An Abstract Model of Routing in Mobile Ad Hoc Networks

Cong Yuan[1], Jonathan Billington[1], and Jörn Freiheit[2]

[1] Computer Systems Engineering Centre
University of South Australia
Mawson Lakes Campus, SA 5095, AUSTRALIA
Email: yuacy003@students.unisa.edu.au, jonathan.billington@unisa.edu.au
[2] Programming Logics Group, Max-Planck Institute
Saarland, 66123, GERMANY
Email: freiheit@mpi-inf.mpg.de

Abstract. Modelling Mobile Ad Hoc Networks (MANETs) is a challenge because the topology of such networks changes dynamically and unpredictably. We create a highly abstract Coloured Petri Net model of routing in a MANET based on the Destination-Sequenced Distance-Vector (DSDV) routing protocol. Our experiments show that this model can simulate the required dynamic changes of network topology and reveal that incorrect routing information can be created and propagated in the MANET.

Keywords: MANETs, DSDV routing protocol, Modelling and Simulation, Coloured Petri Nets.

1 Introduction

A Mobile Ad Hoc Network (MANET) [1] is a collection of wireless mobile nodes, such as laptops, mobile phones and Personal Digital Assistants (PDAs), which establish a temporary network without the help of any pre-existing infrastructure or centralized administration. Packet forwarding, routing, and other network operations are carried out by the individual nodes themselves [18]. In addition, nodes freely join in and move out, which results in the network topology constantly changing, so that conventional routing protocols do not work well for MANETs. Most current routing protocols designed for MANETs are still under development [2]. So far their definitions are not mature enough for Internet standards and they have mainly been evaluated by simulation and live testing. Unfortunately, simulation and testing are not sufficient to verify that there are no subtle errors or design flaws left in a protocol [28]. To achieve this goal we need to formally verify its operation. Formal verification firstly requires the creation of a formal model of the system.

Concerning formal verification of routing protocols for MANETs, Bhargavan et al. [4] and Obradovic [22] verify the Routing Information Protocol (RIP) [19] and the Ad-hoc On-Demand Distance Vector (AODV) Routing Protocol [23]. They analysed AODV and identified a flaw that could lead to a loop. This was done using SPIN [11] and the HOL Theorem Proving System [10]. In order to realise loop free behaviour, a modification was suggested and verified. Their approach verified the general case, but required a significant amount of user interaction. Wibling et al. [28] consider an automatic verification strategy, and use two model checking tools, SPIN and UPPAAL [17], to verify both the data and control aspects of the Lightweight Underlay Network Ad hoc Routing (LUNAR) protocol. However, they just studied a limited set of topologies.

The purpose of this paper is to illustrate the dynamic operations of a MANET using Coloured Petri Nets (CPNs) [13, 14]. However, it is not easy to create a CPN model of a MANET because the topology of such a network changes dynamically and unpredictably. Only a few attempts have been made to model a highly dynamic system topologies using CPNs. Findlow and Billington [7] used High-Level Petri Nets [5] to model dynamic dining

philosophers, where philosophers can come and go unpredictably. The topology of the system is circular, but arbitrarily expanding and contracting as philosophers come and go and can take different positions in the circle. In MANETs there is no regular structure but an arbitrarily changing network topology.

Xiong et al. [29] present a timed CPN model for AODV. They considered that it was too difficult to model the highly dynamic topology of MANETs with CPNs directly and proposed a topology approximation (TA). They assume that every node has the same transmission range and thus that the neighbourhood relation is symmetric. They also assume that each node in the MANET has the same number of neighbours, a rather unrealistic assumption. Further, in simulation experiments, the number of neighbours needs to be supplied by the analyst.

Kristensen and Jensen [15] model and analyse the Edge Router Discovery Protocol (ERDP), a protocol for connecting gateways in MANETs to edge routers in fixed networks. Thus their CPN model does not involve the dynamically changing topology of MANETs. Kristensen et al [16] use hierarchical CPNs to model a fixed number (in their case 4) of MANETs which communicate with each other via an IPv6 network. Nodes can join and leave a particular MANET and can move from one MANET to another. The topology of a MANET is described explicitly by storing pairs of nodes, where each pair represents a one directional link. The model includes simple forwarding of user packets directly from the source node to the destination node, irrespective of which MANET they belong to. The model thus abstracts from the mechanism by which these packets are forwarded through different nodes of the various MANETs to their destination, which we believe is a key part of MANET design. Thus no attempt is made to model a routing protocol or how the packets are routed in the network.

In this paper, we consider the Destination-Sequenced Distance-Vector (DSDV) routing protocol [24, 25], because DSDV has relatively low complexity compared with many other ad hoc routing protocols. DSDV is a representative *proactive protocol*. The primary characteristic of such a routing approach is that each node tries to maintain a route to every other node in the network at all times. It has the advantage that there is no delay to begin a session, and tends to perform well in networks where there are a significant number of data sessions within the network [2].

The main aim of this paper is to provide the first CPN model of the basic functions of the DSDV routing protocol as a fisrt step towards its formal specification and verification. We introduce the basic features of DSDV and then provide an abstract CPN model of the DSDV routing mechanism. We perform some simulation experiments to demonstrate the way the model captures the dynamic changes in network topology. In doing so, we uncover some interesting errors and suggest modifications to the procedures to eliminate these errors.

There are several contributions of this paper. Firstly, we demonstrate that CPNs can model the dynamically changing network topologies associated with MANETs in an elegant and simple way, without using the assumptions made in [29] and without requiring the relatively complex 4-level hierarchical structure used in [16]. Secondly, we provide the first CPN model of the DSDV routing procedures and discover two errors in these procedures. This provides the first analysis of the key component of DSDV, the use of sequence numbers to discard old information. Thirdly we discuss modifications to the DSDV procedures for updating routing tables that we believe will remove these errors. These modifications are implemented in a revised CPN model and our simulations have shown that they have been effective in eliminating these errors in the scenarios run so far.

The rest of the paper is organised as follows. Section 2 gives an introduction to DSDV and explains the basic operations of the protocol. In Section 3, we describe a highly abstract

CPN model of the DSDV protocol. Simulation experiments are conducted in Section 4, and discussed in Section 5. Finally, conclusions and future work are presented in Section 6.

2 Destination-Sequenced Distance-Vector Routing Protocol

2.1 Background

The destination-sequenced distance-vector routing protocol is derived from distance-vector routing algorithms [20]. Such algorithms are often referred as the Distributed Bellman-Ford (DBF) algorithm since they are based on a shortest path computation algorithm presented by Bellman [3]. The first description of the distributed algorithm was given by Ford and Fulkerson [8]. This algorithm was the original Arpanet routing algorithm, and was also used in the Internet under the name RIP and in early versions of DECnet and Novell's Internet Packet eXchange (IPX). AppleTalk and Cisco routers use improved distance vector protocols [26]. The distance vector algorithm can cause the formation of both short-lived and long-lived loops [6]. The main cause of the formation of routing loops is that nodes choose their next hops in a completely distributed fashion based on information that may be stale and therefore incorrect. The modifications [9, 12, 21] designed to eliminate the looping problem are not feasible in MANETs because of the rapidly changing topology of such a network [25].

RIP [19] is a simple and practical distance vector protocol. It is easy to understand and modify. However, like other distance-vector algorithms, RIP also suffers from very slow convergence (the *counting to infinity* problem [26]). Despite this problem, without the ability to handle rapid topological changes, the usefulness of RIP in MANET is limited [25]. Furthermore, the techniques designed in [19] to solve *counting to infinity* are not useful within the wireless environment. For these reasons, in 1994 C. E. Perkins and P. Bhagwat [24] presented the destination-sequenced distance-vector routing protocol for ad hoc networks. In 2001, a more comprehensive protocol specification was given by Perkins [25], and a recent description by Royer can be found in [2]. This protocol preserves the simplicity of RIP and avoids the looping problem by using sequence numbers. The sequence number is attached to each route entry in the routing tables stored in nodes, so they can quickly distinguish stale routes from new ones, and thus avoid formation of routing loops. A brief introduction to the basic routing algorithm of DSDV is given in the next subsection mainly based on [25]. The purpose is not to give a complete description of DSDV, but to provide sufficient information to understand the CPN model in the next section. In this model, we relax the assumption that links are bidirectional [25], because asymmetries of transmission ranges are prevalent in a wireless environment.

2.2 Protocol Overview

In DSDV, every mobile node maintains a route to every other node (i.e. destination) in the network. Thus its routing table comprises a list of route entries. A route entry corresponding to a destination contains the following attributes:

- **Destination**: IP address of the destination;
- **Nexthop**: IP address of next node along the route to the destination;
- **Metric**: the number of nodes (hops) required to reach the destination;
- **SeqNr**: last recorded *sequence number* for the destination;
- **Installtime**: the time when this route entry is received.

The purpose of sequence numbers is to track changes in topology. Each node keeps its own sequence number, which is increased whenever important changes are made to its routing table. When a route entry to a destination is established, it is stamped with the current sequence number of this destination. As the topology of the network changes, more recent route entries have higher sequence numbers, so that nodes can distinguish between current and stale route entries by comparing the sequence numbers of these entries. In order to keep routing tables consistent in a dynamically changing topology, each node periodically transmits updates using a *full dump* packet, and transmits an *incremental* update immediately after a significant change to its routing table. These updates comprise a list of triples of the form: (**Destination**, **Metric**, **SeqNr**), derived directly from the routing table of a node. A full dump comprises a list of triples where each triple corresponds to an entry in the routing table. An incremental update only carries triples that have changed since the last full dump. Each node may receive routing information sent by another node. It utilizes this information to recompute its routing table entries. For every triple received, the node firstly checks its own routing table to find whether or not an entry to the same destination exists. If such an entry does not exist, the node adds this route entry received to its table after: incrementing the metric of this entry by one hop; adding the sender as the next hop; and including the current sequence number of this destination. If such an entry exists, the node will choose the entry with the higher sequence number. If two entries have the same sequence number, the entry with the shorter metric will be chosen. These two situations can guarantee loop-free paths to each destination in DSDV (A proof of correctness of this point is given in [25]). Nodes in a MANET may cause *broken links* as they move from place to place. A broken link may be inferred by a node if no message has been received for a while (e.g. one time interval of the periodic broadcast) from a former neighbour. The metric of a broken link is represented by ∞. When a link to the next hop is broken, any route through this next hop is immediately assigned an ∞ metric, and its sequence number is increased by 1 [25].

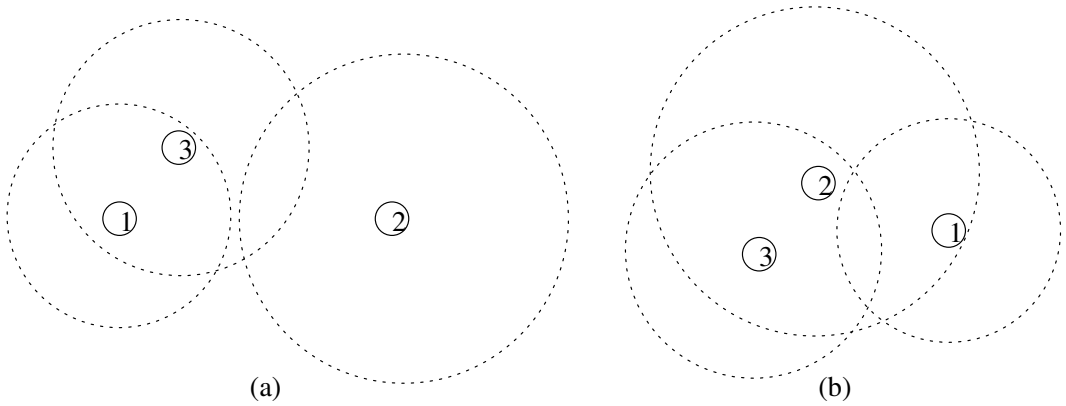

Fig. 1. The neighbourhood relation between the nodes in a MANET

In the context of mobile ad hoc networking, each node can communicate directly with any other node within its transmission range. If the node wants to communicate with nodes which reside beyond this range, intermediate nodes between them are used to relay the messages. We consider that node A is a neighbour of node B if node A is within node B's transmission range. In a realistic environment, mobile hosts have different software/hardware configurations and different radio interfaces [18], which can lead to variability in their transmission ranges. Consider the example of a MANET comprising 3 nodes (node 1, node 2 and node 3) shown in Fig. 1. In this figure, the full circles represent nodes, and the dotted circles represent transmission ranges of the nodes. The number positioned in each of the small circles represents

the node's identity (i.e. IP address). Fig. 1 shows 2 snapshots of the network as time increases and the nodes move in the network. As shown in Fig. 1(a), node 1 and node 3 are neighbours of each other. Node 2 is not a neighbour of node 1 or node 3, because it is not within the transmission range of either node. Similarly, node 2 has no neighbours because no other nodes are in its range. Within a MANET, each node can move at will, changing its neighbours arbitrarily. Thus in Fig. 1(b), node 1 is now a neighbour of node 2, node 2 and node 3 are neighbours of each other, but node 2 is not a neighbour of node 1. Therefore in a highly dynamic MANET the neighbourhood relation may not be symmetric. In previous work, the general assumption has been made that every node has identical capability and responsibility, so that a MANET is fully symmetric [4, 25, 29]. However, a key objective of our research is to precisely mimic the dynamic nature of a MANET, and thus we do not make this assumption.

In [25], many parameters which control the behaviour of DSDV, such as the frequency of broadcast, the frequency of full dumps versus incremental updates and the percentage change in the routing metric which can trigger an incremental update are not given. Moreover, when and how each node updates its sequence number in DSDV are not described explicitly. There is also not any description of the initial state in the DSDV process. Therefore, in order to illustrate the behaviour of DSDV completely and unambiguously, we need to make some assumptions. According to the example presented in [25], there is a route entry to itself in the routing table of each node. For our modelling and analysis, we assume that each node initially only has the route entry to itself in its routing table with its sequence number set to 0. The node increases its sequence number by 2 whenever there is a significant change to its routing table. Then this node broadcasts the triples associated with the changed route entries immediately. This assumption is indeed safe for DSDV because it satisfies the need of DSDV: more recent route entries should have higher sequence numbers in the routing table. Meanwhile, it does not influence the routing algorithm of DSDV in [25]. To simplify the problem, we ignore the install time of each route entry.

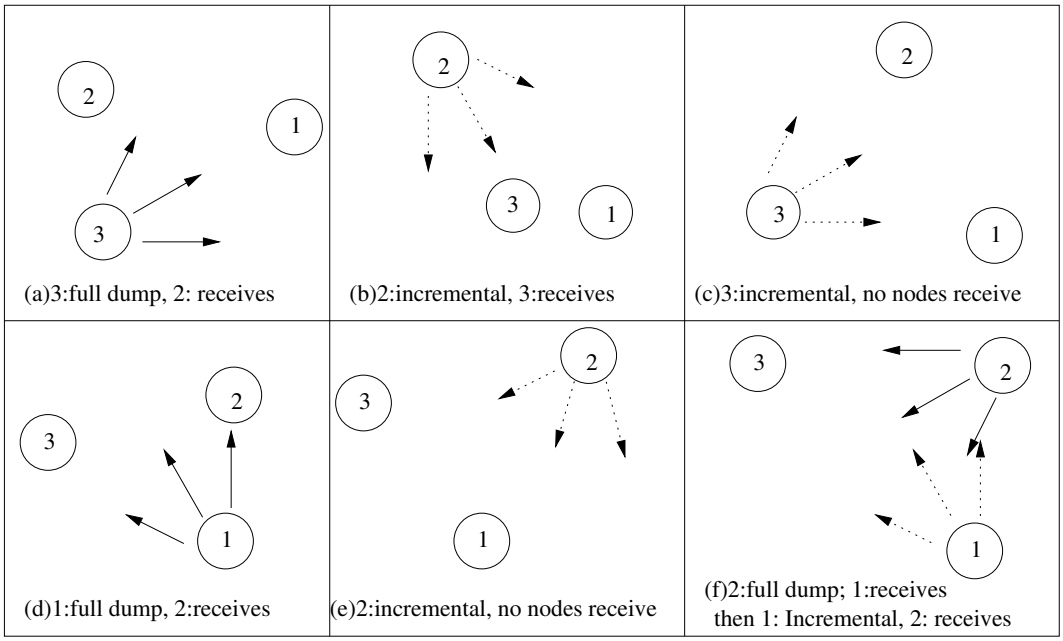

Fig. 2. An example of DSDV in operation (1) route advertisements

An example of DSDV in operation with unidirectional links is depicted in Fig. 2. In this figure, the solid arrows indicate that the sender broadcasts a full dump to its neighbours, and the dashed arrows indicate that the sender broadcasts an incremental update. In order

Table 1. Initial routing tables of nodes 1, 2 and 3

Dest.	Next	Met.	SeqNr
1	1	0	(1, 0)

Dest.	Next	Met.	SeqNr
2	2	0	(2, 0)

Dest.	Next	Met.	SeqNr
3	3	0	(3, 0)

Table 2. Routing tables of nodes 1, 2 and 3 in **Fig. 2(a)**

Dest.	Next	Met.	SeqNr
1	1	0	(1, 0)

Dest.	Next	Met.	SeqNr
2	2	0	(2, 2)
3	3	1	(3, 0)

Dest.	Next	Met.	SeqNr
3	3	0	(3, 0)

Table 3. Routing tables of nodes 1, 2 and 3 in **Fig. 2(b)** and **(c)**

Dest.	Next	Met.	SeqNr
1	1	0	(1, 0)

Dest.	Next	Met.	SeqNr
2	2	0	(2, 2)
3	3	1	(3, 0)

Dest.	Next	Met.	SeqNr
2	2	1	(2, 2)
3	3	0	(3, 2)

Table 4. Routing tables of nodes 1, 2 and 3 in **Fig. 2(d)** and **(e)**

Dest.	Next	Met.	SeqNr
1	1	0	(1, 0)

Dest.	Next	Met.	SeqNr
1	1	1	(1, 0)
2	2	0	(2, 4)
3	3	1	(3, 0)

Dest.	Next	Met.	SeqNr
2	2	1	(2, 2)
3	3	0	(3, 2)

Table 5. Routing tables of nodes 1, 2 and 3 in **Fig. 2(f)**

Dest.	Next	Met.	SeqNr
1	1	0	(1, 2)
2	2	1	(2, 4)
3	2	2	(3, 0)

Dest.	Next	Met.	SeqNr
1	1	1	(1, 2)
2	2	0	(2, 4)
3	3	1	(3, 0)

Dest.	Next	Met.	SeqNr
2	2	1	(2, 2)
3	3	0	(3, 2)

Table 6. Routing tables of nodes 1 and 2 corresponding to **Fig. 3**

Dest.	Next	Met.	SeqNr
1	1	0	(1, 4)
2	2	1	(2, 6)
3	2	∞	(3, 1)

Dest.	Next	Met.	SeqNr
1	1	1	(1, 2)
2	2	0	(2, 6)
3	3	∞	(3, 1)

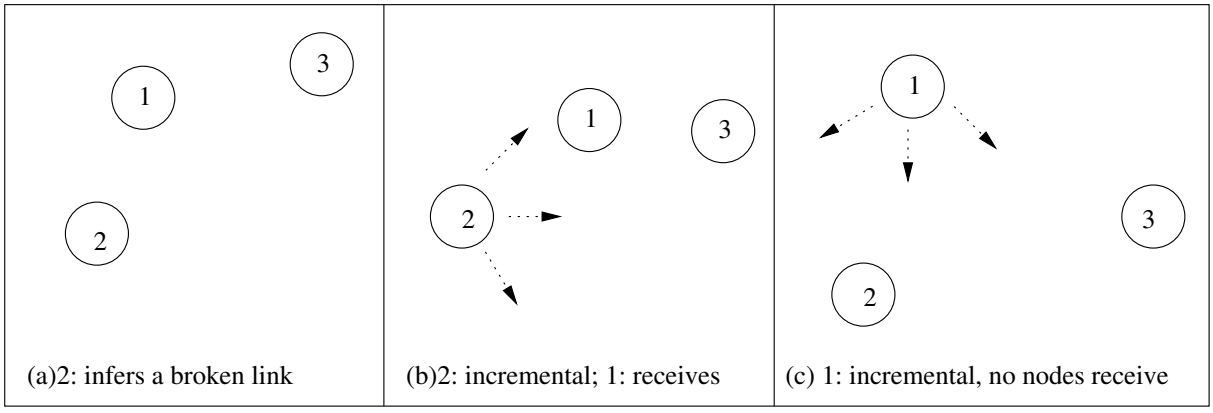

Fig. 3. An example of DSDV in operation (2) link breakage

to keep the figure succinct, the transmission ranges of the nodes are not given in Fig. 2. If a node receives the routing information sent by another node that means the receiver is a neighbour of the sender, i.e. it is in the range of the sender.

Fig. 2 shows possible route advertisements of 3 nodes running DSDV in a MANET from the initial state as time increases. These advertisements include a full dump of each node and consequential incremental updates. The initial routing tables of 3 nodes are shown in Table 1. (To save space, 1 is used for node 1, 2 for node 2 and 3 for node 3.) In the table, (1, 1, 0, (1, 0)) means that node 1 has a routing table comprising one route entry to itself, in which the destination and the next hop both are node 1, the number of hops is 0 and its initial sequence number is set to 0, represented as (1, 0).

We assume that node 3 is the first to broadcast its full dump (3, 0, (3, 0)) within the MANET. Only node 2 receives this information as indicated in Fig. 2(a). Node 2 updates its routing table based on this information. It adds the new entry for node 3 to its routing table, after increasing the metric of this entry by 1 and adding node 3 as the next hop of this entry. After that, it increases its own sequence number to 2. The updated routing tables of the nodes are given in Table 2. As shown in Fig. 2(b), node 2 immediately broadcasts an incremental update, ((2, 0, (2, 2)), (3, 1, (3, 0))). Node 3 receives this information and updates its routing table in a similar way, adding a new route entry for node 2 to its routing table, and increases its sequence number to 2, as shown in Table 3. In Fig. 2(c), node 3 broadcasts an incremental update, ((2, 1, (2, 2)), (3, 0, (3, 2))), but no other nodes receive it, because node 2 is out of the range of node 3. Now, node 1 originates its full dump broadcasting (1, 0, (1, 0)) which node 2 receives, as shown in Fig. 2(d). Node 2 updates its routing table as shown in Table 4. In Fig. 2(e), node 2 broadcasts an incremental update ((1, 1, (1, 0)), (2, 0, (2, 4))), but no others receive it. Now in Fig. 2(f), node 2 originates its full dump and transmits ((1, 1, (1, 0)), (2, 0, (2, 4)), (3, 1, (3, 0))), and node 1 receives this information. Node 1 keeps the entry to itself unchanged because of the identical sequence number 0. It adds the entries for node 2 and node 3 to its routing table respectively, and it increases its sequence number by 2. Then node 1 broadcasts an incremental update ((1, 1, (1, 2)), (2, 1, (2, 4)), (3, 2, (3, 0))) while node 2 receives it (to save space, also shown in Fig. 2(f)). Node 2 only updates the route entry for node 1 as (1, 1, (1, 2)), and keeps the other entries unchanged because there is no new information received, so it does not broadcast any incremental update. The modified routing tables are shown in Table 5.

An example that illustrates how a node deals with a broken link is given in Fig. 3. Assume the routing tables of nodes 1 and 2 are the same as those in Table 5. Consider node 2 finds that it has not received a broadcast from node 3 for a while (e.g. a periodic interval), as shown in Fig. 3(a). It infers the link between them is broken. Hence, node 2 assigns the metric of this link to ∞ and increases the sequence number by 1. Then it increases its own sequence number by 2 and immediately broadcasts an incremeantal update ((2, 0, (2, 6)), (3, ∞, (3, **1**))), while node 1 receives it in Fig. 3(b) (note: at this moment, the routing table of node 1 is the same as that in Table 5, while the routing table of node 2 is shown in Table 6). Node 1 updates the route entry for node 2 to the received sequence number 6. For the route entry for node 3, node 1 selects the one received since it has the higher sequence number. Because this is a broken link, node 1 increases its own sequence number by 2. The modified routing table of node 1 is shown in Table 6. Then node 1 immediately broadcasts an incremental update ((1, 0, (1, 4)), (2, 1, (2, 6)), (3, ∞, (3, **1**))) and no other nodes receive it, as shown in Fig. 3(c). So routing tables of node 1 and node 2 are the same as those in Table 6.

3 Abstract CPN Model of a MANET based on DSDV

3.1 Design Rationale

The intent of our CPN model is to show that CPNs can be used for the modelling of routing protocols in a MANET environment where arbitrary changes of network topology are possible. We therefore start by modelling the basic operation of the routing protocol: nodes discover other nodes by receiving broadcast messages and update their routing tables accordingly; and nodes discover that previously established links are no longer valid, and mark them as broken in their routing tables.

We do not model the routing messages explicitly. Instead we just consider events where the information from the message is received and processed. We assume that this cannot occur simultaneously in different nodes, i.e. no two nodes receive and process the broadcast at exactly the same time. Instead, these events are interleaved in the different nodes. Further, because of arbitrary movements of the nodes, there is no synchronisation between different nodes for a broadcast. Hence the broadcast by node A may be received by: no nodes (corresponding to A being an isolated node); one node (all other nodes are out of range); or any number of nodes. The interpretation is that all nodes that have updated their routing tables are in range at that time. Conversely, all nodes that have not updated their routing tables are not in range at that time. Moreover, a lost message has the same effect as being out of range. With this understanding, we model the updating of routing tables in the MANET non-deterministically. Thus updating a routing table is considered an arbitrary event.

Further, we model broken links in a similar way. Because a functional model abstracts from time, we consider that a broken link can be interleaved with any other event and is thus modelled by an arbitrary event. Hence it is possible for a node to receive a full dump from a neighbour and then to declare the link down as the next event. This could correspond to the node leaving the MANET and other nodes becoming dispersed so that they are all out of range.

We believe this captures the asymmetry between nodes and their arbitrary movement realistically at a high-level of abstraction. Because the functional model abstracts from probabilities, it includes many situations that would be considered rare events.

3.2 CPN Model

A CPN diagram of the MANET routing protocol, based on DSDV, is given in Fig. 4. The purpose of the CPN is to model how nodes update their routing tables and deal with broken links. Thus we model the nodes in the MANET by just their identity and their routing table. The nodes are stored in the only place in the CPN, Nodes. This place is typed by the colour set MNode as shown in line 13 in Fig. 5. MNode is a product of its identity, NodeId, which corresponds to its address, and its routing table, RT. The routing table is represented as a list of route entries, one for each destination.When ignoring the install time, the route entry is a 4-tuple comprising the destination, nexthop, metric and its sequence number.The destination and nexthop are node addresses, represented by the NodeId. We represent the NodeId as a positive integer up to the maximum number of nodes in the MANET (see line 4 in Fig. 5). The metric field is a little more interesting. It is normally the hopcount, the number of nodes that need to be traversed to reach the destination. This can be represented as a non-negative integer. However, when a link with a neighbour is considered to be down, the metric that is used is ∞. Thus the metric is a union of the hopcount and ∞ (represented as *infinity*) as seen in line 7 in Fig. 5. The sequence number can also be represented as a non-negative integer. In Fig. 5, SeqNr is a product set, including the identity of the destination originating the

sequence number, and the value of this sequence number. This corresponds to its description in DSDV.

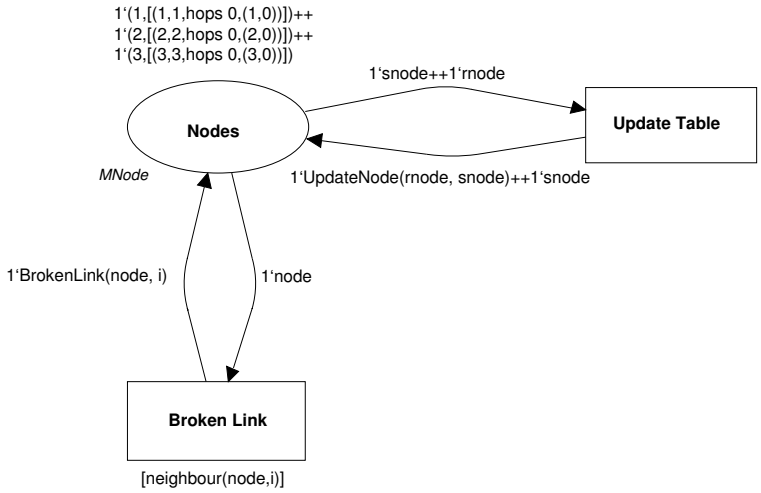

Fig. 4. CPN Model of a MANET

We consider a configuration of 3 nodes in the MANET for our initial experiments (see line 1 of Fig. 5). Nodes is thus initially marked by 3 nodes (1, 2 and 3), where each routing table just has a single entry to itself. This corresponds to **Table 1**.

In line 2 of Fig. 5, *UpdateSeq* is an ML reference variable [27]. It is used as a flag to indicate if a sequence number is to be updated by the functions described later in this section.

```
1  val MaxNodes = 3;
2  val UpdateSeqNr = ref false;
3
4  color NodeId = int with 1..MaxNodes;
5  color Destination = NodeId;
6  color Nexthop = NodeId;
7  color Metric = union hops:Hopcount + infinity declare of_hops;
8  color Hopcount = int;
9  color Number = int;
10 color SeqNr = product Destination * Number;
11 color RTEntry = product Destination * Nexthop * Metric * SeqNr;
12 color RT = list RTEntry;
13 color MNode = product NodeId * RT;
14
15 var i: NodeId;
16 var snode, rnode, node: MNode;
```

Fig. 5. Global declarations of the CPN model

Transition Update Table models the process of a node updating its routing table based on the information it receives from another node. The arc inscription from place Nodes to this transition has two variables, *snode* and *rnode*, which represent two arbitrary nodes in the MANET. In our model, *rnode* receives the routing information broadcast by *snode*. The arc inscription from Update Table to the place Nodes includes a function, UpdateNode(rnode, snode). This function returns the updated routing table of *rnode*, while *snode*'s routing table is maintained unchanged when Update Table occurs. This is where we abstract from the

DSDV protocol mechanisms. We interpret the occurrence of Update Table to mean that any node, *rnode*, can update its table, based on the information from another node, *snode*, at any time. Thus if Update Table occurs, *rnode* must have been within the transmission range of *snode*. If Update Table does not occur, then either the update was not sent by *snode* or when *snode* broadcasts the update *rnode* was out of its transmission range. Update Table conceptually implements both full dump and incremental updates, as the information that is in the incremental update is the only information that has any affect when updating the routing table. So although clearly the full information is available, because only the incremental information is used, the incremental update is modelled. This is the case because we do not model the messages explicitly. So at our level of abstraction the two mechanisms are not distinguishable. However, when a certain scenario is created by executing transitions, then it can be interpreted as incremental or full dump as is appropriate.

Transition Broken Link models a node detecting that it has not received an update from another node within the expected time, and updating its routing table accordingly. This can occur for any node and for any one of its neighbours. On the occurrence of Broken Link, a node (bound to the variable *node*) and one of its neighbours (i) are chosen arbitrarily. The guard neighbour(node,i) ensures that the destination in the route entry of *node* that corresponds to i does have a metric (hopcount) of 1 (which indicates that it is a neighbour). The function BrokenLink(node,i) updates *node*'s routing table accordingly.

After Broken Link occurs, DSDV requires that an incremental update is broadcast. This broadcast may or may not be received by the nodes in the MANET. If no node receives the broadcast, then conceptually this is modelled by Update Table not occurring with *snode* bound to the node that has just updated its table for the broken link. On the other hand if a node does receive the incremental broadcast, then this is modelled by Update Table occurring with that node binding to variable *rnode*, and the node that is broadcasting, binding to *snode*. The effect of the function UpdateNode is the same whether or not it is a full dump or an incremental update. Thus transition Update Table represents the behaviour of updating a node's routing table, irrespective of whether it receives a full dump or an incremental update.

3.3 Functions for Update Table

All functions needed for updating the routing table of a node are described in this subsection (see Listing 1.1 and Listing 1.2). The main function is UpdateNode(rnode, snode) shown in lines 1-9 of Listing 1.1. By this function, a node, *rnode*, updates its routing table (including its own sequence number) based on the routing information sent by another node, *snode*. This function contains a local declaration, *Let/in/end*. Using it, one value declaration binds a value returned by $UpdateSeqNr$, a ref-variable with initial value $false$ (see line 3), and the other one binds a value returned by the function, UpdateRT() (see lines 1-34 in Listing 1.2), to an identifier variable *updated*. In line 6, the value of $UpdateSeqNr$ is checked. If it is *true*, the sequence number of *rnode* is increased by 2 using function UpdateOwnRT() (see lines 17-20 in Listing 1.1). Then the updated routing table with the node's ID (obtained from function GetNodeId(), see lines 11-12) is returned (see line 7). Otherwise, *rnode*'s identity and the value contained in *updated* (in lines 4-5) is returned directly (see line 8). The function UpdateNode(rnode, snode) involves the whole procedure of updating the routing table of *rnode*, using two steps as follows.

1. **First Step:** the *rnode* updates its routing table based on the routing information broadcast by *snode*. This step is realised by function UpdateRT().
2. **Second Step:** the *rnode* determines whether or not to increase its own sequence number. This step is done by checking the value of $UpdateSeqNr$.

Listing 1.1. Function UpdateNode

```
1  (*--rnode updates its routing table based on information broadcast by snode--*)
2  fun  UpdateNode(rnode,snode)=
3  let  val _ = (UpdateSeqNr:=false)
4       val updated =  UpdateRT(GetNodeId(rnode),GetRTNode(rnode),
5                               GetNodeId(snode),GetRTNode(snode))
6  in   if(!UpdateSeqNr)
7       then (GetNodeId(rnode),UpdateOwnRT(updated))
8       else (GetNodeId(rnode),updated)
9  end;
10
11 (*--to get a node's identity--*)
12 fun GetNodeId(n,rt)= n;
13
14 (*--to get a node's routing table--*)
15 fun GetRTNode(n,rt)= rt;
16
17 (*--to update a node's own sequence number--*)
18 fun UpdateOwnRT(n,rte::rt)= if OwnRTEntry(n,rte)
19                           then (IncreaseSeqNr(rte))::rt
20                           else rte::UpdateOwnRT(n,rt);
21
22 (*--to find the route entry to the node itself--*)
23 fun OwnRTEntry(n,(des1,next1,metr1,seqnr1)) = (n = des1);
24
25 (*--to increase the value of sequence number by 2--*)
26 fun IncreaseSeqNr(des1,next1,metr1,(des2,num1))= (des1,next1,metr1,(des2,num1+2));
```

Now, we focus on the operations in the first step. Functions designed for this step are shown in Listing 1.2. In lines 36-41, function AddRouteEntry() realises adding new route entries. In line 44, the function hopnumbers() returns an integer value that is of type Hopcount from the union type Metric (see line 7 in Fig. 5). In line 47, function add1() increments the metric that is of type Hopcount by 1. As shown in lines 1-34, the main function UpdateRT() involves the whole procedure of updating the routing table of a node based on the information sent by another node. The node compares each route entry in this information with the corresponding one in its routing table. To give some insight into the operation of this function we illustrate the procedure for updating a route entry in Fig. 6. This is then applied recursively for each route entry (see lines 5-34 of Listing 1.2). For convenience, in our model, all route entries are listed in ascending order of the destination's node identifier (address), to simply the procedure. This has the added benefit reducing state space explosion that would otherwise occur due to arbitrary ordering of the list.

As shown in Fig. 6, the receiver is represented as $(mnid, [(md, mn, mm, (md1, mseq)) :: mrt])$, in which $mnid$ represents the node identity and $(md, mn, mm, (md1, mseq))$ represents the route entry (also represented as $entry1$), currently listed in the first position in its routing table. The rest of the route entries are represented by mrt. Similarly, the sender is represented as $(n, [(nd, nn, nm, (nd1, nseq)) :: rt])$. Its identity is n, and the route entry listed in the first position in its routing table is $(nd, nn, nm, (nd1, nseq))$ (also represented as $entry2$). The metrics of $entry1$ and $entry2$ are represented as mm and nm respectively. In Fig. 6, $mm{:}int$ or $nm{:}int$ means that the metric is of type Hopcount, i.e. an integer. $mm{:}infinity$

Listing 1.2. Function UpdateRT

```
(*--a node updates its route entries based on another node --*)
fun UpdateRT(mnid,[],n,[])= []
| UpdateRT(mnid,[],n,rte::rt)=(UpdateSeqNr:=true; AddRouteEntry(mnid,n,rte::rt))
| UpdateRT(mnid,mrte::mrt,n,[])=  mrte::mrt
| UpdateRT(mnid,(md,mn,mm,(md1,mseq))::mrt,n,(nd,nn,nm,(nd1,nseq))::rt)=
if       md=nd
then
  if     (mseq>nseq)
  then   (md,mn,mm,(md1,mseq))::UpdateRT(mnid,mrt,n,rt)
  else
    if   (mseq=nseq)
    then case (of_hops'Metric(mm)) of
  (true) =>if   (hopnumbers(mm) <= hopnumbers(nm))
           then (md,mn,mm,(md1,mseq))::UpdateRT(mnid,mrt,n,rt)
           else (UpdateSeqNr:=true;(nd,n,add1(nm),(nd1,nseq))::UpdateRT(mnid,mrt,n,rt))
 |(false)=>(md,mn,mm,(md1,mseq))::UpdateRT(mnid,mrt,n,rt)
    else case ((of_hops'Metric(mm)),(of_hops'Metric(nm))) of
 ((true),(true))=>if  (mm = add1(nm))
                  then (nd,n,add1(nm),(nd1,nseq))::UpdateRT(mnid,mrt,n,rt)
                  else (UpdateSeqNr:=true;(nd,n,add1(nm),(nd1,nseq))
                       ::UpdateRT(mnid,mrt,n,rt))
 |((true),(false)) => (UpdateSeqNr:=true;(nd,n,nm,(nd1,nseq))::UpdateRT(mnid,mrt,n,rt))
 |((false),(true)) => (UpdateSeqNr:=true;(nd,n,add1(nm),(nd1,nseq))
                       ::UpdateRT(mnid,mrt,n,rt))
 |((false),(false))=> (nd,n,nm,(nd1,nseq))::UpdateRT(mnid,mrt,n,rt)
else
  if      md<nd
  then   (md,mn,mm,(md1,mseq))::UpdateRT(mnid,mrt,n,(nd,nn,nm,(nd1,nseq))::rt)
  else
    if   of_hops'Metric(nm)
    then (UpdateSeqNr:=true;(nd,n,add1(nm),(nd1,nseq))
         ::UpdateRT(mnid,(md,mn,mm,(md1,mseq))::mrt,n,rt))
    else (UpdateSeqNr:=true;(nd,n,nm,(nd1,nseq))
         ::UpdateRT(mnid,(md,mn,mm,(md1,mseq))::mrt,n,rt));

(*--a node with identity m adds new route entries based on another node--*)
fun AddRouteEntry(m,n,[])=[]
| AddRouteEntry(m,n,(des1,nexthop1,metric1,seqnr1)::rtm)=
if  of_hops'Metric(metric1)
then (des1,n,add1(metric1),seqnr1)::AddRouteEntry(m,n,rtm)
else (des1,n,metric1,seqnr1)::AddRouteEntry(m,n,rtm);

(*--to get the value of the metric--*)
fun hopnumbers(hops metr) = metr;

(*--to increase the metric by 1--*)
fun add1(hops metr)= let val M = metr+1
                     in  hops M
                     end;
```

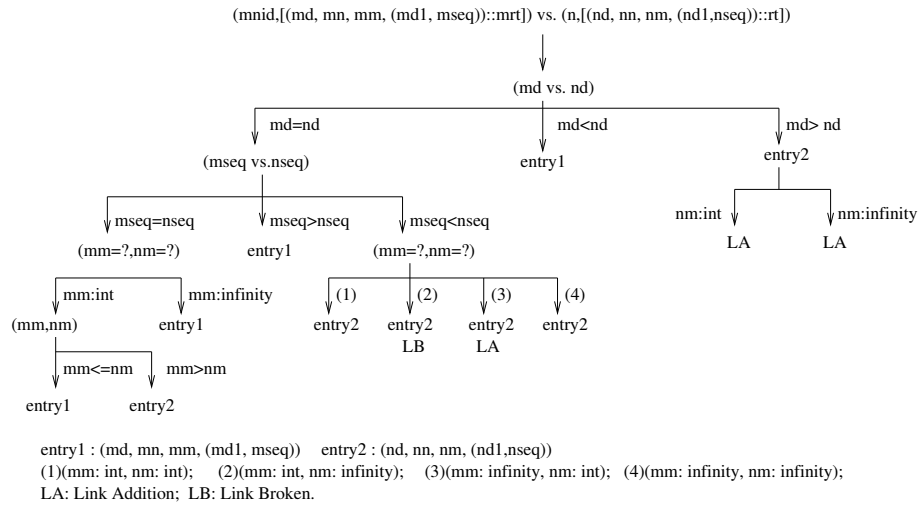

Fig. 6. Procedure for updating a route entry

or $nm{:}infinity$ means that the metric is ∞ (represented as $infinity$ in Fig. 6), namely the corresponding entry is a broken link. The purpose of $UpdateSeqNr$ is to track changes to the routing table of the receiver during updating. The value of $UpdateSeqNr$ is set to $true$, whenever an important change happens, such as a link addition (LA), a link breakage (LB) or metric change, as shown in Fig. 6. In Fig. 6, if $entry1$ occurs under an arrow, it means the receiver does not change its route entry. Otherwise, the route entry sent by the sender, $entry2$, is used to update the receiver's route entry. In this case, if the metric of $entry2$ is of type Hopcount, the receiver increases this metric by 1 by function add1(), and includes the sender's node identifier as the next hop of the entry. If the metric is ∞, the receiver just adds this entry without any change. Now, a comparison between two route entries, $(md, mn, mm, (md1, mseq))$ and $(nd, nn, nm, (nd1, nseq))$, is described as follows.

1. If two entries are to the same destination, $md = nd$ (see lines 6-25 in Listing 1.2). The receiver then compares their sequence numbers, $mseq$ and $nseq$. If $mseq > nseq$, the receiver will choose $entry1$. If $mseq = nseq$, that means the two metrics are of the same type. Thus both entries are available or both are broken. In the first case (corresponding to lines 13-15 in Listing 1.2), the receiver will select the route entry with the shorter metric. If $entry2$ is chosen, the value of $UpdateSeqNr$ is set to $true$ because the metric has changed. If they have the same metric, the receiver arbitrarily chooses one. In our model, $entry1$ is chosen for convenience. In the second case, the receiver can arbitrarily choose, and $entry1$ is chosen for convenience. Otherwise, in the case $mseq < nseq$, the receiver always selects $entry2$ instead of $entry1$. While according to the metrics of two entries, there also are four possible conditions (corresponding to lines 22-26 in Listing 1.2): in (1), $entry2$ is of type Hopcount, so the receiver deals with this available entry. If the metric changes, the value of $UpdateSeqNr$ is set to $true$. In (2), $entry1$ is an available entry but $entry2$ is a broken one. So it is a LB. In (3), $entry1$ is a broken entry but $entry2$ is an available one. So it is a LA. In (4), $entry2$ is a broken link, so the receiver adds it unchanged.
2. If two entries are to different destinations, and $md < nd$ (see lines 27-28 in Listing 1.2). That means there is no route entry to destination md in the routing information broadcast by the sender. So the receiver keeps $entry1$ unchanged.
3. Otherwise, if $md > nd$ (see lines 29-34 in Listing 1.2), which means that the route entry to destination nd (i.e. $entry2$) is a new route entry for the receiver. Both conditions can be taken as LAs, even if $entry2$ in the second condition is a broken link.

The procedure described above is used for the next route entries in both routing tables. This procedure is repeated until at least all route entries in one of the routing tables have been processed. In this case, if some route entries listed in the receiver's routing table have not been compared, the receiver will keep them unchanged (see line 4 in Listing 1.2). In line 3 in Listing 1.2, some route entries listed in the sender's routing table are not in the receiver's routing table. In this case, the receiver will add them (i.e. LAs) using function AddRouteEntry() (see lines 36-41 in Listing 1.2). If all route entries of both routing tables have been processed, the computation stops and an empty list is returned (see line 2 in Listing 1.2).

3.4 Functions for Broken Link

The functions associated with the transition Broken Link are given in Listing 1.3. Lines 1-6 define the function neighbour(node,i), the guard of transition Broken Link in Fig. 4. It ensures that a node, *node*, detects that the link to one neighbour i is down. Function check() (see lines 14-19 in Listing 1.3) describes the metric of each broken link as ∞ (represented as $infinity$ in functions), and increases the sequence number of such a link by 1 when its next hop is i. Then, the updated routing table of this node is returned, and this value is bound to an identifier *Broken* in line 10 of the main function BrokenLink(node, i) (see lines 8-12 of Listing 1.3). In line 11, the node increases its sequence number by function UpdateOwnRT(), and then a node with its IP address and updated routing table is returned.

Listing 1.3. Function BrokenLink

```
1  (*--to ensure a neighbour with IP address i --*)
2  fun neighbour((n,[]),i)= false
3  | neighbour((n,(des1,next1,metr1,(des2,num1))::rt),i)=
4  if (i=des1) andalso (of_hops'Metric(metr1)) andalso (hopnumbers(metr1)=1)
5  then true
6  else neighbour((n,rt),i);
7
8  (*-- a node deals with broken links and updates its sequence number--*)
9  fun BrokenLink(node,i)=
10 let val Broken = check(i,GetNodeId(node),GetRTNode(node))
11 in (GetNodeId(node),UpdateOwnRT(GetNodeId(node),Broken))
12 end;
13
14 (*--to deal with broken links--*)
15 fun check(m,n,[])=[]
16 | check(m,n,(des1,next1,metr1,(des2,num1))::rt)=
17 if (m=next1)
18 then (des1,next1,infinity,(des2,num1+1))::check(m,n,rt)
19 else (des1,next1,metr1,(des2,num1))::check(m,n,rt);
```

4 Simulation of the CPN Model

In order to provide some insight into the operation of our CPN model, we consider the following simulation of its behaviour. The number of nodes in the MANET is governed by the initial marking of the CPN model in Fig. 4 (i.e. the marking of place Nodes), and can

be extended easily. Here, we just consider the operation of the CPN with 3 nodes. Suppose their addresses are represented as 1, 2 and 3 respectively. As shown in Fig. 4, there are two transitions, Update Table and Broken Link, in the CPN model. In order to explain the simulation of the CPN explicitly, we describe the occurrence of transition Update Table: $(UpdateTable, < rnode = a, snode = b >)$, in which $rnode$ and $snode$ are variables in the arc expression of the input arc to this transition, and a and b are values bound to $rnode$ and $snode$ respectively. As described in section 3.2, $rnode$ represents a node that can receive and update its routing table based on the routing information sent by another node, $snode$. For example, $(UpdateTable, < rnode = 1, snode = 3 >)$ means that when transition Update Table occurs, node 1 is bound to $rnode$ and node 3 is bound to $snode$. Thus, node 1 updates its routing table based on the routing information it received from node 3. Analogously, the occurrence of transition Broken Link is depicted as: $(BrokenLink, < node = c, i = d >)$, in which $node$ is the variable in the arc expression of the input arc to this transition, and c is a value assigned to $node$. Here, $node$ represents a node which detects and deals with the broken links to a neighbour i, which is ensured by the guard neighbour(node,i). For example, $(BrokenLink, < node = 1, i = 2 >)$ means that when transition Broken Link occurs, node 1 is bound to $node$ and 2 is bound to i. So node 1 detects that it has not received an update from a former neighbour, node 2, within the expected time, so it makes all route entries through node 2 as broken links. Now, the steps that occur in a execution of the CPN model are depicted as follows, and the markings reached are given in Fig. 7.

```
-------------------------------------------------------------------------
(1)       1`(1,[(1, 1, hops(0), (1, 0))])++
          1`(2,[(2, 2, hops(0), (2, 0))])++
          1`(3,[(3, 3, hops(0), (3, 0))])
-------------------------------------------------------------------------
(2)       1`(1,[(1, 1, hops(0), (1, 0))])++
          1`(2,[(2, 2, hops(0), (2, 2)), (3, 3, hops(1), (3, 0))])++
          1`(3,[(3, 3, hops(0), (3, 0))])
-------------------------------------------------------------------------
(3)       1`(1,[(1, 1, hops(0), (1, 0))])++
          1`(2,[(2, 2, hops(0), (2, 2)), (3, 3, hops(1), (3, 0))])++
          1`(3,[(1, 1, hops(1), (1, 0)), (3, 3, hops(0), (3, 2))])
-------------------------------------------------------------------------
(4)       1`(1,[(1, 1, hops(0), (1, 0))])++
          1`(2,[(2, 2, hops(0), (2, 4)), (3, 3, infinity, (3, 1))])++
          1`(3,[(1, 1, hops(1), (1, 0)), (3, 3, hops(0),  (3, 2))])
-------------------------------------------------------------------------
(5)       1`(1,[(1, 1, hops(0), (1, 0))])++
          1`(2,[(2, 2, hops(0), (2, 4)), (3, 3, infinity, (3, 1))])++
          1`(3,[(1, 1, infinity, (1, 1)), (3, 3, hops(0), (3, 4))])
-------------------------------------------------------------------------
(6)       1`(1,[(1, 3, infinity, (1, 3)), (3, 3, hops(1), (3, 4))])++
          1`(2,[(2, 2, hops(0), (2, 4)), (3, 3, infinity, (3, 1))])++
          1`(3,[(1, 1, infinity, (1, 1)), (3, 3, hops(0), (3, 4))])
-------------------------------------------------------------------------
(7)       1`(1,[(1, 3, infinity, (1, 3)), (3, 3, hops(1), (3, 4))])++
          1`(2,[(2, 2, hops(0), (2, 4)), (3, 3, infinity, (3, 1))])++
          1`(3,[(1, 1, infinity, (1, 3)), (3, 3, hops(0), (3, 4))])
-------------------------------------------------------------------------
```

Fig. 7. Markings of the CPN during the simulation

The initial marking is given in Fig. 7(1). For example, $(1, [(1, 1, hops(0), (1, 0))])$ is a pair comprising the identity and routing table of node 1 (see line 13 in Fig. 5). In its routing table,

there is only one route entry $(1, 1, hops(0), (1, 0))$. According to line 11 in Fig. 5, we know the destination and the next hop both are node 1, the metric is 0, which is of type Hopcount (see line 7 in Fig. 5), and $(1, 0)$ means the sequence number is originated by node 1 and its value is 0.

$(UpdateTable, < rnode = 2, snode = 3 >)$ occurs, which means node 2 receives the routing information sent by node 3, $[(3, hops(0), (3, 0))]$ (see section 2). Node 2 adds this route entry to its routing table, after increasing the metric by 1 and adding node 3 as the next hop of this entry. Because it is a link addition (see Fig. 6), node 2 increments its sequence number by 2. The marking reached is presented in Fig. 7(2).

After that, $(UpdateTable, < rnode = 3, snode = 1 >)$ occurs, which means node 3 receives the routing information, $[(1, 1, hops(0), (1, 0))]$ sent by node 1. Node 3 adds a new entry to node 1 into its routing table and increases its own sequence number by 2. The marking reached is shown in Fig. 7(3).

Then $(BrokenLink, < node = 2, i = 3 >)$ occurs. That means node 2 does not receive any information from a neighbour, node 3, for a while, so it infers the link to node 3 is broken. Node 2 assigns $infinity$ as the metric of this route entry and increases the sequence number of this entry by 1. Because it is a link breakage, node 2 increases its own sequence number by 2. The marking after this step is given in Fig. 7(4).

Next, $(BrokenLink, < node = 3, i = 1 >)$ occurs. Similarly, node 3 deals with the broken link to node 1 and increases its own sequence number as well. The marking is shown in Fig. 7(5).

$(UpdateTable, < rnode = 1, snode = 3 >)$ occurs. Node 1 receives the routing information, $[(1, 1, infinity, (1, 1)), (3, 3, hops(0), (3, 4))]$ sent by node 3. For two route entries to the same destination, a node always selects the one with the higher sequence number. In this case, there are two route entries to node 1, one containing the routing information sent by node 3, and the other kept in the routing table of node 1. Node 1 selects the former one because it has the higher sequence number, 1. Because a broken link, $(1, 1, infinity, (1, 1))$, replaces the available one, $(1, 1, hops(0), (1, 0))$, it can be taken as link removal. Node 1 adds the route entry to node 3 into its routing table, i.e. a link addition. Then node 1 increases its own sequence number from 1 to 3. The marking reached is shown in Fig. 7(6).

$(UpdateTable, < rnode = 3, snode = 1 >)$ now occurs. Node 3 receives the routing information, $[(1, 3, infinity, (1, 3)), (3, 3, hops(1), (3, 4))]$ sent by node 1. For two route entries to node 1, node 3 selects the one received, because it has higher sequence number 3. Node 3 keeps the route entry to itself in its routing table unchanged, because the two entries to node 3 have the same sequence number 4. The marking reached is shown in Fig. 7(7).

5 Discussion

The execution sequence described above shows that the CPN model can simulate the updating of routing tables in an environment where the topology of the MANET can change dramatically. In this section we illustrate how this execution sequence can be related to events in the MANET, and also analyse the trace by considering the validity of the routing table entries shown in the markings of Fig. 7.

A scenario in the MANET that relates to the execution sequence in the previous section is shown in Fig. 8. The dashed circles include the node that is broadcasting and which nodes receive the broadcast. The solid arrows indicate that the sender broadcasts a *full dump*, and the dashed arrows indicate that the sender broadcasts an *incremental* update. We assume that nodes broadcast their *full dump* in the order: node 3, node 1 and node 2. Initially, the nodes just have their own entries as given by the marking of Fig. 7(1).

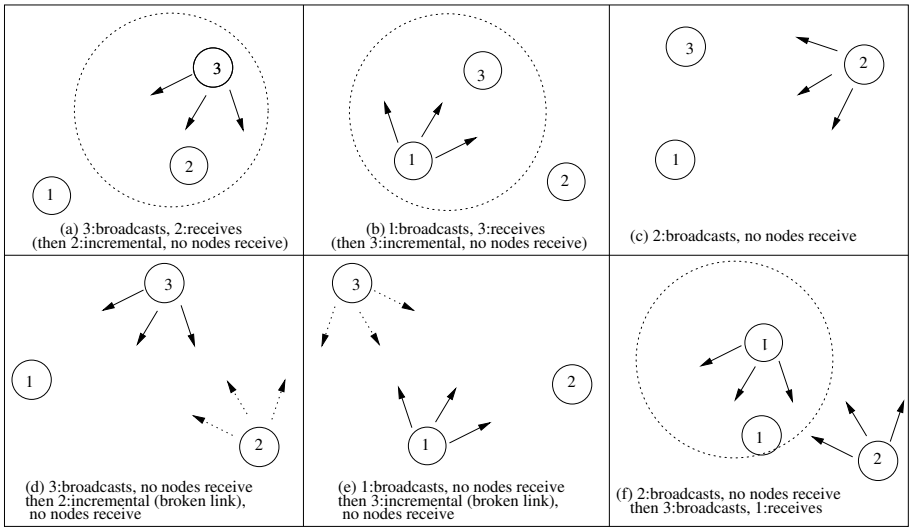

Fig. 8. An illustration of the simulation in section 4

As shown in Fig. 8**(a)**, node 3 broadcasts and node 2 receives and updates its routing table. The updated routing tables are in Fig. 7(2). Node 2 then broadcasts an incremental update, but unfortunately no nodes receive it as they are out of range. This is not shown in Fig. 8. Sometime later (see Fig. 8**(b)**), node 1 broadcasts and node 3 receives and updates its routing table. The updated routing tables of the nodes now correspond to Fig. 7(3). Node 3 now broadcasts an incremental update but it is not received (not shown in the Fig.). Later, as shown in Fig. 8**(c)**, node 2 broadcasts but unfortunately, the other nodes are still too far away to receive it. Similarly, in Fig. 8**(d)**, node 3 broadcasts, but no other nodes receive it. Node 2 has not received any information from node 3 for too long, so it declares this link broken, and immediately broadcasts this route change but no nodes receive it. The updated routing tables at this time are shown in Fig. 7(4). In Fig. 8**(e)**, node 1 broadcasts to no effect. Node 3 judges that the link to node 1 is broken, and immediately broadcasts this broken link in the MANET, but no others receive it. The updated routing tables of the nodes at this stage are shown in Fig. 7(5). Now node 2 broadcasts again, but to no avail (Fig. 8**(f)**). Then node 3 broadcasts and node 1 receives and updates its routing table. The updated routing tables of the nodes now correspond to the marking in Fig. 7(6).

We find that node 1 has not communicated with the others for a while since Fig. 8**(b)**, so its sequence number is not updated. Then in Fig. 8**(e)** node 3 considers that its link to node 1 is broken, so it increases the sequence number of this link to 1. Therefore, there are two route entries to node 1, one in routing table of node 1 being $(1, 1, hops(0), (1, 0))$, and the other in routing table of node 3, $(1, 1, infinity, (1, 1))$. So node 3 has a higher sequence number for node 1 than that of node 1 itself. Thus node 1 updates its own routing entry with an incorrect routing entry from node 3.

The fact that a higher sequence number for a node can occur in a node other than the node itself leads to incorrect routing entries being created in DSDV. We classify these errors caused by this fact into two categories as follows.

1. **A node updates the route entry to itself based on information from another node.** Intuitively, the metric of the route entry to a node itself is 0 and the next hop is the node itself. However, in Fig. 7(6), we find a route entry of $(1, 3, infinity, (1, 3))$ in the routing table of node 1. That means the link from node 1 to itself is broken and should be via another node, node 3. This is obviously wrong.

Listing 1.4. Modified function UpdateRT

```
1  (*--a node updates its route entries based on another node --*)
2  fun UpdateRT(mnid,[],n,[])= []
3  |   UpdateRT(mnid,[],n,rte::rt)=(UpdateSeqNr:=true; AddRouteEntry(mnid,n,rte::rt))
4  |   UpdateRT(mnid,mrte::mrt,n,[])=  mrte::mrt
5  |   UpdateRT(mnid,(md,mn,mm,(md1,mseq))::mrt,n,(nd,nn,nm,(nd1,nseq))::rt)=
6  if  md=nd
7  then
8    if md=mnid
9    then (md,mn,mm,(md1,mseq))::UpdateRT(mnid,mrt,n,rt)
10   else
11     if mseq>nseq
12     then
13       if (md=n) andalso (not(of_hops'Metric(mm))) andalso ((of_hops'Metric(nm))
14       then (UpdateSeqNr:=true;(nd,n,add1(nm),(nd1,nseq))::UpdateRT(mnid,mrt,n,rt))
15       else (md,mn,mm,(md1,mseq))::UpdateRT(mnid,mrt,n,rt)
16 else
17   if (mseq=nseq)
18   ...
```

2. **A node cannot update the broken link to another node even on receiving a broadcast from this node.** Now consider that node 3 receives a broadcast from node 1. Node 3 should update the broken link to node 1 based on the route entry sent by node 1, increase its sequence number, and immediately broadcast this routing change. Unfortunately, node 3 does not update this broken link at all because of the value of the sequence number. The route entry sent by node 1 has the sequence number $(1, 0)$, whereas the broken link to node 1 contained in the routing table of node 3 has a higher sequence number $(1, 1)$.

According to DSDV, more recent route entries should have higher sequence numbers in the routing table. Hence, nodes can distinguish between current and obsolete route entries by comparing the values of their sequence numbers. For this mechanism to be correct, each node should keep its own sequence number as the most current one. Unfortunately, DSDV does not guarantee this point in the general environment of a MANET.

In order to avoid these errors, we modify DSDV by changing the way routing tables are updated. A correction is made to function UpdateRT() (described in subsection 3.3), and thus there is no need to alter the structure of the CPN model given in Fig. 4. The modified function is presented in Listing 1.4 corresponding to lines 1-11 in Listing 1.2.

1. To avoid the first kind of error: a node keeps the route entry to itself unchanged when updating its routing table, as shown in lines 8-9 in Listing 1.4. It only updates its sequence number attached in this route entry when needed in function UpdateNode() (see line 7 of Listing 1.1).
2. To avoid the second kind of error: a node receives routing information from another node, if it already has a route entry to this sender, it will update this route entry regardless of the value of the sequence number, because the route information it just receives is more current. This is implemented in lines 13-15 of Listing 1.4: the receiver receives an available route entry to the sender and updates the broken one to this sender with the new one, even if the new one has a lower sequence number.

The rest of the function is identical to that in Listing 1.2, so it is not included in Listing 1.4. After simulating the modified CPN model, we find that both kinds of errors are eliminated. For example, given the routing tables of the nodes in Fig. 7(5), if $(UpdateTable, < rnode = 1, snode = 3 >)$ occurs (see section 4), node 1 updates its routing table according to the information broadcast by node 3. It keeps the route entry to itself unchanged and adds the route entry to node 3 as a new one, and updates its sequence number, so its updated routing table is: $[(1, 1, hops(0), (1, 2)), (3, 3, hops(1), (3, 4))]$. Whereas, if $(UpdateTable, < rnode = 3, snode = 1 >)$ occurs, node 3 updates its routing table according to the information broadcast by node 1. Regardless of the value of the sequence number, it updates the broken link to node 1, and updates its sequence number, so its updated routing table is: $[(1, 1, hops(1), (1, 0)), (3, 3, hops(1), (3, 4))]$.

6 Conclusions and Future Work

This paper demonstrates the feasibility of using CPNs to faithfully model routing protocols of MANETs given their dynamically changing network topologies. We present the first abstract CPN model for a MANET based on DSDV. Although the model looks deceptively simple, it not only allows for arbitrary changes in topology but also relaxes the assumption that nodes have the same transmission ranges. This allows us to model MANETs in which the nodes are heterogeneous (e.g. a combination of PDAs, notebooks and mobile phones). Our results show that the CPN model captures the highly dynamic topology of such a network, something that has been considered a difficult problem by others [29]. Further, although the model is abstract, it has sufficient detail for us to find errors in the DSDV procedures using simulation.

Two categories of errors have been found. The first is that it is possible for a node to wrongly update its own route entry, replacing its metric of a hopcount of zero (which must always be the case) with *infinity* and directing packets destined for itself to another node. This is a serious error. The second error is that it is possible for a node with a broken link entry for another node to not re-establish the link with that node, even though it has received a broadcast from that node. This is due to incorrect handling of sequence numbers. This is the first time these errors have been discovered in DSDV as far as we are aware. We also suggest modifications to the routing table updating procedures to eliminate these errors. Our simulations have confirmed their effectiveness.

In this paper, we have not attempted to analyse our model using state spaces because the state space is infinite due to the sequence number being unbounded. This faithfully reflects the DSDV specification. However, unbounded sequence numbers are impractical, so we plan to investigate state space analysis of this model when the sequence number space is limited. Further, we would like to enhance the model in two ways by including: a) the install time parameter; and b) nodes powering down and rejoining the MANET. More generally, it may be possible to use the model as a platform to study proactive routing protocols and distance-vector routing algorithms [20] used in MANETs.

Acknowledgements

The authors would like to acknowledge the assistance of their colleagues, Guy Gallasch, Somsak Vanit-Anunchai and Lin Liu, for their assistance with some of the detailed technical parts of the model. We also gratefully acknowledge the constructive comments of the reviewers on this paper. Cong Yuan is supported by an Australian Government International Postgraduate Scholarship.

References

1. S. Basagni, M. Conti, S. Giordano, and I. Stojmenovic. Mobile Ad Hoc Networking. IEEE Press, New York, 2004.
2. E. Belding-Royer. Routing Approaches in Mobile Ad Hoc Networks. In S. Basagni, M. Conti, S. Giordano, and I. Stojmenovic, editors, *Mobile Ad Hoc Networking*. IEEE Press, New York, 2004.
3. R. E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
4. K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal Verification of Standards for Distance Vector Routing Protocols. *Journal of the ACM (JACM)*, 49(4):538–576, July 2002.
5. J. Billington. Many-sorted High-Level Nets. In *the Third International Workshop on Petri Nets and Performance Models*, pages 11–13, Kyoto, Japan, December 1989. IEEE CS Press, Washington, D. C., USA, 1989.
6. C. Cheng, R. Riley, S.P.R Kumar, and J.J Garcia-Luna-Aceves. A Loop-Free Bellman-Ford Routing Protocol without Bouncing Effect. In *ACM SIGCOMM 1989*, pages 224–237, September 1989.
7. G. Findlow and J. Billington. High-Level Nets for Dynamic Dining Philosophers Systems. In M.Z. Kwiatkowska, M.W. Shields, and R.M. Thomas, editors, *Semantics for Concurrency*, pages 185–222. Springer-Verlag, 1990.
8. L. R. Jr. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
9. J.J Garcia-Luna-Aceves. A Unified Approach to Loop-Free Routing Using Distance Vectors or Link States. In *ACM SIGCOMM 1989*, pages 212–223, September 1989.
10. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
11. G. J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.
12. J. M. Jaffe and F. Moss. A Responsive Distributed Routing Algorithm for Computer Networks. In *IEEE Transaction on Communications COM-30(7)*, pages 1758–1762. July 1982.
13. K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1–3. Springer-Verlag, 1997.
14. L. M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
15. L. M. Kristensen and K. Jensen. Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad Hoc Networking. In *INT 2004, LNCS 3147*, pages 248–269. Springer-Verlag, 2004.
16. L. M. Kristensen, J. B. Jørgensen, and K. Jensen. Application of Coloured Petri Nets in System Development. In *Lectures on Concurrency and Petri nets - Advanced in Petri Nets.Proc.of 4th Advanced Course on Petri Nets.Vol.3098 of Lecture Notes in Computer Science*, pages 626–685. Springer-Verlag, 2004.
17. K. G. Larsen, P. Pettersson, and Y. Wang. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, October 1997.
18. Jennifer J. N. Liu and I. Chlamtac. Mobile Ad Hoc Networking with a View of 4G Wireless: Imperatives and Challenges. In S. Basagni, M. Conti, S. Giordano, and I. Stojmenovic, editors, *Mobile Ad Hoc Networking*, pages 1–45. IEEE Press, New York, 2004.
19. G. Malkin. RIP Version 2- Carrying Additional Information. RFC 1723(draft standard). Technical report, Internet Engineering Task Force, November 1994.
20. G. S. Malkin and M. E. Steenstrup. Distance-Vector Routing. In *Routing in Communications Networks*, pages 83–98. Prentice-Hall, 1995.
21. P. M. Merlin and A. Segall. A Failsafe Distributed Routing Protocol. In *IEEE Transactions on Communications COM-27(9)*, pages 1280–1287. September 1979.
22. D. Obradovic. *Formal Analysis of Routing Protocols*. PhD thesis, University of Pennsylvania, 2002.
23. C. E. Perkins, E. Belding-Royer, and S. Das. *Ad-hoc On-Demand Distance Vector (AODV) Routing*. IETF Internet draft, draft-ietf-manet-aodv-09.txt, 2001.
24. C. E. Perkins and P. Bhagwat. Highly Dynamic Destination Sequenced Distance Vector (DSDV) for Mobile Computers. In *ACM SIGCOMM 1994 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, August 1994.
25. C. E. Perkins and P. Bhagwat. *Ad Hoc Networking, Chapter 3: DSDV Routing over a Multihop Wireless Network of Mobile Computers*. Addison-Wesley, 2001.
26. A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, Upper Saddle River, NJ, 4th edition, 2003.
27. J. D. Ullman. *Elements of ML Programming*. Prentice-Hall, 1998.
28. O. Wibling, J. Parrow, and A. Pears. Automatized Verification of Ad Hoc Routing Protocols. In *FORTE 2004, LNCS 3235*, pages 343–358. Springer-Verlag, 2004.
29. C. Xiong, T. Murata, and J. Tsai. Modeling and Simulation of Routing Protocol for Mobile Ad Hoc Wireless Networks Using Colored Petri Nets. In *Proc. Workshop on Formal Methods Applied to Defence Systems in Formal Methods in Software Engineering and Defence Systems, Conferences in Research and Practice in Information Technology*, volume 12, pages 145–153, 2002.