A Rigorous Analysis of AODV and its Variants

Peter Höfner
NICTA, Australia
University of New South Wales, Australia
Peter.Hoefner@nicta.com.au

Wee Lum Tan
NICTA, Australia
University of Queensland, Australia
WeeLum.Tan@nicta.com.au

Annabelle McIver
Macquarie University, Australia
NICTA, Australia
annabelle.mciver@mq.edu.au

Rob van Glabbeek
NICTA, Australia
University of New South Wales, Australia
rvg@cs.stanford.edu

Marius Portmann
NICTA, Australia
University of Queensland, Australia
marius@itee.uq.edu.au

Ansgar Fehnker
NICTA, Australia
University of New South Wales, Australia
Ansgar.Fehnker@nicta.com.au

ABSTRACT

In this paper we present a rigorous analysis of the Ad hoc On-Demand Distance Vector (AODV) routing protocol using a formal specification in AWN (Algebra for Wireless Networks), a process algebra which has been specifically tailored for the modelling of Mobile Ad Hoc Networks and Wireless Mesh Network protocols. Our formalisation models the exact details of the core functionality of AODV, such as route discovery, route maintenance and error handling. We demonstrate how AWN can be used to reason about critical protocol correctness properties by providing a detailed proof of loop freedom. 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. A key contribution of this paper is the demonstration of how the reasoning and proofs can relatively easily be adapted to protocol variants.

Categories and Subject Descriptors

C.2.2 [**Network Protocols**]: Routing protocols; Protocol verification; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Invariants

General Terms

Reliability; Theory; Verification

Keywords

AODV; loop freedom; process algebra; routing protocols; wireless mesh networks

MSWiM'12, October 21–25, 2012, Paphos, Cyprus.

1. INTRODUCTION

Routing protocols are crucial to the dissemination of data packets between nodes in Wireless Mesh Networks (WMNs) and Mobile Ad Hoc Networks (MANETs). One of the most popular routing protocols that is widely used in WMNs and MANETs is the Ad hoc On-Demand Distance Vector (AODV) routing protocol [13]. AODV 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 upcoming IEEE 802.11s wireless mesh network standard [10]. The details of the AODV protocol are laid out in the RFC 3561 [13]. 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 [5, 10, 16, 18, 15] are simulation and test-bed experiments. While these are important and valid methods for protocol evaluation, in particular for quantitative performance evaluation, there are 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 [3]. This problem is illustrated by recent discoveries of limitations in AODV-like protocols that have been under intense scrutiny over many years [12]. 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.

This paper is based on a complete and accurate formal specification of the core functionality of the AODV routing protocol using the specification language AWN (Algebra of Wireless Networks) [7]. AWN 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 ambiguities. A key contribution is to demonstrate how AWN can be used to support reasoning about protocol behaviour and to provide a rigorous proof of key protocol properties, using the example of loop freedom. In contrast to what can be achieved, e.g., by model checking, our proofs apply to all conceivable dynamic network topologies.

We analyse different readings of the AODV RFC, and show which interpretations do satisfy the loop freedom criterion, and which do not. We also discuss two 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 these 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.

The remainder of this paper is organised as follows. We briefly describe AWN in Section 2, and use it to formally specify AODV in Section 3. We discuss one of several ambiguities and contradictions in the AODV RFC, and propose potential resolutions in Section 4. We then summarise the key points of a detailed proof of loop freedom of AODV in Section 5, and demonstrate how the reasoning and proof can relatively easily be adapted to variants of the AODV protocol in Section 6. We discuss related work in Section 7, and summarise our work in Section 8.

2. AWN—AN ALGEBRA FOR WIRELESS NETWORKS

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 allow formal reasoning. For the specification of and for formal reasoning about AODV, we use AWN [7, 8], a process algebra specifically tailored for WMNs. AWN allows us to write a protocol specification in a simple language, which makes it easy to read and to use. Its key operators are *conditional unicast*—allowing error handling in response to failed communications while abstracting from link layer implementations of the communication handling—and *local broadcast*—allowing a node to send messages to all its immediate neighbours as implemented by the physical and data link layer.

In this section we only give an overview of the main operations (Table 1) and illustrate the use of AWN with a simple example. Additional explanations and a full description can be found in [7, 8].

The example considers a network of two nodes on which the same process is running. One node broadcasts an integer value. A received message will be delivered to the application layer if its value is 1. Otherwise the node decrements the value and broadcasts the new value. The behaviour of each node can be modelled by:

$\mathtt{X}(\mathtt{n}) \stackrel{def}{=} \mathbf{broadcast}(\mathtt{n}).\mathtt{Y}(\,)$

$\mathtt{Y}(\,) \stackrel{def}{=} \mathbf{receive}(\mathtt{m}).([\mathtt{m}{=}1]\ \mathbf{deliver}(\mathtt{m}).\mathtt{Y}(\,) + [\mathtt{m}{\neq}1]\ \mathtt{X}(\mathtt{m}{-}1))$

If a node is in a state $\mathtt{X}(\mathtt{n})$ it will broadcast $\mathtt{n}$ and continue

$X(exp_1, \ldots, exp_n)$	process name with arguments
$P + Q$	choice between processes P and Q
$[\varphi]P$	conditional process; execute P only if condition φ holds
$[\![\mathtt{var} := exp]\!]P$	assignment followed by process P
$\mathbf{broadcast}(ms).P$	broadcast message ms followed by P
$\mathbf{groupcast}(dests, ms).P$	iterative unicast to all destinations $dests$ (if broadcast is inappropriate)
$\mathbf{unicast}(dest, ms).P \blacktriangleright Q$	unicast ms to $dest$; if successful proceed with P; otherwise with Q
$\mathbf{deliver}(data).P$	deliver data to application layer
$\mathbf{receive}(\mathtt{msg}).P$	receive a message
$P \| Q$	parallel composition of nodes

Table 1: Process expressions

in state $\mathtt{Y}(\,)$. If a node is in state $\mathtt{Y}(\,)$, and it receives $\mathtt{m}$, it has two ways to continue. Process $[\mathtt{m}{=}1]\,\mathbf{deliver}(\mathtt{m}).\mathtt{Y}(\,)$ is enabled if $\mathtt{m}{=}1$. In that case $\mathtt{m}$ will be delivered to the application layer, and the process returns to $\mathtt{Y}(\,)$. Alternatively, if $\mathtt{m} \neq 1$, the process continues as $\mathtt{X}(\mathtt{m}{-}1)$. Note that calls to processes use expressions as parameters, in this case $\mathtt{m}{-}1$.

Assume that the nodes A and B are within communication range of each other; node A in state $\mathtt{X}(2)$, and node B in $\mathtt{Y}(\,)$. Then, node A broadcasts 2 and continues as $\mathtt{Y}(\,)$. Node B receives 2, and continues as $\mathtt{X}(1)$. Next B broadcasts 1, and continues as $\mathtt{Y}(\,)$, while node A receives 1, and, since the condition $\mathtt{m}{=}1$ is satisfied, **deliver**s 1 and continues as $\mathtt{Y}(\,)$. This gives rise to transitions from one state to the other:

$$\mathtt{X}(2) \,\|\, \mathtt{Y}(\,) \xrightarrow{A:\mathbf{broadcast}(2)} \mathtt{Y}(\,) \,\|\, \mathtt{X}(1) \xrightarrow{B:\mathbf{broadcast}(1)} \xrightarrow{A:\mathbf{deliver}(1)} \mathtt{Y}(\,) \,\|\, \mathtt{Y}(\,)$$

In state $\mathtt{Y}(\,) \,\|\, \mathtt{Y}(\,)$ no further activity is possible; the network has reached a *deadlock*.

3. A FORMAL SPECIFICATION OF AODV

AODV is a reactive protocol, which means that routes are only established on demand. If a node S wants to send a data packet to node D, but currently does not know a route, it buffers the packet and initiates a route discovery process by broadcasting a route request (RREQ) message in the network. An intermediate node A that receives this message creates a routing table entry for a route towards S, referred to as a *reverse route*, and re-broadcasts the RREQ. This is repeated until the RREQ reaches the destination D, or alternatively a node with a route to D. In both cases, the node replies by unicasting a route reply (RREP) back to the source S, via the previously established reverse route. When forwarding RREP messages, a node creates a routing table entry for node D, called the *forward route*. When the RREP reaches S, a route between S and D is established and data packets can start to flow. In the event of link and route breaks, AODV uses route error messages (RERR) to notify the affected nodes. AODV uses sequence numbers to indicate the freshness of routes and to avoid routing loops. Full details are given in [13].

3.1 Modelling AODV

We present a model of AODV using AWN. The formalisation is a faithful representation of the core functionality of AODV as defined in [13]. 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, since AWN currently does not

Process 1 The basic routine

```
AODV(ip,sn,rt,rreqs,store) =def
 1.  receive(msg) .
 2.  /* depending on the message, different processes are called */
 3.  (
 4.    [msg = newpkt(data,dip)] /* new DATA packet */
 5.      PKT(data,dip,ip , 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,oip,osn,sip)] /*RREQ*/
 9.      /* update the route to sip in rt */
10.      ⟦rt := update(rt,(sip, 0, val, 1, sip, ∅))⟧
11.      RREQ(hops,rreqid,dip,dsn,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, 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, val, 1, sip, ∅))⟧
19.      RERR(dests,sip , ip,rt,sn,rreqs,store)
20.  )
21. +[Let dip ∈ qD(store) ∩ vD(rt)] /* send a queued data packet */
22.  ...
```

support time. In concrete terms, this means that the AODV timing parameters `ACTIVE_ROUTE_TIMEOUT`, `DELETE_PERIOD` and `PATH_DISCOVERY_TIME` are set to infinity.

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.

Our AODV model consists of the following six processes:

- `AODV`, the main process, reads a message from the message queue (Line 1 of Process 1) and calls the appropriate process `PKT`, `RREQ`, `RREP`, or `RERR` to handle it (Lines 4–19). The process also handles the forwarding of any queued data packets if a valid route to their destination is known (Lines 21 ff.).
- `PKT` deals with received data packets, including forwarding if a route exists, and sending an error message if the route is broken. If the data packet originates at the local node and no route to the destination exists, the process buffers the data packet and initiates a new route discovery process.
- `RREQ` deals with received RREQ messages, and will be discussed in detail below.
- `RREP` deals with received RREP messages, including the updating of routing tables and handling of errors.
- `RERR` models the processing of AODV error messages.
- `QMSG` describes the general handling of incoming AODV messages: whenever a message is received, it is first stored in a FIFO queue. As soon as the corresponding node is able to handle a message it retrieves the oldest message from the queue and handles it.

Each node in an AODV network maintains a *routing table* to keep track of the node's routing information collected so far. A routing table consists of sets of entries of the form $(dip, dsn, flag, hops, nhip, pre)$, with dip being the node identifier (typically IP address) of the ultimate destination node, and dsn the destination sequence number, which represents the "freshness" of this routing table entry. The $flag$ parameter indicates whether an entry is valid or invalid, and $hops$ represents the distance to the destination node dip in number of hops. $nhip$ identifies the next hop node along the route to node dip, and pre is the set of *precursors*—nodes that "rely" on this routing table entry for their own routes. Following [13], a routing table entry would also contain a *sequence-number-status flag*. In the present paper we abstract from this flag, since (a) the main results are independent of the existence of the flag, and (b) none of the common implementations (AODV-UU [2], Kernel-AODV [1], AODV-UIUC [11], AODV-UCSB [6], AODV-ns2[1]) maintains this flag.[2] Hence the specification here follows the implementations available.

In a routing table rt there is at most one entry for each destination dip; $\mathtt{sqn}(rt, dip)$ denotes the sequence number of that entry and $\mathtt{flag}(rt, dip)$, $\mathtt{dhops}(rt, dip)$ and $\mathtt{nhop}(rt, dip)$ its validity, hop count and next hop. Furthermore the sets $\mathtt{kD}(rt)$ and $\mathtt{vD}(rt)$ of destinations contain all entries of rt for which there is an (arbitrary) entry or a valid entry, resp. The function `update` updates a routing table rt with an entry r, which is one of the major activities of `AODV`:

$$\mathtt{update}(rt,r) := \begin{cases} rt \cup \{r\} & \text{if } \pi_1(r) \notin \mathtt{kD}(rt) & \textit{//r is new} \\ nrt \cup \{nr\} & \text{if } \mathtt{sqn}(rt,\pi_1(r)) < \pi_2(r) & \textit{//fresher} \\ nrt \cup \{nr\} & \text{if } \mathtt{sqn}(rt,\pi_1(r)) = \pi_2(r) & \\ & \quad \wedge\, \mathtt{dhops}(rt,\pi_1(r)) > \pi_4(r) & \textit{//shorter} \\ nrt \cup \{nr\} & \text{if } \mathtt{sqn}(rt,\pi_1(r)) = \pi_2(r) & \textit{//replaces} \\ & \quad \wedge\, \mathtt{flag}(rt,\pi_1(r)) = \mathtt{inv} & \textit{invalid} \\ nrt \cup \{nr'\} & \text{if } \pi_2(r) = 0 & \textit{//unk. sqn} \\ nrt \cup \{ns\} & \text{otherwise} , & \end{cases}$$

where the projections π_1, π_2 and π_4 select the respective component from an entry, namely the destination, the destination sequence number and the hop count. s is the current entry in rt for destination $\pi_1(r)$ (if it exists); and $nrt := rt - s$ removes s from rt. The entry nr is identical to r except that the precursors from the corresponding routing table entry are added and ns is generated from s by adding the precursors of r. The entry nr' is identical to nr except that the sequence number is replaced by the one from the routing table (route s).

If a route is not valid any longer, instead of deleting it, AODV sets its validity flag to invalid. This way, the stored information on the route, such as the sequence number and hop count, remains accessible. We model route invalidation by a function `invalidate` whose arguments are a routing table and a set $dests$ of pairs (rip, rsn) of a destination rip to be invalidated, and the sequence number of the invalidated routing table entry. Normally, rsn is obtained by incrementing the last known sequence number of the route.

In our formalisation, a route request message has the form $\mathtt{rreq}(hops, rreqid, dip, dsn, oip, osn, sip)$, where $hops$ is the number of hops the RREQ has already travelled from its origin oip, and $rreqid$ (in combination with oip) is a unique identifier of the message. dip is the destination node identifier (IP address) of the route request and dsn the last known corresponding sequence number. The parameter oip is the

[1] `www.auto-nomos.de/ns2doku/aodv_8cc-source.html`

[2] Kernel-AODV implements the flag, but does not use it.

address of the originator of the route request and *osn* is its sequence number. Finally, *sip* represents the sender IP address, i.e., the address of the intermediate node from which the request was received. Any node forwarding such a message updates *sip* with its own address, increments *hops*, and retains all other parameters. A reply to such a message has the form `rrep`(*hops*,*dip*,*dsn*,*oip*,*sip*), where *dip* and *oip* are copied from the corresponding RREQ message and *hops* is the distance from *dip* to *sip*. The processes `RREQ` and `RREP` that handle incoming RREQ and RREP messages maintain variables `dip`, `oip`, etc. to store the values of the parameters of these messages, as summarised below.

Variables	Used for
`ip`	address of current node
`dip`	destination address
`oip`	originator of a route request or data packet
`rip`	destination of invalid route
`sip`	sender of AODV control message
`nhip`	next hop towards some destination

The process `AODV`, specified by Process 1, deals with the message handling of the node. It stores its own address in the variable `ip`, its own sequence number in `sn`, manages its routing table `rt`, records all route requests seen so far in `rreqs` and maintains in `store` data packets to be sent. Initially, `rt`, `rreqs` and `store` are set to empty, and `sn` to 1.

3.2 Route Request Handling

In this paper, we discuss only the model of the RREQ process; see [8] for a complete model of all AODV processes.[3]

A route discovery in AODV is initiated by a source node broadcasting a RREQ message. Process 2 shows our process algebra specification of the handling of a RREQ message received by a node *ip*.

If the RREQ with the same *oip* and *rreqid* has been seen previously by the node, it is ignored, and we go back to the main AODV process (Lines 1–2). If the RREQ is new (Line 3), we update the routing table by adding a "reverse route" entry to *oip*, the originator of the RREQ, via node *sip*, with distance *hops*+1 (Line 5). If there already is a route to *oip* in the node's routing table *rt*, it is only updated with the new route, if the new route is "better", i.e., fresher and/or shorter and/or replacing an invalid route (cf. Section 3.1). The process also adds the message to the list of known RREQs (Line 7).

Lines 9–20 deal with the case where the node receiving the RREQ is the intended destination, i.e., *dip*=*ip* (Line 9). In this case, a RREP message needs to be sent to the originating node *oip*. According to the AODV RFC, the node's sequence number is set to the maximum of the node's current sequence number and the destination sequence number (*dsn*) in the RREQ message (Line 10).

The RREP message is initialised as follows: hop count (*hops*) is set to 0, the destination (*dip*) and originator (*oip*) are copied from the corresponding RREQ message and the destination's sequence number is the node's sequence number `sn`. Of course, the sender's IP address (*sip*) is set to the node's ip (Line 12). The RREP message is unicast to the next hop along the reverse route back to the originator of the corresponding RREQ message, and if this is successful, the process goes back to the AODV process (Line 13).

[3]There, the sequence-number-status flag is modelled as well.

Process 2 RREQ handling

```
RREQ(hops,rreqid,dip,dsn,oip,osn,sip,ip,sn,rt,rreqs,store) ≝
 1. [(oip,rreqid) ∈ rreqs] /* the RREQ has been handled before */
 2.   AODV(ip,sn,rt,rreqs,store) /* silently ignore RREQ */
 3. +[(oip,rreqid) ∉ rreqs] /* the RREQ is new to this node */
 4.   /* update the route to oip in rt */
 5.   ⟦rt := update(rt,(oip,osn,val,hops + 1,sip,∅))⟧
 6.   /* update rreqs by adding (oip,rreqid) */
 7.   ⟦rreqs := rreqs ∪ {(oip,rreqid)}⟧
 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 */
12.       unicast(nhop(rt,oip),rrep(0,dip,sn,oip,ip)).
13.         AODV(ip,sn,rt,rreqs,store)
14.       ▶ /* if transmission fails, a RERR is generated */
15.         ⟦dests := {(rip,inc(sqn(rt,rip)))|rip ∈ vD(rt) ∧
                      nhop(rt,rip) = nhop(rt,oip)}⟧
16.         ⟦rt := invalidate(rt,dests)⟧
17.         ⟦pre := ⋃{precs(rt,rip) | (rip,*) ∈ dests}⟧
18.         ⟦dests := {(rip,rsn) | (rip,rsn) ∈ dests ∧
                      precs(rt,rip) ≠ ∅}⟧
19.         groupcast(pre,rerr(dests,ip)).
20.         AODV(ip,sn,rt,rreqs,store)
21.     +[dip ≠ ip] /* this node is not the destination node */
22.       (
23.         /* valid route to dip that is fresh enough */
24.         [dip ∈ vD(rt) ∧ dsn ≤ sqn(rt,dip) ∧ sqn(rt,dip) ≠ 0]
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 */
29.           unicast(nhop(rt,oip),
                      rrep(dhops(rt,dip),dip,sqn(rt,dip),oip,ip)).
30.             AODV(ip,sn,rt,rreqs,store)
31.           ▶ /* if transmission fails, a RERR is generated */
32.             ⟦dests := {(rip,inc(sqn(rt,rip))) | rip ∈ vD(rt) ∧
                          nhop(rt,rip) = nhop(rt,oip)}⟧
33.             ⟦rt := invalidate(rt,dests)⟧
34.             ⟦pre := ⋃{precs(rt,rip) | (rip,*) ∈ dests}⟧
35.             ⟦dests := {(rip,rsn) | (rip,rsn) ∈ dests ∧
                          precs(rt,rip) ≠ ∅}⟧
36.             groupcast(pre,rerr(dests,ip)).
37.             AODV(ip,sn,rt,rreqs,store)
38.         /* no valid route that is fresh enough */
39.         +[dip ∉ vD(rt) ∨ sqn(rt,dip) < dsn ∨ sqn(rt,dip) = 0]
40.           /* no further update of rt */
41.           broadcast(rreq(hops + 1,rreqid,dip,
                          max(sqn(rt,dip),dsn),oip,osn,ip)).
42.           AODV(ip,sn,rt,rreqs,store)
43.       )
44.   )
```

If the unicast of the RREP fails, we proceed with Lines 14–20, in which a route error (RERR) message is generated and sent. This conditional unicast is implemented in our model with the AWN construct **unicast**(*dest*,*ms*).*P* ▶ *Q* (Lines 12ff.) We assume that, as is the case for relevant wireless technologies such as IEEE 802.11, unicast messages are acknowledged, and we therefore can determine whether the transmission was unsuccessful and the link to the next node towards *oip* is broken. In this case, the node sends a RERR message to all nodes that rely on the broken link for one of their routes. For this, we first determine which destination nodes are affected by the broken link, i.e., the nodes that have this unreachable node listed as a next hop in the routing table (Line 15). Here, the operator `inc` increments the sequence numbers of those entries. Then, we invalidate any affected routing table entries (Line 16), and determine the list of *precursors*, which are the neighbouring nodes that have a route to one of the affected destination nodes via the broken link (Line 17). Finally, using the AWN **groupcast** primitive, a RERR message is sent via unicast to all these precursors (Line 19), listing only those invalidated

destinations with a non-empty set of precursors (Line 18).

Lines 21–42 deal with the case where the node receiving the RREQ is not the destination, i.e., $dip \neq ip$ (Line 21). The node can respond to the RREQ with a corresponding RREP on behalf of the destination node *dip*, if its route to *dip* is "fresh enough" (Line 24). This means that (a) the node has a valid route to *dip*, (b) the destination sequence number in the node's current routing table entry ($\texttt{sqn}(rt,dip)$) is greater than or equal to the requested sequence number to *dip* in the RREQ message, and (c) the sequence number is valid, i.e., it is not unknown ($\texttt{sqn}(rt,dip)\neq 0$). If these three conditions are met (Line 24), the node generates a RREP message and unicasts it back to the originator node *oip* via the reverse route. 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 ($\texttt{dhops}(rt, dip)$) in the corresponding field of the new reply (Line 29). As usual, the unicast might fail, which causes the same error handling (Lines 32–35). Just before unicasting the RREP message, the intermediate node updates the forward routing table entry to *dip* by placing the last hop node (*sip*) into the precursor list for that entry (Line 26). Likewise, it updates the reverse routing table entry to *oip* by placing the first hop $\texttt{nhop}(rt, dip)$ towards *dip* in the precursor list for that entry (Line 27).

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 41. 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 destination sequence number in the RREQ packet and the the current sequence number for *dip* in the routing table. In case *dip* is an unknown destination, $\texttt{sqn}(rt, dip)$ returns the unknown sequence number 0.

4. AMBIGUITIES IN THE RFC

The formal specification of AODV, outlined above and given in full detail in [8], closely follows the RFC 3561 [13], the official specification of the protocol. However, the RFC contains several ambiguities and contradictions; an inventory is presented in [8], and for each ambiguity or contradiction a number of ways to resolve them is listed. An *interpretation* of the RFC is given by the allocation of a resolution to each of the ambiguities and contradictions. Each reading, implementation, or formal analysis of AODV must pertain to one of its interpretations. The formal specification of AODV in [8] constitutes one interpretation; the inventory of ambiguities and contradictions is formalised 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.

A crucial contradiction in the RFC concerns the question of what would happen if a node has a valid routing table entry for a destination D, with destination sequence number n, and an error message is received from the next hop towards D, saying that the route to D is broken, and stating for this route a destination sequence number m, which may be smaller than n. Section 6.11 of the RFC unambiguously states that in such a case the node updates its routing table entry to D by marking the route as invalid, and *copying* the destination sequence number from the incoming route error message. However, Section 6.1 of the RFC states that if $m < n$, any information related to D in the AODV message must be discarded.

One can show [8] that in case no node will ever store a routing table entry to itself (a self-entry), the above situation will never occur. However, the RFC does not explicitly exclude self-entries[4], and they can in fact occur [8] in response to the standard handling of RREP messages.

The following ways to resolve this contradiction have been listed in [8]:

(a) 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.[5]

(b) 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 $m \geq n$.

(c) Always invalidate the routing table entry, but update the destination sequence number to $\max(m, n)$.

(d) Always invalidate the routing table entry, but update the destination sequence number to $\max(m, n+1)$.

(e) Invalidate the routing table entry and update the destination sequence number to $\max(m, n+1)$ only if $m \geq n$.[6]

(f) Invalidate the routing table entry only if $m > n$.[7]

(g) Forbid self-entries; if an incoming RREP message would create a self-entry, discard that message.

(h) Forbid self-entries; if an incoming RREP message would create a self-entry, forward that message without updating the node's routing table.

It should be noted that only resolutions (a) and (b) are compliant with the RFC. However, in [9] we have shown that any interpretation based on resolutions (a) or (b) gives rise to routing loops, so in order to arrive at a loop-free version of AODV, one has to deviate from the RFC. Here, as in [8], we do so by choosing resolution (f).

The above is only one of many ambiguities; another one is presented in Section 6.1.

5. LOOP FREEDOM

We now formalise loop freedom and sketch a proof that our detailed specification of AODV cannot create routing loops. We also show how such a formal proof can form a baseline for evaluating variants of AODV—some of them will be loop free, others can yield loops.

First we formalise what it means for the routing tables established by AODV (our specification) to be free of loops. Let **IP** be the set of network nodes and $dip \in \mathbf{IP}$ a particular destination; let N be a state of the network, encompassing the current values of all variables maintained by all nodes.

[4]The Kernel-AODV, AODV-UIUC, AODV-UCSB and AODV-ns2 implementations allow self-entries to occur.

[5]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.

[6]The case $\max(m, n)$ if $m \geq n$ need no separate consideration, since it is equivalent to (b).

[7]Here, it does not matter whether we copy, take $\max(m, n)$ or $\max(m, n+1)$; they are all equivalent.

The *routing graph* $\mathcal{R}_N(dip)$ for destination dip in state N is the directed graph $(\mathbf{IP}, E)$ with set of vertices $\mathbf{IP}$ and set of edges $E \subseteq \mathbf{IP}\times\mathbf{IP}$ consisting of the pairs (ip, ip') such that $ip \neq dip$ and $(dip, *, \mathtt{val}, *, ip', *)$ occurs in the routing table of ip in state N. Thus, there is an edge (ip, ip') if node ip is not the destination dip, but has a valid entry for dip, and ip' is the next hop according to that entry. Loops in directed graphs are defined to be paths following edges which return to a vertex.

A (network) state N is *loop free* if the routing graphs $\mathcal{R}_N(dip)$ are loop free for all $dip \in \mathbf{IP}$. The specification of AODV is *loop free* iff all reachable states are loop free.

Let us now turn towards a proof of loop freedom for AODV. It relies on a number of *invariants*—statements that hold for all reachable states of our model. An invariant is usually verified by showing that it holds for all possible initial states, and that, for any transition $N \xrightarrow{\ell} N'$ derived by our operational semantics [8], if it holds for state N then it also holds for state N', reached after performing some action ℓ. These transitions can be traced back to the line numbers in our process declarations `AODV`, `PKT`, `RREQ`, `RREP`, `RERR` and `QMSG`.

A proof of AODV's loop freedom using invariants has first been proposed in [4].[8] The main invariant of [4] states that *if node ip has a routing table entry for destination dip with next hop nhip, then also node nhip has a routing table entry for dip, and the latter has a larger destination sequence number, or an equal one with a strictly smaller hop count.*

This invariant is claimed to hold regardless whether the routing table entries for dip at ip and $nhip$ are marked as valid or invalid. Nevertheless, the following example shows that it does not hold for the current version of AODV.

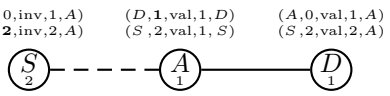


Figure 1: Sequence numbers can go down[9]

The network state depicted in Figure 1 can occur when, after a standard RREQ-RREP cycle, a link break is detected. We assume a simple linear topology of 3 nodes. Below the node names, the circles list the nodes' own sequence numbers, which we assume to be 1 at the initial state. The example starts with empty routing tables and node S searching for a route to node D. Before broadcasting a RREQ message, node S increments its sequence number by 1. Due to the successful exchange of RREQ-RREP messages, all routing tables are updated. After that, the link between nodes S and A goes down, S detects the link break and updates its routing table: it sets all entries in its routing table with next hop A to *invalid* and increases the destination sequence number of the route to D to 2; the destination sequence number in the route to A is unknown (0) and hence not incremented. Now the destination sequence number on the route to node D is 2 in the routing table of S and 1 inside A's routing table, hence this number can go down.

[8]In fact, the same idea occurs already in [14], but without the formalisation in terms of invariants. However, that proof fails to consider some cases that do occur in AODV and might yield routing loops [9].

[9]We omit the precursor set; the routing table entries in each node are represented as $(dip, dsn, flag, hops, nhip)$, described in Section 3.1.

One way to avoid this problem is to claim the invariant only for the case that the routing table entries at ip and $nhip$ are both marked as valid. This is what we do in Theorem 5.4 below, and it suffices to obtain loop freedom of AODV. However, in order to prove Theorem 5.4, we need an invariant that also takes invalid routing table entries into account (cf. Proposition 5.3), so it is not possible to avoid the above problem altogether.

To compensate for the increase of a sequence number in case of route invalidation, we introduce the concept of a *net sequence number* of a route to dip according to the routing table of node ip in state N, which combines "freshness" and validity:

$$\mathtt{nsqn}_N^{ip}(dip) := \begin{cases} \mathtt{sqn}_N^{ip}(dip) & \text{if } \mathtt{flag}_N^{ip}(dip) = \mathtt{val} \vee \mathtt{sqn}_N^{ip}(dip) = 0 \\ \mathtt{sqn}_N^{ip}(dip) - 1 & \text{otherwise .} \end{cases}$$

Here, we write $\mathtt{sqn}_N^{ip}(dip)$ for $\mathtt{sqn}(rt, \mathtt{dip})$ in case rt happens to be the routing table maintained by a node with IP address ip in state N of the network. Likewise $\mathtt{flag}_N^{ip}(dip)$ denotes the validity of the route from ip to dip according to the routing table of ip in state N, $\mathtt{dhops}_N^{ip}(dip)$ its hop count, and $\mathtt{nhop}_N^{ip}(dip)$ its next hop. Furthermore $\mathtt{kD}_N^{ip}$, abbreviating $\mathtt{kD}(rt)$, is the set of destinations for which there is a valid entry in the routing table of ip.

In this section we state the key theorems and sketch some of the proofs; all details can be found in [8]. In particular, we show only proofs w.r.t. Process 2, the RREQ handling, and the displayed portion of Process 1.

PROPOSITION 5.1. *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.*

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

The next proposition says that in the routing table of a node ip, an entry to a destination dip will never be deleted, and the net sequence number of the entry will never go down.

PROPOSITION 5.2. *Let $ip, dip \in \mathbf{IP}$, and assume $N \xrightarrow{\ell} N'$, i.e. the network proceeds from state N to state N' by the occurrence of some action ℓ. If $dip \in \mathtt{kD}_N^{ip}$ then $dip \in \mathtt{kD}_{N'}^{ip}$ and $\mathtt{nsqn}_N^{ip}(dip) \leq \mathtt{nsqn}_{N'}^{ip}(dip)$.*

PROOF. In our model of AODV, the only way to change a routing table is by means of the operations `update`, `addpreRT` and `invalidate`. None of them ever deletes an entry altogether, or decreases its net sequence number. □

In this proof it is essential that we use resolution (f) of the ambiguity in the RFC presented in Section 4. Proposition 5.2 would not hold under resolutions (a), (b) or (c).

PROPOSITION 5.3. *If, in a reachable network state 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.*

$$\begin{aligned} & dip \in \mathtt{kD}_N^{ip} \ \wedge\ nhip \neq dip \\ \Rightarrow\ & dip \in \mathtt{kD}_N^{nhip} \ \wedge\ \mathtt{nsqn}_N^{ip}(dip) \leq \mathtt{nsqn}_N^{nhip}(dip)\ , \end{aligned}$$

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

PROOF. In the initial state, the invariant holds since all routing tables are empty (cf. Section 3.1). Next we assume that the property holds and check each line in Pro. 1 and Pro. 2 which could invalidate it.

A modification of the routing table of *nhip* is harmless, as it can only increase $\mathtt{kD}_N^{nhip}$ as well as $\mathtt{nsqn}_N^{nhip}(dip)$ (cf. Proposition 5.2).

Adding precursors to routes of *ip* does not harm since the invariant does not depend on precursors. It remains to examine all calls of `update` and `invalidate` to the routing table of *ip*. 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.

Pro. 1, Lines 10, 14, 18: The entry $(\mathtt{sip}, 0, \mathtt{val}, 1, \mathtt{sip}, \emptyset)$ is used for the update; its destination is $dip := \mathtt{sip}$. We assume this entry is actually inserted in the routing table of *ip*. Since $dip = \mathtt{sip} = \mathtt{nhop}_N^{ip}(dip) = nhip$, the antecedent of the invariant to be proven is not satisfied.

Pro. 2, Line 5: The entry $(\mathtt{oip}, \mathtt{osn}, \mathtt{val}, \mathtt{hops}{+}1, \mathtt{sip}, *)$ is used for the update; again we assume it is inserted into the routing table of node *ip*. So $dip := \mathtt{oip}$, $nhip := \mathtt{sip}$, $\mathtt{nsqn}_N^{ip}(dip) := \mathtt{osn}$ and $\mathtt{dhops}_N^{ip}(dip) := \mathtt{hops}{+}1$. This information is distilled from a received route request message (cf. Lines 1 and 8 of Pro. 1). By Proposition 7.1 of [8], this message was sent before, say in state N'; by Proposition 7.8 of [8], the sender of this message has identified itself correctly, and is `sip`.

By Proposition 5.1, with $ip_c := \mathtt{sip} = nhip$, $oip_c := \mathtt{oip} = dip$, $osn_c := \mathtt{osn}$ and $hops_c := \mathtt{hops}$, and using that $ip_c = nhip \neq dip = oip_c$, we get that $dip \in \mathtt{kD}_{N'}^{nhip}$ and

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

We first assume that the first line holds. Then, by Proposition 5.2,

$$\begin{aligned}\mathtt{nsqn}_N^{nhip}(dip) &\geq \mathtt{nsqn}_{N'}^{nhip}(dip) \geq \mathtt{sqn}_{N'}^{nhip}(dip){-}1\\ &\geq \mathtt{osn} = \mathtt{nsqn}_N^{ip}(dip)\ .\end{aligned}$$

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

$$\begin{aligned}\mathtt{nsqn}_N^{nhip}(dip) &\geq \mathtt{nsqn}_{N'}^{nhip}(dip) = \mathtt{sqn}_{N'}^{nhip}(dip)\\ &= \mathtt{osn} = \mathtt{nsqn}_N^{ip}(dip)\ .\end{aligned}$$

Pro. 2, Lines 16, 33: In these applications of `invalidate`, the next hop *nhip* is not changed. Since the invariant has to hold before the execution, it follows that $dip \in \mathtt{kD}_N^{nhip}$ also holds after execution. Furthermore, in view of Lines 15 and 32, the route is invalidated while the destination sequence number is incremented. For this reason the net sequence number stays the same, and the invariant is maintained. □

THEOREM 5.4. *If, in a state N, a node $ip \in \mathbf{IP}$ has a valid entry to dip, and the next hop is not dip and has a valid dip-entry as well, then the latter entry has a larger destination sequence number or an equal one with a smaller hop count.*

$$\begin{aligned}&dip \in \mathtt{vD}_N^{ip} \cap \mathtt{vD}_N^{nhip} \ \wedge\ nhip \neq dip\\ \Rightarrow\ &\mathtt{sqn}_N^{nhip}(dip) > \mathtt{sqn}_N^{ip}(dip) \vee \big(\mathtt{sqn}_N^{nhip}(dip) = \mathtt{sqn}_N^{ip}(dip) \wedge\\ &\mathtt{dhops}_N^{nhip}(dip) < \mathtt{dhops}_N^{ip}(dip)\big)\ ,\end{aligned}$$

where $nhip := \mathtt{nhop}_N^{ip}(dip)$, the next hop in the routing table entry at ip for the route to dip.

The proof [8] is similar to the previous one, but makes use of Proposition 5.3—including the case where $dip \notin \mathtt{vD}_N^{nhip}$—in an essential way.

From Theorem 5.4, we can conclude

THEOREM 5.5. *AODV is loop free.*

PROOF. If there were a loop in a routing graph $\mathcal{R}_N(dip)$, then for any edge $(ip, nhip)$ on that loop one would have $\mathtt{sqn}_N^{ip}(dip) \leq \mathtt{sqn}_N^{nhip}(dip)$, by Theorem 5.4. Hence the value of $\mathtt{sqn}_N^{ip}(dip)$ is the same for all nodes *ip* on the loop. Thus, by Theorem 5.4, the sequence numbers keep decreasing when travelling around the loop, which is impossible. □

6. ANALYSING VARIANTS OF AODV

In this section, we use AWN to model interpretations and variants of the AODV protocol. Interpretations are just different readings of the RFC, variants are the result of modifications to address existing limitations. Thanks to the use of process algebra, we can easily adapt the proofs of established correctness properties of the protocol, such as loop freedom. This is in contrast to the analysis of variants of existing protocols via simulations and testbed experiments, where all the work typically has to be redone from scratch (and even then cannot provide the same level of assurance).

6.1 Interpretations

In this section we briefly discuss two ambiguities of the RFC together with possible interpretations. More can be found in [8]. Of course each interpretation may possibly create routing loops and hence has to be examined separately. The specification and the proofs formalised in AWN can easily be adapted.

6.1.1 Invalidating Routing Table Entries

We have already presented one contradiction of the AODV RFC in Section 4. It was based on the question of what would happen if a node that has a valid routing table entry for a destination D receives an error message and invalidates the corresponding routing table entry. In the same section we also list eight possible resolutions. The first two, the only interpretations compliant with the RFC, violate Theorem 5.4, and yield routing loops [9]. The same holds for Interpretation (c). As stated before, to guarantee loop freedom one has to create an interpretation of AODV that is (literally speaking) not compliant with the RFC. Interpretations (d) and (e) are loop free—the proof is identical to the one of (f), given in Section 5. The remaining two resolutions can be proven to be loop free as well [8].

6.1.2 Updating with the Unknown Sequence Number

The AODV RFC [13] states that whenever a node receives a forwarded AODV control message from a 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. In the presented specification, this update is modelled in Lines 10, 14 and 18 of Process 1. In the event a new routing table entry is created, the sequence-number-status flag is set to false to signify that the sequence number corresponding to the neighbour is unknown. This interpretation is modelled in [8] and is compliant with the RFC.

However, in most implementations of AODV (e.g. [2, 6]), an unknown sequence number is simply represented by the value 0, rather than by setting a flag. In the specification of Section 3 we follow this approach of using the value 0.

Since the RFC does not make the update mechanism clear, different interpretations arise when an existing valid routing table entry for the neighbour has to be updated. While it is clear that expiry values for timers associated with the routing table entry will be updated, it is not clear if a valid sequence number with value n (>0) will remain unmodified as it is, or be updated to the value 0. In order to verify which interpretations are reasonable, we check which ones satisfy the invariants specified before (e.g., Theorem 5.4).

If we assume that an entry $(dip, 0, \texttt{val}, hops', *, *)$ replaces an entry $(dip, dsn, \texttt{val}, hops, *, *)$ (where $dsn>0$) in a routing table, it is easy to see that Theorem 5.4 is violated. In fact, it is not hard to exploit this to create a routing loop, since it allows the possibility of decreasing destination sequence numbers [8, Sect. 9.1]. This is the interpretation that is implemented by AODV-UIUC and AODV-UCSB. Our interpretation follows AODV-UU: an entry $(dip, dsn, \texttt{val}, hops, nhip, *)$ is replaced by $(dip, dsn, \texttt{val}, 1, sip, *)$, i.e., in the existing entry only the next hop and the hop count is updated—the sequence number stays the same. This interpretation is not in line with the RFC—the RFC never merges information of two routes. However, it is loop free (cf. Section 5). AODV-ns2 does not perform an update if a routing table entry already exists, i.e., it uses a version of `update` as in Section 3.1, but without the fifth clause. This interpretation is also loop free; the proof is identical to the one presented in the previous section.

There are more interpretations possible if the sequence-number-status flag (the flag indicating whether a sequence number is known or unknown) is modelled.

(a) Set the flag to unknown and the sequence number to 0. Using the same argument as before, this can yield routing loops.

(b) Set the flag to unknown, but keep the destination sequence number stored in the routing table. This interpretation is indeed loop free and is most likely the intention of the AODV RFC.

Detailed proofs can be found in [8].

6.2 Variants

Let us now turn to variants of AODV and look at (known) shortcomings of the AODV protocol, present possible improvements and then use AWN to verify that the modified AODV is still loop free.

6.2.1 Non-Optimal Route Selection

In AODV's route discovery process, a destination node (or an intermediate node with an active route to the destination node) will generate a RREP message in response to a received RREQ message. The RREQ message is then discarded 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 [12], where route optimality is defined in terms of a metric, for example hop count. In [12] 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

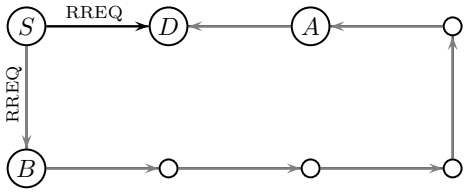


Figure 2: Non-optimal route selection

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.

We illustrate this by the example in Figure 2. There, node S wants to find a route to node D. It generates and broadcasts a RREQ message that is received by its neighbours D and B. Since D is the destination, it responds with a RREP message. The received RREQ message is discarded and not forwarded. On the other hand, B continues to forward its received RREQ message, which eventually arrives at A. 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 D, A is prevented from discovering its optimal route to S.

A possible modification to AODV to solve this problem is to allow the destination node to continue to forward the RREQ message. This will then enable A in Figure 2 to discover its optimal route to S. In addition, the forwarded RREQ message from the destination node is modified to include a flag that indicates a RREP message has already been generated and sent in response to the former message. This is to prevent other nodes (with active routes to the destination) from sending a RREP message in response to their reception of the forwarded RREQ message.

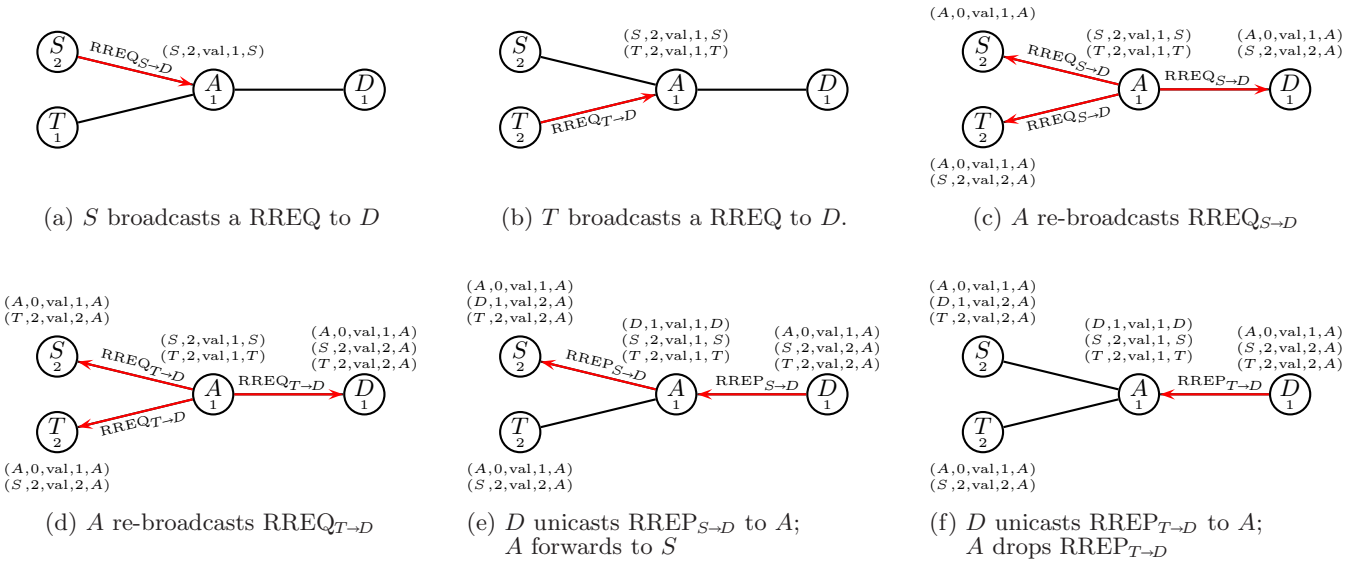
The entire specification of this variant (in [8]) differs only in five lines from the original—all of which are contained in the process `RREQ`; the other processes remain unchanged. The changes introduce the new flag and a case distinction based on that, as well as three new broadcasts. For example, after initiating a route reply at the destination (Process 2, Line 12), the route request message is forwarded:

broadcast(`rreq`(`hops`+1,`rreqid`,`dip`,`dsn`,`oip`,`osn`,`ip`,`true`)) ,

where the last component of the RREQ message is the newly introduced flag. The proofs of important properties (e.g., loop freedom) are still valid. The proofs of the invariants proceed by examining lines in our processes where the invariant might be invalidated: The proof of Proposition 5.1, which can be found in [8], checks all occurrences of sending a RREQ message, and the proofs of Proposition 5.3 and Theorem 5.4 check all occurrences of `update` and `invalidate`. For the former, three new **broadcast**-commands have to be examined; however these cases are similar to the **broadcast** already implemented in the original process `RREQ` (Line 41). For the latter, no extra effort is needed, as the modification does not involve occurrences of `update` and `invalidate`.

6.2.2 Failure of Route Discovery Process

In AODV's route discovery process, a RREP message from the destination node is unicast back along the selected route



(a) S broadcasts a RREQ to D

(b) T broadcasts a RREQ to D.

(c) A re-broadcasts RREQ$_{S\to D}$

(d) A re-broadcasts RREQ$_{T\to D}$

(e) D unicasts RREP$_{S\to D}$ to A; A forwards to S

(f) D unicasts RREP$_{T\to D}$ to A; A drops RREP$_{T\to D}$

Figure 3: Failure of route discovery process

towards the originator of the RREQ message. Every intermediate node on the selected route will process the RREP message and, in most cases, forwards it towards the originator node. However, there is a possibility that the RREP message is discarded at an intermediate node and hence results in the originator node not receiving a reply. The discarding of the RREP message is due to the RFC specification of AODV [13] 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. 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. We illustrate this problem with an example.[10]

Figure 3 shows a four-node topology. In Figures 3(a) and 3(b), source nodes S and T, respectively, initiate a route discovery process to search for a route to D. When generating a RREQ message, the source node increments its sequence number before inserting it into the message. In processing the RREQ messages from S and T, node A creates routing table entries in its routing table.

In Figures 3(c) and 3(d), node A re-broadcasts the RREQ messages that it received previously. The destination node D receives the RREQ messages, and creates corresponding entries in its routing table. In Figure 3(e), D replies with a RREP message in response to the RREQ message from node S. Since the RREQ message from S does not contain any information on the destination sequence number for D, node D inserts its sequence number of 1 into the RREP message. This message is then processed by A (a routing table entry to D is created) and forwarded to S.

Similarly, in Figure 3(f), node D replies with a RREP message in response to the RREQ message from node T. Since the RREQ message from node T does not contain any information on the destination sequence number for node D, node D again inserts its sequence number of 1 into its RREP message. When the intermediate node A receives the RREP message, it processes the message. However, the existing routing table entry that node A already has for destination node D contains the same information (same destination sequence number and same hop count) as in the received RREP message. Therefore, node A does not update its routing table entry for node D and thus, according to the RFC specification, will not forward the RREP message to the source node T. This then leads to an unsuccessful route discovery process for node T.

A solution to this problem is to require intermediate nodes to forward *all* RREP messages that they receive using the newest available information on the route to the destination node: if the node's routing table contains an entry for the destination node that is valid and "fresher" than that in the RREP message, the intermediate node updates the contents of the RREP message to reflect this. The intermediate node A will then forward a RREP message containing up-to-date information on the destination node D.

As in Section 6.2.1, this solution does not violate any of the invariants; again this follows by adapting the original proofs [8].

7. RELATED WORK

Previous attempts to prove loop-freedom of AODV have been reported in [14, 4, 20], but none of these proofs are complete and valid for the current version of AODV [13]. [14] fails to consider the effect of RERR messages, which can result in routing loops; [4] analyses an earlier draft of AODV and uses an invariant that does not hold when following the AODV RFC; and [20] only considers a restricted version of AODV, not covering the important case of route replies by intermediate nodes. Details of the limitations of these proof attempts are provided in [9].

Graph Transformation Systems were used in [17] to model DYMO v10, a protocol derived from AODV. The paper provides a semi-algorithm, based on graph rewriting, which was

[10] A slightly different example was given on the MANET mailing list `http://www.ietf.org/mail-archive/web/manet/current/msg05702.html`

used to verify loop-freedom for DYMO. Model checking is also used to verify properties of routing protocols for WMNs. For example, [19] shows loop freedom of the ad-hoc protocol LUNAR, for fixed topologies or set changes in the topology. Model checking in general lacks the ability to verify protocols for an arbitrary and changing topology. It is used to check specific scenarios only.

8. CONCLUSIONS

In this paper, we have presented a complete and accurate model of the core functionality of AODV using the process algebra AWN, which has been tailored specifically for the formal modelling of wireless mesh networks and MANETs. The unique set of features and primitives of AWN allows the creation of accurate, concise and readable models of relatively complex and practically relevant network protocols, which we have demonstrated with AODV. This is in contrast to some prior related work, which either modelled only very simple protocols, or modelled only a subset of the functionality of relevant WMN or MANET routing protocols.

The currently predominant practice of informally specifying WMN and MANET protocols via English prose has a potential for ambiguity and inconsistent interpretation. The ability to provide a formal and unambiguous specification of such protocols via AWN is a significant benefit in its own right. We have demonstrated how AWN can be used as a basis for reasoning about critical protocol correctness properties, illustrated with the example of loop freedom. We have further shown how relevant proofs can relatively easily be adapted to protocol variants. 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 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.

9. REFERENCES

[1] Kernel AODV (ver. 2.2.2), NIST. `http://www.antd.nist.gov/wctg/aodv_kernel/`.

[2] AODV-UU: An implementation of the AODV routing protocol (IETF RFC 3561). `http://sourceforge.net/projects/aodvuu/`.

[3] K. Bhargavan, C. A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan. Verisim: Formal analysis of network simulations. *IEEE Transactions on Software Engineering*, 28(2):129–145, 2002.

[4] K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal verification of standards for distance vector routing protocols. *J. ACM*, 49(4):538–576, 2002.

[5] I. Chakeres and C. Perkins. Dynamic MANET on-demand (AODVv2) routing. Internet Draft (Standards Track), draft-ietf-manet-dymo-22, 2012. `tools.ietf.org/html/draft-ietf-manet-dymo-22`.

[6] I. D. Chakeres and E. M. Belding-Royer. AODV routing protocol implementation design. In *Workshop on Wireless Ad Hoc Networking (WWAN'04)*, 2004.

[7] A. Fehnker, R. J. van Glabbeek, P. Höfner, A. McIver, M. Portmann, and W. L. Tan. A process algebra for wireless mesh networks. In H. Seidl, editor, *European Symposium on Programming (ESOP'12)*, volume 7211 of *LNCS*, pages 295–315. Springer, 2012.

[8] A. Fehnker, R. J. van Glabbeek, P. Höfner, A. McIver, M. Portmann, and W. L. Tan. A process algebra for wireless mesh networks used for modelling, verifying and analysing AODV. Technical Report 5513, NICTA, 2012. `http://www.nicta.com.au/pub?id=5513`.

[9] R. J. van Glabbeek, P. Höfner, W. L. Tan, and M. Portmann. Sequence numbers do not guarantee loop freedom—AODV can yield routing loops, 2012. `http://rvg.web.cse.unsw.edu.au/pub/AODVloop.pdf`

[10] IEEE P802.11s. IEEE draft 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, July 2010.

[11] V. Kawadia, Y. Zhang, and B. Gupta. System services for ad-hoc routing: Architecture, implementation and experiences. In *Mobile Systems, Applications and Services (MobiSys'03)*, pages 99–112. ACM, 2003.

[12] S. Miskovic and E. W. Knightly. Routing primitives for wireless mesh networks: Design, analysis and experiments. In *Information Communications (INFOCOM'10)*, pages 2793–2801. IEEE, 2010.

[13] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc on-demand distance vector (AODV) routing, RFC 3561 (experimental), 2003. `http://www.ietf.org/rfc/rfc3561.txt`.

[14] C. Perkins and E. Royer. Ad-hoc On-Demand Distance Vector Routing. In *Mobile Computing Systems and Applications (WMCSA'99)*, pages 90–100, 1999.

[15] A. A. Pirzada, M. Portmann, and J. Indulska. Performance analysis of multi-radio AODV in hybrid wireless mesh networks. *Computer Communications*, 31(5):885–895, 2008.

[16] K. Ramachandran, M. M. Buddhikot, G. Chandranmenon, S. Miller, E. Belding-Royer, and K. Almeroth. On the design and implementation of infrastructure mesh networks. In *Workshop on Wireless Mesh Networks (WiMesh'05)*. IEEE, 2005.

[17] M. Saksena, O. Wibling, and B. Jonsson. Graph grammar modeling and verification of ad hoc routing protocols. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, pages 18–32. Springer, 2008.

[18] A. Subramanian, M. Buddhikot, and S. Miller. Interference aware routing in multi-radio wireless mesh networks. In *IEEE Workshop on Wireless Mesh Networks (WiMesh'06)*. IEEE, 2006.

[19] O. Wibling, J. Parrow, and A. N. Pears. Automatized verification of ad hoc routing protocols. In D. de Frutos-Escrig and M. Núñez, editors, *Formal Techniques for Networked and Distributed Systems (FORTE '04)*, volume 3235 of *LNCS*, pages 343–358. Springer, 2004.

[20] M. Zhou, H. Yang, X. Zhang, and J. Wang. The proof of AODV loop freedom. In *Wireless Communications & Signal Processing (WCSP'09)*. IEEE, 2009.