time-independent.

Previous attempts to prove loop freedom of AODV have been reported in [82, 6, 106], but none of these proofs are complete and valid for the current version of AODV [79]:

- The proof sketch given in [82] uses the fact that when a loop in a route to a destination $Z$ is created, all nodes $X_i$ on that loop must have route entries for destination $Z$ with the same destination sequence number. "*Furthermore, because the destination sequence numbers are all the same, the next hop information must have been derived at every node $X_i$ from the same RREP transmitted by the destination Z*" [82, Page 11]. The latter is not true at all: some of the information could have been derived from RREQ messages, or from a RREP message transmitted by an intermediate node that has a route to $Z$. More importantly, the nodes on the loop may have acquired their information on a route to $Z$ from different RREP or RREQ messages, that all carried the same sequence number. This is illustrated by our loop created in Figure 11 (Section 8.2.3).

- Based on an analysis of an early draft of AODV[79] [6] suggests three improvements. The modified version is then proved to be loop free, using the following invariant (written in our notation):

$$\begin{array}{ll} \multicolumn{2}{l}{\text{if } \mathit{nhip} = \mathtt{nhop}_N^{ip}(\mathit{dip}), \text{ then}} \\ (1) & \mathtt{sqn}_N^{ip}(\mathit{dip}) \leq \mathtt{sqn}_N^{\mathit{nhip}}(\mathit{dip}), \text{ and} \\ (2) & \mathtt{sqn}_N^{ip}(\mathit{dip}) = \mathtt{sqn}_N^{\mathit{nhip}}(\mathit{dip}) \;\Rightarrow\; \mathtt{dhops}_N^{ip}(\mathit{dip}) < \mathtt{dhops}_N^{\mathit{nhip}}(\mathit{dip})\ . \end{array}$$

  This invariant does not hold for this modified version of AODV nor for the current version, documented in the RFC. It can happen that in a state $N$ where $\mathtt{sqn}_N^{ip}(\mathit{dip}) = \mathtt{sqn}_N^{\mathit{nhip}}(\mathit{dip})$, node *ip* notices that the link to *nhip* is broken. Consequently, *ip* invalidates its route to *dip*, which has *nhip* as its next hop. According to recommendation (**A1**) of [6, Page 561]), node *ip* increments its sequence number for the (invalid) route to *dip*, resulting in a state $N'$ for which $\mathtt{sqn}_{N'}^{ip}(\mathit{dip}) > \mathtt{sqn}_{N'}^{\mathit{nhip}}(\mathit{dip})$, thereby violating the invariant.

  Note that the invariant of [6] does not restrict itself to the case that the routing table entry for *dip* maintained by *ip* is *valid*. Adapting the invariant with such a requirement would give rise to a valid invariant, but one whose verification poses some problems, at least for the current version of AODV. These problems led us, in this paper, to use *net sequence numbers* instead (cf. Section 7.5).

  Recommendation (**A1**) is assumed to be in effect for the (improved) version of AODV analysed in [6], although it was not in effect for the draft of AODV existing at the time. Since then, recommendation (**A1**) has been incorporated in the RFC. Looking at the proofs in [6], it turns out that Lemma 20(1) of [6] is invalid. This failure is surprising, given that according to [6] Lemma 20 is automatically verified by SPIN. A possible explanation might be that this lemma *is* obviously valid for the version of AODV prior to the recommendations of [6].

**Model Checking** Bhargavan et al. [6] not only found problems in an early draft of AODV, they were also among the first to apply model checking techniques to AODV, thereby demonstrating the feasibility and value of automated verification of routing protocols. For their studies they use the model checker SPIN.

Musuvathi et al. [73] introduced the model checker CMC primarily to search for coding errors in implementations of protocols written in C. They use AODV (draft version 10) as an example and were able to discover 34 distinct errors in three different implementations: mad-hoc (version 1), Kernel AODV (version 1.5) and AODV-UU (version 0.5).[80] They also found a problem with the specification itself: they discovered that routing loops can be created when sequence numbers are just copied from an incoming RERR message, without checking the value. We have discovered the same problem (see Ambiguity 8 in Section 8.2.3) and proposed the same solution as they do, namely introducing a check prior to invalidating routes (in our specification the check is $\mathtt{sqn}(\mathtt{rt},\mathtt{rip}) < \mathtt{rsn}$ in Line 2 of Pro. 6). However, the routing loops found in [73] crucially depend on the use of an unordered message queue, in which messages can

[79] Draft version 2 is analysed, dated November 1998; the RFC can be seen as version 14, dated July 2001.

[80] For our analysis in Section 8.3, we use version 0.9.5 of AODV-UU and version 2.2.2. of Kernel AODV. We did not analyse mad-hoc since it is no longer actively supported.

overtake each other after being sent. Our loop, on the other hand, manifests itself even when using FIFO queues, as specified in the RFC. Although [73] testifies that both the bug and the fix where accepted by the protocol authors, the proposed solution is not incorporated in the current standard [79].

Chiyangwa and Kwiatkowska [14] use the timing features of the model checker UPPAAL to study the relationship between the timing parameters and the performance of route discovery in AODV, and find some route discovery failures.

Using the model checkers SPIN and UPPAAL, [103] demonstrates that the ad-hoc protocol LUNAR satisfies a strong variant of the packet delivery property for a number of routing scenarios.

All this related work show that model checking can be used as a diagnostic tool for MANETs and WMNs. Although model checking generally lacks the ability to verify protocols for an arbitrary and changing topology, it can be efficiently used to check specific scenarios (topologies) and to reveal problems in the specification in an early stage of protocol development; even before anybody starts to verify interesting properties by pen-and-paper proofs or with support of interactive theorem provers.

We believe that model checking as a diagnostic tool can complement the process-algebraic approach presented in this paper. Having the ability of model checking specifications written in AWN will allow the confirmation and detailed diagnostics of suspected errors which arise during modelling. The availability of an executable model will become especially useful in the evaluation of proposed improvements. A first step to this complementation was taken in [24] and further elaborated in [25]. In [24], we generated a ("time-free") UPPAAL model of AODV from our AWN specification, confirmed some of the problems discovered by Chiyangwa and Kwiatkowska [14], and show their independence of time. In [25] we continued the analysis of AODV by model-checking techniques by an exhaustive exploration of AODV's behaviour in all network topologies up to 5 nodes. We were able to automatically locate problematic and undesirable behaviours. In that paper, we moreover sketched possible modifications of AODV, which also were subjected to rigorous analysis by means of model checking. In these experiments we created an environment in which we can test a range of different topologies in a systematic manner. This will allow us to do a fast comparison between standard protocols (e.g. given by RFCs) and proposed variations in contexts known to be problematic.

**Statistical Model Checking** Unfortunately, current state-of-the art (exhaustive) model checkers are unable to handle protocols of the complexity needed for WMN routing in realistic settings: network size (usually dozens, sometimes even hundreds of nodes) and topology changes yield an explosion in the state space. Another limitation of (exhaustive) model checking is that a quantitative analysis is often not possible: finding a shortcoming in a protocol is great but does not show how often the shortcoming actually occurs. Statistical model checking (SMC) [105, 93] is a complementary approach that can overcome these problems. It combines ideas of model checking and simulation with the aim of supporting quantitative analysis as well as addressing the size barrier. Among others, SMC has been used to analyse AODV and DYMO.

[51] first develops timed models for AODV and DYMO. These models are based on the UPPAAL models created from our AWN specifications. The paper then carries out a systematic analysis across all small networks. In contrast to simulation and test bed studies, the analysis based on quality and quantity enables the examination of reasons for observed differences in performance between AODV and DYMO. [51] then examines the feasibility of SMC w.r.t. scalability; the results imply that networks of realistic size (up to 100 nodes) can be analysed.

For small networks it is possible to analyse all topologies. This gives a good overall view of the performance and behaviour in any situation. For large networks this is not feasible, and so the selection of topologies as well as their dynamic behaviour becomes something of a 'stab in the dark'. The Node Placement Algorithm for Realistic Topologies (NPART) [70] is a tool that allows the generation of arbitrary-sized topologies and transmission ranges; it has been shown that the generated topologies have graph characteristics similar to realistic wireless multihop ones. [27] proposes a topology-based mobility model that abstracts from physical behaviour and models mobility as probabilistic changes in the topology. It is demonstrated how this model can be instantiated to cover the main aspects of the random walk and the random waypoint mobility model. The model is not a stand-alone model, but intended to be used in combination with protocol models. As one application a brief analysis of the Ad-hoc On demand Distance Vector (AODV) routing protocol is given.

A more thorough (quantitative) analysis of AODV based on this topology-based mobility model is performed in [49]. Here, variants of AODV, such as always forwarding route replies (see Section 10.2), are analysed as well. The paper makes surprising observations on the behaviour of AODV. For example, it is shown that some optional features (D-flag) should not be combined with others (resending). Another observation of [49] is that a well-known shortcoming occurs more often than expected and has a significant effect on the success of route establishment.

**Other Approaches** Next to process algebra and model checking other approaches have been used to analyse WMNs. A frequently used approach is *coloured Petri nets* (CPNs) [55].

The idea to use CPNs to model routing protocols for MANETs was first employed in [104]: the paper proposes a *topology approximation* (TA) mechanism for modelling mobility and, based on this, presents a CPN model of AODV. Using this formal model the network behaviour for a network with 5 nodes is simulated.

Mandatory parts of DYMO are modelled as a hierarchy of CPNs in [23]. The paper analyses draft-version 10 and identifies and resolves some ambiguities in specification. Moreover, it points at problematic behaviour; six of these findings have been reported to the IETF MANET Working Group mailing list, and have been resolved by the DYMO developers in version 11 of the DYMO specification. The model presented in [23] has a complex net structure, comprising 4 levels of hierarchy and 14 modules. A much smaller model of DYMO, which even covers some optional features, is presented in [7].[81] Reducing the size of the model also reduces the state space, so larger networks can be analysed. Experiments performing test runs on small topologies confirm specified behaviour. However, similar to model checking, networks with a few nodes only can be analysed.

*Graph Transformation Systems* are used in [91] to model DYMO (version 10), but without the feature of route reply by intermediate nodes. The paper provides a semi-algorithm, based on graph rewriting, which was used to verify loop freedom for this version of DYMO.

Other formal approaches are *algebraic techniques* involving a.o. semirings and matrices. Sobrinho was the first who brought algebraic reasoning into the realm of hop-by-hop routing [95]. He uses algebraic properties to argue about the relationship between routing algorithms and Dijkstra's shortest path algorithm. This approach has been further elaborated for the analysis of path vector protocols like the Border Gate Protocol BGP [96, 45]. Similar algebraic reasoning has been performed in [52] to present algebraic versions of the algorithms of Dijkstra and Floyd-Warshall. [50] presents first steps towards an algebraic characterisation of AODV using these algebraic techniques.

## 12 Conclusion and Future Work

In this paper we have proposed AWN, a novel process algebra that can be used to model, verify and analyse (routing) protocols for Wireless Mesh Networks (WMNs). The applicability of the process algebra has been demonstrated by a careful analysis of the Ad hoc On-Demand Distance Vector (AODV)

[81] A detailed comparison between the models given in [23] and [7] is given in [7, Sect. 4].

Routing Protocol. To the best of our knowledge it is by far the most detailed analysis of a routing protocol for WMNs.

The introduced process algebra AWN covers major aspects of WMN routing protocols, for example the crucial aspect of data handling, such as maintaining routing table information. Amongst others, the assignment primitive, which is used to manipulate data, turns AWN into an easy to read language—its syntax is close to the syntax of programming languages. Key operators of AWN are *local broadcast* and *conditional unicast*. Local broadcast allows a node to send messages to all its immediate neighbours as implemented by the physical and data link layer. Conditional unicast models an abstraction of an acknowledgment-of-receipt mechanism that is typical for unicast communication but absent in broadcast communication, as typically implemented by the link layer of relevant wireless standards such as IEEE 802.11. AWN can capture the bifurcation depending on the success of the unicast; it allows error handling in response to failed communications while abstracting from link layer implementations of the communication handling.

The unique set of features and primitives of AWN allows the creation of accurate and concise models of relatively complex and practically relevant network protocols in a simple language. We have demonstrated this by giving a complete and accurate model of the core functionality of AODV, a widely used protocol of practical relevance. We currently do not model optional features such as local route repair, expanding ring search, gratuitous route reply and multicast. We also abstract from all timing issues. In addition to modelling the complete set of core functionalities of the AODV protocol, our model also covers the interface to higher protocol layers via the injection and delivery of application layer data, as well as the forwarding of data packets at intermediate nodes. Although this is not part of the AODV protocol specification, it is necessary for a practical model of any reactive routing protocol, where protocol activity is triggered via the sending and forwarding of data packets.

Process algebras are standard tools to describe interactions, communications and synchronisations between a collection of independent agents, processes or network nodes. They provide algebraic laws that facilitate formal reasoning. To demonstrate the strength of formal reasoning we performed a careful analysis of AODV, in particular with respect to the loop-freedom property. By establishing invariants that remain valid in a network running AODV, we have shown that our model is in fact loop free. In contrast to protocol evaluation using simulation, test-bed experiments or model checking, where only a finite number of specific network scenarios can be considered, our reasoning with AWN is generic and the proofs hold for any possible network scenario in terms of topology and traffic pattern. None of the experimental protocol evaluation approaches can deliver this high degree of assurance about protocol behaviour. We have also shown that, in contrast to common belief, sequence numbers do not guarantee loop freedom, even if they are increased monotonically over time and incremented whenever a new route request is generated.

Our analysis of AODV uncovered several ambiguities in the RFC, the de facto standard of AODV. In this paper we have analysed *all* interpretations of the AODV RFC that stem from the ambiguities revealed. It turned out that several interpretations can yield unwanted behaviour such as routing loops. We also found that implementations of AODV behave differently in crucial aspects of protocol behaviour, although they all follow the lines of the RFC. As pointed out, this is often caused by ambiguities, contradictions or unspecified behaviour in the RFC. Of course a specification "needs to be reasonably implementation independent"[82] and can leave some decisions to the software engineer; however it is our belief that any specification should be clear and unambiguous enough to guarantee the same behaviour when given to different developers. As demonstrated, this is not the case for AODV, and likely not for many other RFCs provided by the IETF.

Finding ambiguities and unexpected behaviour is not uncommon for RFCs, since the currently pre-

[82] `http://www.ietf.org/iesg/statement/pseudocode-guidelines.html`

dominant practice is an informal protocol specification via English prose. This shows that the specification of a reasonably rich protocol such as AODV cannot be described precisely and unambiguously by simple (English) text only; formal methods are indispensable for this purpose. We believe that formal specification languages and analysis techniques—offering rigorous verification and analysis techniques—are now able to capture the full syntax and semantics of reasonably rich IETF protocols. These are an indispensable augmentation to natural language, both for specifying protocols such as AODV, AODVv2 and HWMP, and for verifying their essential properties.

Our analysis of AODV also uncovered several shortcomings of the protocol, including a failure in route discovery, and the creation of non-optimal routes. In this paper, we have not only listed the shortcomings, we have proposed (small) modifications of AODV to overcome these deficiencies. All proposed variants have been carefully analysed as well, in particular with respect to loop freedom. By this we have shown how proofs based on AWN can relatively easily be adapted to protocol variants.

A further analysis of AODV will require an extension of AWN with time and probability: the former to cover aspects such as AODV's handling (deletion) of stale routing table entries and the latter to model the probability associated with lossy links. We expect that the resulting algebra will be also applicable to a wide range of other wireless protocols.

Next to this on-going work, we also aim at a complementation of AWN by model checking. Having the ability of automatically deriving a model for model checkers such as UPPAAL from an AWN specification allows the confirmation and detailed diagnostics of suspected errors in an early phase of protocol development. Surely, model checking is limited to particular topologies, but finding shortcomings in some topologies is useful to identify problematic behaviour. These shortcomings can be eliminated, even before a more thorough and general analysis using AWN.

"Time is the nurse and breeder of all good."

W. Shakespeare, The Two Gentlemen of Verona

## References


[1] *Kernel AODV* (ver. 2.2.2), NIST. http://www.antd.nist.gov/wctg/aodv_kernel/ (accessed 27 September 2013).

[2] *AODV-UU:* An Implementation of the AODV routing protocol (IETF RFC 3561). http://sourceforge.net/projects/aodvuu/ (accessed 27 September 2013).

[3] J.C.M. Baeten, J.A. Bergstra & J.W. Klop (1987): *On the Consistency of Koomen's Fair Abstraction Rule*. Theoretical Computer Science 51(1/2), pp. 129–176, doi:10.1016/0304-3975(87)90052-1.

[4] J.A. Bergstra & J.W. Klop (1986): *Algebra of Communicating Processes*. In J.W. de Bakker, M. Hazewinkel & J.K. Lenstra, editors: Mathematics and Computer Science, CWI Monograph 1, North-Holland, pp. 89–138.

[5] K. Bhargavan, C.A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky & M. Viswanathan (2002): *Verisim: Formal Analysis of Network Simulations*. IEEE Transactions on Software Engineering 28(2), pp. 129–145, doi:10.1109/32.988495.

[6] K. Bhargavan, D. Obradovic & C.A. Gunter (2002): *Formal Verification of Standards for Distance Vector Routing Protocols*. Journal of the ACM 49(4), pp. 538–576, doi:10.1145/581771.581775.

[7] J. Billington & C. Yuan (2009): *On Modelling and Analysing the Dynamic MANET On-Demand (DYMO) Routing Protocol*. In K. Jensen, J. Billington & M. Koutny, editors: Transactions on Petri Nets and Other Models of Concurrency III, Lecture Notes in Computer Science 5800, Springer, pp. 98–126, doi:10.1007/978-3-642-04856-2_5.

[8] T. Bolognesi & E. Brinksma (1987): *Introduction to the ISO Specification Language LOTOS*. Computer Networks 14, pp. 25–59, doi:10.1016/0169-7552(87)90085-7.

[9] J. Borgström, S. Huang, M. Johansson, P. Raabjerg, B. Victor, J.Å. Pohjola & J. Parrow (2011): *Broadcast Psi-calculi with an Application to Wireless Protocols*. In G. Barthe, A. Pardo & G. Schneider, editors: Software Engineering and Formal Methods (SEFM'11), Lecture Notes in Computer Science 7041, Springer, pp. 74–89, doi:10.1007/978-3-642-24690-6_7.

[10] M. Bugliesi, L. Gallina, S. Hamadou, A. Marin & S. Rossi (2013): *Behavioral Equivalences and Interference Metrics for Mobile Ad-hoc Networks*, doi:10.1016/j.peva.2013.11.003. Performance Evaluation, In Press, Corrected Proof, December 2013.

[11] A. Cerone & M. Hennessy (2013): *Modelling Probabilistic Wireless Networks*. Logical Methods in Computer Science 9(3), doi:10.2168/LMCS-9(3:26)2013.

[12] A. Cerone, M. Hennessy & M. Merro (2013): *Modelling MAC-Layer Communications in Wireless Systems*. In R. De Nicola & C. Julien, editors: Coordination Models and Languages (COORDINATION '13), Lecture Notes in Computer Science 7890, Springer, pp. 16–30, doi:10.1007/978-3-642-38493-6_2.

[13] I.D. Chakeres & E.M. Belding-Royer (2004): *AODV Routing Protocol Implementation Design*. In: Conference on Distributed Computing Systems Workshops (WWAN'04), IEEE, pp. 698–703, doi:10.1109/ICDCSW.2004.1284108.

[14] S. Chiyangwa & M. Kwiatkowska (2005): *A Timing Analysis of AODV*. In: Formal Methods for Open Object-based Distributed Systems (FMOODS'05), Lecture Notes in Computer Science 3535, Springer, pp. 306–322, doi:10.1007/11494881_20.

[15] T. Clausen & P. Jacquet (2003): *Optimized Link State Routing Protocol (OLSR)*. RFC 3626 (Experimental), Network Working Group. Available at http://www.ietf.org/rfc/rfc3626.txt.

[16] R. Cleaveland, G. Lüttgen & V. Natarajan (2001): *Priority in Process Algebra*. In J.A. Bergstra, A. Ponse & S.A. Smolka, editors: Handbook of Process Algebra, chapter 12, Elsevier, pp. 711–765, doi:10.1016/B978-044482830-9/50030-8.

[17] S. Cranen, M.R. Mousavi & M.A. Reniers (2008): *A Rule Format for Associativity*. In F. van Breugel & M. Chechik, editors: Concurrency Theory (CONCUR '08), Lecture Notes in Computer Science 5201, Springer, pp. 447–461, doi:10.1007/978-3-540-85361-9_35.

[18] S.R. Das, R. Castañeda & J. Yan (2000): *Simulation-based Performance Evaluation of Routing Protocols for Mobile Ad Hoc Networks*. Mobile Networks and Applications 5(3), pp. 179–189, doi:10.1023/A:1019108612308.

[19] R. De Nicola & F.W. Vaandrager (1995): *Three Logics for Branching Bisimulation*. Journal of the ACM 42(2), pp. 458–487, doi:10.1145/201019.201032.

[20] R. de Simone (1985): *Higher-Level Synchronising Devices in* MEIJE-*SCCS*. Theoretical Computer Science 37, pp. 245–267, doi:10.1016/0304-3975(85)90093-3.

[21] S. Edenhofer & P. Höfner (2012): *Towards a Rigorous Analysis of AODVv2 (DYMO)*. In: Rigorous Protocol Engineering (WRiPE '12), IEEE, doi:10.1109/ICNP.2012.6459942.

[22] C. Ene & T. Muntean (2001): *A Broadcast-based Calculus for Communicating Systems*. In: Parallel & Distributed Processing Symposium (IPDPS '01), IEEE Computer Society, pp. 1516–1525, doi:10.1109/IPDPS.2001.925136.

[23] K.L. Espensen, M.K. Kjeldsen & L.M. Kristensen (2008): *Modelling and Initial Validation of the DYMO Routing Protocol for Mobile Ad-Hoc Networks*. In K.M. van Hee & R. Valk, editors: Applications and Theory of Petri Nets (PETRI NETS '08), Lecture Notes in Computer Science 5062, Springer, pp. 152–170, doi:10.1007/978-3-540-68746-7_13.

[24] A. Fehnker, R.J. van Glabbeek, P. Höfner, A.K. McIver, M. Portmann & W.L. Tan (2011): *Modelling and Analysis of AODV in UPPAAL*. In: Rigorous Protocol Engineering (WRiPE' 11).

[25] A. Fehnker, R.J. van Glabbeek, P. Höfner, A.K. McIver, M. Portmann & W.L. Tan (2012): *Automated Analysis of AODV using UPPAAL*. In C. Flanagan & B. König, editors: Tools and Algorithms for the Construction and Analysis of Systems (TACAS '12), Lecture Notes in Computer Science 7214, Springer, pp. 173–187, doi:10.1007/978-3-642-28756-5_13.

[26] A. Fehnker, R.J. van Glabbeek, P. Höfner, A.K. McIver, M. Portmann & W.L. Tan (2012): *A Process Algebra for Wireless Mesh Networks*. In H. Seidl, editor: European Symposium on Programming (ESOP '12), Lecture Notes in Computer Science 7211, Springer, pp. 295–315, doi:10.1007/978-3-642-28869-2_15.

[27] A. Fehnker, P. Höfner, M. Kamali & V. Mehta (2013): *Topology-based Mobility Models for Wireless Networks*. In L. Alvisi & D. Giannakopoulou, editors: Quantitative Evaluation of Systems (QEST '13), Lecture Notes in Computer Science 8054, Springer, pp. 368–383, doi:10.1007/978-3-642-40196-1_32.

[28] D.M. Gabbay, A. Pnueli, S. Shelah & J. Stavi (1980): *On the Temporal Analysis of Fairness*. In P.W. Abrahams, R.J. Lipton & S.R. Bourne, editors: Principles of Programming Languages (POPL '80), ACM Press, pp. 163–173, doi:10.1145/567446.567462.

[29] L. Gallina, S. Hamadou, A. Marin & S. Rossi (2011): *A Probabilistic Energy-Aware Model for Mobile Ad-Hoc Networks*. In K. Al-Begain, S. Balsamo, D. Fiems & A. Marin, editors: Analytical and Stochastic Modeling Techniques and Applications (ASMTA '11), Lecture Notes in Computer Science 6751, Springer, pp. 316–330, doi:10.1007/978-3-642-21713-5_23.

[30] L. Gallina, A. Marin, S. Rossi, T. Han & M.Z. Kwiatkowska (2013): *A Process Algebraic Framework for Estimating the Energy Consumption in Ad-hoc Wireless Sensor Networks*. In: Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM '13), ACM Press, pp. 255–262, doi:10.1145/2507924.2507958.

[31] L. Gallina & S. Rossi (2013): *A Process Calculus for Energy-aware Multicast Communications of Mobile Ad Hoc Networks*. Wireless Communications and Mobile Computing 13(3), pp. 296–312, doi:10.1002/wcm.2207. An earlier version appeared as *A Calculus for Power-Aware Multicast Communications in Ad Hoc Networks*, in C.S. Calude & V. Sassone, editors: Theoretical Computer Science (TCS '10), IFIP Advances in Information and Communication Technology 323, Springer, pp. 20–31, doi:10.1007/978-3-642-15240-5_2.

[32] J.J. Garcia-Luna-Aceves (1989): *A Unified Approach to Loop-free Routing using Distance Vectors or Link States*. In: Symposium Proceedings on Communications, Architectures & Protocols (SIGCOMM '89), ACM SIGCOMM Computer Communication Review 19(4), ACM Press, pp. 212–223, doi:10.1145/75246.75268.

[33] J.J. Garcia-Luna-Aceves & H. Rangarajan (2004): *A New Framework for Loop-free On-demand Routing using Destination Sequence Numbers*. In: *Mobile Ad-hoc and Sensor Systems (MASS' 04)*, IEEE, pp. 426–435, doi:10.1109/MAHSS.2004.1392182.

[34] F. Ghassemi, W. Fokkink & A. Movaghar (2008): *Restricted Broadcast Process Theory*. In A. Cerone & S. Gruner, editors: *Software Engineering and Formal Methods (SEFM '08)*, IEEE Computer Society, pp. 345–354, doi:10.1109/SEFM.2008.25.

[35] F. Ghassemi, W. Fokkink & A. Movaghar (2010): *Equational Reasoning on Ad Hoc Networks*. In F. Arbab & M. Sirjani, editors: *Fundamentals of Software Engineering (FSEN '09)*, *Lecture Notes in Computer Science* 5961, Springer, pp. 113–128, doi:10.1007/978-3-642-11623-0_6.

[36] F. Ghassemi, W. Fokkink & A. Movaghar (2010): *Equational Reasoning on Mobile Ad Hoc Networks*. *Fundamenta Informaticae* 105(4), pp. 375–415, doi:10.3233/FI-2010-371.

[37] F. Ghassemi, W. Fokkink & A. Movaghar (2011): *Verification of Mobile Ad Hoc Networks: An Algebraic Approach*. *Theoretical Computer Science* 412(28), pp. 3262–3282, doi:10.1016/j.tcs.2011.03.017.

[38] F. Ghassemi, M. Talebi, A. Movaghar & W. Fokkink (2011): *Stochastic Restricted Broadcast Process Theory*. In N. Thomas, editor: *Computer Performance Engineering (EPEW '11)*, *Lecture Notes in Computer Science* 6977, Springer, pp. 72–86, doi:10.1007/978-3-642-24749-1_7.

[39] R.J. van Glabbeek, P. Höfner, W.L. Tan & M. Portmann (2013): *Sequence Numbers Do Not Guarantee Loop Freedom —AODV Can Yield Routing Loops—*. In: *Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM '13)*, ACM Press, pp. 91–100, doi:10.1145/2507924.2507943.

[40] J.C. Godskesen (2007): *A Calculus for Mobile Ad Hoc Networks*. In A.L. Murphy & J. Vitek, editors: *Coordination Models and Languages (COORDINATION '07)*, *Lecture Notes in Computer Science* 4467, Springer, pp. 132–150, doi:10.1007/978-3-540-72794-1_8.

[41] J.C. Godskesen (2009): *A Calculus for Mobile Ad-hoc Networks with Static Location Binding*. *Electronic Notes in Theoretical Computer Science* 242(1), pp. 161–183, doi:10.1016/j.entcs.2009.06.018.

[42] J.C. Godskesen (2010): *Observables for Mobile and Wireless Broadcasting Systems*. In D. Clarke & G.A. Agha, editors: *Coordination Models and Languages (COORDINATION '10)*, *Lecture Notes in Computer Science* 6116, Springer, pp. 1–15, doi:10.1007/978-3-642-13414-2_1.

[43] J.C. Godskesen, H. Hüttel & M. Kühnrich (2009): *Verification of Correspondence Assertions in a Calculus for Mobile Ad Hoc Networks*. *Electronic Notes in Theoretical Computer Science* 229(2), pp. 77–93, doi:10.1016/j.entcs.2009.06.030.

[44] J.C. Godskesen & S. Nanz (2009): *Mobility Models and Behavioural Equivalence for Wireless Networks*. In J. Field & V.T. Vasconcelos, editors: *Coordination Models and Languages (COORDINATION '09)*, *Lecture Notes in Computer Science* 5521, Springer, pp. 106–122, doi:10.1007/978-3-642-02053-7_6.

[45] T.G. Griffin & J. Sobrinho (2005): *Metarouting*. *SIGCOMM Computer Communication Review* 35(4), pp. 1–12, doi:10.1145/1090191.1080094.

[46] J.F. Groote & A. Ponse (1995): *The Syntax and Semantics of $\mu$CRL*. In A. Ponse, C. Verhoef & S.F.M. van Vlijmen, editors: *Algebra of Communicating Processes '94*, Workshops in Computing, Springer, pp. 26–62, doi:10.1007/978-1-4471-2120-6_2.

[47] C.A.R. Hoare (1985): *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs.

[48] P. Höfner, R.J. van Glabbeek, W.L. Tan, M. Portmann, A.K. McIver & A. Fehnker (2012): *A Rigorous Analysis of AODV and its Variants*. In: *Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM '12)*, ACM Press, pp. 203–212, doi:10.1145/2387238.2387274.

[49] P. Höfner & M. Kamali (2013): *Quantitative Analysis of AODV and its Variants on Dynamic Topologies using Statistical Model Checking*. In V. Braberman & L. Fribourg, editors: *Formal Modelling and Analysis of Timed Systems (FORMATS '13)*, *Lecture Notes in Computer Science* 8053, Springer, pp. 121–136, doi:10.1007/978-3-642-40229-6_9.

[50] P. Höfner & A. McIver (2011): *Towards an Algebra of Routing Tables*. In H. de Swart, editor: *Relational and Algebraic Methods in Computer Science (RAMiCS '11)*, *Lecture Notes in Computer Science* 6663, Springer, pp. 212–229, doi:10.1007/978-3-642-21070-9_17.

[51] P. Höfner & A. McIver (2013): *Statistical Model Checking of Wireless Mesh Routing Protocols*. In G. Brat, N. Rungta & A. Venet, editors: *NASA Formal Methods Symposium (NFM '13)*, *Lecture Notes in Computer Science* 7871, Springer, pp. 322–336, doi:10.1007/978-3-642-38088-4_22.

[52] P. Höfner & B. Möller (2012): *Dijkstra, Floyd and Warshall meet Kleene*. *Formal Aspects of Computing* 24(4-6), pp. 459–476, doi:10.1007/s00165-012-0245-4.

[53] IEEE (2011): *IEEE Standard for Information Technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications Amendment 10: Mesh Networking*. Available at http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6018236.

[54] P. Jacquet, A. Laouiti, P. Minet & L. Viennot (2002): *Performance of Multipoint Relaying in Ad Hoc Mobile Routing Protocols*. In E. Gregori, M. Conti, A.T. Campbell, G. Omidyar & M. Zukerman, editors: *Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communications (NETWORKING '02)*, Lecture Notes in Computer Science, Springer, pp. 387–398, doi:10.1007/3-540-47906-6_31.

[55] K. Jensen (1997): *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use*. Monographs in Theoretical Computer Science. An EATCS Series, Springer, doi:10.1007/978-3-642-60794-3.

[56] D. Johnson, Y. Hu & D. Maltz (2007): *The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4*. RFC 4728 (Experimental), Network Working Group (Errata Exist). Available at http://www.ietf.org/rfc/rfc4728.txt.

[57] V. Kawadia, Y. Zhang & B. Gupta (2003): *System Services for Ad-Hoc Routing: Architecture, Implementation and Experiences*. In: *Mobile Systems, Applications and Services (MobiSys '03)*, ACM Press, pp. 99–112, doi:10.1145/1066116.1189040.

[58] L. Lamport (2000): *Fairness and Hyperfairness*. *Distributed Computing* 13(4), pp. 239–245, doi:10.1007/PL00008921.

[59] I. Lanese & D. Sangiorgi (2010): *An Operational Semantics for a Calculus for Wireless Systems*. *Theoretical Computer Science* 411(19), pp. 1928–1948, doi:10.1016/j.tcs.2010.01.023.

[60] R. Lanotte & M. Merro (2011): *Semantic Analysis of Gossip Protocols for Wireless Sensor Networks*. In J.-P. Katoen & B. König, editors: *Concurrency Theory (CONCUR '11)*, *Lecture Notes in Computer Science* 6901, Springer, pp. 156–170, doi:10.1007/978-3-642-23217-6_11.

[61] N. Lynch & M. Tuttle (1989): *An Introduction to Input/Output Automata*. *CWI-Quarterly* 2(3), pp. 219–246. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[62] D. Macedonio & M. Merro (2014): *A Semantic Analysis of Key Management Protocols for Wireless Sensor Networks*. *Science of Computer Programming* 81, pp. 53–78, doi:10.1016/j.scico.2013.01.005. An earlier version, coauthored with F. Ballardin and M. Tirapelle, appears at http://arxiv.org/abs/1109.5088, 2011.

[63] D. Maltz, J. Broch & D.B. Johnson (2001): *Lessons from a Full-scale Multihop Wireless Ad hoc Network Testbed*. *IEEE Personal Communications* 8(1), pp. 8–15, doi:10.1109/98.904894.

[64] Z. Manna & A. Pnueli (1992): *The Temporal Logic of Reactive and Concurrent Systems — Specification*. Springer, doi:10.1007/978-1-4612-0931-7.

[65] P.M. Merlin & A. Segall (1979): *A Failsafe Distributed Routing Protocol*. *IEEE Transactions on Communications* 27(9), pp. 1280–1287, doi:10.1109/TCOM.1979.1094552.

[66] M. Merro (2009): *An Observational Theory for Mobile Ad Hoc Networks (full version)*. *Information and Computation* 207(2), pp. 194–208, doi:10.1016/j.ic.2007.11.010.

[67] M. Merro, F. Ballardin & E. Sibilio (2011): *A Timed Calculus for Wireless Systems*. *Theoretical Computer Science* 412(47), pp. 6585–6611, doi:10.1016/j.tcs.2011.07.016.

[68] M. Merro & E. Sibilio (2013): *A Calculus of Trustworthy Ad Hoc Networks*. *Formal Aspects of Computing* 25(5), pp. 801–832, doi:10.1007/s00165-011-0210-7.

[69] N. Mezzetti & D. Sangiorgi (2006): *Towards a Calculus For Wireless Systems*. *Electronic Notes in Theoretical Computer Science* 158, pp. 331–353, doi:10.1016/j.entcs.2006.04.017.

[70] B. Milic & M. Malek (2009): *NPART—Node Placement Algorithm for Realistic Topologies in Wireless Multihop Network Simulation*. In: Simulation Tools and Techniques (Simutools '09), ACM Press, doi:10.4108/ICST.SIMUTOOLS2009.5669.

[71] R. Milner (1989): *Communication and Concurrency*. Prentice Hall.

[72] S. Miskovic & E.W. Knightly (2010): *Routing Primitives for Wireless Mesh Networks: Design, Analysis and Experiments*. In: Conference on Information Communications (INFOCOM '10), IEEE, pp. 2793–2801, doi:10.1109/INFCOM.2010.5462111.

[73] M. Musuvathi, D.Y.W. Park, A. Chou, D.R. Engler & D.L. Dill (2002): *CMC: A Pragmatic Approach to Model Checking Real Code*. SIGOPS Operating Systems Review 36(SI), pp. 75–88, doi:10.1145/844128.844136.

[74] S. Nanz & C. Hankin (2006): *A Framework for Security Analysis of Mobile Wireless Networks*. Theoretical Computer Science 367, pp. 203–227, doi:10.1016/j.tcs.2006.08.036.

[75] A. Neumann, M. Aichele, C. Lindner & S. Wunderlich (2008): *Better Approach To Mobile Ad-hoc Networking (B.A.T.M.A.N.)*. Internet-Draft (Experimental), Network Working Group. Available at http://tools.ietf.org/html/draft-openmesh-b-a-t-m-a-n-00.

[76] *The Network Simulator ns-2*. http://nsnam.isi.edu/nsnam/index.php/Main_Page (accessed 20 December 2013).

[77] E.-R. Olderog & C.A.R. Hoare (1986): *Specification-oriented Semantics for Communicating Processes*. Acta Informatica 23(1), pp. 9–66, doi:10.1007/BF00268075.

[78] C.E. Perkins, E.M. Belding-Royer & I.D. Chakeres (2003): *Ad hoc On-Demand Distance Vector (AODV) Routing*. Internet Draft, Mobile Ad hoc Networks Working Group. Available at http://tools.ietf.org/id/draft-perkins-manet-aodvbis-00.txt.

[79] C.E. Perkins, E.M. Belding-Royer & S. Das (2003): *Ad hoc On-Demand Distance Vector (AODV) Routing*. RFC 3561 (Experimental), Network Working Group. Available at http://www.ietf.org/rfc/rfc3561.txt.

[80] C.E. Perkins, E.M. Belding-Royer, S.R. Das & M.K. Marina (2001): *Performance Comparison of Two On-demand Routing Protocols for Ad hoc Networks*. IEEE Personal Communications 8(1), pp. 16–28, doi:10.1109/98.904895.

[81] C.E. Perkins, S. Ratliff & J. Dowdell (2013): *Dynamic MANET On-demand (AODVv2) Routing*. Internet Draft (Standards Track), Mobile Ad hoc Networks Working Group. Available at http://tools.ietf.org/html/draft-ietf-manet-aodvv2-02.

[82] C.E. Perkins & E.M. Royer (1999): *Ad-hoc On-Demand Distance Vector Routing*. In: Mobile Computing Systems and Applications (WMCSA '99), IEEE, pp. 90–100, doi:10.1109/MCSA.1999.749281.

[83] A.A. Pirzada, M. Portmann & J. Indulska (2008): *Performance Analysis of Multi-radio AODV in Hybrid Wireless Mesh Networks*. Computer Communications 31(5), pp. 885–895, doi:10.1016/j.comcom.2007.12.012.

[84] A.A. Pirzada, M. Portmann, R. Wishart & J. Indulska (2009): *SafeMesh: A Wireless Mesh Network Routing Protocol for Incident Area Communications*. Pervasive and Mobile Computing 5(2), pp. 201–221, doi:10.1016/j.pmcj.2008.11.005.

[85] G.D. Plotkin (2004): *A Structural Approach to Operational Semantics*. Journal of Logic and Algebraic Programming 60–61, pp. 17–139, doi:10.1016/j.jlap.2004.05.001. Originally appeared in 1981.

[86] A. Pnueli (1977): *The Temporal Logic of Programs*. In: Foundations of Computer Science (FOCS '77), IEEE, pp. 46–57, doi:10.1109/SFCS.1977.32.

[87] K.V.S. Prasad (1991): *A Calculus of Broadcasting Systems*. In S. Abramsky & T.S.E. Maibaum, editors: Theory and Practice of Software Development (TAPSOFT '91), Lecture Notes in Computer Science 493, Springer, pp. 338–358, doi:10.1007/3-540-53982-4_19.

[88] K.V.S. Prasad (1995): *A Calculus of Broadcasting Systems*. Science of Computer Programming 25(2-3), pp. 285–327, doi:10.1016/0167-6423(95)00017-8.

[89] K. Ramachandran, M.M. Buddhikot, G. Chandranmenon, S. Miller, E.M. Belding-Royer & K. Almeroth (2005): *On the Design and Implementation of Infrastructure Mesh Networks*. In: *IEEE Workshop on Wireless Mesh Networks (WiMesh'05)*, IEEE.

[90] H. Rangarajan & J.J. Garcia-Luna-Aceves (2005): *Making On-demand Routing Protocols Based on Destination Sequence Numbers Robust*. In: *Communications (ICC '05)*, 5, pp. 3068–3072, doi:10.1109/ICC.2005.1494958.

[91] M. Saksena, O. Wibling & B. Jonsson (2008): *Graph Grammar Modeling and Verification of Ad Hoc Routing Protocols*. In C.R. Ramakrishnan & J. Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, *Lecture Notes in Computer Science* 4963, Springer, pp. 18–32, doi:10.1007/978-3-540-78800-3_3.

[92] SCALABLE Network Technologies: *QualNet Communications Simulation Platform*. http://web.scalable-networks.com/content/qualnet (accessed 20 December 2013).

[93] K. Sen, M. Viswanathan & G.A. Agha (2005): *VESTA: A Statistical Model-checker and Analyzer for Probabilistic Systems*. In: *Quantitative Evaluation of Systems (QEST '05)*, IEEE, pp. 251–252, doi:10.1109/QEST.2005.42.

[94] A. Singh, C.R. Ramakrishnan & S.A. Smolka (2010): *A process calculus for Mobile Ad Hoc Networks*. *Science of Computer Programming* 75, pp. 440–469, doi:10.1016/j.scico.2009.07.008.

[95] J. Sobrinho (2002): *Algebra and Algorithms for QoS Path Computation and Hop-by-hop Routing in the Internet*. *IEEE/ACM Transactions on Networking* 10(4), pp. 541–550, doi:10.1109/TNET.2002.801397.

[96] J. Sobrinho (2003): *Network Routing with Path Vector Protocols: Theory and Applications*. In: *Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '03)*, ACM Press, pp. 49–60, doi:10.1145/863955.863963.

[97] L. Song & J.C. Godskesen (2010): *Probabilistic Mobility Models for Mobile and Wireless Networks*. In C.S. Calude & V. Sassone, editors: *Theoretical Computer Science (TCS '10)*, *IFIP Advances in Information and Communication Technology* 323, Springer, pp. 86–100, doi:10.1007/978-3-642-15240-5_7.

[98] L. Song & J.C. Godskesen (2012): *Broadcast Abstraction in a Stochastic Calculus for Mobile Networks*. In J.C.M. Baeten, T. Ball & F. de Boer, editors: *Theoretical Computer Science (TCS '12)*, *Lecture Notes in Computer Science* 7604, Springer, pp. 342–356, doi:10.1007/978-3-642-33475-7_24.

[99] A.P. Subramanian, M.M. Buddhikot & S. Miller (2006): *Interference Aware Routing in Multi-radio Wireless Mesh Networks*. In: *IEEE Workshop on Wireless Mesh Networks (WiMesh '06)*, IEEE.

[100] C.F. Tschudin (2004): *Lightweight Underlay Network Ad hoc Routing (LUNAR) Protocol*. Internet Draft (Expired), Mobile Ad Hoc Networking Working Group. Available at http://user.it.uu.se/~rmg/pub/draft-tschudin-manet-lunar-00.txt.

[101] C.F. Tschudin, R. Gold, O. Rensfelt & O. Wibling (2004): *LUNAR: A Lightweight Underlay Network Ad-Hoc Routing Protocol and Implementation*. In Y. Koucheryavy, J. Harju & A. Koucheryavy, editors: *Next Generation Teletraffic and Wired/Wireless Advanced Networking (NEW2AN '04)*.

[102] M. Wang & Y. Lu (2012): *A Timed Calculus for Mobile Ad Hoc Networks*. In P.C. Ölveczky & C. Artho, editors: *Formal Techniques for Safety-Critical Systems (FTSCS '12)*, *Electronic Proceedings in Theoretical Computer Science* 105, pp. 118–134, doi:10.4204/EPTCS.105.9.

[103] O. Wibling, J. Parrow & A.N. Pears (2004): *Automatized Verification of Ad Hoc Routing Protocols*. In D. de Frutos-Escrig & M. Núñez, editors: *Formal Techniques for Networked and Distributed Systems (FORTE '04)*, Lecture Notes in Computer Science, Springer, pp. 343–358, doi:10.1007/978-3-540-30232-2_22.

[104] C. Xiong, T. Murata & J. Tsai (2002): *Modeling and Simulation of Routing Protocol for Mobile Ad Hoc Networks Using Colored Petri Nets*. In: *Formal Methods in Software Engineering and Defence Systems (CRPIT '02)*, 12, Australian Computer Society, Inc., pp. 145–153.

[105] H.L.S. Younes (2004): *Verification and Planning for Stochastic Processes with Asynchronous Events*. Ph.D. thesis, Carnegie Mellon University.

[106] M. Zhou, H. Yang, X. Zhang & J. Wang (2009): *The Proof of AODV Loop Freedom*. In: *Wireless Communications & Signal Processing (WCSP '09)*, IEEE, doi:10.1109/WCSP.2009.5371479.

## List of Processes

## List of Figures

# List of Tables

# Index