MStack: a communications stack for mobile ad-hoc networks

by

David Fusté Vilella

Thesis Supervisor: Jorge García Vidal
Department of Computer Architecture
Technical University of Catalonia

Submitted to the Department of Computer Architecture in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
at the
Technical University of Catalonia

Barcelona, October 2011

MStack: a communications stack for mobile ad-hoc networks

by

David Fusté Vilella

Submitted to the Department of Computer Architecture on October 2011 in partial fulfillment of the requirements for the degree of Doctor of Philosophy

ABSTRACT

Mobile ad-hoc networks have received considerable attention in the wireless communications research community. Nevertheless, its importance in industry is still low. Self-organized MANETs of smartphones or laptops can be used, for example, in military, public safety and disaster relief, conference, or campus environments. In order to enable communications between nodes inside a MANET, a communications stack is needed in each node of the network. Research in this area was always focused on optimizing the TCP/IP stack, the stack par excellence in the majority of our current networks. Instead of designing a complete new communications stack, researchers focused their work mainly on improving already existing solutions coming from the wired networks. However, we think that a complete and new communications stack specifically designed for MANETs is needed if we want to achieve robust communications in this type of networks, which have properties very different from what wired networks have. In this thesis we present what nowadays is still missing: a novel communications stack specifically designed for mobile ad-hoc networks. The MStack is an example of how some of the basic assumptions and mechanisms used in wired or wireless infrastructure networks must be fundamentally modified when dealing with MANETs. All in all, we believe that the results presented in this thesis provide interesting insights into the potential of the MStack in MANETs.

Thesis Supervisor: Jorge García Vidal
Title: Assistant Professor

Agraïments

Ves per on, hi havia una vegada un noi de Ponts, un petit poble de Lleida. Aquest noi, gràcies als seus pares, va tenir la sort de poder anar a l'escola. Així doncs, va anar passant a través de la guarderia (on va aprendre a odiar el pernil dolç i a estimar a la padrina Ramona i el padrí Mingo), a través de primària (on es va fer un fart de riure amb el Vincen i un fart de barallar-se amb la seva germaneta Ares), a través d'ESO (on gràcies a l'Helena va aprendre que les cantimplores canten i ploren, i gràcies a l'Esther va aprendre que Mystic, en català, és Místic...així de Natural), i a través de batxillerat (on va adoptar els tics matemàtics de la Josefina i on va descobrir l'escalada). Quines coses...

Ves per on, a aquest noi se li donava força bé això d'estudiar. Quan va acabar el batxillerat va rebre matrícula d'honor pels seus estudis. I llavors va arribar el moment de prendre una decisió important. Al noi li van dir que a partir de llavors havia d'escollir, que podia anar a treballar, o que podia seguir estudiant a la universitat. A més, si escollia tirar per la universitat, havia de triar quins estudis en concret realitzar. Quin mal de cap...

Ves per on, el noi, sense massa plantejaments interns, va pensar que donat que això dels ordinadors sempre se li havia donat bé, doncs que lo més fàcil i còmode era anar a la universitat a estudiar informàtica. Quin pal anar a treballar...

Ves per on, va resultar que això d'estudiar, encara que fos ja en un altre lloc, se li continuava donant igual de bé. Al cap d'uns anys va acabar els estudis universitaris, i va ser condecorat com al tercer millor estudiant de la seva promoció. Durant aquests anys però, a part de descobrir moltes coses a la universitat, també va descobrir un dels pilars més importants de la seva vida. La Meritxell, la Glòria, l'Albertet i el Nick van passar a ser la seva segona família. I de nou, va arribar un altre moment de prendre un decisió important. Al noi li van dir que havia de tornar a escollir, podia anar a treballar, o podia seguir estudiant a la universitat, aquest cop mitjançant el doctorat. A més, si escollia el doctorat, havia de triar sobre quin tema basar la seva tesi. Quin mal de cap...

Ves per on, per aquell llavors el noi just acabava de conèixer al Jorge García, el que va ser el seu professor a l'última assignatura de la carrera i el seu director de PFC (que mira tu, també va realitzar amb matrícula d'honor). Donat que el Jorge es dedicava a les xarxes inhalàmbriques, i que al noi, de totes les coses que havia vist durant la carrera, possiblement aquella era la que més li agradava, doncs va pensar que lo més fàcil i còmode era realitzar el doctorat amb el Jorge. Quin pal anar a treballar...

Ves per on, es va tornar a donar el cas que això dels estudis se li seguia donant bé. A mesura que anava passant els cursos de doctorat anava obtenint matrícules d'honor. A més, a part de tot lo referent a la universitat, una altra meravella va tenir lloc a la seva vida. L'Ingrid, la Wadi, els Peques, la Nuri i l'Edgar van passar a ser la seva tercera família. I com no podia ser menys, de nou li va arribar el moment de prendre una nova decisió important. Resulta que el Jorge, juntament amb el Mario i el Danny Nemirovsky, li van proposar de fundar conjuntament una empresa als Estats Units basada en el treball que el Jorge i ell havien dut a terme en la seva tesi durant aquests anys que portaven junts. De repent, el noi va rebre l'oportunitat de fer realitat els seus experiments, i de portar les seves invencions a la vida real. Quin "bon" de cap...

Ves per on, la cosa se li va continuar anant força bé. L'empresa va anar creixent materialment a la vegada que el noi també va anar creixent professionalment. Fins i tot el noi va rebre juntament amb el Jorge un premi menció de qualitat per part de la UPC on es reconeixia la seva tasca en recerca durant l'any 2009. Quin pal anar a estudiar...

Ves per on, gràcies a tot aquest camí, on el noi més que caminar va deambular, resulta que ara aquest noi té un present més que emocionant (com diria el Sebastian, an exciting present!). És per tot això que aquest noi no pot evitar donar les gràcies d'una forma inimaginable a les quatre persones que més han influït en el seu camí, ja que sort en va tenir d'elles quan cada cop que ell es desorientava en el seu camí, aquestes persones el tornaven a posar a lloc. Gràcies Pare, Gràcies Mare, Gràcies Jorge, Gràcies Mario. Vosaltres heu estat els timoners al llarg del meu camí.

Ves per on...

TOC

ABSTRACT 2
Agraïments 3
TOC 5
List of Figures 10
List of Tables 13
Publications derived from the Thesis 14
Patents derived from the Thesis 15
1. Introduction 16
2. State of the Art 17
2.1 Open System Interconnection 17
2.2 TCP/IP DARPA Stack 19
2.3 WAP (Wireless Application Protocol) 20
2.4 Why do we need a new communications stack for MANETs? 21
2.5 Network Layer: Routing 21
2.5.1 Ad-hoc On-Demand Distance Vector Routing (AODV) 22
2.5.2 Optimized Link State Routing (OLSR) 23
2.6 Network Layer: Forwarding 23
2.6.1 Selection Diversity Forwarding (SDF) 25
2.6.2 Extremely Opportunistic Routing (ExOR) 27
2.6.3 MORE 30
2.7 Transport Layer 31
2.7.1 AIRMAIL 32
2.7.2 I-TCP 33
2.7.3 SCTP 33
2.7.4 ATCP 34
2.8 Session Layer 34
2.8.1 Common Object Request Broker Architecture (CORBA) 35
2.9 Other Layers 36
3. MStack 37
4. Cooperative Forwarding 41
4.1 Cloud Addresses 41
4.2 Cooperative Forwarding Protocol Operation 42
4.3 Comparison of the Four Main Proposals of Opportunistic Forwarding 45

5. MICO Table 47
5.1 DTDO Table 47
5.2 DOCK Table 47
5.3 CHAIN Table 51
5.3.1 RXCHAIN Table 52
5.3.2 RXUNCHAIN Table 52
6. DELTOYA 53
6.1 DTDO Querying 53
6.2 DTDO Requesting (Pulling DTDOs) 54
6.3 DTDO Sending (Pushing DTDOs) 57
6.4 DOCKs and CHAINs 60
7. PTP 61
7.1 PTP Segmentation 61
7.2 PTP Messages 61
7.3 PTP Operation 62
8. MIFO 64
8.1 MIFO Header 64
8.2 MIFO Table 64
8.3 Unicast/Broadcast Forwarding 65
8.4 Look Up 67
8.4.1 Broadcast Packets 67
8.4.2 Unicast Packets 67
9. DTDODM 70
10. DOCKDM 71
11. RDM 72
12. AGENET 73
12.1 Aggressive Routing 73
12.2 Routing Information: AGgressive ROuting Hellos (AGROHellos) 74
12.3 Scalability: Routing Zone (RZ) 75
12.4 Routing Heuristic (RH) 75
13. MIMI 76
14. MIMA 77
14.1 MANET Scanning 77
14.2 SSID management 77

14.3 Merging and splitting MANETs 78
14.4 MANET Horizons 80
15. Results 81
15.1 Cooperative Forwarding 81
15.1.1 Prototype 81
15.1.2 Experimental Setting 90
15.1.3 Experimental Results 96
15.1.4 Conclusions 112
15.2 MStack 113
15.2.1 Prototype 113
15.2.2 First Insights 113
15.2.3 Real Experiments in Real Scenarios 120
16. Conclusions 122
16.1 Beyond the Conclusions 123
Bibliography 125
Appendix A. Cooperative Forwarding Details 129
A.1 Cooperative Forwarding Calculations 129
Appendix B. MICO Table Details 133
B.1 DTDO Table 133
B.2 DOCK Table 138
B.3 CHAIN Table 141
B.3.1 RXCHAIN 141
B.3.2 RXUNCHAIN 142
Appendix C. DELTOYA Details 143
C.1 DTDO Creation 143
C.2 DTDO Modification 144
C.3 DTDO Deletion 145
C.4 DTDO Publishing 145
C.5 DTDO Unpublishing 146
C.6 DTDO Querying 146
C.6.1 DTDO Query Queue (DTDOQQ) 146
C.6.2 Querying DTDOs 148
C.6.3 Cancelling DTDO Queries 153
C.7 DTDO Requesting 154

C.7.1 DTDO Request Queue (DTDORQ) 154
C.7.2 Requesting specific DTDOs 156
C.7.3 Requesting generic DTDOs 159
C.7.4 Cancelling DTDO Requests 163
C.8 DTDO Sending 163
C.8.1 DTDO Sending Queue (DTDOSQ) 163
C.8.2 DTDO Asking Queue (DTDOAQ) 165
C.8.3 ASK2ME Header 166
C.8.4 Sending DTDOs 168
C.8.5 Resending DTDOs 180
C.8.6 Receiving DTDOs 183
C.9 DOCKs and CHAINs 187
C.9.1 DOCK Creation 187
C.9.2 DOCK Destruction 187
C.9.3 DOCK Querying 188
C.9.4 CHAIN Creation 192
C.9.5 CHAIN Destruction 192
Appendix D. PTP Details 194
D.1 PTP Header 194
D.2 PTP Request Queue (PTPRQ) 195
D.3 PTP Serving Queue (PTPSQ) 197
D.4 Pull Mode 198
D.4.1 getDTDO (Sending PTP Request messages) 198
D.4.2 stopGetDTDO 216
D.5 PTP Calculations 216
Appendix E. MIFO Details 226
E.1 MIFO Header 226
E.2 MIFO Table 227
E.3 Source Learning 227
Appendix F. DTDODM Details 229
F.1 Flooding (The Aggregator) 230
F.2 DTDO Discovery Request (DTDODREQ) 231
F.2.1 DTDODREQ for queryDTDOs 231
F.2.2 DTDODREQ for requestDTDOs 235

F.2.3 DTDODREQ due to pushDTDO 238
F.3 DTDO Discovery Reply (DTDODREP) 244
F.3.1 DTDODREP for queryDTDOs 245
F.3.2 DTDODREP for requestDTDOs 247
F.3.3 DTDODREP due to pushDTDO 250
Appendix G. DOCKDM Details 257
G.1 Flooding (The Aggregator) 257
G.2 DOCK Discovery Request (DOCKDREQ) 258
G.3 DOCK Discovery Reply (DOCKDREP) 261
Appendix H. RDM Details 264
H.1 RDM Queue (RDMQ) 264
H.2 startRDM 264
H.2.1 Route Discovery Request (RDREQ) 266
H.2.2 Route Discovery Reply (RDREP) 268
H.3 stopRDM 271
Appendix I. AGENET Details 272
I.1 Routing Information: AGgressive ROuting Hellos (AGROHellos) 272
I.2 Scalability: Routing Zone (RZ) 272
I.2.1 Starting the RZ 272
I.2. 2 Maintaining the RZ 272
I.2.3 Ending the RZ 273
I.3 AGROHello Format 273
I.3.1 Sending AGROHellos 275
I.3.2 Receiving AGROHellos 276
I.4 Routing Heuristic (RH) 278
Appendix J. MIMI Details 281
J.1 Miraveo Nodes (ID) 281
J.1.1 IP 281
J.1.2 Miraveo Applications 281
J.2 MIMITable 281
J.3 MIMIHello 282
J.3.1 Sending MIMIHellos 283
J.3.2 Receiving MIMIHellos 284
Appendix K. MStack Parameters 287

List of Figures

Figure 1. Mobile Ad-hoc Network 16
Figure 2. Communications Stack 17
Figure 3. OSI/ISO Stack 18
Figure 4. TCP/IP Stack 19
Figure 5. WAP Stack 20
Figure 6. Candidate nodes in SDF 25
Figure 7. Transmission of the Data Packet in SDF 26
Figure 8. Handshake in SDF to select the actual forwarder 26
Figure 9. Example of SDF protocol operation 27
Figure 10. Candidate nodes in ExOR 28
Figure 11. Transmission of the Data Packet in ExOR 28
Figure 12. Handshake in ExOR to determine the actual relay 29
Figure 13. Example of ExOR protocol operation 29
Figure 14. Example of MORE protocol operation 31
Figure 15. MStack 38
Figure 16. MStack (without Cooperative Forwarding) 39
Figure 17. Cloud Addresses in Cooperative Forwarding 41
Figure 18. Forwarding through Clouds: frames are addressed to the next cloud 42
Figure 19. Master and Relay Nodes examples 43
Figure 20. The Cooperative Forwarding mechanism: how it works 44
Figure 21. Example of Cooperative Forwarding protocol operation 45
Figure 22. DOCKBuffer Example 1 48
Figure 23. DOCKBuffer Example 2 49
Figure 24. DOCKBuffer Example 3 50
Figure 25. DOCKBuffer Example 4 51
Figure 26. Creating and Publishing a DTDO 54
Figure 27. Requesting the DTDO 55
Figure 28. Pulling the DTDO 55
Figure 29. Timing the DTDO 56
Figure 30. Requesting again the DTDO 56
Figure 31. Pulling again the DTDO 57
Figure 32. Expiring the DTDO 57
Figure 33. Pushing a DTDO 59
Figure 34. DTDO Waiting in the DOCK 60
Figure 35. PTP PDU 61
Figure 36. PTP Push and Pull Modes 62
Figure 37. IP Route Table 65
Figure 38. Headers of a Packet 65
Figure 39. IP and MIFO Headers 66
Figure 40. BSSID partitioning 78
Figure 41. MANET Merging 79
Figure 42. Click Scheme of the Cooperative Forwarding with AODV 82
Figure 43. Wi-Fi Reception Module 82
Figure 44. Wi-Fi Transmission Module 83

Figure 45. Data Structure Module 83
Figure 46. 802.11 ACK Reception Module 83
Figure 47. Cooperative ACK (CACK) Reception Module 84
Figure 48. Cooperative HELLO (CHELLO) Module 84
Figure 49. Routing Module 85
Figure 50. Cooperative Forwarding Module 86
Figure 51. Map of the Two-hop Scenario Using UDP 91
Figure 52. Difference between Relaying and ARQ Situations 91
Figure 53. Map of the Multi-hop Scenario Using UDP 92
Figure 54. Pictures of the Multi-hop Scenario Using UDP 93
Figure 55. Map of the Two-hop Scenario Using TCP 95
Figure 56. Laptops in the Two-hop Scenario Using TCP 95
Figure 57. Behavior of Relaying in LUNAR (before Route Discovery) 96
Figure 58. Behavior of ARQ in LUNAR (before Route Discovery) 97
Figure 59. Behavior of Relaying in LUNAR (Route Discovery) 97
Figure 60. Behavior of ARQ in LUNAR (Route Discovery) 97
Figure 61. Behavior of Relaying in AODV (before Route Discovery) 98
Figure 62. Behavior of ARQ in AODV (before Route Discovery) 98
Figure 63. Behavior of Relaying in AODV (Route Discovery) 98
Figure 64. Behavior of ARQ in AODV (Route Discovery) 99
Figure 65. Screenshot of Flow A in the multi-hop experiment 101
Figure 66. Consecutive packet losses in Flow A for cooperative and non-cooperative experiments 102
Figure 67. TCP Throughput between client and server in the two-hop experiment 103
Figure 68. Number of Retransmissions in the Non-cooperative case 105
Figure 69. Number of Retransmissions in the Cooperative Case 106
Figure 70. Milestones of the Non-cooperative Case 106
Figure 71. Milestones of the Cooperative Case 108
Figure 72. Cooperative Nature of the TCP Data Packets 111
Figure 73. Cooperative Nature of the TCP ACK Packets 111
Figure 74. Scenario 1 114
Figure 75. Throughput in Scenario 1 114
Figure 76. Packets #0 to #50 for the Receiver Node 115
Figure 77. Packets #0 to #50 for the Sender Node 115
Figure 78. Packets #100 to #150 for the Receiver Node 116
Figure 79. Packets #100 to #150 for the Sender Node 116
Figure 80. Scenario 2 117
Figure 81. Throughput in Scenario 2 117
Figure 82. Scenario 3 118
Figure 83. Throughput in Scenario 3 118
Figure 84. Scenario 4 119
Figure 85. Throughput in Scenario 4 119
Figure 86. Grey Area in Scenario 4 120
Figure 87. Zooming the Grey Area of Scenario 4 120
Figure 88. Format and Size of TCP Frames 129

Figure 89. Mean Backoff Counter Per Retransmission 130
Figure 90. Mean Backoff Counter Per Retransmission (Cooperative case) 132
Figure 91. DTDO Querying Backoff..... 147
Figure 92. DTDO Requesting Backoff 155
Figure 93. DOCK Querying Backoff..... 189
Figure 94. Format and Size of PTP frames 217
Figure 95. Mean Backoff Counter Per Retransmission 218
Figure 96. PTP Transmission Algorithm..... 219
Figure 97. EXPECTED_PTPDATA 222
Figure 98. The DTDOQQ/DTDORQ/DTDOAQ Aggregator 230
Figure 99. The DOCKQQ Aggregator 257
Figure 100. RDM Backoff..... 265
Figure 101. AGROHello message..... 274
Figure 102. MIMIHello..... 282

List of Tables

Table 1. Comparison of different opportunistic forwarding proposals: SDF, ExOR, MORE and Cooperative Forwarding 46
Table 2. Wi-Fi Parameters 86
Table 3. AODV Parameters 87
Table 4. Cooperative Forwarding Parameters 88
Table 5. TCP Parameters 89
Table 6. Average Path Hop Count per Flow in the Multi-hop experiment 94
Table 7. Contingency Table for Chi-square Homogeneity Test for Flow A 94
Table 8. Results of Two-hop Scenario and controlled mobility Tests 100
Table 9. Results of Flows A and B in the multi-hop experiment 101
Table 10. Results of Flows C and D in the multi-hop experiment 101
Table 11. Average size of consecutive packet losses for cooperative and non-cooperative experiments 102
Table 12. Comparison of the overhead in the Multi-hop Scenario 103
Table 13. 802.11a and OFDM parameters 129
Table 14. 802.11a and OFDM parameters (Cont.) 129
Table 15. queryDTDOs calls 148
Table 16. requestDTDOs calls 160
Table 17. queryDOCKs calls 190
Table 18. 802.11b and DSSS parameters 217
Table 19. 802.11b and DSSS parameters (Cont.) 217

Publications derived from the Thesis

David Fusté-Vilella, Jorge García-Vidal, Julián David Morillo-Pozo. “Cooperative Forwarding in IEEE 802.11-based MANETs”. 1st IFIP Wireless Days Conference 2008, Dubai. Best Student Paper Award.

David Fusté-Vilella, Jorge García-Vidal, Julián David Morillo-Pozo. “A Click-based Prototype for the Cooperative Forwarding Mechanism”. Symposium on Click Modular Router. November 2009, Belgium.

David Fusté-Vilella, Jorge García-Vidal, Julián David Morillo-Pozo. “Opportunistic Forwarding in MANETs and Mesh Networks”. Accepted for publication in the book “Cross Layer Designs in WLAN Systems”, ISBN: 9781848762275, Troubador Publishing Ltd, Leicester, UK.

Patents derived from the Thesis

Systems and Methods for Creating, Managing and Communicating Users and Applications on Spontaneous Area Networks. **Inventors**: David Fusté-Vilella, Jorge García Vidal, Daniel Nemirovsky, Mario Nemirovsky. **Assignee**: Miraveo, Inc.

System and method for routing a data packet in a wireless network, computing system in a system for routing a data packet in a wireless network and method for routing a data packet in a computing system. **Inventors**: Jorge García Vidal, Julián David Morillo Pozo, David Fusté Vilella. **Assignee**: Miraveo, Inc.

1. Introduction

A Mobile Ad-hoc NETwork (MANET) [**1**] is a self-configuring network of devices connected by means of wireless links; see Figure 1.

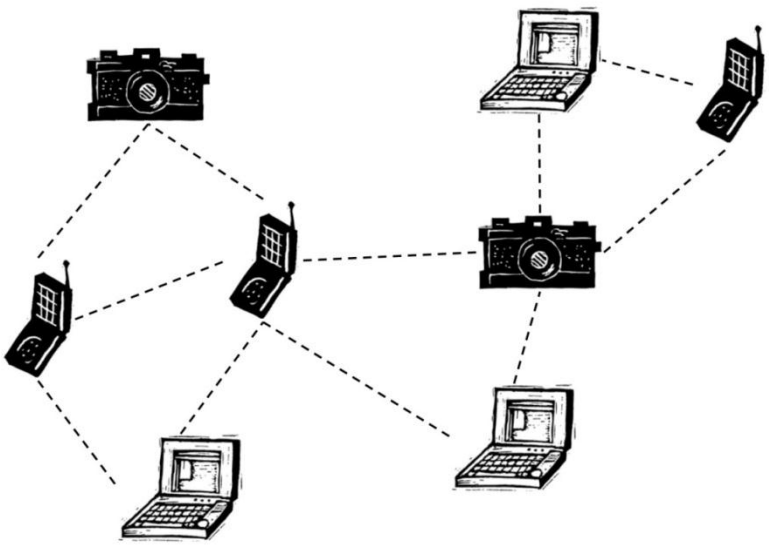

Figure 1. Mobile Ad-hoc Network

Nodes in a MANET communicate directly amongst themselves, operating in a distributed peer-to-peer mode. Usually, nodes can forward traffic unrelated to their own use, therefore becoming routers and creating multi-hop routing paths. Such networks may operate in isolation or may be connected to other networks such as the Internet.

In a MANET, nodes are free to move, thus changing their links to other nodes frequently. The primary challenge in building a MANET is to continuously maintain the information required to properly route traffic.

These networks inherit the traditional problems of wireless and mobile communications, such as bandwidth optimization, power control, and transmission-quality enhancement. In addition, their multi-hop nature and the possible lack of a fixed infrastructure introduce new research problems such as network configuration, device discovery, and topology maintenance, as well as ad-hoc addressing and self-routing. Various approaches and protocols have been proposed to address ad-hoc networking problems, and multiple standardization efforts are under way within the Internet Engineering Task Force, as well as academic and industrial research projects.

MANETs have received considerable attention in the wireless communications research community, although its importance in industry is still low. Self-organized MANETs of smartphones or laptops can be used, for example, in military, public safety and disaster relief, conference, or campus environments.

Many variations of the concept of MANETs are possible, such as Mesh Networks [**2**], Sensor Networks [**3**] or Delay Tolerant Networks [**4**].

2. State of the Art

In order to enable communications between nodes inside a network, a communications stack (or protocol stack) is needed in each node of the network, and each type of network needs its own specific communications stack.

A communications stack is a suite of networking protocols organized in a way that they work together in order to solve the problems arising from communicating nodes through the network. Individual protocols within the suite are often designed with a single purpose in mind. This modularization makes design and evaluation easier. Because each protocol usually communicates with two others, they are commonly imagined as layers in a stack of protocols; see Figure 2. Layers are connected each other through a given interface.

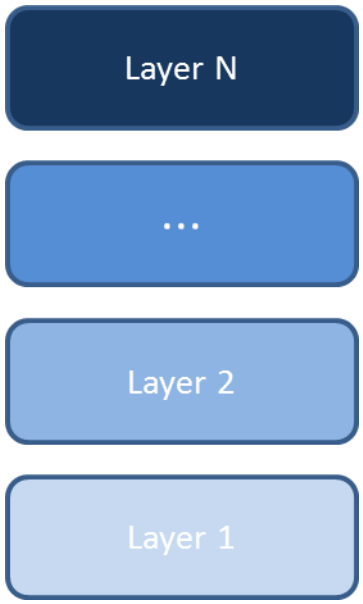

Figure 2. Communications Stack

Usually, the lowest protocol always deals with low-level, physical interaction of the hardware. Every higher layer adds more features. User applications usually deal only with the topmost layers. The three well-known communications stack examples are the OSI [**5**], TCP/IP [**6**], and WAP [**7**] stacks.

2.1 Open System Interconnection

The Open Systems Interconnection (OSI) stack was a product of the International Organization for Standardization (ISO) and has a major historical relevance. In the OSI/ISO stack, there are seven layers and each layer is a collection of similar functions that provide services to the layer above it and receives services from the layer below it; see Figure 3.

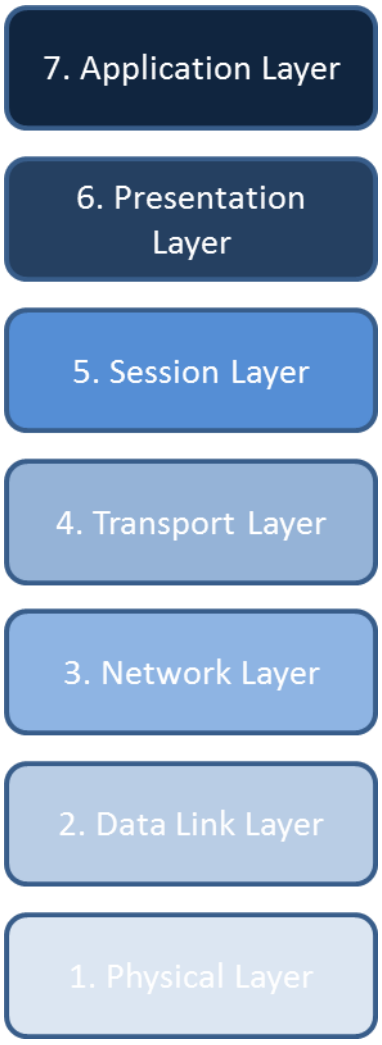

Figure 3. OSI/ISO Stack

The Physical Layer defines the electrical and physical specifications for devices. In particular, it defines the relationship between a device and a transmission medium.

The Data Link Layer provides the functional and procedural means to transfer data between network entities and to detect and possibly correct errors that may occur in the Physical Layer.

The Network Layer provides the functional and procedural means of transferring variable length data sequences from a source host on one network to a destination host on a different network, while maintaining the quality of service requested by the Transport Layer (in contrast to the data link layer which connects hosts within the same network). The Network Layer performs network routing functions, and might also perform fragmentation and reassembly, and report delivery errors.

The Transport Layer provides transparent transfer of data between end users, providing reliable data transfer services to the upper layers. The Transport Layer controls the reliability of a given link through flow control, segmentation/desegmentation, and error control.

The Session Layer controls the dialogues (connections) between computers. It establishes, manages and terminates the connections between the local and remote application.

The Presentation Layer establishes context between Application Layer entities, in which the higher-layer entities may use different syntax and semantics if the presentation layer provides a mapping between them.

Finally, the Application Layer is the OSI layer closest to the end user, which means that both the OSI application layer and the user interact directly with the software application. This layer interacts with software applications that implement a communicating component.

2.2 TCP/IP DARPA Stack

The TCP/IP stack was created in the 1970s by DARPA, an agency of the United States Department of Defense, and is the stack currently used in most of our networks. It evolved from ARPANET, which was the world's first wide area network and a predecessor of the Internet. The TCP/IP stack is presented using four layers; see Figure 4.

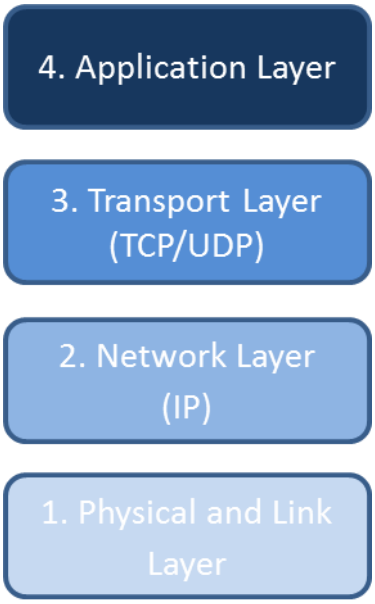

Figure 4. TCP/IP Stack

In the TCP/IP stack, each layer provides function for layers above. The Physical and Link Layer is responsible for delivering bits over a link, for example a wireless transmission link, and for framing, flow control and error detection on the link.

The Network Layer is responsible for internetworking and uses the Internet Protocol (IP) [**8**] to deliver data from upper layers between end hosts. IP is a best effort protocol and the delivery of packets to destinations is not guaranteed. Whereas the lower layers only pertain to a link, the network layer and higher layers are end-to-end. A routing protocol is responsible for building and maintaining routing tables, and a forwarding algorithm is responsible for determining the proper next hop link given a destination address.

The Transport Layer consists of two protocols, namely Transmission Control Protocol (TCP) [**9**] and User Datagram Protocol (UDP) [**10**]. TCP provides reliable end-to-end delivery for data streams with flow control, congestion control, and ordered error free transfer. UDP provides unreliable packet delivery.

Finally, the Application Layer refers to the higher-level protocols used by most applications for network communication. Data coded according to application layer protocols is then encapsulated into a transport layer protocol, which in turn use a lower layer protocol to cause actual data transfer.

2.3 WAP (Wireless Application Protocol)

Another example of communications stack is the Wireless Application Protocol (WAP). It was designed in 1997 by several companies organized as an industry group called the WAP Forum. WAP includes a long and detailed series of technical documents that define standards for implementing wireless network applications. WAP specifies an architecture based on layers that follows the OSI model fairly closely; see Figure 5.

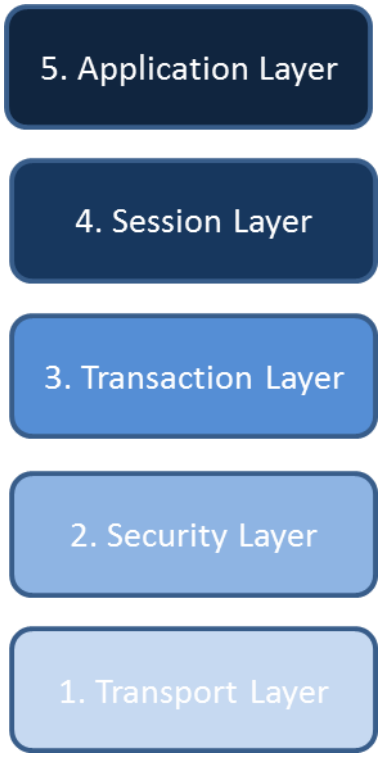

Figure 5. WAP Stack

The motivation was to build a single platform for competing network technologies such as GSM and CDMA. The main usage scenario of WAP is to enable access to the Internet from a mobile phone or PDA.

The Application Layer is the Wireless Application Environment (WAE). WAE directly supports WAP application development with Wireless Markup Language (WML) instead of HTML and WMLScript instead of JavaScript.

The Session Layer is the Wireless Session Protocol (WSP). WSP is the equivalent to HTTP for WAP browsers. WAP involves browsers and servers just like the Web, but HTTP was not a practical choice for WAP because of its relative inefficiency on the wire. WSP conserves precious bandwidth on wireless links; in particular, WSP works with relatively compact binary data where HTTP works mainly with text data.

The Transaction Layer is the Wireless Transaction Protocol (WTP). WTP provides transaction-level services for both reliable and unreliable transports. It prevents duplicate copies of packets from being received by a destination, and it supports retransmission, if necessary, in cases where packets are dropped. In this respect, WTP is analogous to TCP. However, WTP also differs from TCP. WTP is essentially a pared-down TCP that squeezes some extra performance from the network.

The Security Layer is the Wireless Transaction Layer Security (WTLS). WTLS provides authentication and encryption functionality analogous to Secure Sockets Layer (SSL) in Web networking. Like SSL, WTLS is optional and used only when the content server requires it.

The Transport Layer is the Wireless Datagram Protocol (WDP). WDP implements an abstraction layer to lower-level network protocols; it performs functions similar to UDP. WDP is the bottom layer of the WAP stack, but it does not implement physical or data link capability. To build a complete network service, the WAP stack must be implemented on some low-level legacy interface not technically part of the model.

2.4 Why do we need a new communications stack for MANETs?

Concerning MANETs, research in this area was mainly focused on optimizing the TCP/IP stack, the stack par excellence in the majority of our current networks. Instead of designing a complete new communications stack, researchers focused their work mainly on improving already existing solutions coming from the wired networks. Already existing routing protocols, transport protocols, etc. were improved in order to bring a certain level of robustness to MANETs.

But the problem is that optimizing a communications stack specifically designed for wired networks will never be the perfect solution. We think that a complete and new communications stack specifically designed for MANETs is needed if we want to achieve robust communications in this type of networks, which have properties very different from what wired networks have [**11**]. Moreover, the current TCP/IP stack faces several challenges with wireless mobile nodes in a MANET. Many of these mobility related problems stem from the fact that the IP address specifies both the identity and location of a host. This means that a location management scheme is needed to update any changes in a mobile node's IP address to its peers.

For all these reasons, in this thesis we designed what nowadays is still missing: a novel communications stack for mobile ad-hoc networks, called MStack.

In the next subchapters, we will examine in more detail the state of the art of the layers involved in our communications stack, and always compared with the corresponding layers of the TCP/IP stack, since it is the dominant communications stack available today.

2.5 Network Layer: Routing

One of the main problems of MANETs is the possible lack of robustness of the routing paths due to the fact that:

- Wireless links are highly variable; see [**12**].

- Node movement exacerbates the intrinsic variability of the wireless links and can produce fast changes in the network connectivity graph.
- A detailed and timely view of the network topology and links status can be obtained only at a cost of introducing a high signaling overhead, so nodes usually have only a partial knowledge of the instantaneous network state, often obtained by the periodic exchange of signaling packets. This leads to reaction times that can be of the order of seconds.

Many approaches to increase robustness in MANETs at routing layer have been proposed, including highly adaptive dynamic routing, multipath routing, stable routes selection, path and link diversity, optimized link failure detection and repair mechanisms, and network coding [**13**] [**14**] [**15**] [**16**] [**17**] [**18**].

Routing protocols in MANETs can be mainly classified into two types of protocols: reactive and proactive routing protocols. On the one hand, reactive routing protocols base their operation on discovering routing paths only when they are needed, i.e., when two nodes of the network want to establish communication between each other. This can lead to high latency times when starting new communications. On the other hand, proactive routing protocols base their operation on maintaining the routing information of the whole network all the time. This can lead to a certain amount of traffic overhead for the maintenance of the network topology.

2.5.1 Ad-hoc On-Demand Distance Vector Routing (AODV)

Ad-hoc On-Demand Distance Vector routing (AODV) [**13**] can be considered the main proposal inside the big set of reactive routing protocols for MANETs. It was jointly developed by the Nokia Research Center, the University of California Santa Barbara and the University of Cincinnati.

AODV is a distance-vector routing protocol and avoids the counting-to-infinity problem of the distance-vector protocols by using sequence numbers on route updates, a technique firstly introduced by DSDV [**19**].

From the point of view of a source node, when a connection is needed AODV broadcasts a request message through the MANET. Other AODV nodes of the network that receive the request message rebroadcast again the message, causing a flooding of the request message through the whole network. When the desired destination node receives the corresponding request message, it sends back a reply message through a temporary routing path set by the request message. In this way, a routing path in both directions is created between the source and the destination nodes.

AODV uses the number of hops of the routing path as routing metric. If several routing paths are discovered between two nodes, the one with less number of hops will be used.

When a link break occurs, an error message is sent back to the source node, and the discovery process of the routing path repeats again. In order to detect link breaks as soon as possible, AODV uses hello messages between each pair of adjacent nodes in the routing path. When a node of the pair stops receiving hello messages from its neighboring nodes, then it decides that a link break has occurred.

As said before, AODV uses sequence numbers in order to avoid using obsolete request and reply messages that could bring to routing loop problems in the network. With the sequence numbers incremented each time an AODV node sends a request or a reply message, AODV is able to identify the most recent routing information.

Dynamic Source Routing (DSR) [**20**] is another reactive routing protocol quite similar to AODV, the major difference being the fact that DSR uses source routing information. In DSR, data packets carry the complete routing path to be traversed. In AODV, the source node and the intermediate nodes of the routing path only use the next-hop node information for the data packet transmission.

2.5.2 Optimized Link State Routing (OLSR)

Optimized Link State Routing (OLSR) [**14**] can be considered the main proposal of the proactive routing protocols for MANETs.

OLSR is a proactive link-state routing protocol that uses Hello and Topology Control (TC) messages to discover and disseminate link state information throughout the whole network. Nodes use this topology information to compute next hop destinations for all nodes in the network using the shortest hop routing paths.

Since OLSR needs to disseminate routing information through the whole network, it uses a different approach in order to optimize the flooding process. OLSR nodes use Hello messages to discover their 2-hop neighbors and to perform a distributed election of a set of Multipoint Relays (MPRs). OLSR nodes select MPRs in a way that all their 2-hop neighbors are reachable via one of their selected MPRs. Then, these MPR nodes will be the responsible for forwarding the TC messages that contain the routing information to be sent to the whole network in order to let every node of the network to build its own topology map of the network. In this way, by only using MPRs to flood topology information, OLSR removes some of the redundancy of the flooding process.

OLSR does not include any mechanism for sensing the quality of a link, only Hello messages are used in order to consider whether a link is up or down.

2.6 Network Layer: Forwarding

Cooperative protocols in wireless networks allow nodes to use the resources of adjacent nodes to increase their communication and processing capacities. Although the idea of cooperation is fairly general, it combines particularly well with the characteristics of wireless networks.

Most of the early work on cooperative networking was focused on the PHY layer in improving transmission parameters through an emulation of a multiple-antenna antenna array [**21**]. The idea of cooperation, however, has been progressively applied to upper layers of the communications stacks in order to increase link reliability, to obtain a more resilient path or to increase network capacity [**22**].

One way of exploiting link diversity is known as *opportunistic forwarding* or relaying. The main characteristic of opportunistic forwarding is that next-hop node selection is done *after* the packet has been transmitted, while usually deterministic methods select the next-hop node *before* packet transmission. By postponing the decision as to which node will forward the

packet, opportunistic methods are able to take advantage of random and, perhaps, rare opportunities. For example, due to the probabilistic nature of wireless transmission, a transmission may reach a node that is quite distant from the transmitter and close to the final destination. While this node might not be a reliable next-hop, when the chance arises, it might be considerably better suited to forward the packet than a node that has more reliable communication with the sender.

Opportunistic forwarding combines particularly well with the characteristics of wireless networks since it exploits the broadcast advantage inherent in wireless transmission and enables basic techniques for wireless transmission, such as space diversity. We believe that it can be especially beneficial in the case of MANETs as their unplanned nature and the mobility of nodes make communication in these networks particularly hazardous.

These forwarding techniques operate closely with the routing protocols. Routing was one of the main focuses for the early research into MANETs, followed by an important standardization activity by the IETF. Although MANETs are usually small networks, it quickly became apparent that routing was not an easy task. Optimal routing implies a detailed and timely view of the network topology and links status. However, due to the required signaling overhead, nodes usually have only a partial knowledge of the instantaneous network state, often obtained by the periodic exchange of signaling packets. This leads to reaction times that can be on the order of seconds. Thus, in real scenarios, routing protocols can behave poorly, leading to a degraded performance of the applications. It is here important to differentiate both types of mechanisms: routing is the technique for finding and maintaining optimal paths between source and destination nodes, and forwarding is the technique to relay the packets along the routing paths.

Note also that the definition used here for opportunistic forwarding (widely used in the literature) is slightly different than that used in other works such as in [**23**]. In these works, opportunistic forwarding refers to protocols that take advantage of contact opportunities to route data in intermittent connected environments. A similar concept than the latter is considered in works like [**24**], but combined with the concept of duty cycling (a common technique that constrains the RF operations of wireless devices for saving the battery energy) in a way that contact opportunities are more due to the duty cycling of nodes than due to mobility. The same authors define in [**25**] the concept of stateless opportunistic forwarding as a simple fault-tolerant distributed approach for data delivery and information querying in wireless ad-hoc networks, where packets are forwarded to one of the next available neighbors in a “random walk” fashion, until they reach the destination or expire. A key concept difference with the works surveyed before is that in our opportunistic forwarding concept, packet expiration is not considered. Moreover, this “random walk” technique does not use any topology information while the proposals surveyed here work with a routing path established by a routing protocol.

A hallmark of opportunistic forwarding is that when a packet is transmitted, the next hop of a multi-hop routing path is not known, but depends on the outcome of the transmission. The motivation for this flexibility is that it allows multiple nodes to act as relays. For example, if the

transmission fails to reach the primary relay, then a backup node may receive and relay the packet. In this way, opportunism increases robustness.

Opportunistic forwarding is also able to improve other performance metrics such as hop count and bit-rate. To see how this is possible, recall that in traditional, deterministic forwarding, a link is selected to forward data packets if it makes good progress toward the desired destination, and can support the desired bit-rate with an acceptable probability of success. Consequently, deterministic forwarding neglects links that can deliver packets closer to the desired destination, but that have large transmission error probability. However, if, by chance, the transmission across such a link succeeds, opportunistic forwarding is able to make use of the success. As a result, opportunistic forwarding can make use of links with high error probability, effectively decreasing the hop count and/or increasing the bit-rate.

In the following sections, we describe the three main proposals of opportunistic forwarding.

2.6.1 Selection Diversity Forwarding (SDF)

Selection Diversity Forwarding (SDF) [**26**] was the first proposal for opportunistic forwarding. In SDF, the forwarding decision of a packet in a multi-hop routing path is performed subsequent to the transmission of the packet and by means of a handshake between the transmitter of the packet and a set of candidate relays.

2.6.1.1 SDF Protocol Operation

First of all, a node selects a possible set of relays before the packet transmission. For example, in Figure 6, node S selects R_1 and R_2 as candidate nodes to relay packets towards D.

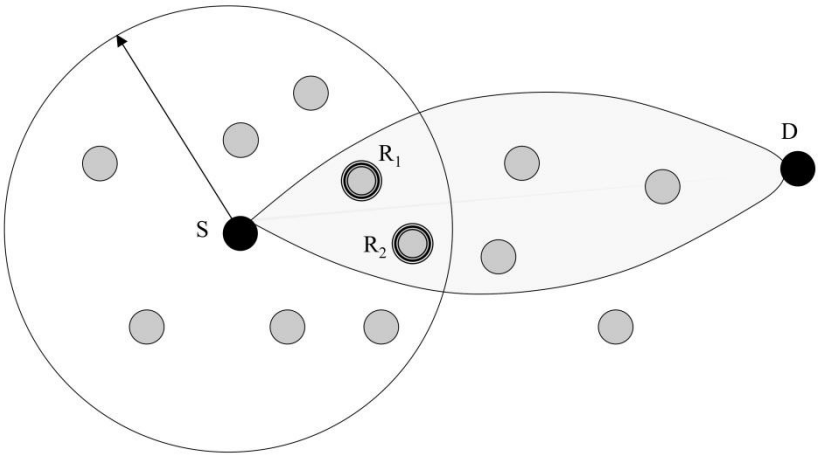

Figure 6. Candidate nodes in SDF

When the packet is transmitted, it is multicasted to these candidate nodes. Note that in wireless links, only one transmission will be needed in order to transmit the packet to all the candidate nodes.

Once the packet is received, every candidate node that has received correctly the packet will transmit an acknowledgement message back to the transmitter of the packet; see Figure 7. In order to avoid the problem of collisions between the different acknowledgements, the candidate nodes will send their acknowledgements following a predetermined order established by the transmitter node inside the packet.

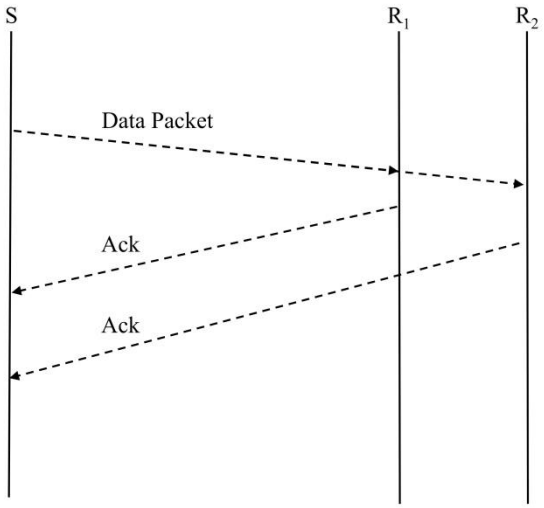

Figure 7. Transmission of the Data Packet in SDF

Based on the acknowledgements returned by the potential relays subsequent to the packet transmission, the node determines the best relay in terms of positive forward progress. The forwarding node then sends a forwarding order granting responsibility for packet relaying to the selected relay. Finally, this forwarding order is acknowledged and the packet forwarded; see Figure 8.

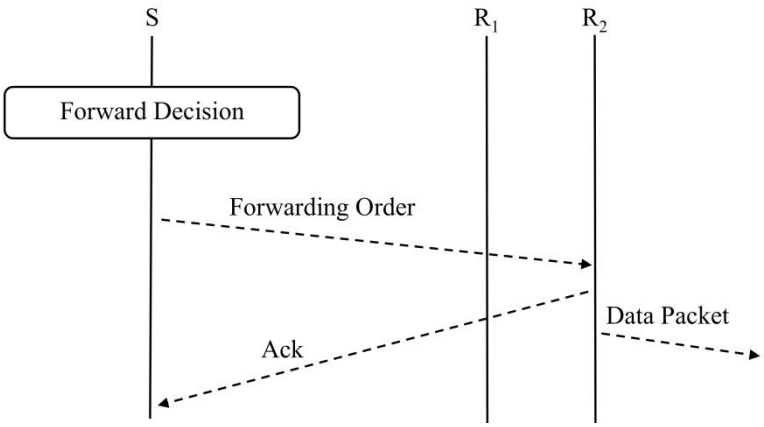

Figure 8. Handshake in SDF to select the actual forwarder

As an example, in Figure 9, node A chooses two routing paths (A,B,E,G) and (A,D,F,G) to reach G. Based on these elected routing paths, the candidate nodes of A for sending the packets to G will be B and D. So, node A forwards the packets to B and D (using just one wireless transmission). When A receives the acknowledgements from both nodes (if both B and D have received the packet correctly), it decides who will be the next forwarder of the packet. For example, it sends the forwarding order to B, and from now on, B will send the acknowledgement for the forwarding order and the packet will keep going through the path (A,B,E,G) in order to reach G.

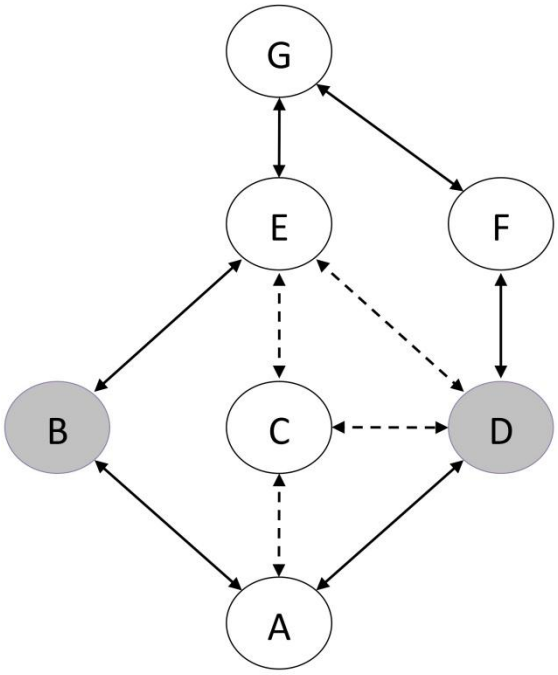

Figure 9. Example of SDF protocol operation

2.6.2 Extremely Opportunistic Routing (ExOR)

Extremely Opportunistic Routing (ExOR) [**27**] [**28**] uses a similar idea than SDF, the main difference being that potential relays coordinate in a distributed way, without the responsibility of the transmitter node.

There are two published versions of ExOR. The first one [**27**] requires a multiple-node handshake per packet. In addition, it requires that each node will know the approximate loss rate between every pair of nodes of the network. In the second version of ExOR [**28**], adapted to IEEE 802.11 hardware, the system operates on batches of packets, and receiving nodes acknowledge the fragment of the batch correctly received. Once again, the sender selects the set of optimal next relays, and the negotiation is distributed.

2.6.2.1 ExOR v1 Protocol Operation

In ExOR v1, the forwarding decision of a packet in a multi-hop routing path is performed subsequent to the transmission of the packet and by means of a handshake between the set of candidate relays.

First of all, a node selects a possible set of relays before the packet transmission. For example, in Figure 10, node S selects R_1 and R_2 as candidate nodes to relay packets towards D. ExOR's performance is determined by its ability to choose a prioritized candidate set of nodes that brings a packet closest to its destination. ExOR chooses the prioritized candidate list as follows: it first identifies the shortest path to the destination, breaking ties between equally short paths using information from a *delivery ratio matrix* (ExOR assumes each node in the network has a matrix containing an approximation of the loss rate for direct radio transmission between every pair of nodes; this matrix can be built using a link-state flooding scheme, in which nodes measure loss rates and periodically flood statistics updates). The first node in this path is the highest priority candidate. Then ExOR deletes that node from the delivery ratio matrix, again

finds the shortest route, and uses the first hop on that route as the candidate with second priority. It repeats this process to find the remaining candidates. The resulting candidate set for a given destination can be cached until the next update of the delivery ratio matrix.

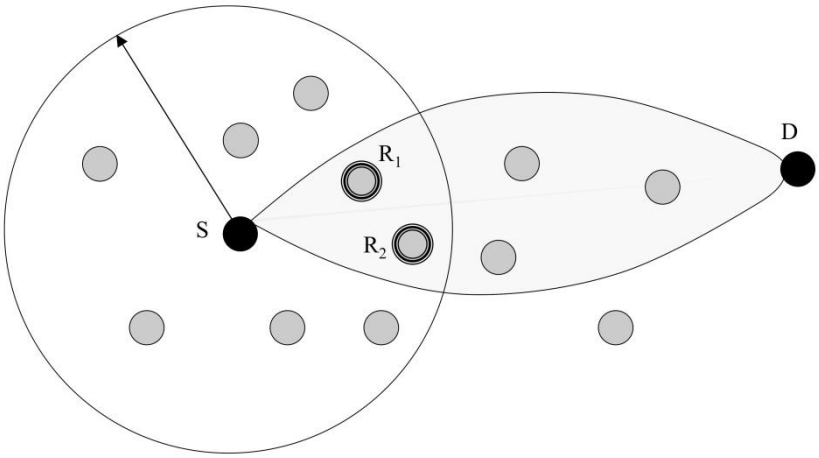

Figure 10. Candidate nodes in ExOR

When the packet is transmitted, it is multicasted to these candidate nodes. Note that in wireless links, only one transmission will be needed in order to transmit the packet to all the candidate nodes. Once the packet is received, every candidate node that has received correctly the packet will transmit an acknowledgement message back to the transmitter of the packet; see Figure 11. Instead of only indicating if the packet was successfully received, each acknowledgement contains the ID of the highest priority successful recipient known by the sender of the acknowledgement. All candidates listen to all acknowledgement slots before deciding whether to forward or not the packet. In order to avoid the problem of collisions between the different acknowledgements, the candidate nodes will send their acknowledgements following a predetermined order established by the transmitter node inside the packet.

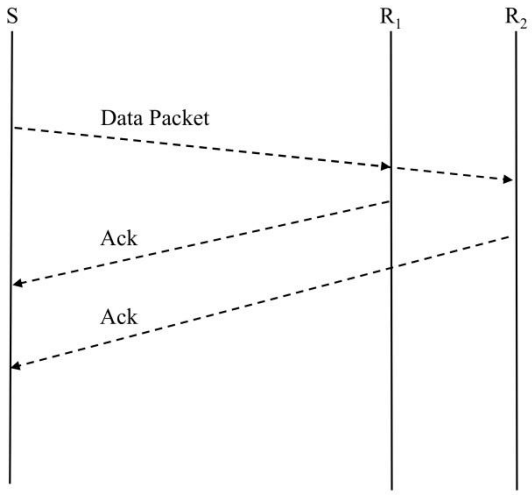

Figure 11. Transmission of the Data Packet in ExOR

Based on the acknowledgements returned by all the potential relays subsequent to the packet transmission, each potential relay determines whether to forward or not the packet; see

Figure 12. Only nodes that have not received acknowledgements containing an ID of a higher priority candidate will forward the packet.

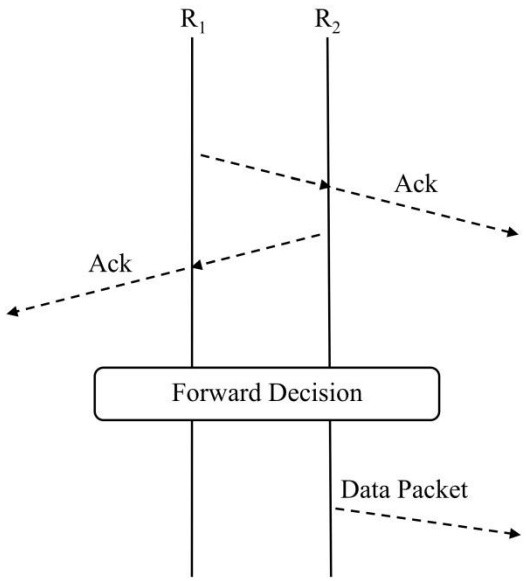

Figure 12. Handshake in ExOR to determine the actual relay

As an example, in Figure 13, node A chooses two routing paths (A,C,E,G) and (A,D,F,G) to reach G. Note that C and D should be neighbors. Node A forwards the packets to C and D (using just one wireless transmission). When a packet is received, nodes C and D decide who will be the next forwarder of the packet by overhearing their acknowledgements.

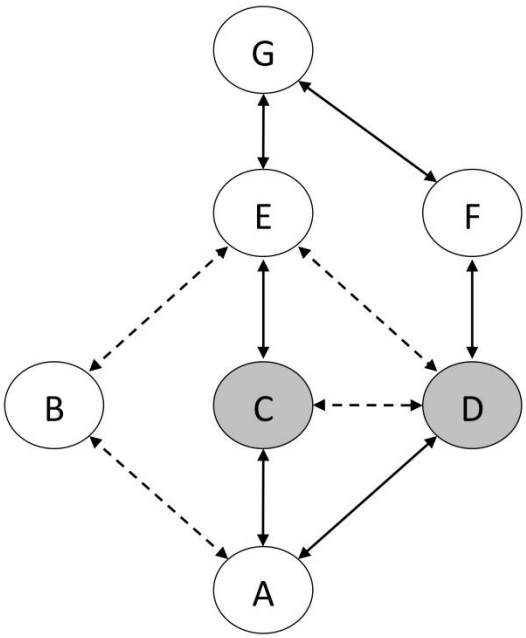

Figure 13. Example of ExOR protocol operation

2.6.2.2 ExOR v2 Protocol Operation

The main difference between the two versions of ExOR is that ExOR v2 operates on batches of packets in order to reduce the communication overhead due to the handshake during the relay election.

The source node collects a batch of packets destined to the same destination node and sends the batch of packets to the prioritized set of candidate forwarders.

The candidate nodes buffer the successfully received packets and await the end of the batch. The highest priority forwarder then broadcasts the packets in its buffer, including its copy of the "batch map" in each packet. The batch map contains, for each packet of the batch, the highest priority node that has received that packet. The remaining forwarders then transmit in order, but only sending packets that were not acknowledged in the batch maps of the higher priority forwarders (i.e., previous forwarders). The result is that a low priority forwarder is unlikely to forward a packet that has already been received by a higher priority forwarder.

The forwarders continue to cycle through the priority list until the destination has 90% of the packets. The remaining packets are transferred with conventional routing.

2.6.3 MORE

MORE [**29**] is a proposal for opportunistic forwarding based on the concept of network coding.

2.6.3.1 MORE Protocol Operation

In MORE, and as it is usually done in network coding, packets are organized into batches or generations.

The sender node sends packets of the same batch in broadcast. When a node overhears a packet, it checks (i) if the packet is innovative, meaning that it is linearly independent of the previously packets received from the same batch, and (ii) if it is closer to the destination than the sender node. If both conditions are fulfilled, the packet is linearly combined with the other packets of the same batch, and broadcasted again.

These packets will eventually reach the destination node, and when it can recover the information of a batch, it sends a signaling packet to stop subsequent transmissions of packets of this batch. At this time, the sender can move on to the next batch.

As an example, in Figure 14, if node A wants to reach G, it broadcasts packets that are overheard by B, C and D. These nodes compute linear combinations of these packets with the previously received packets belonging to the same batch, and rebroadcast the obtained packets. When G has enough information to recover the full batch, it sends back to A an acknowledgement packet to stop transmission of packets of this batch.

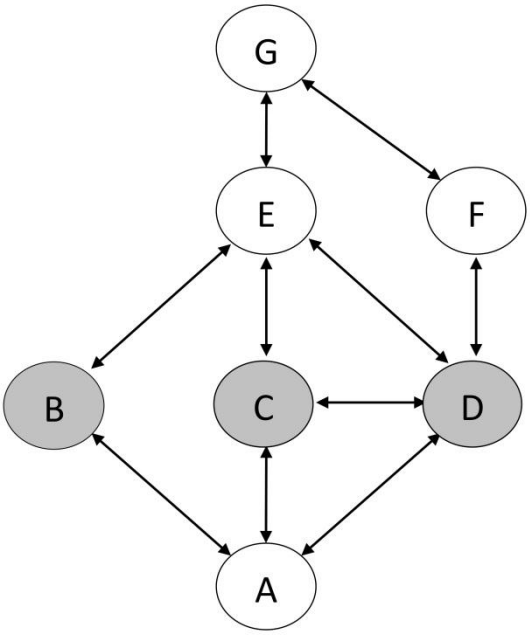

Figure 14. Example of MORE protocol operation

2.7 Transport Layer

Perhaps in the transport layer is where the problem of having optimized current solutions of wired networks for wireless networks instead of having designed new solutions is more visible.

Since the majority of the Internet's traffic is TCP traffic, such as WWW, email, and File Transfer Protocol, TCP has been the subject of intense research. Instead of defining new transport protocols taking into account the properties of the MANETs, researchers have focused their efforts on improving or adapting TCP [**9**], a transport protocol specifically designed for wired networks.

Nevertheless, all these solutions have the same problem: TCP was originally developed for wired and reliable networks. TCP's congestion control mechanism is based on a feedback control algorithm via acknowledgments packets that uses linear increments to the congestion window, and an exponential reduction when congestion takes place. The TCP congestion control mechanism assumes that packet losses are always due to congestion in the network and so, TCP slows down the transmission rate to alleviate congestion.

In MANETs, packet drops are not only due to congestion but also typically due to the conditions of the wireless environment, such as mobility, temporary losses due to fading, high error rates and frequent route changes and network partitions. All these characteristics result in packet loss besides network congestion. As a consequence, TCP does not perform well in mobile wireless environments. Moreover, TCP's reliability has a drawback in terms of latency. This is why many real-time streaming protocols, such as the Real-time Transport Protocol (RTP) [**30**], use UDP to deliver data streams over the Internet.

A number of solutions to adapt TCP to the wireless world have been proposed, which can be classified into link layer solutions, proxy-based solutions, and end-to-end solutions [**31**] [**32**].

The intuition behind all these solutions is that since the problems affecting TCP in a wireless environment are local, they should be solved locally.

Link layer solutions provide local reliability at link level. They can be divided into two classes. The first class includes protocols that use error correction, using techniques such as forward error correction (FEC). The second class includes protocols that use retransmission of lost packets in response to automatic repeat request (ARQ) messages and decide to look at how the link layer protocol interacts with the transport layer. The AIRMAIL [**33**] protocol uses a combination of FEC and ARQ techniques for loss recovery.

Proxy-based solutions, or split-connection solutions, break the end-to-end connection in two parts. They split the TCP connection into two parts at the wireless network edge using a specialized transport protocol over the wireless hop. Indirect TCP (I-TCP) [**34**] was the first protocol to use the split-connection solution. I-TCP splits each TCP connection between a sender and receiver into one TCP connection between the sender and the wireless base station, and the other between the wireless base station and the receiver. I-TCP uses regular TCP for its connection over the wireless link.

In the end-to-end solutions, the sender is aware of the wireless link. These solutions are implemented in the communicating end hosts, namely senders and receivers. The cumulative acknowledgments used by TCP do not allow efficient recovery from multiple packet losses. Selective Acknowledgments (SACK) specified in [**35**] are a commonly used TCP improvement that allows additional information to be sent from receiver to sender about missing packets. SACK can be used to inform the sender what packets are still missing and it thus alleviates some of the efficiency problems with the cumulative acknowledgments. SACK is a TCP option that reports discontinuous blocks of received data. This allows more efficient retransmissions. The Stream Control Transmission Protocol (SCTP) specified in [**36**] is a reliable transport protocol similar to TCP that supports SACK, preserves message boundaries, supports multiple streams in a single connection, and supports multi-homing. SCTP offers separate congestion control for each address pair with multi-homing. On the other hand, ATCP [**37**] is as an end-to-end solution to improve TCP throughput specifically designed for the ad-hoc case. It is implemented as a thin layer inserted between the standard TCP and IP layers and it relies on detecting and distinguishing congestion loss from error loss.

2.7.1 AIRMAIL

AsymmetrIc Reliable Mobile Access In Link-layer (AIRMAIL) [**33**] is a link-layer protocol specifically designed for wireless networks.

The key ideas in the AIRMAIL design consist of placing most of the intelligence in the base station as opposed to placing it symmetrically in the mobile nodes, since the mobile nodes have limited power and smaller processing capability than the base stations. The idea is to require the mobile nodes to combine several acknowledgments into a single acknowledgment to conserve power, and to design the base stations to send periodic status messages, while making the acknowledgment from the mobile nodes event-driven.

AIRMAIL also uses a forward error correction (FEC) technique that incorporates three levels of channel coding which interact adaptively. The coding overhead is changed adaptively so that

bandwidth expansion due to forward error correction is minimized. Integrity of the link during mobility is handled by window management and state transfer.

The motivation for using a combination of forward error correction and link-layer retransmissions is to obtain better performance in terms of end-to-end throughput and latency by correcting errors in an unreliable wireless channel in addition to end-to-end correction rather than by correcting errors only by end-to-end retransmissions.

2.7.2 I-TCP

Indirect TCP (I-TCP) [**34**] is an indirect transport layer protocol for mobile nodes. I-TCP was the first protocol to use the split-connection solution.

I-TCP splits each TCP connection between a sender and receiver into two separate connections at the wireless base station: one TCP connection between the sender and the wireless base station over the wireless link and the other between the wireless base station and the receiver over the wired link.

I-TCP utilizes the resources of Mobility Support Routers (MSRs) to provide transport layer communication between mobile nodes and nodes on the fixed network. With I-TCP, the problems related to mobility and the unreliability of wireless links are handled entirely within the wireless link; the TCP/IP software on the fixed nodes is not modified.

I-TCP, like other split-connection proposals, attempts to separate loss recovery over the wireless link from that across the wired network, thereby shielding the original TCP sender from the wireless link.

2.7.3 SCTP

Stream Control Transmission Protocol (SCTP) [**36**] is a reliable transport protocol operating on top of a connectionless packet network such as IP.

SCTP was designed to supply some of the limitations of TCP. For example, TCP provides both reliable data transfer and strict order-of-transmission delivery of data, but some applications need reliable transfer without sequence maintenance, while others would be satisfied with partial ordering of the data. In both of these cases the head-of-line blocking offered by TCP causes unnecessary delay. On the other hand, the stream-oriented nature of TCP is often an inconvenience. Applications must add their own record marking to delineate their messages, and must make explicit use of the push facility to ensure that a complete message is transferred in a reasonable time. Moreover, the limited scope of TCP sockets complicates the task of providing highly-available data transfer capability using multi-homed nodes.

SCTP offers acknowledged error-free non-duplicated transfer of user data, data fragmentation to conform with the corresponding MTU size, sequenced delivery of user messages within multiple streams with an option for order-of-arrival delivery of individual user messages, optional bundling of multiple user messages into a single SCTP packet, and network-level fault tolerance through supporting of multi-homing at either or both ends of an association. Moreover, the design of SCTP includes appropriate congestion avoidance behavior and resistance to flooding and masquerade attacks.

2.7.4 ATCP

Ad-hoc TCP (ACTP) [**37**] is a TCP adaptation specifically designed for ad-hoc networks. It is implemented as a thin layer inserted between the standard TCP and IP layers.

ATCP relies on network layer feedback to detect and distinguish congestion loss from error loss, and on the ICMP Destination Unreachable message to detect a change of route or temporary partition in the ad-hoc network. According to these feedbacks from the network, the ATCP layer puts the TCP sender into either a persist state, congestion control state or retransmit state.

When the ICMP message indicates a route change or network partition, ATCP puts the TCP sender into persist mode waiting for the route to reconnect so that it does not needlessly transmit and retransmit packets. On the other hand, when packets are lost due to high BER (as opposed to congestion) the ATCP sender retransmits packets without invoking congestion control; for packet reordering, ATCP reorders the packets so that TCP would not generate DUPACKs. Finally, when the network is truly congested, the TCP sender invokes congestion control normally.

2.8 Session Layer

Another problem of adapting the current solutions for MANETs is that they do not exploit the inherent properties of this type of networks:

- Broadcast properties of the wireless medium
- Diversity of nodes in a MANET
- Delay-tolerance properties in a mobile environment

Exploiting the advantages of the broadcast medium of the wireless networks instead of the unicast medium of the wired networks, exploiting the diversity that the plurality of nodes of a MANET offer and exploiting the delay-tolerance properties that any mobile environment have seem to be obligatory for building a powerful communications stack for MANETs.

The basis for exploiting these three properties is the concept of what is called Distributed Object. Encapsulating data as distributed objects can be of special benefit in the case of MANETs. Distributed objects can be stored in any node of the network (thus exploiting the diversity of nodes), can be obtained from any of these nodes (thus exploiting the broadcast properties of the wireless medium), and can be obtained at any time independently of if the originator of the object has left the network or not (thus exploiting the delay-tolerance properties of a mobile environment).

In TCP/IP, the communications stack ends up with the socket interface [**38**] and researchers have tried to incorporate improvements to this layer in higher level solutions. On the one hand, the Common Object Request Broker Architecture (CORBA) [**39**] is the main proposal of a distributed object technology. On the other hand, the Java Remote Method Invocation (Java RMI) [**40**] is a Java application programming interface that performs the object-oriented equivalent of CORBA. And as a third alternative, the PYthon Remote Object (Pyro) [**41**] is a distributed object system written entirely in Python.

The basis of all these distributed objects solutions is to never worry about writing network communication code. These solutions take care of the network communication between the distributed objects once they are splitted over different machines on the network. All the tricky socket programming details are taken care of. The idea is just to call methods on these remote objects as if they were local objects.

2.8.1 Common Object Request Broker Architecture (CORBA)

As the name indicates, CORBA [**39**] is a Broker architecture. That is, there is a logically centralized entity called an ORB (for Object Request Broker) that mediates all communication. CORBA is a distributed object platform, meaning that the components of the system are objects. The method invocations on objects thus go through the ORB, which is responsible for locating the target object, invoking the method, and returning the result.

Current CORBA versions have incorporated the CORBA Messaging specification inside. The Messaging specification defines, primarily, two different parts, the Asynchronous Messaging Interface (AMI) and Time Independent Invocations (TII). AMI provides the possibility of asynchronous invocations and TII provides message routing functionality.

As is common with CORBA-related specifications, AMI includes both options for asynchronous request-response interaction, namely polling and callbacks. CORBA also takes advantage of the fact that an invocation always looks the same to the server, so an AMI implementation is purely a client-side matter. Also, in the callback style, the callback object provided in the invocation is a CORBA object like anything else, so it doesn't have to live in the same address space or even host as the client.

TII is a way for the client to specify certain CORBA objects that act as routers for the message. The idea behind this is that if the client and server are only intermittently connected to the network, it is possible that they are too rarely connected together, making direct communication between them infeasible. Thus, the object reference can specify a number of routers through which the message can be passed. These routers form a store-and-forward or delay-tolerant network, allowing intermittent connectivity of the client and server.

CORBA is nothing if not flexible in its use of protocol. Originally, there was no protocol specified by CORBA, so ORBs from different vendors would not interoperate unless the vendors made a specific effort to do so. This changes with version 2 and the interoperability specification for CORBA. Still, CORBA allows any communication protocol to be used, and provides only some guidelines for implementing protocols.

One specific protocol was defined to be mandatory for interoperable CORBA implementations, though. This is the General Inter-ORB Protocol, or GIOP for short. GIOP is a message-based protocol that is intended to be run over some transport protocol. The requirements that GIOP places on the transport protocol are pretty much those that are provided by TCP, and accordingly, the Internet Inter-ORB Protocol, or IIOP, which is GIOP running on top of TCP, is also a mandatory part of CORBA interoperability.

The primary messages of GIOP are Request and Reply. These comprise the regular invocation behavior in CORBA. It is also possible for the client to send a LocateRequest message, the reply

to which will tell the client where the object is located. This allows the client to locate the object for certain without needing to invoke with the full request, as an invocation reply can be a redirection. This is especially useful for load balancing with a so-called Implementation Repository.

Other parts of GIOP are less often used. It is possible to fragment a GIOP message, similarly to what IP does, but here the intent is to save buffer space of the ORB if the message is too large to hold in memory while being constructed. A client can also cancel a request at any time before receiving a reply. Finally, there is some connection management in the form of the ability to close the transport-layer connection from the server end. This is required at the GIOP level because, while a closed transport connection can be detected, it is possible that a client request had been initiated when the server decided to close the connection, so the client needs to be informed of this to be able to resend any outstanding requests on a newly-opened connection.

The roles in basic GIOP are rigidly defined: the party opening the transport connection is the client and the other party is the server, and the client is allowed to send only requests and the server only replies. This causes problems, for instance, in callback scenarios, which are very useful in distributed systems. For one, opening a second connection is wasteful of resources. But more importantly, it is not always even possible for the server to open a connection to the client, as firewalls may be blocking the communication. This is why GIOP now also includes bidirectional functionality: the ability to use the same transport connection for invocations in both directions. This is commendable, but the original design does show quite clearly how people become fixated on the idea of reflecting transport-layer roles on the application layer as well.

2.9 Other Layers

In a mobile wireless environment, security and power saving are two other important layers for the correct and efficient operation of the communications stack. Both security and power saving layers are out of the scope of this thesis.

3. MStack

Our main goal in the design of the Miraveo Stack (MStack) was to build a complete communications stack for MANETs ready to be installed as a simple piece of software in any kind of Wi-Fi-enabled mobile device. For this reason, our communications stack is placed on top of the TCP/IP stack, since the TCP/IP stack is the most common communications stack found in any network device. However, the MStack could be implemented inside the operating system of the device as a native communications stack, as an alternative to the TCP/IP stack.

We placed the MStack over UDP, in order to be able to implement our own networking protocols over the TCP/IP stack without being constrained by the flow control mechanisms of TCP. We use UDP just as a port selector, as many real-time streaming protocols and applications do.

In Figure 15 we can see the scheme of all the protocols involved in the MStack. Regarding the Cooperative Forwarding protocol, this is the only layer included in the OS of the device. Specific configurations of the Wi-Fi device are needed in order to be able to use the Cooperative Forwarding protocol, and so, it cannot be included as part of the MStack as a simple piece of software over the TCP/IP stack. In Figure 16 we can see the scheme of the MStack when Cooperative Forwarding is not used.

All the protocols of the MStack are grouped in three different planes: the data plane, the control plane, and the management plane. The data plane is the plane in charge of performing the real per-packet processing of the packets between the application and the network. Mainly, the data plane uses a data structure (the Miraveo Communications Table or MICO Table) to know for each packet how to process it, and it must be efficient enough to allow high packet processing rates. The data plane is composed by the Delay Tolerant Object Layer Abstraction (DELTOYA), the Patient Transport Protocol (PTP) and the Miraveo Forwarding (MIFO).

DELTOYA offers to the applications a set of primitives to manage pieces of data that we call Delay Tolerant Distributed Objects (DTDOs). A DTDO is any piece of data managed by DELTOYA. Examples of DTDOs are data files, directories, voice streams, etc. DTDOs can be referred to independently of the node which has originated them, meaning that a DTDO can be stored in nodes different from the one which has originated it, and that the DTDO can be accessed even when the node which has originated the DTDO is not already part of the network. DTDOs can be originated with an assigned Time to Life (TTL), which provides a mechanism for limiting the time the DTDO is kept in the network. DELTOYA offers primitives to the Applications for sending, requesting or sharing DTDOs, creating and assigning an identifier or a TTL to a DTDO, etc.

PTP includes mechanisms for ensuring a reliable transport of information through the network and for performing the control of transmission rates so that the network does not come into a congestion condition. PTP offers communication services to the mechanisms of DELTOYA and uses the communication services of MIFO.

A distinctive characteristic of the mechanisms of PTP is that they can handle transparently the possibility of retrieving the data of a DTDO from different nodes of the network, taking into

account the delay-tolerant properties that characterize the MStack. For instance, if a DTDO is a data file, PTP can retrieve different parts of the data file from different nodes and at the same or different time.

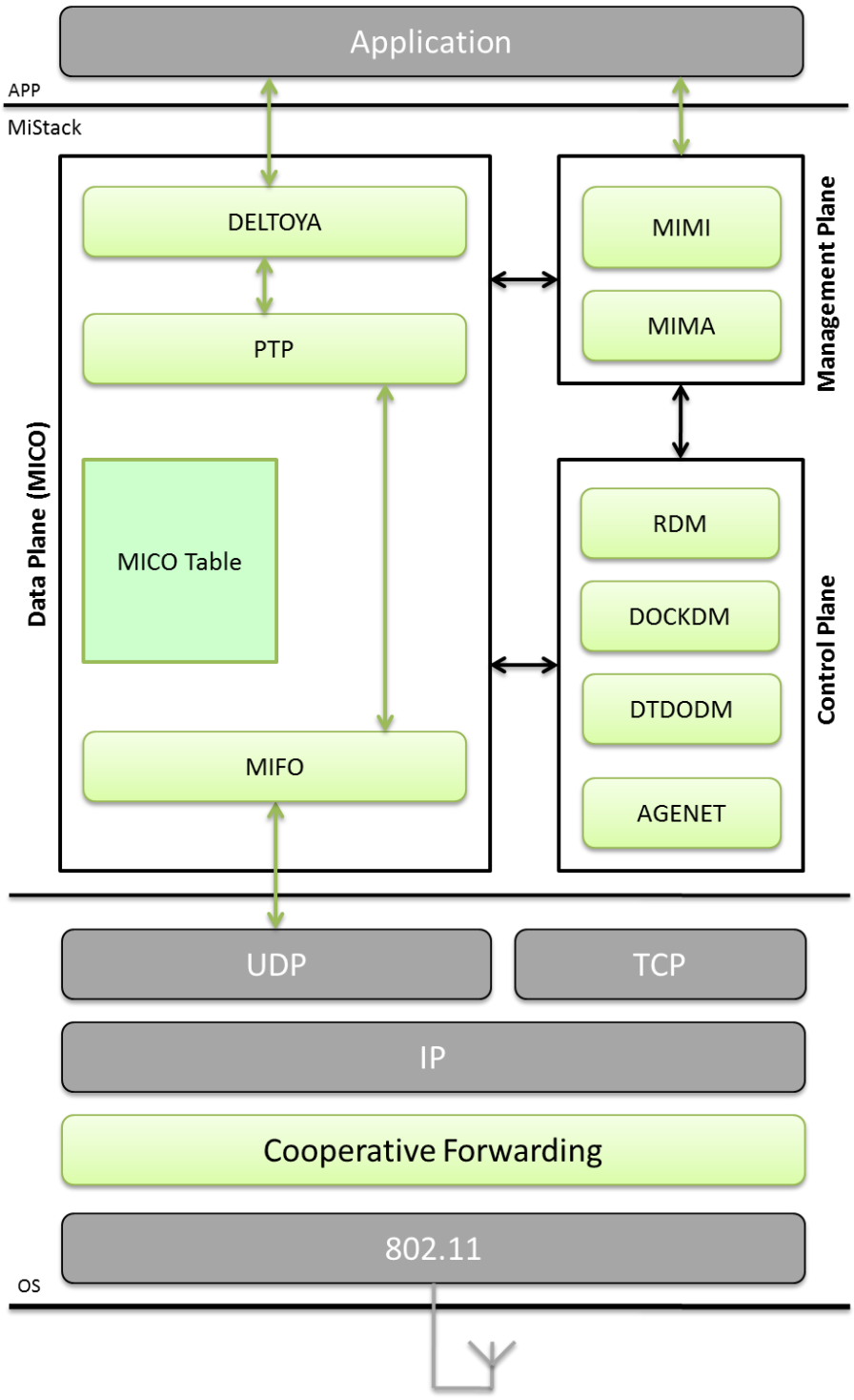

Figure 15. MStack

The retrieval mechanisms of PTP use a Maximum Retrieval Time (MRT), meaning that they are persistent in the sense that even if during a given period of time, the data is not available in the network, the mechanisms persists in their task of data retrieval, until the overall object is retrieved or the given timeout expires.

MIFO is the forwarding mechanism of the MStack. In a MANET, where multihop routing is present, MIFO is needed in order to be able to send a packet between a source and a destination node, using zero, one or more intermediate nodes as forwarders of the packet if it is the case.

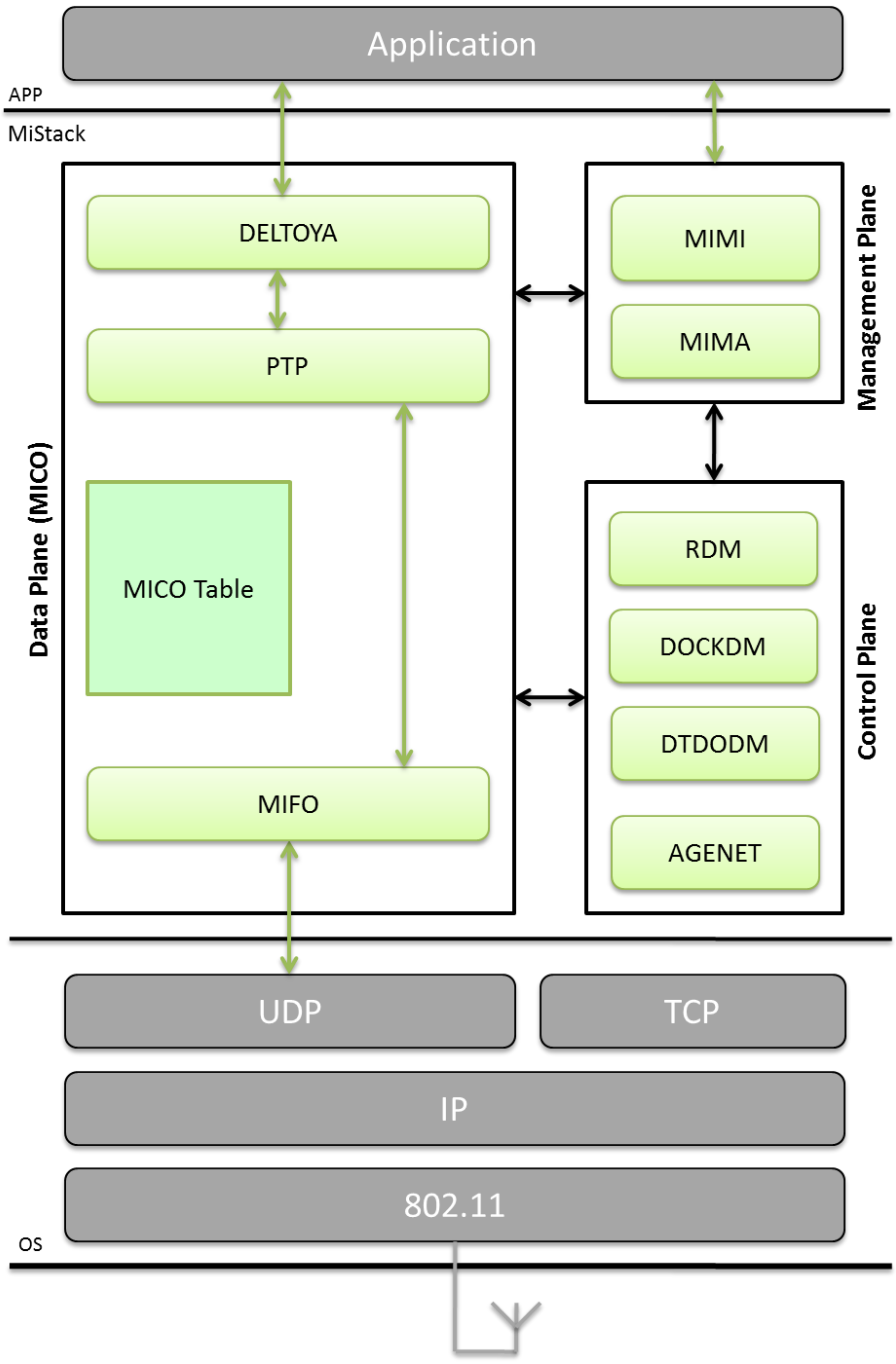

Figure 16. MStack (without Cooperative Forwarding)

Moreover, MIFO is in charge of decoupling the MStack from the IP protocol as a network identifier. In the TCP/IP stack, IP addresses are used as node identifiers in the network. With MIFO, unique node identifiers independent of the IP addresses are used and managed in order to support mobility at network layer. In other words, even when a node changes of IP address

(due to for example a handover between two MANETs), the MStack and even the communications in progress will not be affected.

On the other hand, the control plane is the plane in charge of providing all the information that the MICO Table has to have in order to allow the correct operation of the data plane. The control plane is composed by the DTDO Discovery Mechanism (DTDODM), the DOCK Discovery Mechanism (DOCKDM), the Route Discovery Mechanism (RDM) and the AGgrEssive Networking Protocol (AGENET). All these components include mechanisms for determining routing paths along a network and for discovering DTDOs in the network.

Finally, the management plane is the plane where additional functionalities not directly involved in the data plane but needed for its correct operation are implemented. The management plane is composed by the Miraveo Middleware (MIMI) and the Miraveo Management (MIMA).

MIMI and MIMA are the entities in charge of managing all the functionalities related with the management of a MANET, providing at the same time both address and name resolution services and location based services.

4. Cooperative Forwarding

In section 2.6 we described three proposals of opportunistic forwarding. In the current section, a new proposal of opportunistic forwarding called Cooperative Forwarding is presented.

4.1 Cloud Addresses

Groups of cooperators, called *clouds*, are formed in a distributed and opportunistic manner around the nodes chosen by the routing protocol. Nodes in a *cloud* coordinate their operation by overhearing data and acknowledgment frames. As a rule, the existence of cooperators cannot be guaranteed, and hence if a node fails to find cooperator nodes, it will continue operating as if it was in a non-cooperative network.

Let S be a node in a wireless network. R(S) is the set of nodes receiving the signal transmitted by S within certain parameters of quality, which might correspond, for example, to minimum signal-to-noise ratio (SNR).

Let us assume that S decides to transmit a frame to node D $\in$ R(S). Let us also assume that there is a set of nodes, which we will call *cooperators*, whose conditions for transmission to D are especially favorable, probably because they are in the vicinity of D, and which are also willing to cooperate with D. We will call this set, in which node D is included, a *cloud*, and it is denoted by C(D). When S transmits the frame addressed to D using the wireless medium, the nodes R(S) $\cap$ C(D) will also receive the corresponding signal and will be able to decode it within the minimum quality parameters established. Membership to C(D) will vary over time due to node mobility, and it is established through exchange of periodic *Cooperative HELLO* (CHELLO) messages at times t_n. To introduce this variability into our formulation, clouds will be noted as $C(D,t_n)$, where t_n is the *cloud number*; see Figure 17.

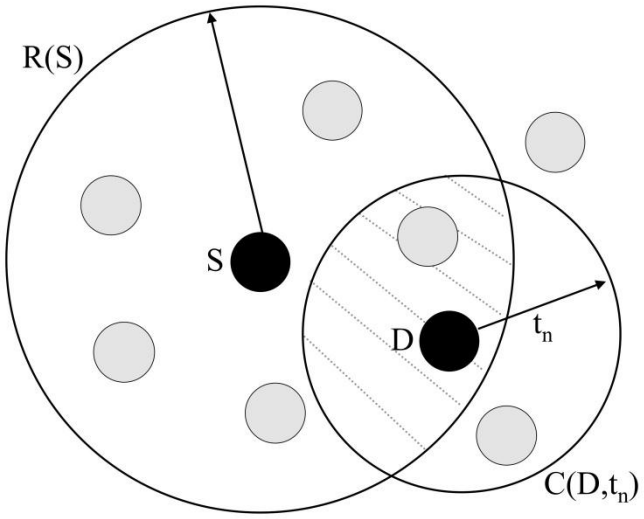

Figure 17. Cloud Addresses in Cooperative Forwarding

The cloud number is a value managed by every node of the network, and it is incremented periodically. It is sent through CHELLO messages to both the possible cooperator nodes (in order to update the cloud throughout time) and to the previous and nexthop nodes (in order to let them know which cloud number to use when they have to send data packets to this

cloud). The cloud number enables *obsolete cooperators* to be detected and thus preventing useless or duplicated cooperation.

The operation of the Cooperative Forwarding mechanism can be simplified by introducing an addressing scheme that enables us to refer not only to the D nodes, but also to the cloud $C(D,t_n)$. The addressing scheme that we propose is as follows:

- Address of node D: <@D,0>
- Address of cloud $C(D,t_n)$: <@D,t_n>

where @D is the link layer address (for example, the MAC address) of the wireless interface of node D. We will assume that addresses such as the ones specified above are used as transmission and reception addresses in every hop of the routing path. Source and destination nodes of the routing path are identified by their network addresses (for example, the IP addresses).

In general terms, the proposed forwarding mechanism can be thought as *forwarding through clouds*, meaning that frames are addressed to the next cloud instead of the next hop node; see Figure 18. The robustness is achieved by means of diversity: the likelihood that at least one node of a cloud can forward the packet will be higher than the likelihood that the packet can be forwarded by the specific node chosen by the routing protocol. Notice that in wireless transmissions, the cost, in terms of number of transmissions, of sending the data frame to all nodes of the cloud is the same as sending the frame to only the nexthop node.

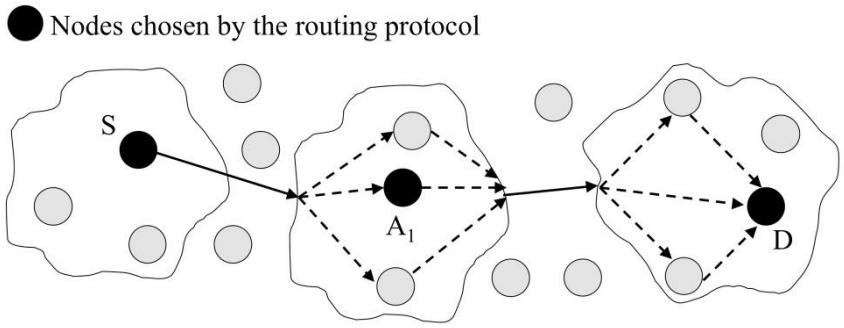

Figure 18. Forwarding through Clouds: frames are addressed to the next cloud

4.2 Cooperative Forwarding Protocol Operation

Let us assume now that S wishes to send a frame addressed to $D \notin R(S)$. We need to use a routing protocol to establish a routing path between the two nodes. Let us say that an ordered set of nodes $(A_0,A_1,A_2,...,A_{n-1},A_n)$ with A_0=S, A_n=D is a path between S and D when $A_i \in R(A_{i-1})$ and $A_{i-1} \in R(A_i)$, i=1...n.

At this point it is important to distinguish between the nodes that define the path -the elements A_i selected by the routing protocol called *master* nodes-, and the nodes that perform the actual per-packet-forwarding function –the elements L_i called *relay* nodes-. Each hop has a single relay node $L_i \in C(A_i,t_n)$. L_i forwards the frames to the set of nodes $C(A_{i+1},t_n) \cap R(L_i)$, which will include the next relay node L_{i+1}. Normally, the master nodes will also be the relay nodes used in the communication, although this behavior will not always be the case. See an example in Figure 19.

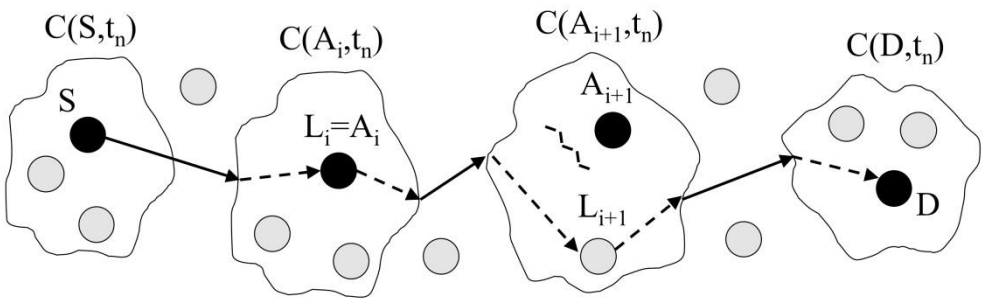

Figure 19. Master and Relay Nodes examples

Then, the Cooperative Forwarding mechanism is as follows:

L_{i-1} sends a (unicast) data frame with reception address <@A_i,τ>, τ being the last cloud number of the cloud $C(A_i,t_n)$ denoted by L_{i-1}. All nodes of $C(A_i,t_n)$, with their last known cloud number of the cloud $C(A_i,t_n)$ *greater or equal than* τ, and which receive the frame correctly will store the data frame in a buffer, waiting for a possible cooperation.

When node A_i receives the data frame correctly, it generates back an 802.11 ACK frame which will be overheard by the other nodes of $C(A_i,t_n)$ and will cause these nodes to stop waiting for cooperating and to discard the buffered data frame. Therefore A_i will forward the data frame to $C(A_{i+1},t_n)$ (if it is not the destination node).

If, after a certain period of time, a node of $C(A_i,t_n)$ other than A_i has not received the link layer acknowledgement (for example, the 802.11 ACK frame) of A_i, waiting for it and taking into account all the possible retransmissions of the data frame by the relay L_{i-1}, it will be able to cooperate. This situation can occur when the link layer acknowledgement of A_i is lost or when the link layer acknowledgement is not generated due either to the fact that A_i has not correctly received the data frame or that A_i is not present. At this point, this cooperator node will send a special frame, called *Cooperative ACK* (CACK), which will be overheard by L_{i-1} and some nodes of $C(A_i,t_n)$, causing the same effect as the link layer acknowledgement. Therefore, it will forward the data frame to $C(A_{i+1},t_n)$ (or to A_i in the case that A_i is the destination node). Note that neighborhood is not required between cooperators of the same cloud.

Moreover, if the node which forwards the data frame is the master of the cloud (i.e. $L_i=A_i$), while the previous node which has sent the data frame is not the master of the previous cloud (i.e. $L_{i-1}\neq A_{i-1}$), then the master node A_i will not only generate the link layer acknowledgement but also a CACK, in order to send to the previous relay node L_{i-1} the feedback necessary to modify its cooperative behavior.

In order to coordinate the operation of different possible cooperators in the cloud, cooperator nodes delay their cooperative action following a predetermined order set by the master node of the cloud and based on the SNR of the CHELLO messages received from these possible cooperators. Moreover, in order to avoid cooperation of *bad cooperators* (i.e. cooperators which cannot reach a valid relay in the next cloud), cooperator nodes use a *heuristic* based on both the SNR of the CHELLO messages received from the next cloud and the CACKs that they should receive as a result of good cooperation.

In Figure 20 we can see an example where L_i (=A_i) receives a data packet (1), sends the corresponding link layer acknowledgement (2) and forwards the data packet (3) to L_{i+1} (≠A_{i+1}), which will send the corresponding CACK (4). Notice that the cooperative forwarding mechanism is performed in a per-packet basis.

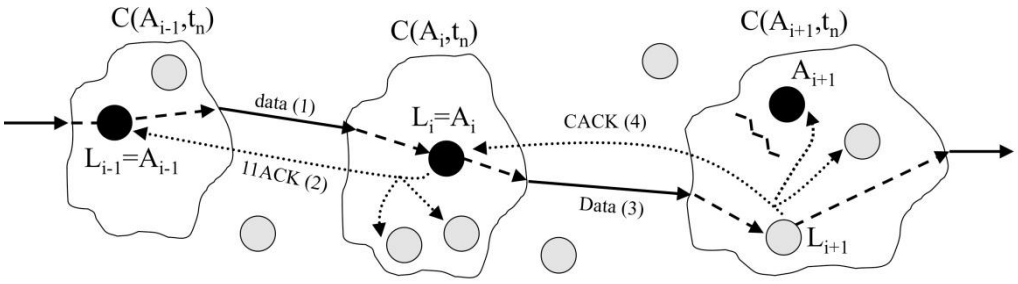

Figure 20. The Cooperative Forwarding mechanism: how it works

A possible problem in this mechanism is what we call *broken clouds*. They arise when a node of $C(A_i,t_n)$ does not receive the link layer acknowledgement or CACK frame of the relay node of its cloud, due to (i) fading, shadowing or temporal failures of the wireless links inside the cloud, to (ii) node mobility, when this node of $C(A_i,t_n)$ goes beyond the limits of the cloud or due to (iii) there is no link between this node of $C(A_i,t_n)$ and the relay node of the cloud. A broken cloud has the effect of forwarding the same packet for duplication by two or more relay nodes of the same cloud. This behavior must be considered in order to prevent the overhead of these duplicated packets. In (ii), the cloud number t_n is used to detect the broken cloud. However, in order to solve the cases (i) and (iii), and make the case (ii) more efficient, our mechanism uses the packet global identifier field of the cooperative header to detect these duplicated packets. This detection will be used both to prevent sending CACKs for duplicated packets and to prevent forwarding the duplicated packets more than one hop.

Cooperation implies that cooperator nodes should have a copy of the forwarding tables of the masters of their clouds, in order to act as theses masters in the case that they have to cooperate. These copies are obtained in the periodic CHELLO messages sent by the masters of the clouds.

As an example, in Figure 21, node A chooses a single routing path (A,C,E,G) to reach G. Node A forwards the packets to C and to its cooperators (with just one wireless transmission). Node D, which is a cooperator of C, overhears the acknowledgements from C and only if it detects that C has not received (i.e., acknowledged) a packet, it forwards the packet to E.

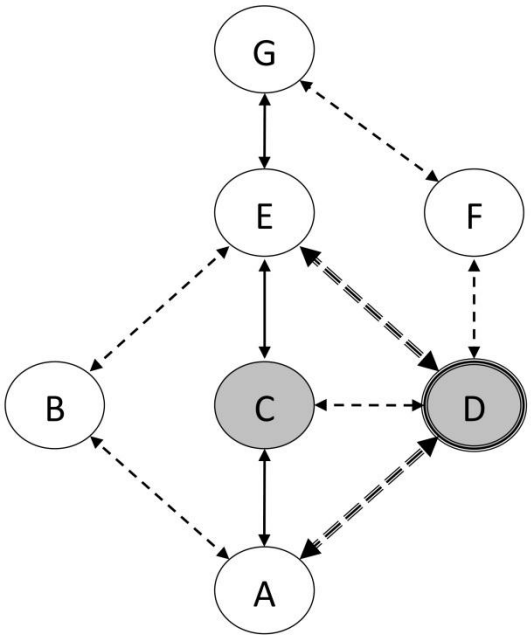

Figure 21. Example of Cooperative Forwarding protocol operation

4.3 Comparison of the Four Main Proposals of Opportunistic Forwarding

SDF [**26**] exploits diversity by selecting a set of possible relays before packet transmission. This leads to an increased likelihood that a packet will reach its intended destination, although it introduces an extra overhead due to handshake packets needed for relay coordination. SDF works well with standard TCP, although it cannot be implemented using the IEEE 802.11 Standard. The handshake process requires several modifications to the link layer technology, as for example, the need of adding all the MAC addresses of the several candidate nodes to the link layer header. Moreover, although no mechanism is specified in SDF, a multipath routing protocol or a similar mechanism will be needed in order to select the possible set of candidate nodes, meaning that this election will be performed by the transmitter of the packet. In the same way, the transmitter of the packet, in a centralized manner, will be the responsible of the relay election.

ExOR v1 [**27**] works well with standard TCP, but it cannot be implemented using the IEEE 802.11 Standard. Again the handshake process requires several modifications to the link layer technology, as for example, the need of adding all the MAC addresses of the several candidate nodes to the link layer header. In ExOR v1, the transmitter of the packet performs the election of the candidate nodes. However, the receivers of the packet, in a distributed manner, are the responsible of the relay election, meaning that, unlike SDF, in ExOR v1 candidate nodes should overhear each other in order to avoid duplicated packet transmissions.

ExOR v2 [**28**] can be implemented using the IEEE 802.11 Standard, but it does not work well with standard TCP due to the use of batches of packets. If ExOR v2 were layered under TCP, ExOR's batches would interact badly with TCP's window mechanism. If the end-to-end loss rate were not very low, TCP would use a window size too small to allow ExOR to accumulate the minimum number of packets required for an efficient batch.

MORE [**29**] avoids relay coordination through the use of intra-flow network coding. Although MORE can be implemented using the IEEE 802.11 Standard, it does not work well with standard TCP due to the use of batches of packets. Moreover, although no mechanism is specified in MORE, a multipath routing protocol or a similar mechanism will be needed in order to be able to check whether the nodes are closer to the destination node than the previous hop of the overheard packets.

Cooperative Forwarding exploits diversity by selecting a set of possible relays during (and not before) packet transmission. Cooperative Forwarding works well with standard TCP, and it can be implemented using the IEEE 802.11 Standard, although slightly modifications in the link layer technology could highly improve its performance.

Table 1 summarizes the main characteristics of the four proposals.

Table 1. Comparison of different opportunistic forwarding proposals: SDF, ExOR, MORE and Cooperative Forwarding

	SDF	**ExOR v1**	**ExOR v2**	**MORE**	**Cooperative Forwarding**
Proposed with 802.11 Standard	No	No	Yes	Yes	Yes
Works well with standard TCP	Yes	Yes	No	No	Yes
Routing	Multipath	Multipath	Multipath	Multipath	Unipath
Cooperators election	Transmitter	Transmitter	Transmitter	Transmitter	Receiver
Relay election	Transmitter (centralized)	Receiver (distributed)	Receiver (distributed)	Not needed	Receiver (distributed)

5. MICO Table

DELTOYA, PTP and MIFO have their own functionalities and provide their own services. However, they have to work together in order to optimize at maximum their operation. At the same time, other protocols of other planes, like AGENET of the control plane, also need to work together with the protocols of MICO. For all these reasons, a data structure called Miraveo Communications Table (MICO Table) is shared by all of them. The MICO Table is divided into three sub tables: the DTDO Table, the DOCK Table and the CHAIN Table.

5.1 DTDO Table

The DTDO Table is divided into two sub tables, and entries of one sub table can be linked with some entries of the other sub table in order to establish recipient relationships.

The first sub table has information related with DTDOs: each entry of this sub table corresponds to a Delay Tolerant Distributed Object (DTDO). A DTDO is any piece or source of data that in general may support the properties of being distributed and delay-tolerant. DTDOs may be referred to independently of the node which has originated them, meaning that DTDOs may be stored in nodes different from the one which has originated them. For instance, they may be accessed even when the node that has originated them is not already part of the MANET.

The second sub table of the DTDO Table has information related with DTDO recipients: each entry of this sub table identifies a routing path to a destination node.

On the other hand, recipient relationships between entries of each sub table can exist: a recipient relationship indicates which nodes (destination nodes of the second sub table) are recipients of which DTDOs (entries of the first sub table), i.e., in which nodes of the network a DTDO could be found.

See appendix “B.1 DTDO Table” for more details on the operation of the DTDO Table.

5.2 DOCK Table

The DOCK Table has information related with DOCKs. A DOCK is a container of DELTOYA where DTDOs can be pushed by other nodes of the network or where DTDOs can be picked up by the application that created the DOCK.

Each entry of the DOCK Table has a DOCK Buffer. The DOCK Buffer is the data structure where the DTDOs pushed into a DOCK are kept until they are picked up from the DOCK.

When a DTDO has to be pushed to a DOCK, we will proceed to insert the new entry to the DOCK Buffer of the corresponding DOCK. The DOCK Buffer can be seen as a particular type of a FIFO queue: we will maintain the entries of the DOCK Buffer ordered by arrival time between different senders and by order inside the CHAIN between entries of the same sender and CHAIN. See four examples in Figure 22, Figure 23, Figure 24 and Figure 25.

On the other hand, when an entry has to be picked up from a DOCK Buffer, we will start checking if the first entry can be picked up (depending on the CHAIN properties of the entry) and we will keep checking entries until we found the first one that can be picked up or until we discover that no entry can be picked up at that moment.

See appendix “B.2 DOCK Table” for more details on the operation of the DOCK Table.

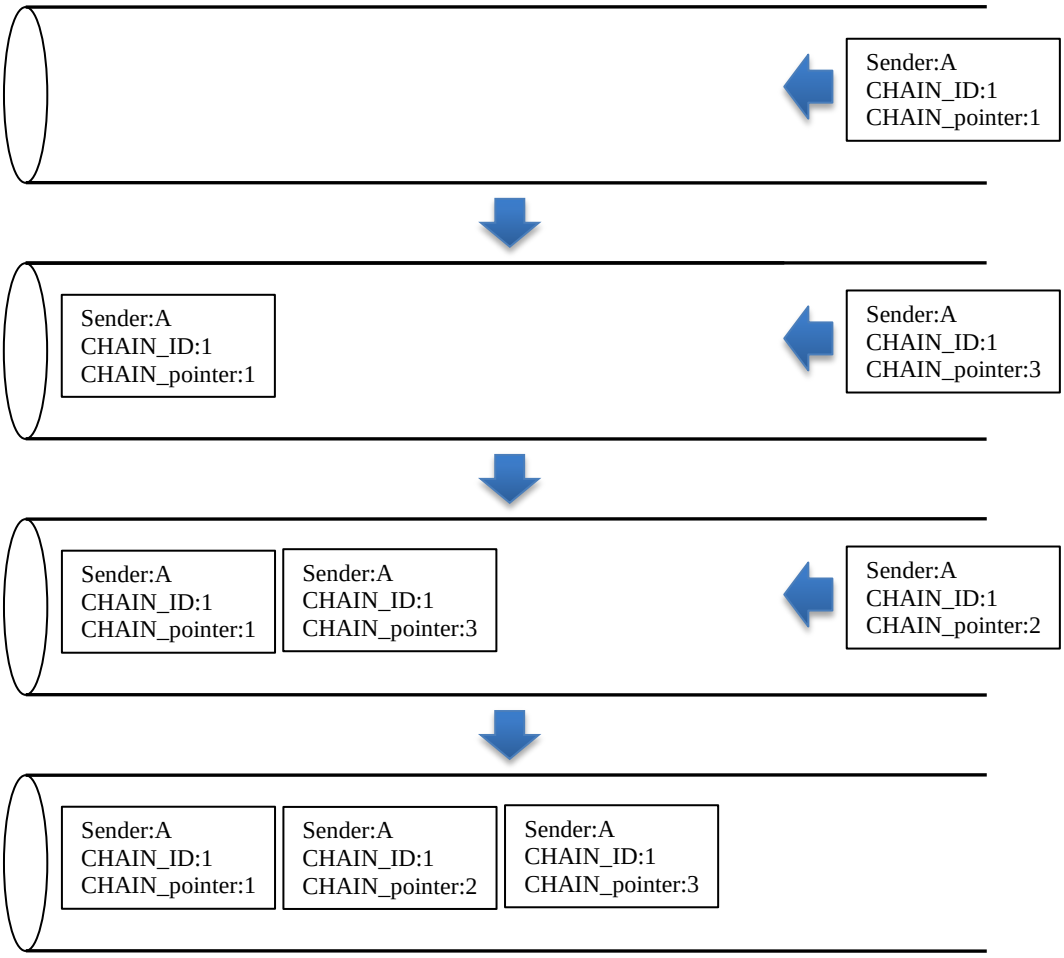

Figure 22. DOCKBuffer Example 1

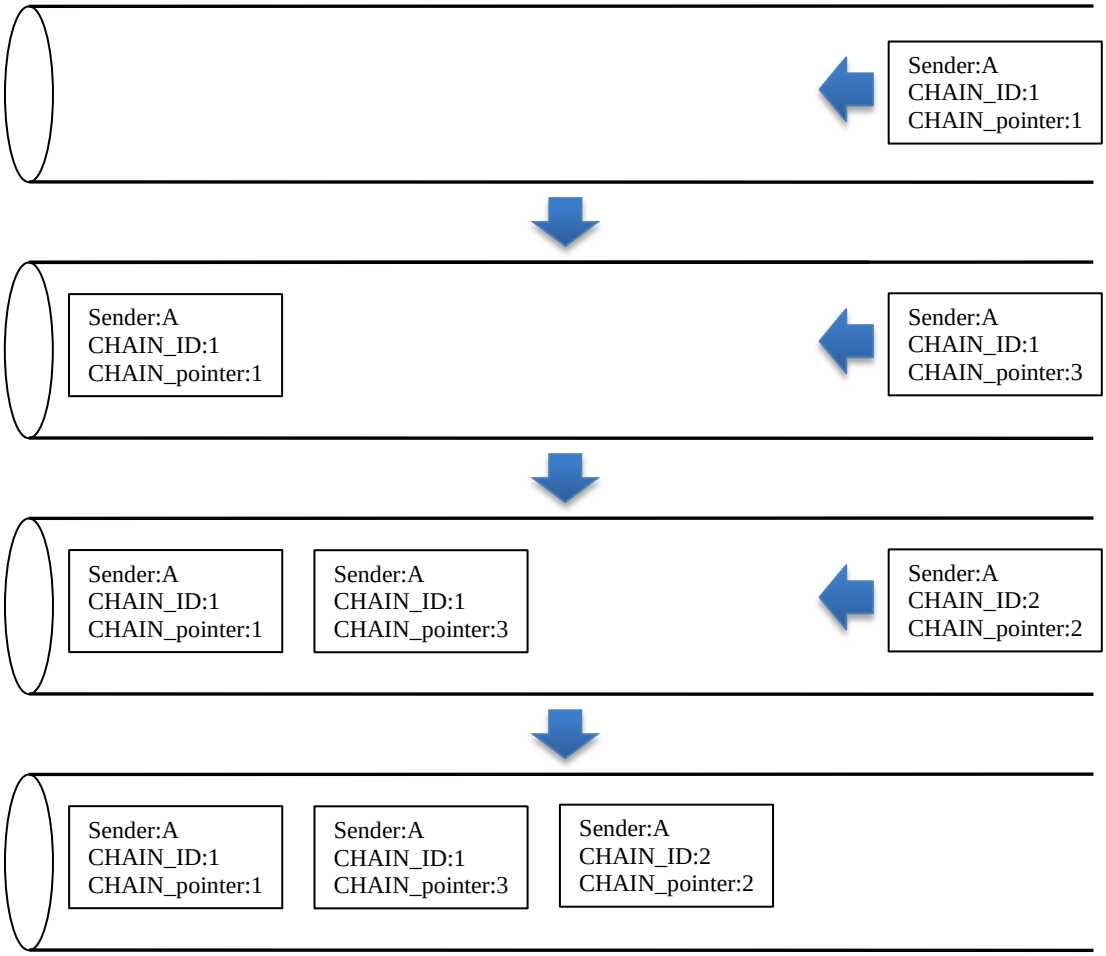

Figure 23. DOCKBuffer Example 2

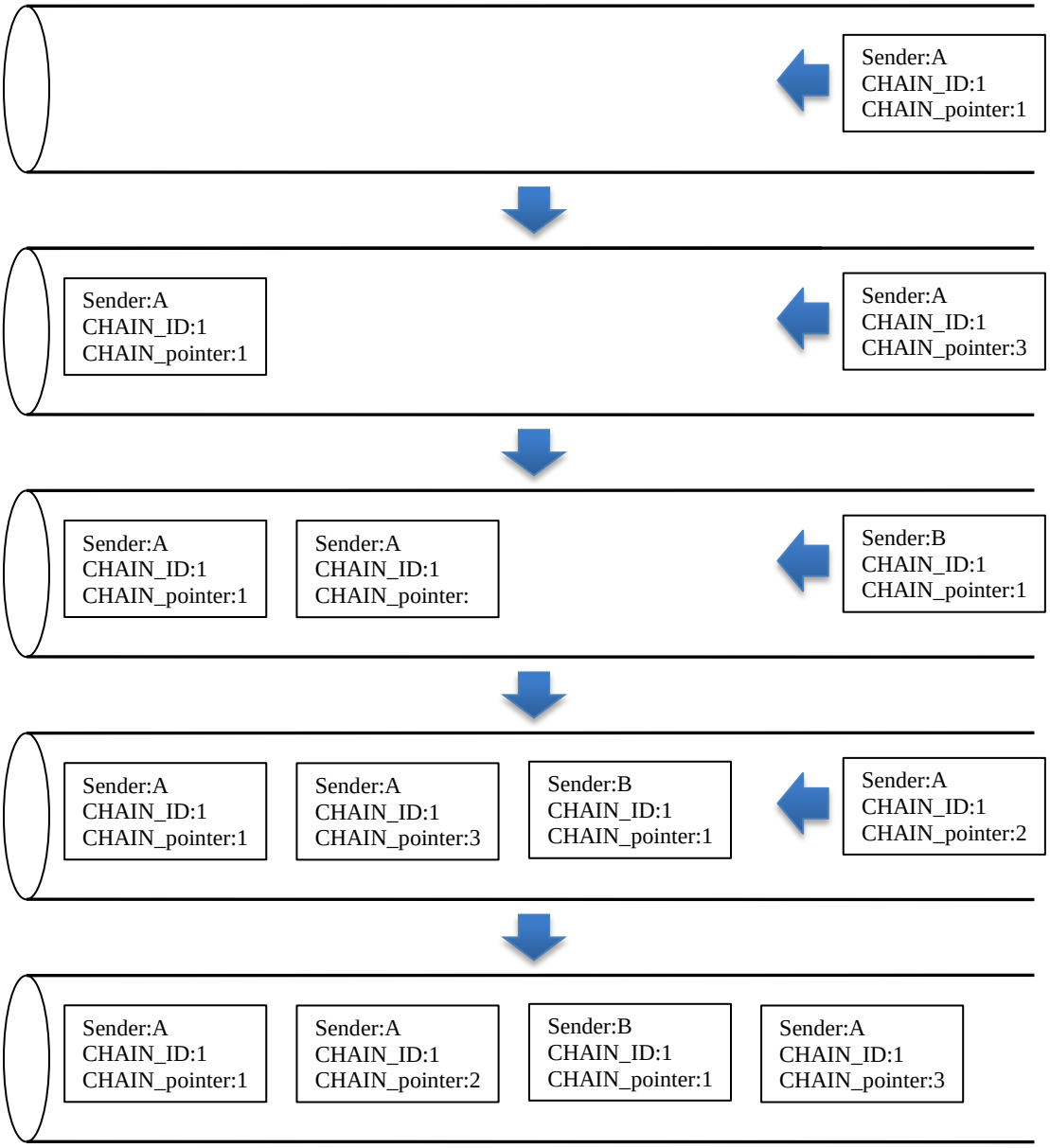

Figure 24. DOCKBuffer Example 3

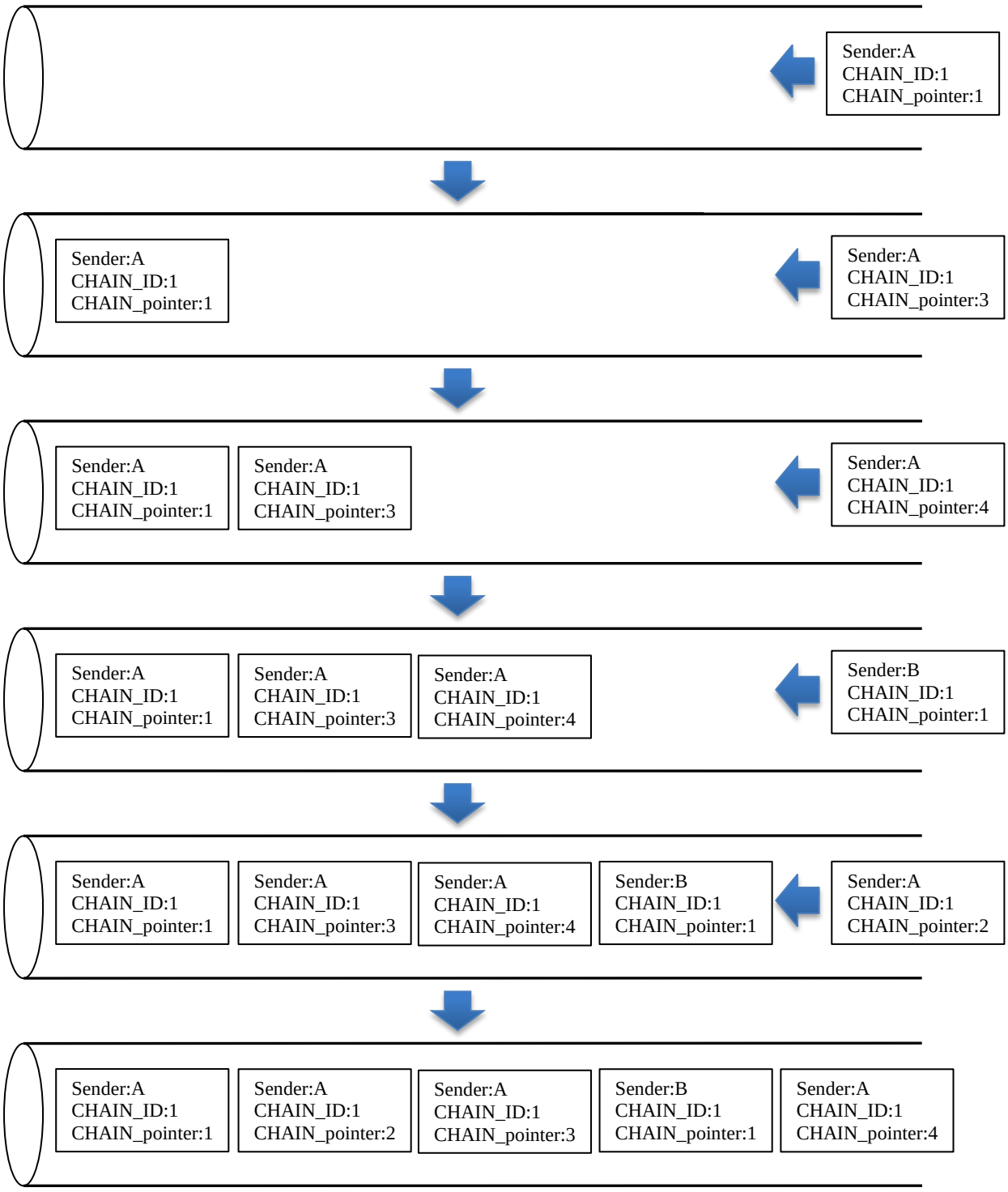

Figure 25. DOCKBuffer Example 4

5.3 CHAIN Table

The CHAIN Table is the data structure used to manage with CHAINs of DTDOs (or just CHAINs to simplify) created by an application. Each entry of the CHAIN Table corresponds to a specific CHAIN of an application.

A CHAIN is an ordered sequence of DTDOs indicating in which order they have to be picked up from a DOCK once they are pushed to this DOCK by a sender node. At the same time, a CHAIN

has a degree of resistance, meaning that if there is a gap in the CHAIN (i.e., some intermediate DTDOs of the CHAIN have been lost through the network), then we can define if we allow continuing picking up the subsequent DTDOs of the CHAIN from the DOCK after a certain period of time (i.e., the CHAIN resistance) while giving up the lost DTDOs, or if we do not allow continuing picking up DTDOs of this broken CHAIN from the DOCK and we wait forever for the DTDOs of the gap (i.e., CHAIN resistance equivalent to infinity).

See appendix “B.3 CHAIN Table” for more details on the operation of the CHAIN Table.

5.3.1 RXCHAIN Table

The RXCHAIN Table is the data structure used to manage with CHAINs of DTDOs received by an application.

Each entry of the RXCHAIN Table corresponds to a specific CHAIN of a specific sender node in the sense that at least one of the DTDOs of this CHAIN pushed by the sender node has been received into one of the DOCKs of the receiver node.

See appendix “B.3.1 RXCHAIN” for more details on the operation of the RXCHAIN Table.

5.3.2 RXUNCHAIN Table

The RXUNCHAIN Table is the data structure used to manage with unchained DTDOs received by an application, in order to be able to detect already pushed and picked up unchained DTDOs.

Each entry of the RXUNCHAIN Table corresponds or identifies a specific push of an unchained DTDO in the node receiving the push.

See appendix “B.3.2 RXUNCHAIN” for more details on the operation of the RXUNCHAIN Table.

6. DELTOYA

The DELay Tolerant Object laYer Abstraction (DELTOYA) is a novel mechanism that exploits cooperation, delay-tolerance and diversity in order to build a robust communication system among nodes of a MANET. The idea behind DELTOYA is, instead of building a node-oriented communications stack like TCP/IP, to build a data-based communications stack: the important thing is the data, not how or from where to get these data.

DELTOYA provides means for handling Delay Tolerant Distributed Objects (DTDOs). A DTDO is any piece or source of data that in general may support the properties of being distributed and delay-tolerant. Examples of DTDOs are data files, directories, etc. Some types of data that in general are not distributed or delay-tolerant, like for instance voice or video streams, may be also considered as special cases of DTDOs, for instance, with an almost zero delay-tolerance.

DTDOs may be referred to independently of the node which has originated them, meaning that DTDOs may be stored in nodes different from the one which has originated them. For instance, they may be accessed even when the node that has originated them is not a part of the MANET anymore.

DTDOs may be originated with an assigned *Time to Life* (TTL), which provides a mechanism for limiting the time the object is kept in the MANET. The TTL refers to the maximum time elapsed from the creation of an object until its destruction, whether it resides in its originator node or in any other node of the MANET. When an application running on top of DELTOYA has copied the DTDO data, the expiration of TTL does not mean that the application will destroy this data, but rather that DELTOYA will not maintain a copy of the retrieved DTDO for being shared with other nodes. If a DTDO does not have assigned a TTL, we can think that the assigned TTL is 0, meaning in fact that it is not a delay-tolerant object. On the other hand, if the TTL is unlimited, we can think in setting TTL to infinity.

DELTOYA provides means for creating, modifying, deleting and publishing DTDOs. All these operations are performed using the DTDO Table. To publish a DTDO means that the DTDO will be made accessible to the other nodes of the network in case they request for it. See appendixes “C.1 DTDO Creation”, “C.2 DTDO Modification”, “C.3 DTDO Deletion”, “C.4 DTDO Publishing” and “C.5 DTDO Unpublishing” for more details on the behavior of these operations.

DELTOYA also provides means for querying, requesting and sending DTDOs through the network.

6.1 DTDO Querying

DELTOYA provides means for querying DTDOs that are stored in other nodes of the network.

A key characteristic of DELTOYA is that nodes may store DTDOs that have not been locally generated, making these objects accessible to other nodes of the MANET. A node that requested a DTDO may store it during a time determined using the TTL field. The node acts as a proxy of the node that originated the DTDO.

The DTDO Querying mechanism must determine the set of nodes where the DTDOs are available. See appendix "C.6 DTDO Querying" for more details on the behavior of querying DTDOs.

6.2 DTDO Requesting (Pulling DTDOs)

DELTOYA provides means for requesting or pulling DTDOs that are stored in other nodes of the network.

The DTDO Requesting mechanism determines the set of nodes where the DTDOs are available and at the same time obtains these DTDOs. Once the decisions are taken and a request is made, the DTDOs will be obtained by means of a transport protocol, which is PTP (see section 7) in the case of the MStack.

Figure 26 shows an example of a diagram where three nodes (nodes A, B and C) are connected in a MANET. In the three nodes there are some applications that exchange data by using the services of DELTOYA.

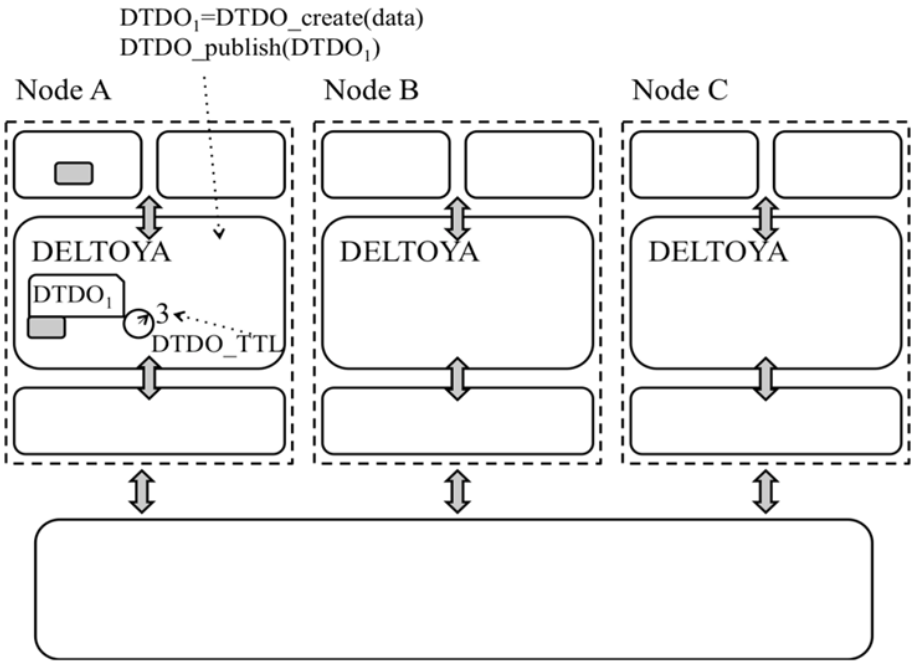

Figure 26. Creating and Publishing a DTDO

In node A, one of the applications has created and published a DTDO $DTDO_1$, associating some data to this DTDO. The $DTDO_1$ has a DTDO_TTL of 3 seconds in this example. In Figure 26 we show the primitives `createDTDO` and `publishDTDO` invoked by the application in order to create and publish the DTDO.

In Figure 27, node B's application requests $DTDO_1$. In the example, this is invoked by the application by means of the `requestDTDOs` primitive. This triggers the transmission of a message requesting for the DTDO (DTDO Request message). Node A will answer that it has the requested DTDO available by means of the DTDO Response message.

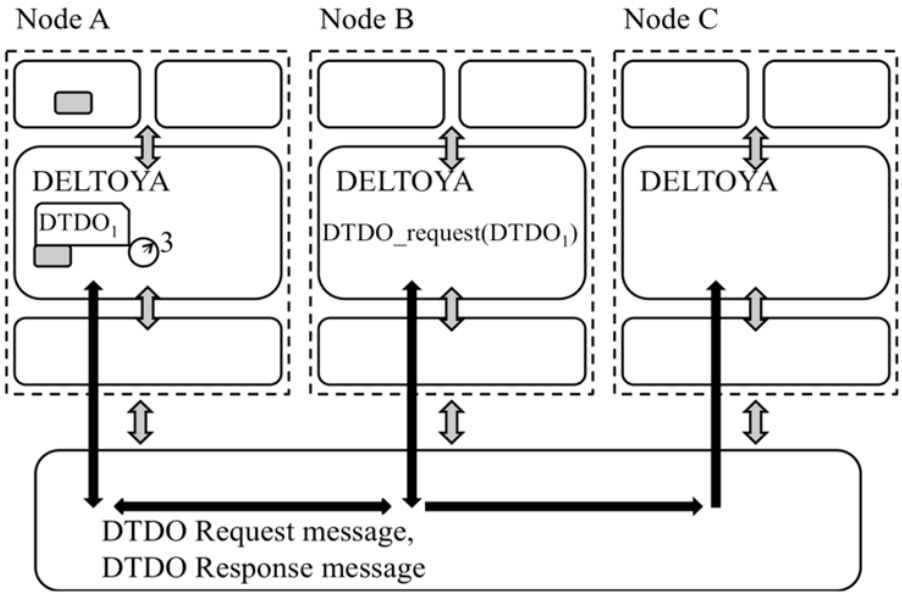

Figure 27. Requesting the DTDO

Figure 28 shows node B finally getting the DTDO. The DTDO_data (i.e. the field of the DTDO containing the actual data) is copied to the application which has requested the DTDO. Moreover, the DTDO resides for some time in the DELTOYA memory space. The time the DTDO will reside there will be determined by the DTDO_TTL field. The DTDO_TTL of the DTDO stored in node B is shown in Figure 29.

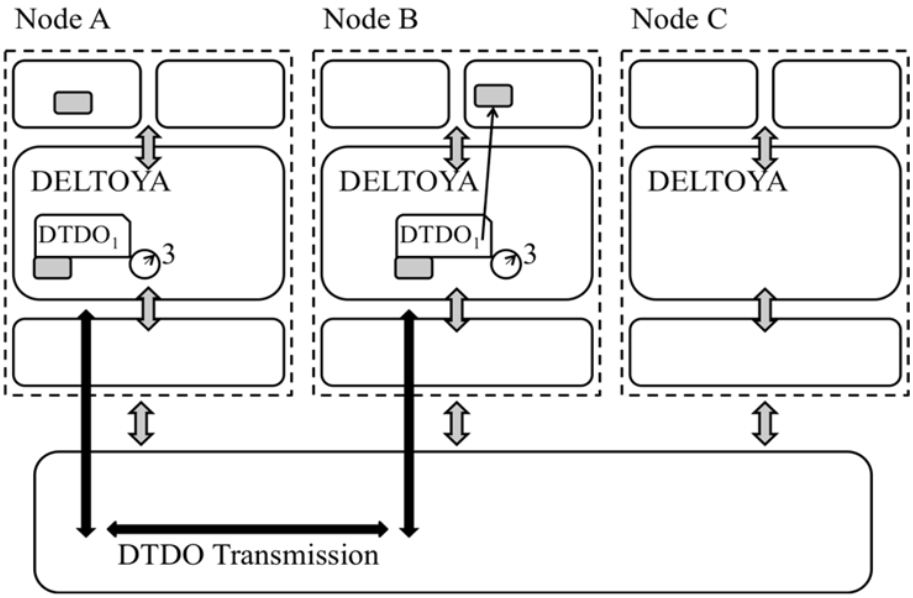

Figure 28. Pulling the DTDO

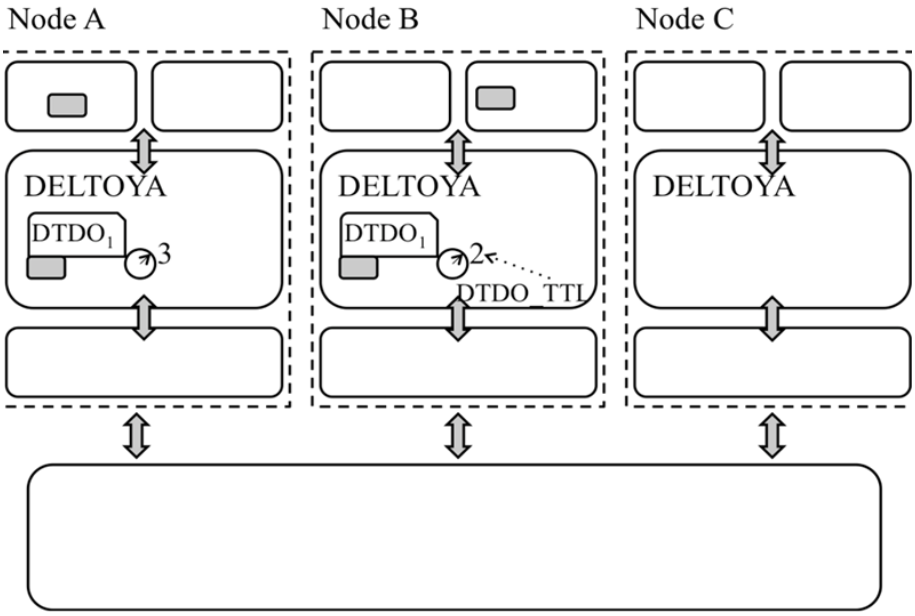

Figure 29. Timing the DTDO

In Figure 30 we see that an application residing in node C requests the same $DTDO_1$, again using a DTDO Request message. At this time, node A, which created the DTDO, has left the network. The request made for node C is now only answered by node B, with a DTDO Response message.

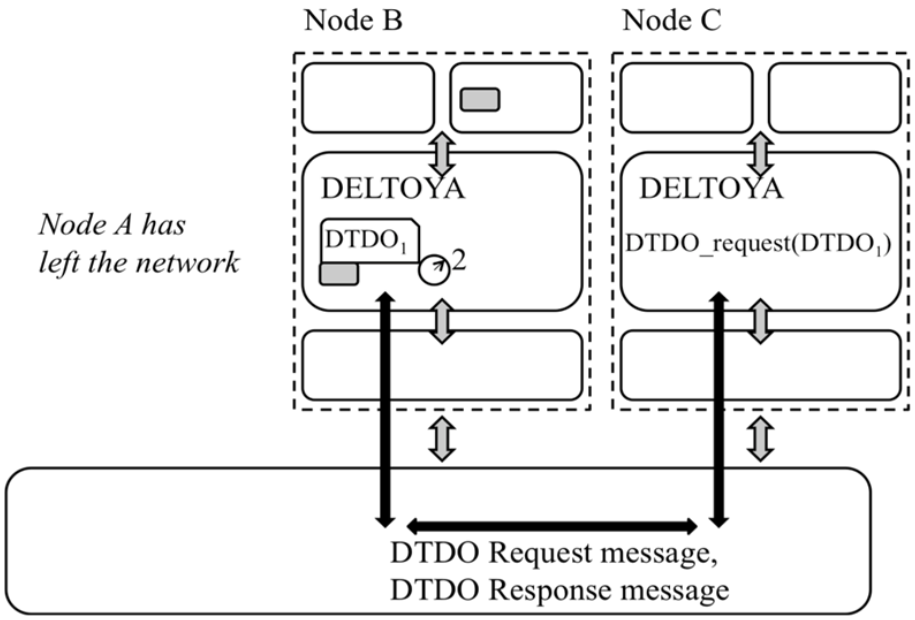

Figure 30. Requesting again the DTDO

In case node A was in the network, both nodes will answer the request, and it would be a decision of node C to choose from which node to retrieve the DTDO.

In Figure 31, node C finally gets the DTDO from B by means of the `requestDTDOs` primitive. The DTDO_data is delivered to the application in node C. The DTDO is again stored in the memory space of DELTOYA in node C.

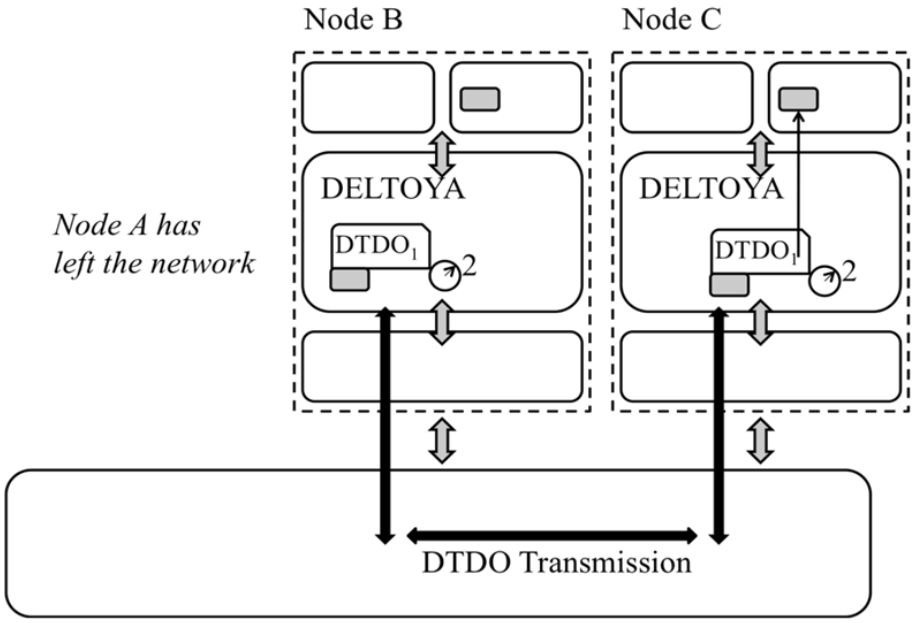

Figure 31. Pulling again the DTDO

In Figure 32 we see that the DTDO_TTL expires in both nodes B and C. These nodes will remove then the DTDO copies of their DELTOYA memory space. Note however that the data stored by the applications is not affected by the TTL expiration.

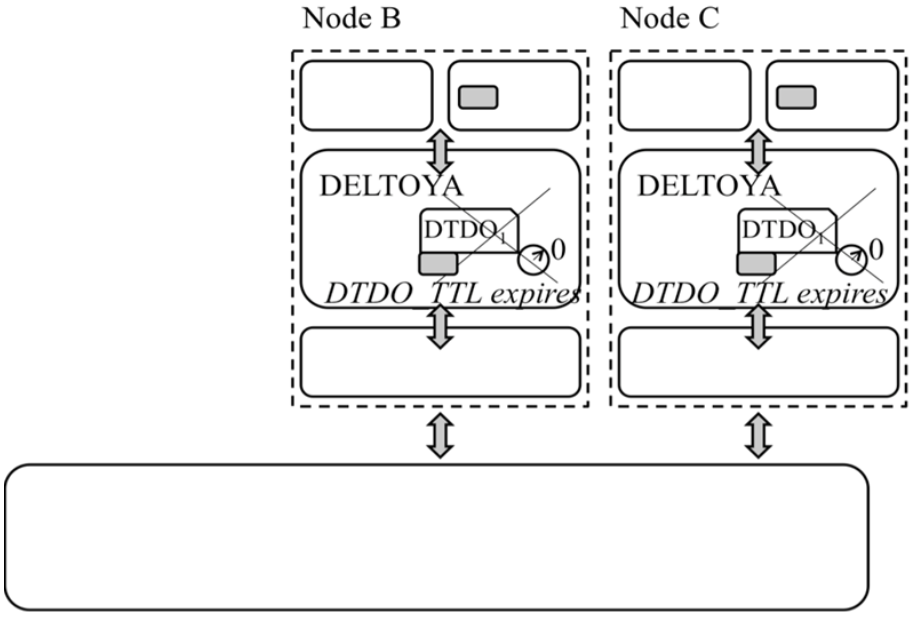

Figure 32. Expiring the DTDO

See appendix "C.7 DTDO Requesting" for more details on the behavior of requesting DTDOs.

6.3 DTDO Sending (Pushing DTDOs)

DELTOYA provides means for sending or pushing a DTDO to other nodes of the network, even when the destination node has not requested the DTDO.

In order for the destination node to receive the DTDOs, we define a new structure called DOCK that defines a virtual deliverable place for DTDOs. A DOCK is a structure where DTDOs sent by other nodes are placed.

DOCKs can receive unchained DTDOs (i.e., DTDOs not belonging to a CHAIN structure) or chained DTDOs (i.e., DTDOs belonging to a CHAIN structure). DTDO CHAINs are a data structure consisting of an ordered sequence of DTDOs. DTDOs include information to identify the CHAIN to which they belong to and their sequence order inside this CHAIN.

DOCKs receiving chained DTDOs deliver these DTDOs in order. This means that DTDO with sequence number N+1 will be delivered after the DTDO with sequence number N has been correctly received by the DOCK and delivered to the entity reading the DTDOs from the DOCK.

DTDO CHAINs have a resistance property. The resistance of a DTDO CHAIN defines the maximum time the DOCK waits until it makes available to the destination node an out-of-order DTDO belonging to this CHAIN. For instance, assume that DTDO with sequence number N+1 is correctly inserted in a given DOCK at time instant T. If at time T, the DTDO with sequence number N is not available for delivering by the DOCK, the DTDO with sequence number N+1 will not be available for delivering by the DOCK. The DTDO with sequence N+1 will be available for delivering whenever one of these two events happens:

- DTDO N is available for delivering by the DOCK
- The time instant T+R is reached, being R the CHAIN resistance of the CHAIN where the DTDO N+1 belongs to.

In Figure 33, we can see an example where node A wants to send the DTDO1 to the DOCK1 of the node B. Node A calls the DTDO_send primitive specifying the DTDO it wants to send and the DOCK where it wants to put the DTDO. Note that in this example, node A is already aware of the existence of the DOCK1 in node B. DOCK1 could be a well-known dock for every node, or if not, node A would have executed previously a DOCK discovery mechanism querying for the possible docks of node B in order to discover the DOCK1 of node B.

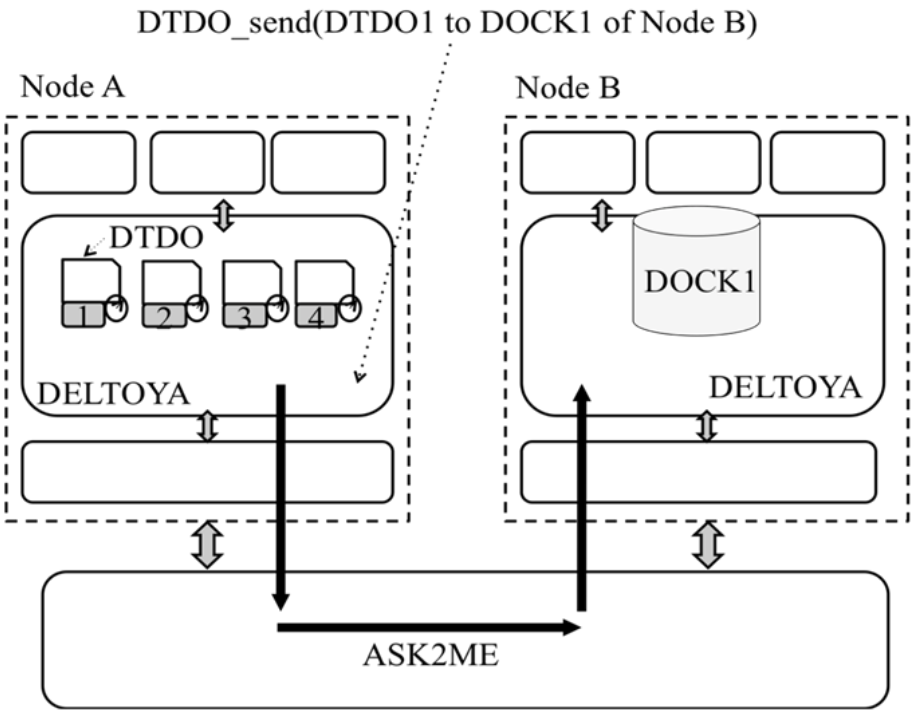

Figure 33. Pushing a DTDO

As a result of the primitive, an ASK2ME message is sent from node A to node B. The ASK2ME message contains all the information needed to allow the node B to get the DTDO1 from node A. If node B accepts receiving the DTDO1 from node A, node B will start a `requestDTDOs` primitive as if node B was the node that was started itself a `requestDTDOs` for this DTDO1. In other words, pushing a DTDO from node A to node B is almost equivalent to pulling this DTDO from node B to node A. See Figure 27 and Figure 28 where there is an example of a `requestDTDOs` done by node B to node A.

In Figure 34, node B has finally received the DTDO1 and, since the `requestDTDOs` of node B was due to a DTDO_send from node A, node B has to put the DTDO1 in the corresponding DOCK1. From now on, the DTDO1 inside the DOCK1 is already waiting for being picked up from the DOCK.

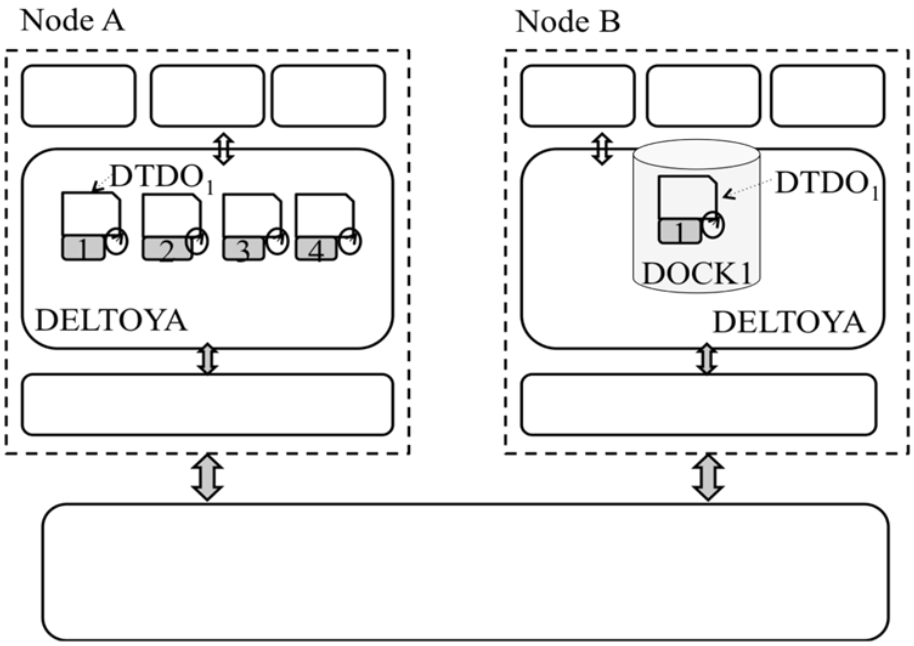

Figure 34. DTDO Waiting in the DOCK

See appendix “C.8 DTDO Sending” for more details on the behavior of sending DTDOs.

6.4 DOCKs and CHAINs

DELTOYA provides means for creating and destroying DOCKs and CHAINs. All these operations are performed using the DOCK and CHAIN Tables. See appendixes “C.9.1 DOCK Creation”, “C.9.2 DOCK Destruction”, “C.9.4 CHAIN Creation” and “C.9.5 CHAIN Destruction” for more details on the behavior of these operations.

DELTOYA also provides means for querying DOCKs that are in other nodes of the network. The DOCK Querying mechanism must determine the set of DOCKs available in the nodes of the network. See appendix “C.9.3 DOCK Querying” for more details on querying DOCKs through the network.

7. PTP

The Patient Transport Protocol (PTP) is a novel mechanism for ensuring a reliable transport of information through the network. It is a transport protocol that adds to the typical functions of transport protocols (order, error and congestion control) the property of persistence, meaning that PTP is patient when it wants to send or receive information through the network, dealing in this way with the problems arisen from the mobile wireless environments.

7.1 PTP Segmentation

Any DTDO that PTP wants to transmit will be segmented into a number of PTP Protocol Data Units (PTP PDUs).

PTP PDUs are the basic transfer units used by PTP, and from the implementation point of view, one PTP PDU cannot be, in size, greater than MTU - size(IP header) - size(UDP header) - size(MIFO header), meaning that never one PTP PDU will need IP fragmentation for sending it through the network.

We will set the maximum size of a PTP PDU to:

- MAX(size(PTP PDU)) = MTU - size(IP header) - size(UDP header) – MAX(size(MIFO header)) Bytes

PTP PDUs include both a PTP header and the payload of the packet. See Figure 35.

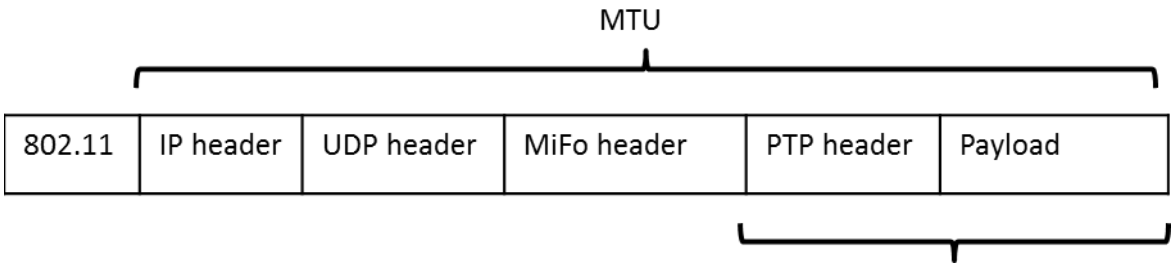

Figure 35. PTP PDU

So, the maximum size of the Payload of a PTP PDU will be:

- MAX(size(PTP PDU Payload)) = MAX(size(PTP PDU)) – size(PTP header) Bytes

See appendix “D.1 PTP Header” for more details on the fields of the PTP Header and Payload.

7.2 PTP Messages

There are three types of PTP messages: PTP Data messages, PTP Request messages and PTP Null messages.

PTP Data messages are the PTP PDUs with a PTP header and Payload identifying which block of data of which DTDO they are carrying in the Portion field (the actual data) of the Payload. The maximum size of the Portion field of a PTP Data message will be M = MAX(size(PTP Data Portion)) = MAX(size(PTP PDU Payload)) – 7 Bytes (see appendix “D.1 PTP Header”).

PTP Request messages are the PTP PDUs with a PTP header and Payload identifying which portions of which DTDO they are requesting to a remote node.

PTP Null messages are the PTP PDUs with a PTP header identifying the DTDO that a remote node has requested and that for some reasons it has become unavailable and it has not been served. The Payload field is used to specify the reason.

7.3 PTP Operation

Unlike TCP that uses a push transfer mode, PTP uses a pull transfer mode. In the pull mode, destination nodes (from the point of view of the data flow) send back to source nodes PTP Request messages (PTPREQ) that request for the transmission of a given number of PTP PDUs. In the pull mode, acknowledgment packets are not needed to guarantee error control. See Figure 36.

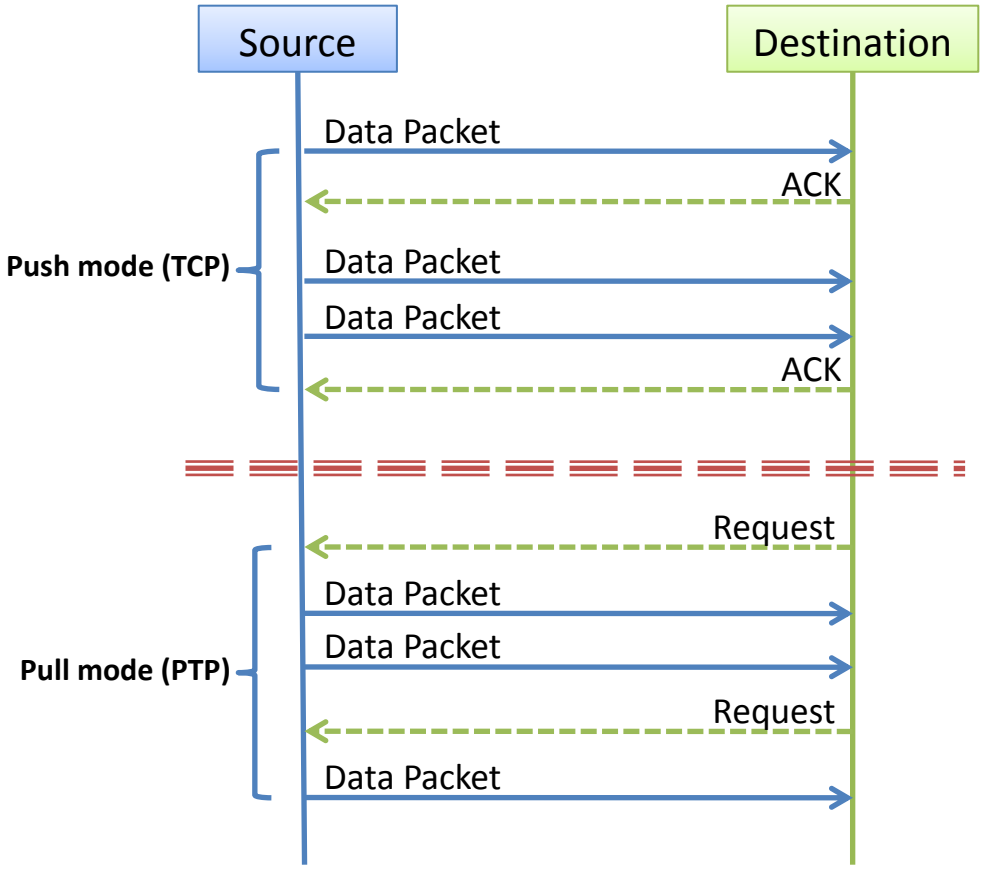

Figure 36. PTP Push and Pull Modes

In the case of PTP working together with DELTOYA, when DELTOYA sends to PTP the information of the nodes that have a certain DTDO and the request to retrieve this DTDO, then PTP starts the retrieval of the DTDO. PTP may choose to perform the whole transfer from a single node, or retrieve different or even the same PTP PDUs of the DTDO from different nodes. As said before, DELTOYA provides the information of the different nodes from which PTP may be able to retrieve the DTDO. Moreover, DELTOYA may keep updating this information to PTP or even PTP may request updates from DELTOYA of this information along time.

If PTP chooses a given node, but the transfer of information is interrupted or has low performance metrics (e.g., packet loss ratio, throughput, etc.), PTP may choose to continue the DTDO transfer from other nodes. From doing so, PTP may ask DELTOYA for issuing a new DTDO

Discovery mechanism across the MANET if it considers, for example, that the information it has about the previous DTDO Discovery mechanism is already obsolete. These actions are performed until the DTDO is completely obtained or a Maximum Retrieval Time (MRT) timeout expires.

The idea is to keep getting the PTP PDUs of the DTDO without bearing in mind if some are lost. The process of getting the different PTP PDUs of one DTDO can be distributed over several nodes in order to exploit data diversity at maximum. The idea is to not worry about which PTP PDUs are lost during the process of getting the DTDO, as PTP does not want to block the flow; lost PTP PDUs will be asked to the same node or other nodes at this time or another time, when PTP determines to be convenient.

In the case of too many loses, PTP can also delay the retrieving process of the DTDO by waiting a configurable period of time for the appearance of new nodes in the network which could contain this DTDO. In this way, PTP does not block when it cannot get the DTDO. Instead, it tries to get the DTDO from other nodes and only when it believes that too much time (e.g., MRT) has been used and no result has been obtained, then PTP may decide to notify the upper layer protocols of the stack (e.g, DELTOYA) with the impossibility of getting this DTDO.

The data rate in PTP is controlled in order to avoid congesting the network. Problems that arise from a congested network may cause poor performance in the data flows of the network and in the functioning and operation of the network. Congestion control in PTP is done in a handshake way by both sides of the data flow. While the originator of the data packets starts at a maximum data rate and keeps decrementing the rate up to a minimum while the data packets are sent, the receiver keeps telling the originator to increment the rate in case it receives the packets without loses. This behavior will let both sides to stabilize in the correct data rate for the MANET in that time.

See appendixes “D.2 PTP Request Queue (PTPRQ)”, “D.3 PTP Serving Queue (PTPSQ)”, “D.4 Pull Mode” and “D.5 PTP Calculations” for more details on the operation of PTP.

8. MIFO

The MIraveo FOrwarding (MIFO) is the forwarding mechanism of the MStack, where the forwarding or relaying of the packets between nodes takes place.

In a MANET, where multihop routing is present, MIFO is needed in order to be able to send a unicast packet between a source node and a destination node, using zero, one or more intermediate nodes as forwarders of the packet if it is the case. In other words, for a unicast packet travelling along the network there is only one source node and one destination node, but there could be several transmitter and receiver nodes (source and destination nodes included).

When multihop routing is used, addressing of packets has to be defined in order to let every packet reach a destination node even when several intermediate nodes need to be used. AGENET (see section 12) is the protocol in charge of providing all the routing information needed to route and forward each packet to a destination node along the network. As explained in section 12, this information will be used to discover to which next node transmit a packet in order to finally be able to reach the destination in a multihop routing path. In other words, to discover the sequence of next hops that have to transmit the packet until it reaches the destination node. For clarification, if we have a routing path composed by the sequence of nodes A_0, A_1, A_2,…, A_{N-1}, A_N, where A_0 is the source node and A_N is the destination node, we will call A_{i-1} the "previous hop" of the node A_i ($1 <= i <= N$), and we will call A_{i+1} the "next hop" of the node A_i ($0 <= i <= N-1$). From the point of view of a node, its previous and next hops are always nodes reachable in one transmission (i.e., nodes that have direct link).

Once the routing information is provided by AGENET, MIFO will use it in order to forward the packet. If we remember the sub table of the DTDO Table (see section 5.1) related with AGENET, we can see that for every destination node in the table, we have one or more routing paths to reach this destination node. This information is the one that will be maintained by AGENET and the one that is going to be used by MIFO in order to forward the packets. Note that although for one destination node and one routing path we have the whole sequence of nodes in order to reach the destination, only the first node (i.e., the first next hop) will be used by MIFO in order to forward the packet. The rest of next hops of the path are only used by AGENET for calculating which the best routing paths are.

8.1 MIFO Header

MIFO headers between the network layer headers and the payload of the packet are needed. MIFO will be the bridge between the MStack (based on unique node IDs) and the UDP/IP protocol (based on port number and IP addresses as node identifiers). This bridge will bring to the MStack mobility support at network layer. See appendix "E.1 MIFO Header" for more details on the MIFO Header.

8.2 MIFO Table

As explained in section 3, the MStack is completely independent of the IEEE 802.11, IP and UDP protocols. It works over UDP/IP, but it could work over any other networking protocol. MIFO is the bridge between the MStack and the networking protocol, the only module of the MStack that is dependent of the network protocol. All other modules of the MStack are

network independent, meaning that they are independent of the network protocol used in the MANET.

In order to map the MStack with the networking protocols, we define the MIFO Table; see appendix "E.2 MIFO Table" for more details on this table.

The goal of this table is to provide to MIFO the information necessary to be able to forward any packet received from the network or generated by the node itself. Once MIFO knows the unique ID of the node to which transmit the packet, MIFO will use the MIFO Table to know the IP address of this node, in order to finally be able to send the packet to the receiver node by using the IP protocol. Concerning the UDP port, the corresponding well-known port `DTDODM_PORT`, `PTP_PORT`, `AGENET_PORT`, `MIMI_PORT`, `RDM_PORT`, `DOCKDM_PORT`, or `ASK2ME_PORT` will be used (see appendix "Appendix K. MStack Parameters").

The Source Learning technique will be used to keep the MIFO table updated. Every packet received by MIFO will be used to update the MIFO table. See appendix "E.3 Source Learning" for more details on the Source Learning algorithm of MIFO.

8.3 Unicast/Broadcast Forwarding

MIFO will forward packets as follows:

1. The IP route table of the OS will be set according the IP address of the node as if the node was in a WLAN without a default gateway, i.e., all IP addresses of the network of the node will have direct link with the node.

For example, if the node IP address is 10.0.0.2/8, the IP route table will look like Figure 37.

Destination	Gateway	Genmask	Flags	Res	Use	NetIf	Expire
10.0.0.0	0.0.0.0	255.0.0.0				ath0	

Figure 37. IP Route Table

2. AGENET will maintain the multihop routing information in the DTDO Table.

3. The MIFO header will be used in every packet (see Figure 38).

IP header	UDP header	MIFO header	Payload

Figure 38. Headers of a Packet

4. The IP header of the packet will be used to route the packet to each next hop in the routing path. The DestinationID node and the SourceRoutingList of the MIFO header will be used to route the packet from the source to the destination node.

In other words, the IP destination and IP source fields of the IP header will be used to set up the receiver and the transmitter nodes of the packet respectively. And the DestinationID field and the first node ID of the SourceRoutingList field of the MIFO header will be used to set up the destination and the source nodes of the packet respectively.

In Figure 39 we can see an example where the source node A wants to send a unicast packet to the destination node C, which is two hops away from A. We can also see the packet

transmitted by A to the intermediate node B, and the packet transmitted by B to the final destination C.

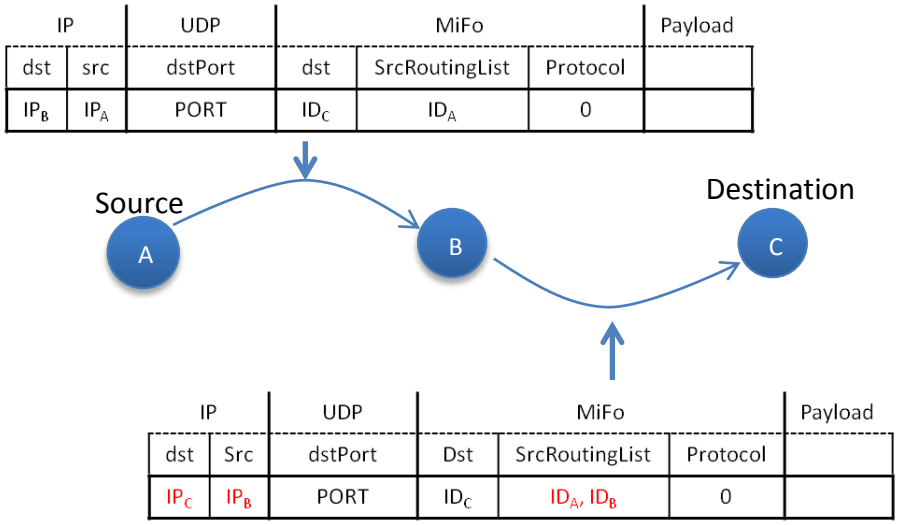

Figure 39. IP and MIFO Headers

5.1. With all these assumptions, when a source node S has to send a packet to a destination node D, it will add the MIFO header to the payload of the packet it wants to send, by setting:

- The DestinationID to the unique ID of node D, or to 0 in the case of destination Broadcast
- The Size of the SourceRoutingList to 1
- The SourceRoutingList to {the unique ID of the node S}
- **If** ((Protocol == (PTP or ASK2ME)) TTL=MAXALLOWED_HOPS **Else** TTL=1
- The payloadSize to the size of the packet
- The Protocol to {PTP | ASK2ME | DTDODM | RDM | DOCKDM | AGENET | MIMI}

Then, it will create a UDP/IP packet with the previous payload and with:

- UDP src_port = {PTP_PORT | ASK2ME_PORT | DTDODM_PORT | RDM_PORT | DOCKDM_PORT | AGENET_PORT | MIMI_PORT}
- UDP dst_port = {PTP_PORT | ASK2ME_PORT | DTDODM_PORT | RDM_PORT | DOCKDM_PORT | AGENET_PORT | MIMI_PORT}
- IP src = the IP address of the node S
- IP dst = see section 8.4

And the packet will be transmitted (to an intermediate node or to the final destination).

5.2. At the same time, when an intermediate node receives a packet, MIFO will check if the DestinationID field of the MIFO header is equal to ID of the node itself or equal to 0 (Broadcast).

- **If** equal, the MIFO header will be removed and the rest of the payload will be sent to the upper layer protocol (identified by the Protocol field of the MIFO header), together

with the first node ID of the SourceRoutingList of the MIFO header (this field will be needed for the upper layer protocols).

- **Else**, MIFO will check the TTL field of the MIFO header, and if it is greater than 1, the node will forward the packet by modifying the MIFO header in this way:
    - The unique ID of the intermediate node will be added at the end of the SourceRoutingList field
    - The Size of the SourceRoutingList will be incremented by 1
    - TTL will be decremented by 1

    And then, it will create a UDP/IP packet with the previous updated payload and with:

    - UDP src_port = {PTP_PORT | ASK2ME_PORT | DTDODM_PORT | RDM_PORT | DOCKDM_PORT | AGENET_PORT | MIMI_PORT}
    - UDP dst_port = {PTP_PORT | ASK2ME_PORT | DTDODM_PORT | RDM_PORT | DOCKDM_PORT | AGENET_PORT | MIMI_PORT}
    - IP src = the IP address of the intermediate node
    - IP dst = see section 8.4

    And the packet will be transmitted (to the next intermediate node or to the final destination).

8.4 Look Up

Every time a node needs to send a packet, it has to look up in MIFO in order to discover which destination IP address it has to use in order to send the packet towards the destination.

8.4.1 Broadcast Packets

For Broadcast packets, the destination IP address will be set to the broadcast IP address of the corresponding network.

8.4.2 Unicast Packets

On the one hand, for unicast packets of the PTP and ASK2ME protocols, every time a node needs to send a packet to a destination node, it has to look up in the DTDO Table in order to discover which entry it has to use in order to send the packet towards the destination. Since several entries may be simultaneously present in the DTDO Table for a single destination, the node has to choose which is the best one to be used at that moment. This entry will be used to determine the nexthop node (the first in the Path field) to which transmit the packet in order to follow the forwarding path towards the final destination node.

In the case that no entry is found, the packet will be discarded (and upper layers of the MStack (e.g., PTP) will be the responsible of managing these loses).

For determining the nexthop node, a Round Robin mechanism will be used (in this way, MIFO will be CPU aware, meaning that it takes into account the CPU usage of every node involved in the process of forwarding, and tries to guarantee a fair use of this resource along these nodes). The entries for a given destination will be chosen following a cyclic order, meaning that each time the node needs to send a packet to this destination, the node uses the entry subsequent

to the entry used with the previous packet sent to the same destination, building a circular list of the possible routes to the destination.

Moreover, in order to avoid loops, once the entry to be used has been chosen, and before using it, the node will check if one of the nodes of the SourceRoutingList of the MIFO header of the packet appears in the list of nodes of the Path field of the entry recently chosen. In this case, this entry will not be used and the next entry, depending on the Round Robin mechanism, will be checked in order to avoid loops with the packet. This next entry of the list will be then checked and the same process will be repeated, until an entry with no loop is found. In the case no entry with no loop is found for this destination, the packet will be discarded.

In summary, if the last entry used was `currentEntry`, the Look up will proceed in this way:

```
noLoopFound = false

next_entry()

For each entry to the same destination starting at current_entry() and
ending at current_entry().previous_entry()

      If (loop)

            next_entry()

      Else

            noLoopFound = true

            If (current_entry().#hops() > the remaining TTL (assuming
            that it has already been decremented) of the packet to
            send)

                  next_entry()

            Else

                  checkTTLok = true

                  currentEntry = current_entry()

                  Break

            EndIf

      EndIf

EndFor


If (!noLoopFound || !checkTTLok)

      Discard packet

Else

      We are able to do unicast forwarding to the first nexthop of the
      Path field of the entry currentEntry:
```

```
        We will look up in the MIFO Table for the entry
        corresponding to nodeID==the first next hop of the Path
        field of the entry currentEntry

        If (exists)

              The destination IP address of the IP header of the
              packet will be set to the nodeIP address of the
              corresponding entry

        Else

              We discard the packet

        EndIf

EndIf
```

On the other hand, for unicast packets of the DTDODM, DOCKDM and RDM protocols, MIFO will look up in the MIFO Table for the entry corresponding to nodeID equal to DestinationID

```
If (exists)

      The destination IP address of the IP header of the packet will
      be set to the nodeIP address of the corresponding entry

Else

      We discard the packet

EndIf
```

9. DTDODM

The DTDO Discovery Mechanism (DTDODM) is the mechanism in charge of discovering a set of specified DTDOs in the network.

The DTDOM starts a flooding mechanism using DTDO Discovery Request (DTDODREQ) messages and DTDO Discovery Reply (DTDODREP) messages in order to discover new recipients for specific DTDOs.

For each DTDODM started, DELTOYA will keep sending DTDODREQ messages following a Binary Exponential Backoff, and it will only stop the DTDODM when all the DTDOs have been obtained or when the remaining time of the DTDODM expires.

However, this is from the point of view of the DTDODM. From the point of view of the network, an Aggregator will be used in order to try to aggregate all the DTDODREQ and DTDODREP messages of the DTDODMs as much as possible, in order to reduce the number of transmissions in the network; see appendix “F.1 Flooding (The Aggregator)” for more details on the Aggregator.

On the one hand, the DTDODREQs for Queries of DTDOs are broadcast messages that will be flooded in the MANET in order to discover which DTDOs and in which nodes they are available at that time in the network. Unlike DTDODREQs for Queries of DTDOs, DTDODREQs for Requests of DTDOs will be also used in order to add routing information in the DTDO Table. On the other hand, DTDODREPs are unicast messages that will be answered back in reply to the corresponding DTDODREQs. See section “Appendix F. DTDODM Details” for all the details on the operation of the DTDO Discovery Mechanism.

10. DOCKDM

The DOCK Discovery Mechanism (DTDODM) is the mechanism in charge of discovering a set of specified DOCKs in the network.

The DOCKDM starts a flooding using DOCK Discovery Request (DOCKDREQ) messages and DOCK Discovery Reply (DOCKDREP) messages in order to discover DOCKs in the network for a period of time.

For each DOCKDM started, DELTOYA will keep sending DOCKDREQ messages following a Binary Exponential Backoff, and only will stop the DOCKDM when the period of time of the DOCKDM expires.

However, this is from the point of view of the DOCKDM. From the point of view of the network, an Aggregator will be used in order to try to aggregate all the DOCKDREQ and DOCKDREP of the DOCKDMs as much as possible, in order to reduce the number of transmissions in the network; see appendix “G.1 Flooding (The Aggregator)” for more details on the Aggregator.

On the one hand, DOCKDREQs are broadcast messages that will be flooded in the MANET in order to discover which DOCKs and in which nodes are available at that time in the network. On the other hand, DOCKDREPs are unicast messages that will be answered back in reply to the corresponding DOCKDREQs. See section “Appendix G. DOCKDM Details” for all the details on the operation of the DOCK Discovery Mechanism.

11. RDM

The Route Discovery Mechanism (RDM) is the mechanism in charge of discovering a specific node (identified by its unique ID) in the network.

Each time a request for discovering a new node is called, a flooding mechanism using Route Discovery Request (RDREQ) messages and Route Discovery Reply (RDREP) messages is started.

The flooding mechanism will follow a cyclic Binary Exponential Backoff until a maximum number of iterations. Once having performed the maximum number of iterations, then the Binary Exponential Backoff will automatically be restarted as if a new Binary Exponential Backoff was started. For each iteration, the flooding mechanism will check if there is a routing path to the node to discover in the DTDO Table. If there is a routing path, the flooding mechanism will do nothing. If there is not a routing path, then the flooding mechanism will send a RDREQ message and will wait for a RDREP message from the node to discover.

On the one hand, RDREQ messages are broadcast messages that will be flooded in the MANET in order to discover routing path between source and destination nodes. On the other hand, RDREP messages are unicast messages that will be answered back in reply to the corresponding RDREQs. See section “Appendix H. RDM Details” for all the details on the operation of the Route Discovery Mechanism.

12. AGENET

AGgrEssive NETwork Protocol (AGENET) is a novel networking protocol that comprises the routing capacities needed to establish communication between different nodes of a MANET.

AGENET exploits cooperation and diversity, two aspects inherent to MANETs, mainly due to the broadcast properties of the wireless medium. Since MANETs are very fragile networks (e.g. due to node mobility and to wireless transmission errors), it becomes apparent that cooperation and diversity are two properties that must be exploited in order to build resilient MANETs.

AGENET is reactive to node mobility. Node mobility may cause frequent changes in the topology of the network, and so, frequent disconnections to communications between applications of different nodes of the MANET.

AGENET works in a pure datagram way, as it does not establish per-flow routing paths between nodes involved in the communication process (e.g. source and destination node of a unicast communication process), then avoiding using obsolete routing paths due to node mobility. On the contrary, it uses recent routing information to decide where to send the packets at every packet transmission time.

Routing Zones (RZs) are created in order to not send the routing information all over the network, and thus avoiding scalability and overhead problems.

Moreover, AGENET is battery aware, meaning that it takes into account the battery level of every node involved in the process of routing, and it tries to guarantee a fair use of this resource along these nodes.

12.1 Aggressive Routing

AGENET does not establish per-flow routing paths between nodes involved in the communication process. Instead, AGENET allows every node of the network to decide, at every packet transmission time, to which next node send the packet in order to finally allow the packet reach its destination. This decision is taken depending on recent topology and state information of the network. This lets the routing to be more reactive in front of node mobility, the main problem of MANETs.

AGENET sends routing information between all nodes of the network, in order to let every node know the information needed to decide to which next node send the packet in order to reach the destinations. This information includes several wireless aware routing metrics. These metrics are the number of hops between source and destination in a multihop MANET, the level of battery of each routing node involved in the routing path, the freshness of the information and the quality of the wireless links of the routing path (i.e., unidirectional or bidirectional links).

AGENET uses flooding to send this information to all nodes of the network. Every node of the network is in charge of sending its own information and the information it has learnt from other nodes of the network. Nodes send this information periodically.

AGENET uses mechanisms to detect which of the routing information that travels through the network is more recent and to avoid routing loops. Packet identifiers (i.e., timestamps) managed by the originator of the information are used to detect the most recent information. Source routing information is also used for the same purpose.

AGENET uses a heuristic to decide how to build the routing table from all the routing information it receives. The heuristic is based on the wireless aware routing information sent by the routing nodes. The heuristic allows nodes to build the routing table with the best routing information at every time instant.

In case of having several possible next-hops to the same destination with the same or similar costs, AGENET uses a multipath mechanism to decide among these possible options, the next node to which the packet must be sent in order to reach the destination. In this case, a Round Robin mechanism is used, meaning that nodes establish a periodic pattern so that the same route does not need to be chosen always even if the costs have not changed. The Round Robin mechanism allows nodes to distribute the data packets for different routing paths, achieving a fair CPU and battery usage of the nodes involved in the routing paths.

AGENET sends routing information messages with high frequency and priority, mainly to supply every node of the network with very recent information of the topology and state of the network. In this way, problems that arise due to node mobility will be almost negligible, since they will be repaired in a short period of time. However, AGENET will have to address potential scalability and overhead problems.

In order to deal with the scalability issues, AGENET tries to minimize the overhead of the routing information messages. The mechanism tries to send routing information messages only when they are needed and only where they are needed. While no data flows appear in the network, the mechanism tries to not send routing information. AGENET addresses scalability by using the concept of Routing Zone (RZ).

The Routing Zone is the set of nodes of a MANET that are involved in the process of routing (i.e., sending routing information, forwarding data packets, etc.) due to the appearance of a data flow. Outside the RZ (i.e., in the areas of the network in which there is no data traffic) no routing information is sent. AGENET includes mechanisms to start, maintain and stop the RZ needed to support each flow of data that can appear in the network.

Moreover, AGENET reduces the overhead of sending the routing information by implementing a scheduled flooding mechanism where nodes distribute their own routing information and the information learnt from others through the network in a periodic way. Routing information has a Time To Live (TTL) period assigned, in order to avoid sending it all the time and to avoid sending it out of the limits of the MANET.

12.2 Routing Information: AGgressive ROuting Hellos (AGROHellos)

AGROHello messages are used to distribute routing information through the network. This distribution is performed by means of flooding, and it will be used to update the DTDO Table.

Nodes of a RZ use periodic flooding of information, meaning that every certain period of time each node in the RZ will send an AGROHello message with its own information and with the

information it has learnt from other nodes of the network. Since AGROHello messages could become large messages, the period of time has to be enough small to avoid work with obsolete information but enough large to avoid a lot of overhead and scalability problems. The information carried out by the AGROHello messages will be used to update the DTDO Table. See appendixes “I.1 Routing Information: AGgressive ROuting Hellos (AGROHellos)” and “I.3 AGROHello Format” for more details on the AGROHEllo messages.

12.3 Scalability: Routing Zone (RZ)

The Routing Zone (RZ) is a set of nodes of a MANET that are involved in the process of routing (for instance, sending routing information, forwarding data packets, etc.). Outside the RZ, the nodes do not neither send any information related to the routing mechanism nor any data packet. In this way, scalability is achieved by limiting the geographical areas in which routing information is sent and routing information will be sent only when it is needed and where it is needed. See appendix “I.2 Scalability: Routing Zone (RZ)” for more details on how starting, maintaining and ending the Routing Zone.

12.4 Routing Heuristic (RH)

The Routing Heuristic (RH) will be used to decide which of all the entries that may be learnt via AGROHello messages will be stored in the routing table.

Only a maximum number of entries can be stored in the routing table for each destination. The entries with the best RH will be the entries stored in the routing table.

For every entry in the sub table of the DTDO Table corresponding to AGENET, the node has the following routing information for a destination node:

- The destination node
- The path to follow in order to reach this destination
- The level of battery of each nexthop in the path
- Information about unidirectional links in the routing path
- The timestamp of this information (i.e., the time instant when this information was generated by the destination node)

The path to follow to the destination node includes the sequence of nexthops, which implicitly tells us the number of hops of the path.

The RH, for a specific routing path, will be calculated in a way that the lower the RH, the better the routing path. See appendix “I.4 Routing Heuristic (RH)” for more details on the routing heuristic of AGENET.

13. MIMI

The MIraveo MIddleware (MIMI) is the entity in charge of managing all the functionalities related with the management of the MANETs, providing both address and name resolution services and location based services.

Each node of a MANET will have associated a unique ID of 64 bits. In fact, this unique ID will be the key field to uniquely identify a node. The first two bytes of the ID will be set to 0 (reserved), and the subsequent six bytes will be set to the MAC address of the Wi-Fi chipset of the node.

Moreover, each node of a MANET will have associated an IP address. The IP address of a node will be randomly chosen in the range between the IP address 10.0.0.1 and the IP address 10.255.255.254. We will assume that never two nodes will collide in the same MANET and at the same time with the same IP address. However, distributed DHCP mechanism could be also used in order to set the IP addresses of the nodes of a MANET.

In order to deal with all the functionalities related with the management of the MANETs, MIMI uses the MIMI Table and the MIMI Hello messages. On the one hand, the MIMI Table is used to store some information of the current nodes of the MANET we are connected to. For example, the unique ID of the node, if this node is a gateway to an external network or not, or if this node has been discovered over unidirectional links. On the other hand, MiMiHellos are the messages used to distribute the node information to the rest of the nodes of the MANET, in order to build up the MIMI Tables. See section “Appendix J. MIMI Details” for more details on the operation of MIMI.

14. MIMA

The Miraveo Management (MIMA) network service is designed so that applications may create, join, leave, scan and, in general, manage MANETs. MIMA allows creating new MANETs or joining already existent MANETs. In order to join already existent MANETs, MIMA allows scanning MANETs.

14.1 MANET Scanning

The MIMA network service is designed so that applications will be able to scan MANETs. The scanning function allows scanning (i.e. discovering) which MANETs are in the near-by geographical area where the scan is being performed.

14.2 SSID management

In the IEEE 802.11 (or Wi-Fi) technology, frames use a field to identify the Basic Service Set (BSS) they belong to, known as BSSID, which has the same format as a MAC address. In infrastructure BSSs, the BSSID is set by the Wireless Access Point, and it is usually set to the MAC address of the Wi-Fi chipset of the Wireless Access Point. In independent BSSs (ad-hoc BSSs or MANETs), and according to the mentioned standard, the BSSID is randomly chosen by the node creating the BSS.

BSSs are usually identified by a Service Set Identifier (SSID), a string sent in the Beacon management frames, which at the same time allows other nodes to discover which BSSs are in their vicinity and join them. When a node is instructed by the user or by an application to join a network with a given SSID in ad-hoc mode (i.e., join a MANET), two possibilities appear:

- If the node detects the existence of this network in its proximity (because it is able to receive Beacon frames with the corresponding SSID), it simply adopts the BSSID, also indicated in the Beacon frames. This means that the node can normally communicate with the other nodes of the BSS, as it shares the same BSSID in the sent and received frames.
- If a node does not detect the existence of this network, it usually creates a new network or MANET, following the procedure described in the IEEE 802.11 Standard, or other suitable method. In the IEEE 802.11 Standard, the BSSID is essentially a random string of bits.

In the second case, the following situation can arise: if the node has created a BSS with a SSID and a randomly chosen BSSID, and it moves to an area where another BSS already exists, with the same SSID but with a different BSSID (as the BSSID was also randomly chosen by the node which created the second BSS), both BSSs cannot communicate directly, and the situation cannot be easily detected, as both have the same SSID. In other words, there can be two or more MANETs at the same place and time with equal SSID and different BSSID.

This situation is often referred to as “BSSID partitioning” (see Figure 40) and represents a serious problem in order to merge independently created MANETs.

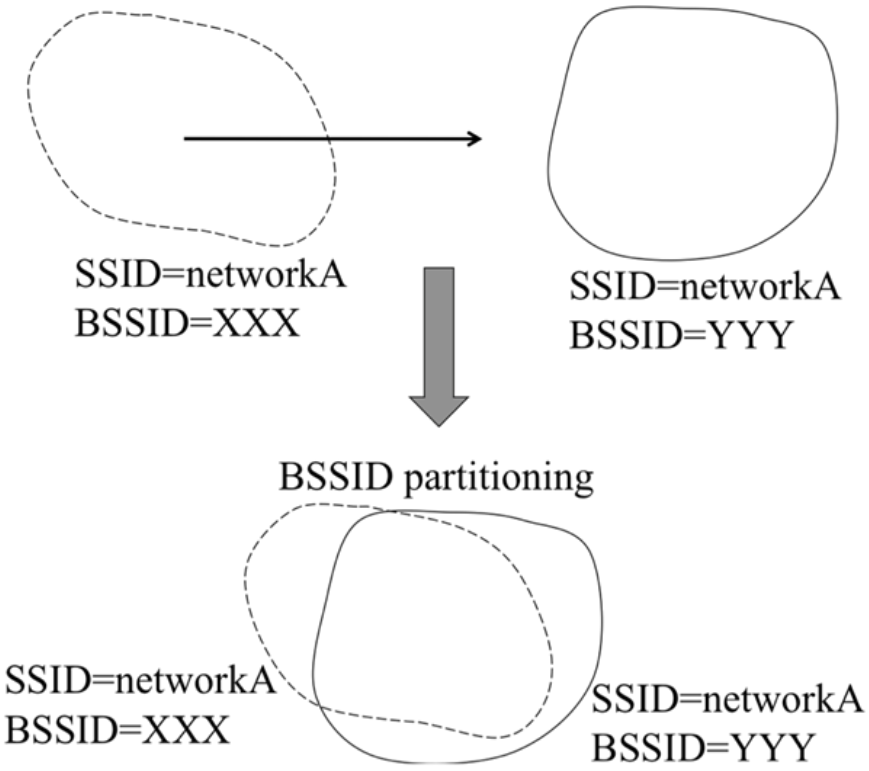

Figure 40. BSSID partitioning

In order to solve this problem, MIMA will restrict the set of possible SSIDs that can be used by a MANET to a large number of strings chosen randomly by MIMA. As an example, they can be defined to be the set of strings of the form "MANETxxxxxxxxxxxxxxxxxxxxxxxxxxxx", where "x…x" is a 28 character string (i.e. 224 bits) randomly chosen by the node which creates the MANET.

When a node decides to connect to a MANET, MIMA will perform a scanning function. If one MANET is found (i.e., SSID equal to "MANETxxxxxxxxxxxxxxxxxxxxxxxxxxxx"), then MIMA will join this MANET using its SSID. If several MANETs are found, then MIMA will join the MANET network with greatest SSID number, being the SSID number the part of the SSID corresponding to the last 28 characters. And if no MANET is found, a new MANET network will be created with SSID = "MANET$x_0x_1…x_{27}$" where x_i = random{'0','1','2','3','4','5','6','7','8','9'}.

The structure of these names is chosen to ensure that the collision probability of having two or more MANETs at the same place and time with equal SSID and different BSSID is small. In this way, when different MANETs are in the same geographical area, MIMA can detect with high probability the existence of the other MANETs using different BSSIDs thanks to the fact of also using different SSIDs. Note that when two MANETs have the same SSID but different BSSIDs they may not be able to communicate each other (i.e. they are in fact different MANETs).

14.3 Merging and splitting MANETs

MANETs may merge or split. Two MANETs merge when the nodes of one MANET move (i.e. join) into the other MANET. A MANET splits when it becomes two or more MANETs, which may be or not geographically separated.

The decision of merging MANETs (see Figure 41) may be taken automatically by MIMA. In order to reduce the number of MANETs coexisting in the same geographical area, the use of automatic procedures for nodes to join other MANETs may be established.

In order to automatically merge MANETs in the same area, MIMA will use a timer that will be executed every T_MERGEMANET milliseconds. Every time the timer expires, MIMA will perform a scanning function. If other MANET networks different from the one we are connected are discovered and at least one of these discovered MANET networks has greater SSID number, then MIMA will change from our MANET network to the one with greatest SSID number.

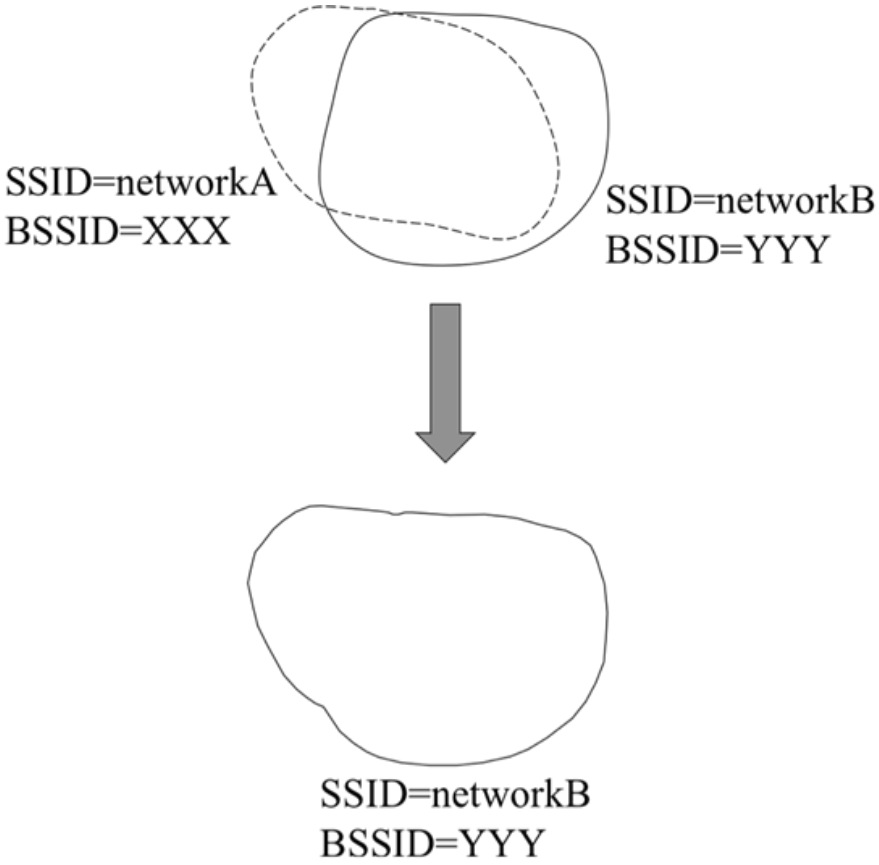

Figure 41. MANET Merging

Even with the method described before, the problem of the BSSID Partitioning in MANETs could still appear, for example when a node fails connecting to an already existent MANET (for example, due to problems we have detected with the scanning function of some Wi-Fi devices) and then, it creates a new MANET network with the same SSID but with different BSSID, or for example when two distant nodes that created a MANET network in the past coincide in the future on the same place and their random part of the SSID of their MANET is the same.

In these special cases, the following can be performed in order to solve the problem. Every time the timer of T_MERGEMANET expires and the scanning function is performed, if we detect two or more MANETs with the same SSID, different BSSID, and we are at this time connected to one of these networks (i.e., BSSID partitioning), then we will proceed to merge the MANETs only if there exists a MANET with greater SSID than the SSID of our MANET. If

there is not any MANET with greater SSID, then, instead of continue residing in the same MANET, we will create a new MANET as explained before but choosing the random part of the new SSID between the number of our current SSID plus 1 and the maximum value for this number. In the case that our current SSID is already the maximum value, then the random will be chosen between 0 and the half of the maximum value. With this solution, after a certain period of time, every node will end to the same MANET, i.e., all MANETs will have merged to a unique MANET, thus solving the BSSID partitioning problem.

14.4 MANET Horizons

MIMA may set some limits in the amount of nodes that may belong to a MANET, or in the communication capabilities of nodes of a MANET.

For instance, MANETs may have a horizon (which can be finite or not). This horizon establishes a limit beyond of which the communication is unreliable or even impossible. The horizon can be defined in terms of forwarding hops, in terms of geographical area (i.e. distance between users), in terms of number or density of users per MANET, in terms of data traffic (i.e. bandwidth usage), etc.

These MANET limits may be of importance when establishing policies for merging and splitting MANETs. For example, a possible horizon for MANETs could be an infinite horizon, only limited by the number of neighbors at distance 1 hop, meaning that every node is allowed to join a MANET until the maximum of 1-hop neighbors allowed per MANET is reached. If reached, a new MANET (for example, in a different channel) may be created to allow joining additional users. In this way, MANETs tend to be as large as possible.

15. Results

Beside the problem of the current communications stacks for MANETs explained in section 2, i.e. trying to adapt the current solutions of the wired networks to the wireless networks instead of designing new solutions, a second big problem has affected the research work in the topic of the MANETs.

The vast majority of the research work has been always based on results obtained via network simulators. Despite the evident advantage of using a network simulator concerning the operational costs of the experiments, this fact has caused an obtaining of results far away from reality. Not only network simulators are far away from simulating real conditions but also a lot of key properties for the correct operation of MANETs have been hided due to the fact of using simulators for the experimentation.

Have results based on simulations considered the problem that certain Wi-Fi devices, such as smartphones, have concerning the directivity of their Wi-Fi antennas? The simple fact of turning a smartphone in one direction or in another or the simple fact of having the smartphone in our pocket instead of in our hands can be of huge impact concerning the results of the experiment. This is important not only for evaluating the new proposed solutions but also for discovering this kind of details (only discoverable with real experimentation and not with simulations) that help a lot on improving our decisions of design.

Obviously, using real experimentation we cannot test one thousand mobile nodes in a five hundred square meters area. Only with network simulators we can test big scenarios. But this fact, far from being an advantage of the network simulators, is also another big error caused by the use of simulators. With difficulty a real MANET will have one thousand mobile nodes and will be located in a five hundred square meters area. A real MANET will usually have a few tens of nodes in a small area. And we should not forget that our task as researchers is not designing better solutions than our research partners; our task is designing better solutions being able to work in the real world.

For all these reasons, our preference has been for real experimentation instead of for the more traditional evaluation through simulation, as we believe that current simulation models hardly capture with sufficient fidelity the particularities of wireless channels in complex scenarios; see [**42**].

15.1 Cooperative Forwarding

15.1.1 Prototype

To test the efficiency and the improvements in the robustness of a MANET of the Cooperative Forwarding mechanism described before we implemented, using Click Modular Router [**43**] and MadWifi driver [**44**], a prototype of the Cooperative Forwarding mechanism combined with two implementations of AODV [**13**] and LUNAR [**45**] as routing protocols.

The Cooperative Forwarding mechanism introduces an additional header between the IEEE 802.11 and the IP headers. The size of this Cooperative header is 7 Bytes and includes several fields such as the Cloud number, a packet global identifier, several flags and the upper layer protocol type.

Concerning the implementation of the prototype, in Figure 42 we can see the Click scheme of the prototype of the Cooperative Forwarding mechanism together with AODV as routing protocol.

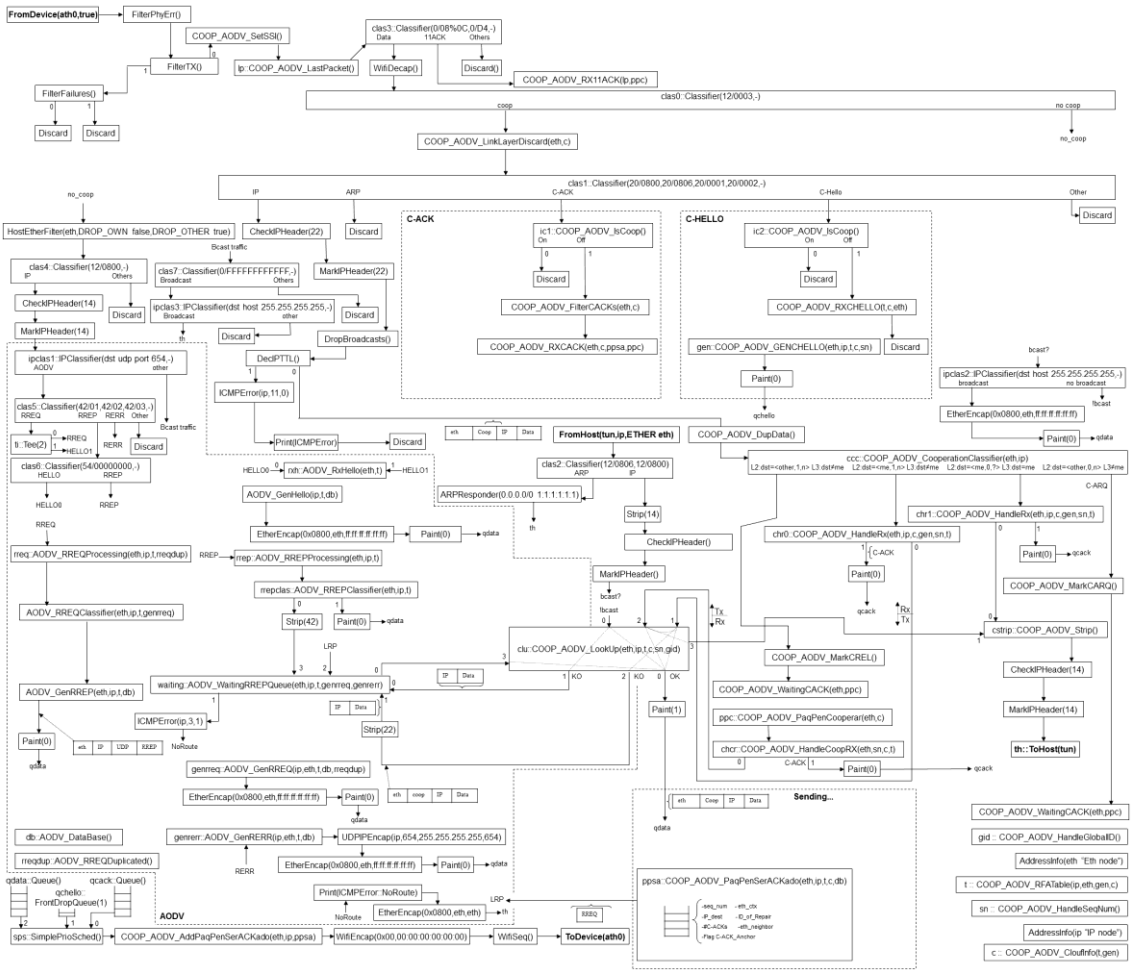

Figure 42. Click Scheme of the Cooperative Forwarding with AODV

If we analyze in detail the Click scheme of Figure 42, we can split it into several modules. In Figure 43 we can see the module corresponding to the reception of any packet from the network. The Monitor mode is needed in the Wi-Fi network device in order to be able to receive not only data packets but also control and management packets. The Radiotap Header of the packets is needed in order to know the SNR of the packets received.

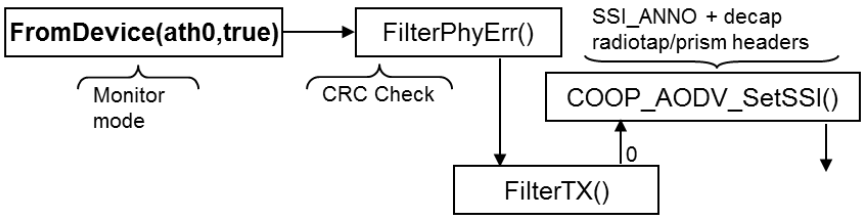

Figure 43. Wi-Fi Reception Module

In Figure 44 we can see the module corresponding to the transmission of any packet to the network. Several output buffers are needed to set priorities between control packets (CACK and CHELLO packets) and data packets.

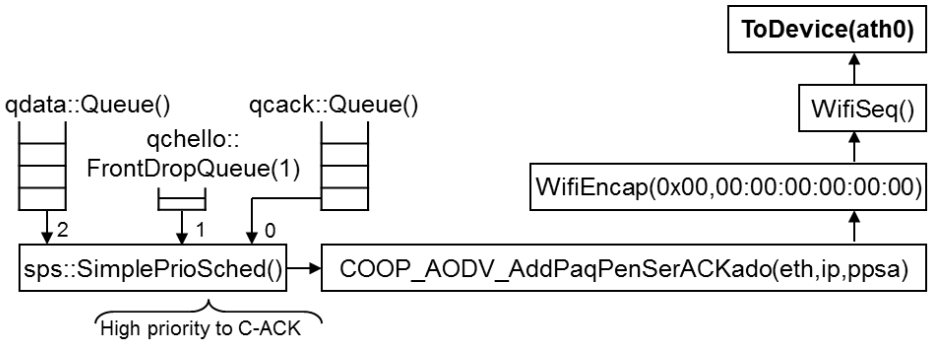

Figure 44. Wi-Fi Transmission Module

In Figure 45 we can see the module corresponding to the data structures used for Cooperative Forwarding and AODV, i.e., the routing, forwarding and ARP tables and the Cloud information.

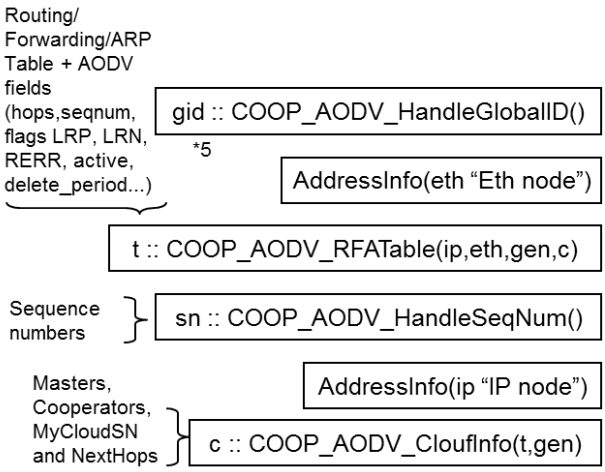

Figure 45. Data Structure Module

In Figure 46 we can see the module corresponding to the reception of 802.11 ACK packets. Master nodes are the explicit receivers of the data packets. When they correctly receive a data packet, they transmit the corresponding 802.11 ACK packet. In Cooperative Forwarding, the 802.11 ACK packets will be also used to signal the cooperators of the cloud in order to avoid useless cooperations.

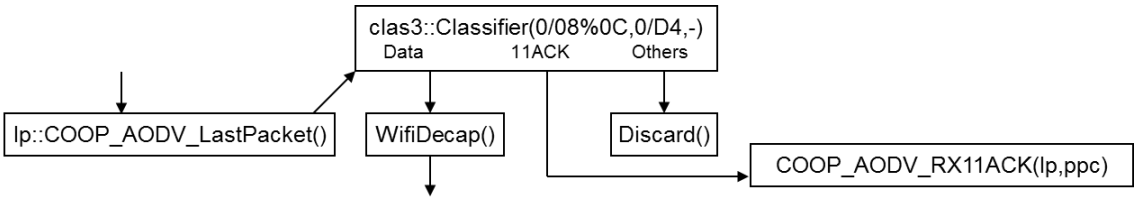

Figure 46. 802.11 ACK Reception Module

In Figure 47 we can see the module corresponding to the reception of Cooperative ACK (CACK) packets. This module is needed when cooperator nodes decide to cooperate. Since

cooperators are not the explicit receivers of the packet, when they cooperate, they do not transmit the corresponding 802.11 ACK packet, and so, a Cooperative ACK packet instead is sent in order to avoid duplicated cooperations.

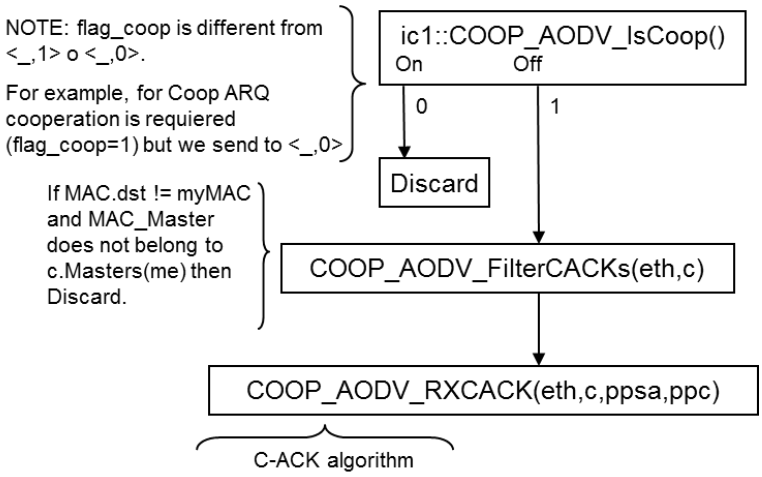

Figure 47. Cooperative ACK (CACK) Reception Module

In Figure 48 we can see the module corresponding to the reception and transmission of Cooperative HELLO (CHELLO) packets, needed to form the clouds of cooperator nodes around the master nodes.

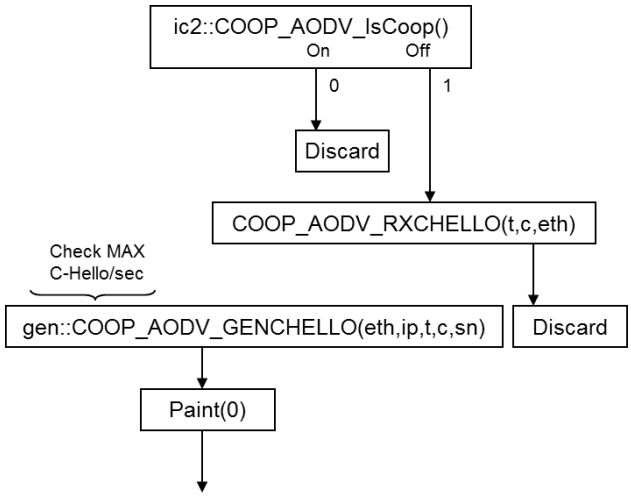

Figure 48. Cooperative HELLO (CHELLO) Module

In Figure 49 we can see the module corresponding to the routing protocol, i.e., AODV (or LUNAR) combined with the Cooperative Forwarding mechanism.

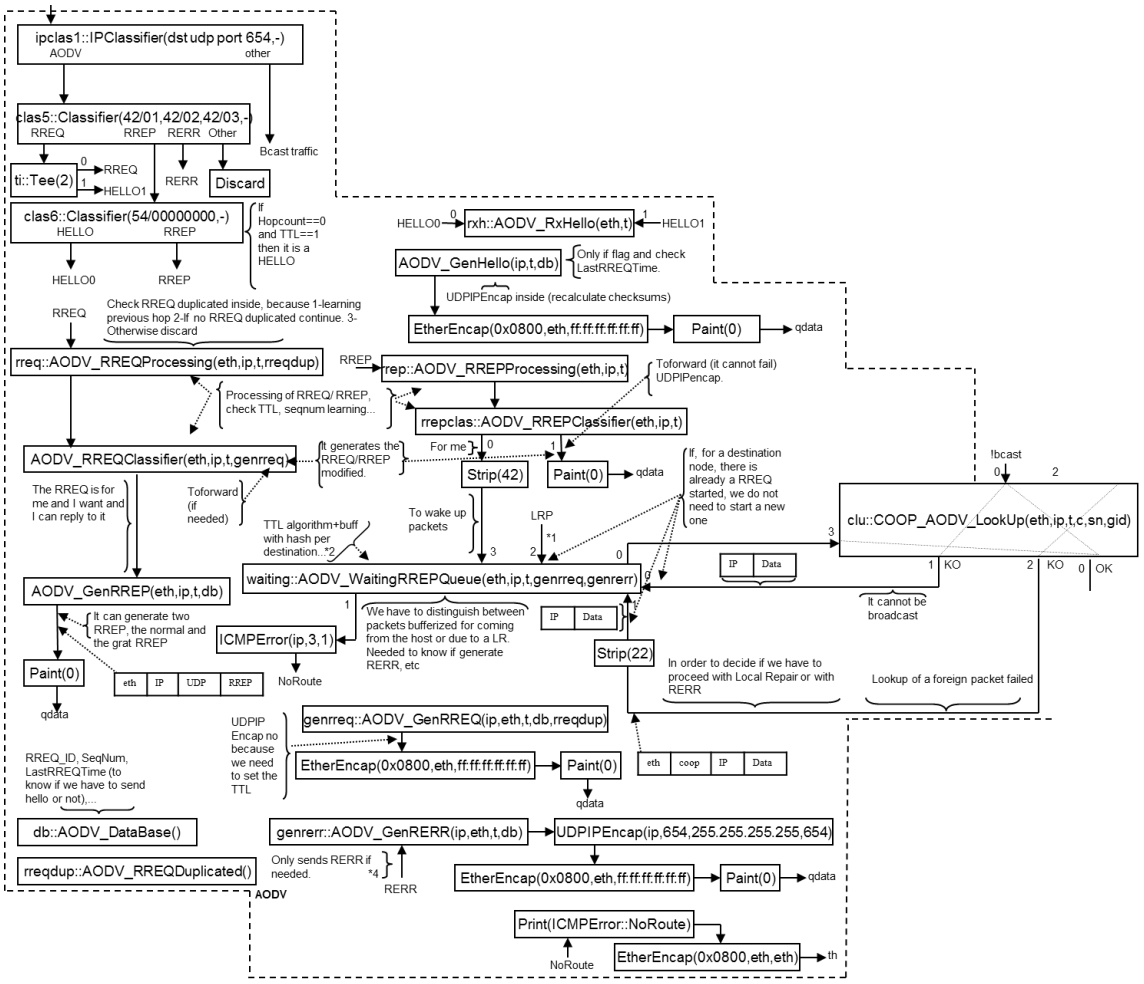

Figure 49. Routing Module

In Figure 50 we can see the core module of the Cooperative Forwarding mechanism. Several actions are performed in this module:

- Classifying the packets depending on if we are masters or cooperators for a packet
- Queuing the packets pending to be sent as a cooperative action (i.e., waiting for the corresponding 802.11 ACK or CACK packet)
- Executing the cooperative actions if it is the case
- Looking up in the Forwarding Table

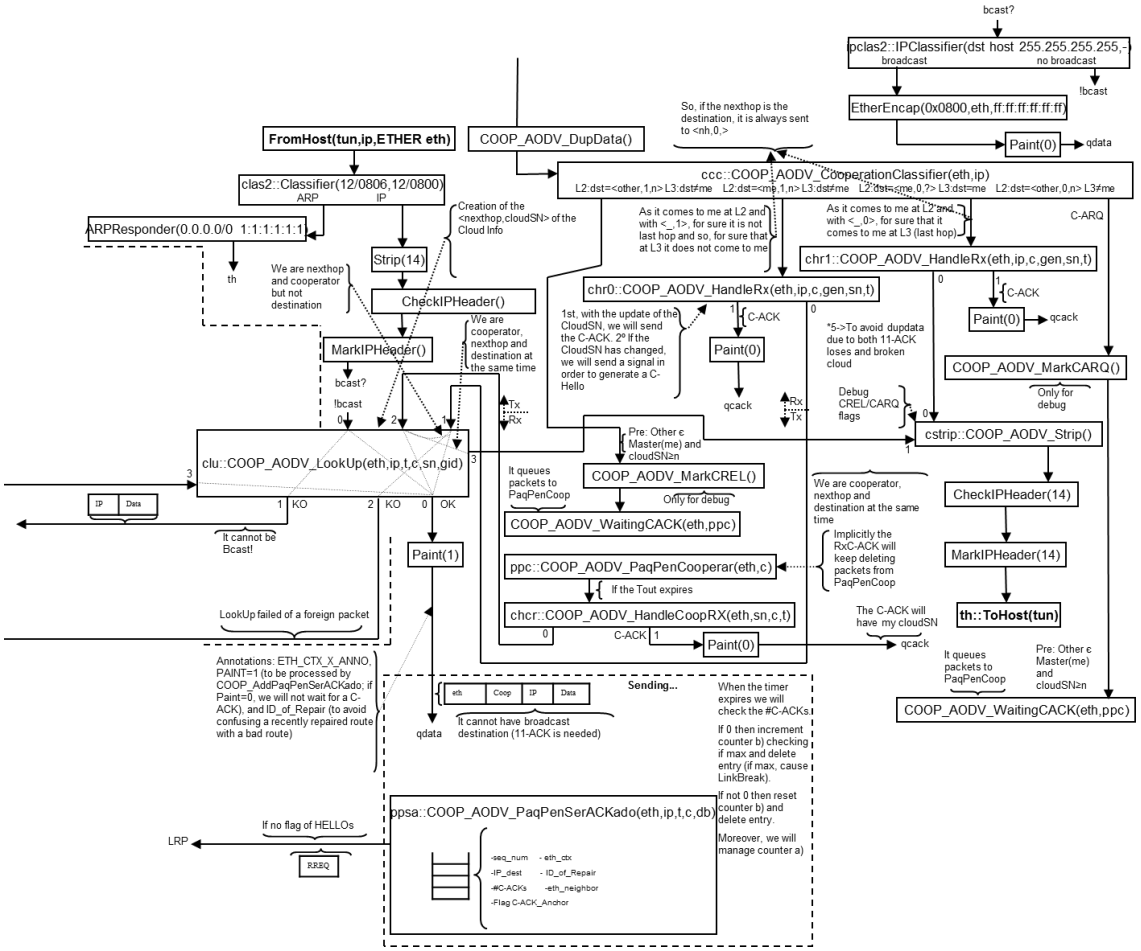

Figure 50. Cooperative Forwarding Module

Concerning the values of the parameters of the prototype, it is important to say that the prototype uses off-the-shelf IEEE 802.11 technology although its performance would undoubtedly be improved if MAC supporting native cooperative signaling were used. The Wi-Fi parameters can be found in Table 2.

Table 2. Wi-Fi Parameters

Parameter	Value
PHY	OFDM
Rate	6 Mbps
Modulation at 6 Mbps	BPSK
Coding rate at 6 Mbps (R)	1/2
Coded bits per carrier at 6 Mbps	1
Coded bits per symbol at 6 Mbps	48
Data bits per symbol at 6 Mbps	24
DIFS	34 microseconds
SIFS	16 microseconds
CW_{min}	15
CW_{max}	1023
Slot time (σ)	9 microseconds
Preamble duration ($T_{PREAMBLE}$)	20 microseconds
PLCP header duration (T_{PLCP})	4 microseconds
ACK timeout (T_{ACK_TOUT})	25 microseconds
Symbol duration (T_{SYM})	4 microseconds
Tail bits	6 bits
Wireless card mode	Monitor

Channel (cooperative case)	40
Channel (non-cooperative case)	140
Number of retransmissions (N)	7
TX power	15 dBm
RTS/CTS	Disabled
MTU	1500 Bytes

We used 802.11a channels to execute the experiments. We also used 7 retransmissions per packet, as it is defined in the IEEE 802.11 standard, and 6 Mbps, as it is the default rate used by MadWifi when the wireless card is in monitor mode. Monitor mode is the only one which allows us to capture not only data frames but also control and management frames. The retransmission algorithm was 7 retransmissions all at 6 Mbps. The transmission power was set to the maximum allowed by the wireless cards. Finally, the RTS/CTS mechanism was disabled, mainly due to limitations of the prototype, and the MTU was set to its default value of 1500 Bytes. All the other parameters are PHY-dependent, and can be found in the IEEE 802.11a standard, with the exception of the ACK timeout, which can be found in the proc file system of the laptops (/proc/sys/dev/wifi0/acktimeout).

The AODV (and LUNAR) parameters can be found in Table 3.

Table 3. AODV Parameters

Parameter	Value
Local repair	Disabled
Hello messages	Enabled
Destination only flag	Enabled
Port	654
RREQ type	1
RREP type	2
RERR type	3
Net diameter	5 hops
RERR rate limit	10 RERR per second
RREQ retries	3
RREQ rate limit	20 RREQ per second
Hello interval	100 milliseconds
Allowed hello loss	20 hello messages
Node traversal time	10 milliseconds
Network traversal time	2 * Node traversal time * Net diameter
Path discovery time	2 * Net traversal time
Active route timeout	6000 milliseconds
My route timeout	2 * Active route timeout
K delete period	3
Delete period	K delete period * MAX(Active route timeout,Hello interval)

The local repair mechanism of AODV was disabled, mainly due to the small size of the network. Hello messages were enabled in order to detect link breaks, and the destination only flag was enabled, also mainly due to the small size of the network. The net diameter was set to 5 hops, enough to cover this type of experiment. We increased the number of allowed RREQ per second set by the RFC from 10 to 20 and the number of RREQ retries from 2 to 3. Moreover, we decreased the hello interval from 1 second to 100 milliseconds, and we increased the allowed hello loss from 2 to 20 hello messages. The reason of these differences between our values and the values of the RFC is that, during the experiments, we realized that the values

specified in the RFC were values not well fitted in real implementations of AODV, causing very poor performance to AODV, specially due to slow route discovery processes and to the detection of fake link breaks due to hello losses. The active route timeout was set to a reasonable value of 6 seconds, the node traversal time to 10 milliseconds, the K delete period to 3, and all the other parameters were set to their default values specified in the RFC.

The Cooperative Forwarding parameters can be found in Table 4.

Table 4. Cooperative Forwarding Parameters

Parameter	Value
Max number of cooperators	4
Native retransmission time	1000 microseconds
Cooperative retransmission time	2000 microseconds
CHello interval	100 milliseconds
Cloud timeout	6000 milliseconds
Cooperator timeout	3000 milliseconds
Next hop cloud timeout	Cloud timeout
Neighbor timeout	Cloud timeout
Duplicated data timeout	200000 microseconds
Increment cloud number timeout	3000 milliseconds
Maximum TX failures counter	1
Ethertype COOPERATIVE	0x0003
Minimum SSI CHello	30 dB
Minimum SSI to cooperate	20 dB
SSI exponential component	Disabled
SSI decrement factor	10 dB
Maximum sequence number	0x3FFF
Minimum sequence number	0
Maximum global ID	0x3FFF
Minimum global ID	0
Maximum random value	100000
CHello rate limit	20 CHELLO per second
Timestamp debugging flag	Disabled
Cooperative debugging flag	Disabled
Radiotap size	26 Bytes
Prism size	144 Bytes

We set the maximum number of cooperators to 4, a reasonable value for this scenario. The native retransmission time and the cooperative retransmission time are the most critical parameters of the prototype. The native retransmission time is the time a node takes to send a packet and to receive the corresponding ACK. The cooperative retransmission time is the time a cooperator node takes to cooperate with a packet and send the corresponding CACK. Since, from the point of view of the application layer, these times depend on the current size of the wireless driver queues and the usage and quality of the wireless channel, it is very difficult to set up these parameters correctly, since their value could vary a lot over time. In these experiments, we decided to set the parameters to 1000 and 2000 microseconds respectively, and are approximated values based on captures of previous experiments. The reason for which these values are critical is that they are used to calculate the value of two important timeouts, the timeout of a packet pending to be used for cooperation (PPC) and the timeout of a packet pending to be cooperative-acknowledged (PPA). PPC is used by cooperator nodes to know when they have to cooperate for one packet or not. Wrong values of this parameter could cause a lot of overhead, due to duplicated packets of useless and premature cooperations, if

the value is too small, or too much delay to cooperate if the value is too big. PPA is used by cooperator nodes to know, after sending a cooperative packet, how much time they have to wait to receive the corresponding CACK. Wrong values of this parameter could cause believing that the cooperator is a bad cooperator when it is good, if the value is too small, or too much delay to penalize a bad cooperator if the value is too big. The CHELLO interval was set to 100 milliseconds (equal to the Hello interval of AODV). The cloud, next hop cloud and neighbor timeouts were set to 6000 milliseconds, and the cooperator timeout was set to 3000 milliseconds, mainly to do the mechanism more reactive in front of node mobility. The timeout used to avoid forward duplicate data was set to 200000 microseconds, and the timeout used to increment periodically the cloud number was set to 3000 milliseconds. Another important parameter of the mechanism is the maximum value of the transmission failures counter. It is used, in conjunction with the PPA timeout, to detect bad cooperators: if the cooperator node does not receive any CACK for "Maximum TX failures counter" consecutive cooperative packets, it will penalize itself as a bad cooperator. This parameter was set to 1. The minimum SSI CHELLO is the minimum SSI in which cooperator nodes are considered by a master node. A master node will only have as cooperators the nodes from which it receives their CHELLO packets with ≥30 dB of signal. The minimum SSI to cooperate is the minimum SSI which cooperator nodes have to have with a next hop cloud in order to be able to cooperate with it. Cooperators will only cooperate with next hop clouds of signal quality 20 dB or more. In order to penalize bad cooperators due to unsuccessful cooperations, two parameters can be used: the SSI exponential component and the SSI decrement factor. The SSI exponential component is used to penalize cooperators who over time and due to node mobility become bad cooperators, and without the need of waiting for its cloud timeout expiration. The SSI decrement factor is used to penalize cooperators who, for "Maximum TX failures counter", they have tried to cooperate and no good cooperations are achieved. In the experiments, only the second one was used, and it was set to 10 dB. The maximum sequence number and maximum global ID parameters are defined by the size of these parameters in the cooperative header, that is, the maximum value you can achieve with 14 bits. Finally, the limit of CHELLO packets per second was set to 20. The rest of the parameters are self-explained or fixed by standard values.

The TCP parameters can be found in Table 5.

Table 5. TCP Parameters

Parameter	Value
tcp_abc	1
tcp_abort_on_overflow	0
tcp_adv_win_scale	2
tcp_app_win	31
tcp_congestion_control	bic
tcp_dsack	1
tcp_ecn	0
tcp_fack	1
tcp_fin_timeout	60
tcp_frto	0
tcp_keepalive_intvl	75
tcp_keepalive_probes	9
tcp_keepalive_time	7200
tcp_low_latency	0

tcp_max_orphans	32768
tcp_max_syn_backlog	1024
tcp_max_tw_buckets	180000
tcp_mem	98304 131072 196608
tcp_moderate_rcvbuf	1
tcp_no_metrics_save	0
tcp_orphan_retries	0
tcp_reordering	3
tcp_retrans_collapse	1
tcp_retries1	3
tcp_retries2	15
tcp_rfc1337	0
tcp_rmem	4096 87380 174760
tcp_sack	1
tcp_stdurg	0
tcp_synack_retries	5
tcp_syncookies	0
tcp_syn_retries	5
tcp_timestamps	1
tcp_tso_win_divisor	3
tcp_tw_recycle	0
tcp_tw_reuse	0
tcp_window_scaling	1
tcp_wmem	4096 16384 131072
tcp_bic_beta	819
tcp_bic_fast_convergence	1
tcp_bic_initial_ssthresh	100
tcp_bic_low_window	14
tcp_bic_max_increment	16
tcp_bic_smooth_part	20

All these parameters are the default parameters used in the Linux Kernel 2.6.16.13, which is the one used in the TCP experiments.

15.1.2 Experimental Setting

We performed a number of experiments in indoor scenarios with pedestrian mobility, and depending on the test, we used UDP or TCP traffic. Moreover, we used IEEE 802.11a at 6 Mbps in order to reduce the complexity of capturing and processing data. The RTS/CTS mechanism was disabled, mainly due to limitations of the prototype, and the MTU was set to its default value of 1500 Bytes.

The IEEE 802.11 retransmission algorithm was the default used by MadWifi when the wireless card is in monitor mode. The number of retransmissions was the default value specified in the IEEE 802.11 Standard, that is, a maximum of 7 retransmissions of the same packet before the 802.11 ACK is received. We performed three sets of measurements and depending on the experiment, we used 4, 11 or 6 laptops respectively, all of the same model (Toshiba Satellite Pro A120) and with the same wireless card (Cisco Aironet AIR-CB21AG-E-K9 802.11a/b/g).

We will see, first, a simple UDP scenario with controlled mobility to show the operation of the Cooperative Forwarding mechanism. Then, a more complex scenario with random mobility and multi-hop communications is presented to compare the performance of the Cooperative and the traditional (or non-cooperative) forwarding mechanisms in a dense MANET. Finally, a

simple scenario is presented to analyze the behavior of TCP against the Cooperative and the traditional (or non-cooperative) forwarding mechanisms during link breaks due to mobility.

15.1.2.1 Two-hop Scenario and Controlled Mobility using UDP

The first set of experiments corresponds to a simple scenario: a single two-hop communication path, 4 nodes and a controlled mobility pattern; see Figure 51.

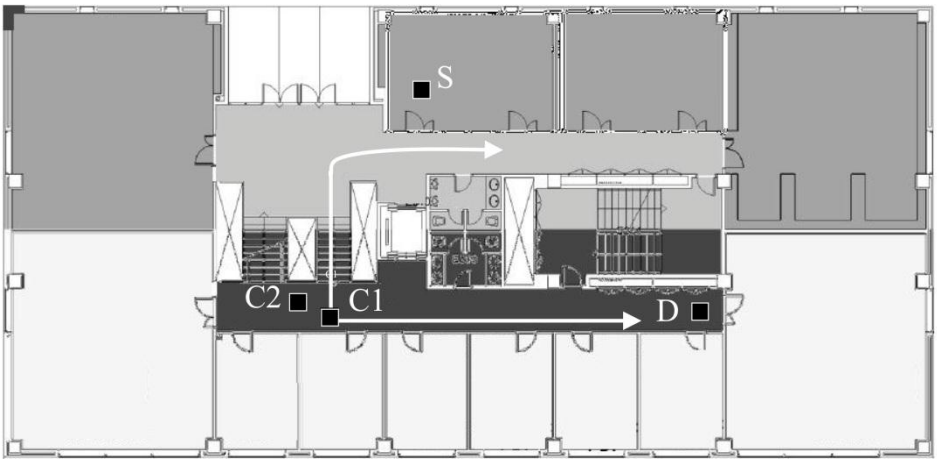

Figure 51. Map of the Two-hop Scenario Using UDP

These tests were designed to show the operation of the Cooperative Forwarding mechanism in two basic situations that we call *Relaying* and *ARQ*. Relaying corresponds to the situation in which cooperators perform the functionality of an intermediate master node (i.e., the case $L_i \neq A_i$), while ARQ corresponds to the situation in which cooperators of the destination node receive the packet and retransmit it to this destination node; see Figure 52. In the Relaying situation, the number of hops of a path does not increase, while in the ARQ situation this number increases by one.

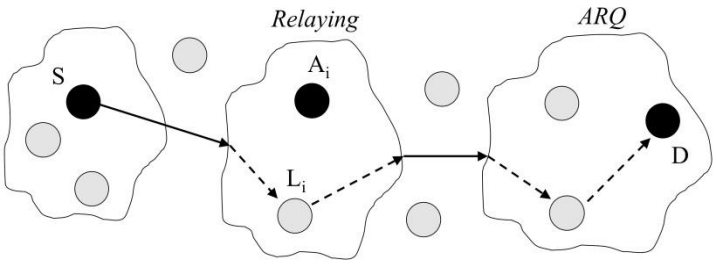

Figure 52. Difference between Relaying and ARQ Situations

During these experiments, we used both AODV (with link layer feedback) and LUNAR as routing protocols. In these experiments a two hops path between nodes S and D was established in a floor of one of the buildings of the UPC campus; see Figure 51. The intermediate hop was one of two nodes, C1 and C2, which were placed in the same area. The scenario was controlled in order to guarantee that there was not a direct communication between S and D, but that there was an acceptable connection between nodes S and D using either C1 or C2 as intermediate node.

A CBR flow of UDP packets of 1000B of payload at a rate of 100 packets/sec between S and D was established. After few seconds, one experimenter took the intermediate node C1 or C2

and moved it toward either node S or node D, causing a link break and, unless the routing protocol reacted fast enough to perform a link repair, giving the chance to the other intermediate node to cooperate.

The Relaying situation occurs when the intermediate node (e.g. C1) is moved towards D. In this case, S losses connectivity with C1 and the cooperator node C2 acts as the intermediate node of the path. The ARQ situation occurs when the intermediate node (e.g. C1) is moved towards S. In this case, C1 losses connectivity with D and the cooperator node C2 helps C1 to retransmit the packets to D.

15.1.2.2 Multi-hop Scenario and Random Mobility using UDP

The second set of experiments corresponds to a more complex scenario, in which a number of nodes move along a floor of one of the buildings of UPC campus. These tests were designed to compare the performance of the cooperative and traditional (or non-cooperative) forwarding mechanisms in a dense MANET. During these tests we used LUNAR as routing protocol. The experimentation area was the second floor of a 20m x 40m building of the UPC campus, and it includes the corridors and 4 offices that remained with the doors opened throughout the experiment. The central part of the building is occupied by service rooms, an elevator, and the main stair of the building; see Figure 53 and Figure 54.

The experiments took place at night, meaning that the other offices were closed and there was no interference from extraneous persons. Moreover, we used the UNII band, and no measured interference from other networks was present. The dimensions and propagation characteristics are such that two hops were usually enough to guarantee connectivity between any two points, although depending on the node placement, the hop-count could increase to 3, 4 or even 5. 11 laptops were used in the experiment: (i) 4 of them were kept static in the same positions throughout the whole set of experiments (marked as squares in Figure 53 (ii) 2 of them suffered slow and short displacements in the middle of the two long corridors; and (iii) 5 of them were carried by experimenters who followed a random pedestrian mobility pattern.

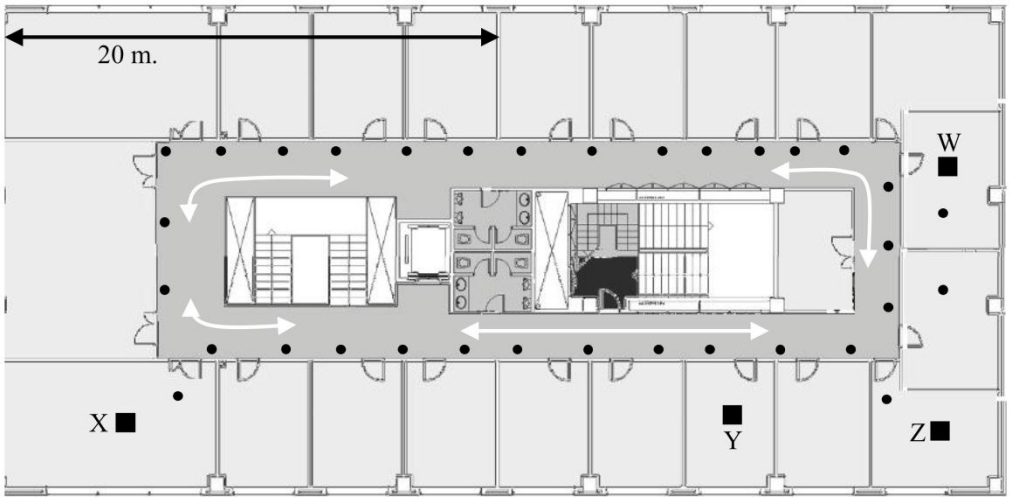

Figure 53. Map of the Multi-hop Scenario Using UDP

Figure 54. Pictures of the Multi-hop Scenario Using UDP

The mobility pattern was the following: each participant in the experiment walked in a given direction (clockwise or counterclockwise) along the corridor, independently of the other experimenters. 33 chairs (marked as small dots in Figure 53) were placed in the experimental area; see Figure 54. The walk of the experimenter consisted in counting 8 (in counterclockwise direction) or 9 chairs (in clockwise direction), which approximately accounted for a time period of around 20 seconds. After this time, she/he sat down in the next free chair for approximately 15 or 20 seconds. After this period, she/he randomly chose a new direction, repeating the whole process. Given the size of the building, during a walking period, a node could almost complete one fourth of a round of the building. The experiment duration was of 20 minutes, meaning that each node made around 30 walking-stop cycles. During the experiments it was already clear that many different configuration of nodes appeared. Sometimes, nodes were alone, whereas in at least one occasion the 5 mobile nodes were concentrated in the same area of the building.

In both cooperative and non-cooperative experiments, we generated traffic consisting of 4 CBR traffic flows of 1000B of payload in UDP packets sent at a rate of 20 packets/sec (24000 packets per flow). The mobility patterns of each flow were the following: (i) *Flow A*: from a fixed node placed at point *X* to a fixed node placed at point *Z*. (ii) *Flow B*: from a fixed node placed at point *Y* to a mobile node. (iii) *Flow C*: from a mobile node to a fixed node placed at point *W*. (iv) *Flow D*: from a mobile node to a mobile node.

Due to the random choice of the walk direction at each cycle, each node moved differently during each experiment, although always following the same mobility procedure. This means that the experiments are *not repeatable*, as we did not record the exact movement patterns of the participants in the experiment. We thus need to justify that, while the nodes moved differently during each experiment, the experiments in which we compare the cooperative and the non-cooperative scheme give results that can be *statistically compared*. To this end, we compared the distribution of hop-count per flow in both experiments, as this is an easily measured parameter that has a direct impact on the packet drop ratio per flow. With a random sample of 100 packets per flow and experiment, we can see the average path hop count in Table 6.

Table 6. Average Path Hop Count per Flow in the Multi-hop experiment

	Cooperative Experiment	Non-cooperative Experiment
Flow A	2,52	2,53
Flow B	1,57	1,56
Flow C	1,43	1,44
Flow D	1,33	1,22

For flows A, B and C we made a chi-square homogeneity test, which gave a very good agreement. The contingency table for flow A is shown in Table 7. For flow D (the only flow with both source and destination nodes being mobile nodes), Table 6 shows that there was a small bias in favor of the non-cooperative experiments.

Table 7. Contingency Table for Chi-square Homogeneity Test for Flow A

	A2	A3	A4	Row tot
Coop	54	40	6	100
Non-coop	53	41	6	100
Col tot	107	81	12	200
χ^2	0,02			
Degree of freedom	2			
$Q(\chi^2_{dg})$	0,99			

15.1.2.3 Two-hop Scenario and Controlled Mobility using TCP

The third set of experiments corresponds again to a simple scenario: a single two-hop communication path, 6 nodes and a controlled mobility pattern; see Figure 55.

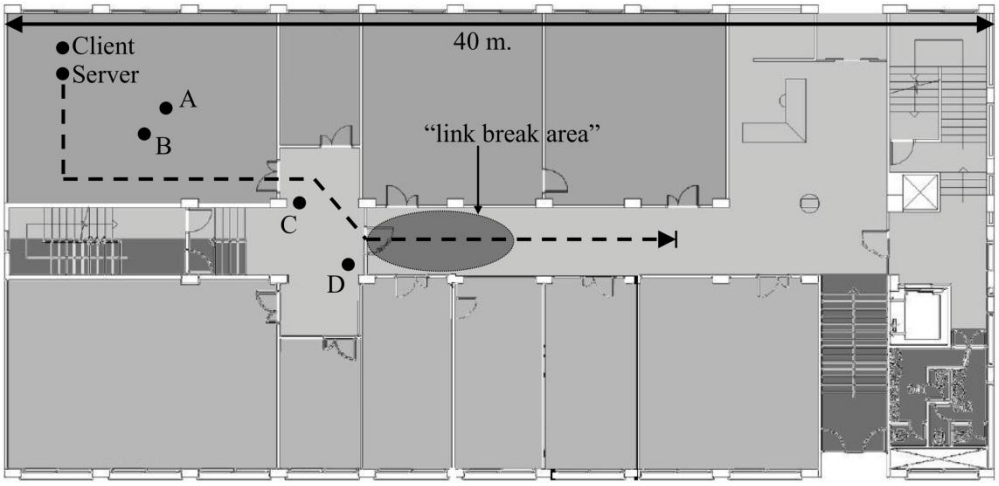

Figure 55. Map of the Two-hop Scenario Using TCP

These tests were designed to analyze the behavior of TCP against Cooperative and traditional (or non-cooperative) forwarding mechanisms during link breaks in the routing path due to mobility of nodes. The routing protocol was AODV and the TCP stack was the one included in the Linux kernel 2.6. The experimentation area was the first floor of a building on the UPC campus; see Figure 55.

We used six laptops for each of both the Cooperative and the non-cooperative cases. We placed all the 6 laptops of the non-cooperative case just over the 6 laptops of the Cooperative case. This set up helps us to compare the results of both cases (although they cannot be strictly compared) since the laptops of each case are placed at the same location and/or moved with the same pattern. In order to not interfering both cases each other, they were executed in orthogonal wireless channels. See Figure 56.

Figure 56. Laptops in the Two-hop Scenario Using TCP

We generated traffic between a client and a server consisting of one flow of 1000 Bytes of payload in TCP packets sent at the maximum rate allowed by the network. The six nodes for each case were placed on the black dots in Figure 55. The nodes placed on the Client, A, B, C and D dots were static nodes. The node placed on the Server dot was moved according to the following mobility pattern: (1) when the TCP stream between the client and the server was started, the server node remained at the starting point for 10 seconds. (2) After 10 seconds, the experimenter took the server node and, with pedestrian mobility, moved it according to

the path marked with the dashed line in Figure 55. (3) When the experimenter arrived at the end of the dashed line, he waited there for 30 seconds before stopping the TCP stream.

In this scenario, the initial one-hop path between the client and the server was maintained until the server entered the “link break area” of Figure 55 (30 seconds from the beginning of the experiment). At this time, a link break was detected by the routing protocol and the route was repaired. A two-hop path was then created using one of the possible nodes between the client and the mobile server.

15.1.3 Experimental Results

15.1.3.1 Two-hop Scenario and Controlled Mobility using UDP

In Figure 57, Figure 58, Figure 59, Figure 60, Figure 61, Figure 62, Figure 63 and Figure 64 we show screenshots of four tests we executed in this scenario. For every packet and for every node of the test, the screenshot marks whether a packet was transmitted (or received in the case of node D) via cooperation or not. The four tests correspond to the Relaying and ARQ situations for both LUNAR and AODV routing protocols. Moreover, in all these tests C1 was the intermediate node that initially belonged to the established routing path between S and D and that was moved to cause the link break. C2 was always the cooperator node.

For every test we show the behavior of the Cooperative Forwarding mechanism when the quality of the link is decreasing (see Figure 57, Figure 58, Figure 61 and Figure 62) and when the routing protocol decides to repair the route (see Figure 59, Figure 60, Figure 63 and Figure 64) due to the movement of C1.

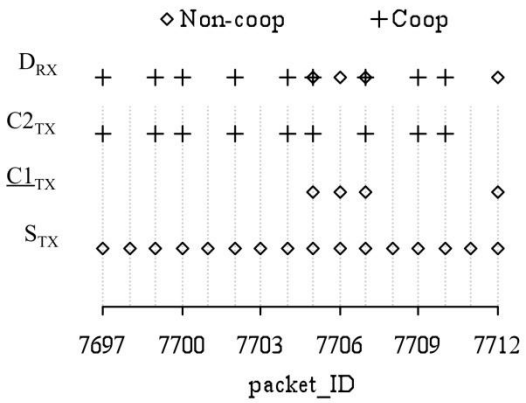

Figure 57. Behavior of Relaying in LUNAR (before Route Discovery)

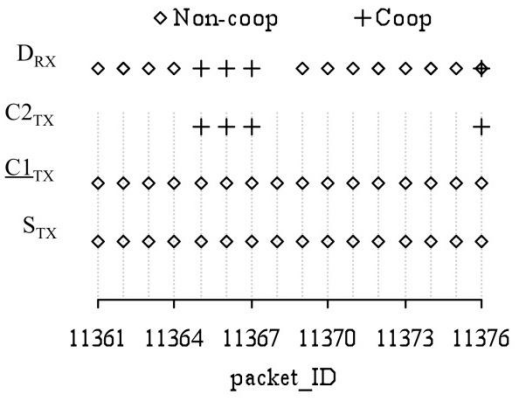

Figure 58. Behavior of ARQ in LUNAR (before Route Discovery)

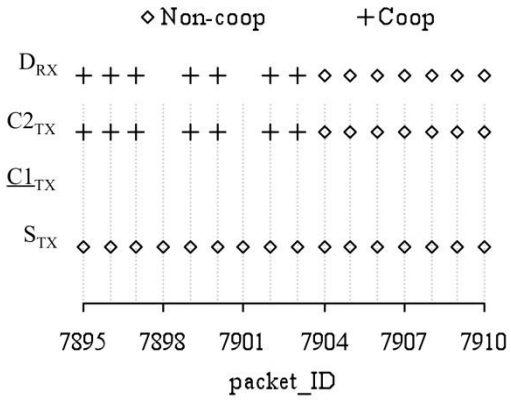

Figure 59. Behavior of Relaying in LUNAR (Route Discovery)

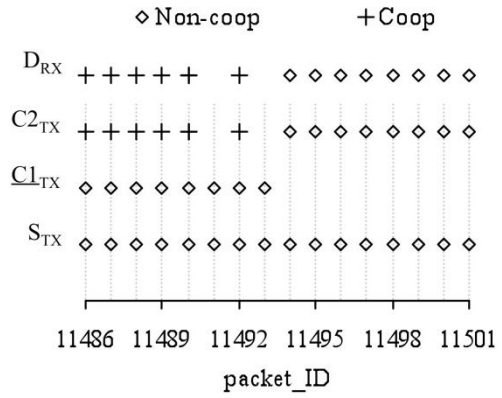

Figure 60. Behavior of ARQ in LUNAR (Route Discovery)

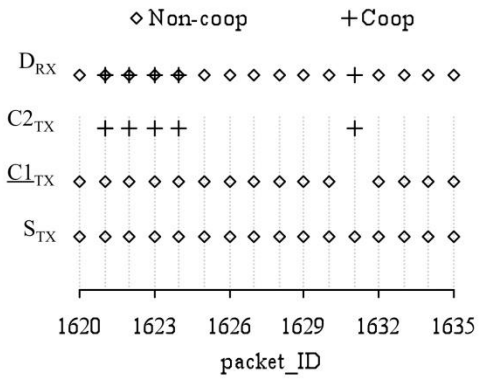

Figure 61. Behavior of Relaying in AODV (before Route Discovery)

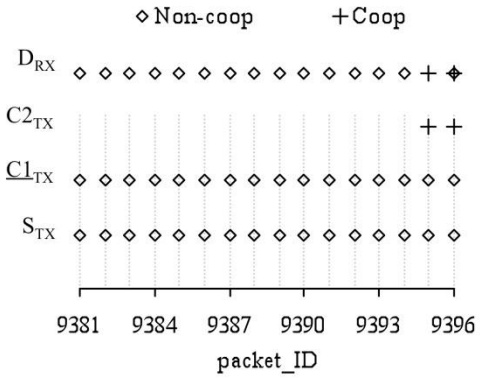

Figure 62. Behavior of ARQ in AODV (before Route Discovery)

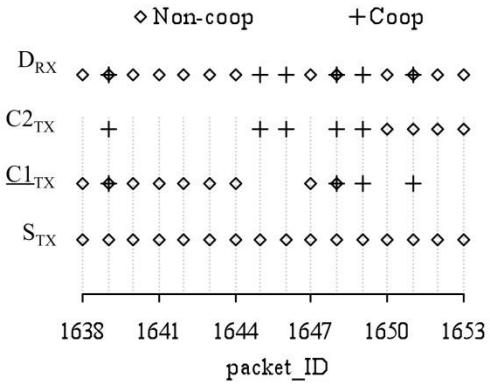

Figure 63. Behavior of Relaying in AODV (Route Discovery)

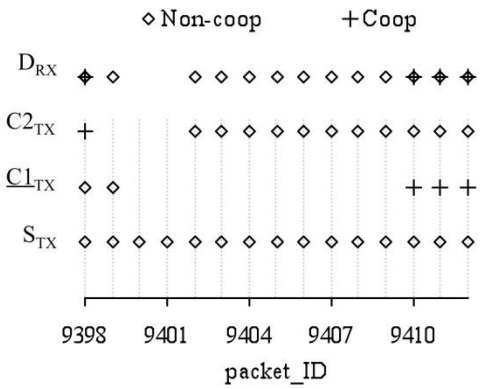

Figure 64. Behavior of ARQ in AODV (Route Discovery)

In Figure 57 we can see the behavior of the Cooperative Forwarding mechanism under LUNAR with a screen-shot of packets 7697 to 7712. In this test node C1 is the intermediate node in the path from S to D, node C2 is one static cooperator node of C1, and C1 is moving towards D, that is, decreasing connectivity with source S. Since LUNAR does not act fast enough to repair the route, we can see how some packets sent by S are correctly received by destination D through C1 but a lot of packets are not received by C1. In the cases where C1 does not receive the packets sent by S, cooperator C2 tries to cooperate in order to forward the packets that C1 has not been able to forward. For example, we can see how packet 7712 reaches node D via C1 using the routing path, but how packet 7700 reaches node D thanks to the cooperator C2, which sends the packet directly to D. In this case, C2 is in the Relaying situation and the hop count of the packets is not increased. Another possible behavior is the case of packet 7707 where the packet is both forwarded by C1 and C2, and then, received by duplicated in node D. This is due to C2 has not received the 802.11 ACK which C1 sends to S, and then C2 cooperates unnecessarily. The last case is the case of packet 7703 where neither C1 nor C2 have received the packet, and so, the network is temporarily disconnected, in which case the Cooperative Forwarding mechanism cannot solve the problem.

In Figure 58 we can see the behavior of the Cooperative Forwarding mechanism under LUNAR but now with node C1 moving towards S. The screenshot corresponds to packets 11361 to 11376 and we can see how some packets sent by S are correctly received by destination D through C1 but how some packets forwarded by C1 cannot reach D. In the cases where D does not receive the packets sent by C1, cooperator C2 tries to cooperate in order to forward the packets that C1 has not been able to forward. For example, we can see how packet 11364 reaches node D via C1 using the routing path, but how packet 11365 reaches node D thanks to the cooperator C2, which retransmits the packet forwarded by C1 to D. In this case, C2 is in the ARQ situation and the hop count of the packets is increased by one. Another possible behavior is the case of packet 11376 where the packet is both forwarded by C1 and retransmitted by C2, and so, received by duplicated in node D. This is due to C2 has not received the 802.11 ACK from D and then it cooperates unnecessarily. The last case is the case of packet 11368 where C1 cannot send the packet to D but moreover C2 also has not received this packet. In this case C2 cannot cooperate and the Cooperative Forwarding mechanism cannot solve the problem.

Similar observations can be seen in Figure 59, Figure 60, Figure 61, Figure 62, Figure 63 and Figure 64. However, depending on the behavior of the routing protocol in front of a link break, the Cooperative Forwarding mechanism performs in a different way. In the case of LUNAR, which does not detect link breaks and only repairs routes every certain time (e.g. 3 seconds), cooperation has a lot of impact in the delivery rate of packets, especially immediately before the route discovery (packets 7895 to 7903 in Figure 59 and packets 11486 to 11493 in Figure 60), when the intermediate node C1 has null connectivity with the source S or the destination D. In the case of AODV with link layer feedback, link breaks are detected and rapidly repaired. For this reason, only few cooperations before the route discovery (only packet 1631 in Figure 61) and during the route discovery (only packets 1645, 1646 and 1649 in Figure 63) can be seen. Nevertheless, link breaks are not always repaired as fast as in this simple topology and the Cooperative Forwarding mechanism not only acts during the route discovery processes.

In Table 8 we can see the full results of the four tests in terms of transmitted and received packets. In the case of Relaying in LUNAR, 1695 packets were transmitted by S while the intermediate node C1 was moving towards D, and node D received a total of 1606 packets for which 141 of them were received due to the Cooperative Forwarding mechanism. In the case of AODV results show how the impact of the Cooperative Forwarding mechanism is less visible but not less important than in LUNAR, as for example in the Relaying situation, the test achieved 0 losses due to cooperation.

Table 8. Results of Two-hop Scenario and controlled mobility Tests

	LUNAR		AODV	
	Relaying	**ARQ**	**Relaying**	**ARQ**
Tx packets	1695	1337	1894	1627
Rx packets	1606	1287	1894	1624
Rx packets without cooperation	1453	1169	1868	1614
Rx duplicated packets	12	19	22	9
Rx packets with cooperation	141	99	4	1
Lost packets	89	50	0	3

15.1.3.2 Multi-hop Scenario and Random Mobility using UDP

In Figure 65 we can see a screenshot of the cooperative experiment for some packets of the flow *A*. For example, between packets 12585 and 12591 the routing path from S to D was composed only by C1, and between packets 12592 and 12599 LUNAR changed the routing path from S to D using the intermediates nodes C2 and C8. In these two cases, we can see how there are packets correctly forwarded by the routing path (e.g. packet 12594), packets that need to be forwarded by cooperator nodes in order to reach D (e.g. packet 12586), packets received by duplicated in D due to useless cooperation (e.g. packet 12595), and packets that cannot reach D (e.g. packet 12589).

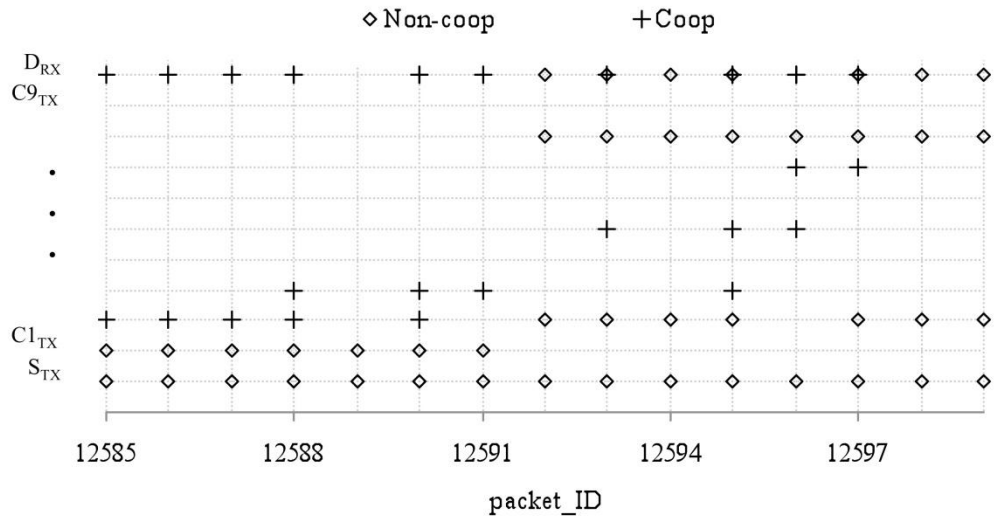

Figure 65. Screenshot of Flow A in the multi-hop experiment

In Table 9 and Table 10 we can see the full results of the test for the four flows and for both the cooperative and non-cooperative experiments. The Cooperative Forwarding mechanism achieves a reduction of packet losses from 12.7% to 5% for flow *A*; from 6.6% to 1.6% for flow *B*; from 8.6% to 0.9% for flow *C*; and from 7.1% to 0.5% for flow *D*. Therefore, results show that, in terms of delivery rate, the Cooperative Forwarding mechanism is a very promising technique to achieve robustness in dense MANETs.

Table 9. Results of Flows A and B in the multi-hop experiment

	Flow A		**Flow B**	
	Coop	**Non-coop**	**Coop**	**Non-coop**
Tx packets	24000	24000	24000	24000
Rx packets	22799 (95%)	20956 (87.3%)	23611 (98.4%)	22420 (93.4%)
Rx packets without cooperation	18369 (76.5%)	20956 (87.3%)	19028 (79.3%)	22420 (93.4%)
Rx duplicated packets	2456 (10.3%)	0	3731 (15.5%)	0
Rx packets with cooperation	1974 (8.2%)	0	852 (3.6%)	0
Lost packets	**1201 (5%)**	**3044 (12.7%)**	**389 (1.6%)**	**1580 (6.6%)**

Table 10. Results of Flows C and D in the multi-hop experiment

	Flow C		**Flow D**	
	Coop	**Non-coop**	**Coop**	**Non-coop**
Tx packets	24000	24000	24000	24000
Rx packets	23787 (99.1%)	21948 (91.4%)	23871 (99.5%)	22303 (92.9%)

Rx packets without cooperation	20835 (86.8%)	21948 (91.4%)	21148 (88.1%)	22303 (92.9%)
Rx duplicated packets	1689 (7%)	0	1775 (7.4%)	0
Rx packets with cooperation	1263 (5.3%)	0	948 (4%)	0
Lost packets	**213 (0.9%)**	**2052 (8.6%)**	**129 (0.5%)**	**1697 (7.1%)**

Another advantage of the Cooperative Forwarding mechanism is its impact in order to reduce the size of consecutive packet losses. Cooperation can act at any time of the experiment, due to fading or shadowing in the wireless links, but especially when mobility of nodes changes the topology of the network and the routing protocol cannot act fast enough to repair the route. Therefore, not only is the Cooperative Forwarding mechanism able to increment the delivery rate of packets, but it is also able to reduce the size of consecutive packet losses. In Table 11 we can see how the average size of consecutive packet losses is reduced from the non-cooperative experiment to the cooperative experiment. In Figure 66 we can compare for flow *A* the distribution of this measure for both cooperative and non-cooperative experiments.

Table 11. Average size of consecutive packet losses for cooperative and non-cooperative experiments

	Flow A	**Flow B**	**Flow C**	**Flow D**
Coop	1,8006	1,752252	1,33125	1,141593
Non-coop	2,368872	2,464899	2,739653	2,41394

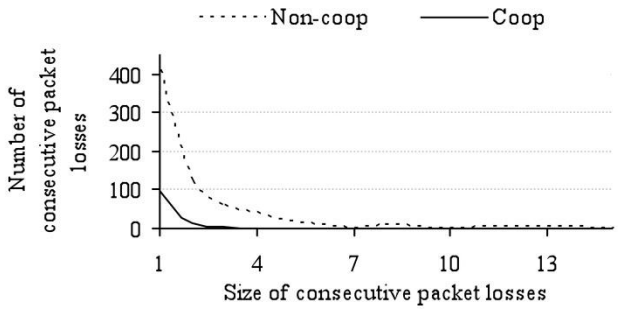

Figure 66. Consecutive packet losses in Flow A for cooperative and non-cooperative experiments

Once the benefits of the Cooperative Forwarding mechanism have been demonstrated, it is necessary to discuss its costs. In Table 12 we can see, for every type of packet, the number of packets transmitted by all nodes of the experiment and the number of bytes of these packets (from the IEEE 802.11 header to the payload of the packet). Instead of comparing the overhead of every flow, we compare the overall overhead produced in executing all the four flows in the cooperative and the non-cooperative experiments. If we divide the total number of bytes transmitted for all the nodes (without including packet retransmissions) by the number of data bytes that the four destination nodes of the test have received, we obtain that the cooperative experiment required 2.02 transmissions for every received data byte to reach its destination, while the non-cooperative experiment required 1.92 transmissions. If we

normalize by dividing the previous values by the average path hop count of the four flows (1.65 and 1.71 for the cooperative and non-cooperative experiments respectively), we obtain that the average number of transmissions per correctly received byte and per hop is 1.21 in the cooperative experiment and 1.11 in the non-cooperative experiment - not a great difference if we bear in mind the huge increment in the delivery rate of packets for the cooperative mechanism. However, this overload may have an impact on congested networks, and while it can probably be decreased with an adequate parameter tuning, this is an aspect requiring further study.

Table 12. Comparison of the overhead in the Multi-hop Scenario

	Coop		Non-coop	
	#packets	**Bytes**	**#packets**	**Bytes**
CHELLO	24217	5658137	0	0
CACK	30200	2325400	0	0
Routing	17207	1829765	18208	1935936
802.11 ACK	166696	6516180	152006	5954526
Non-coop data	155309	169836955	160691	175724893
Coop data	18929	20700563	0	0

Another cost of Cooperative Forwarding to be studied is the level of disorder that cooperation adds to packet reception. While in the non-cooperative experiment there was no disorder of packets, in the cooperative experiment all flows suffered a small number of disordered packets (24, 19, 22 and 17 packets of 24000 packets for flows *A*, *B*, *C* and *D* respectively). However, it is important to bear in mind that all these disordered packets are packets that in the non-cooperative experiment would be lost.

15.1.3.3 Two-hop Scenario and Controlled Mobility using TCP

In Figure 67 we can see the *Throughput over Time* curves achieved by the TCP stream between the client and the server.

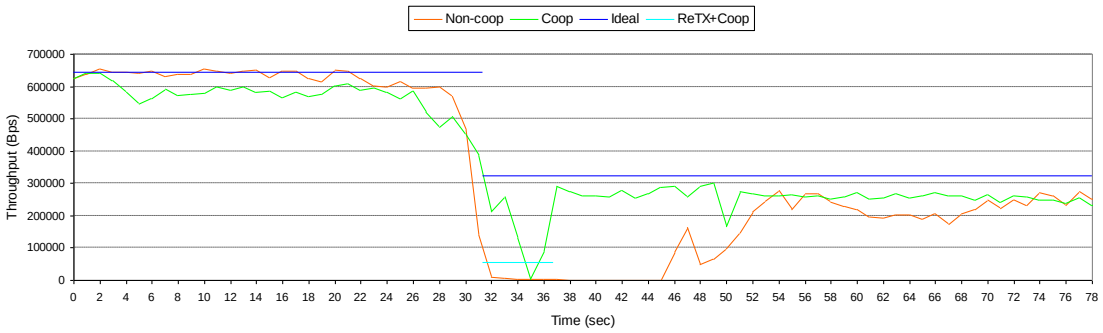

Figure 67. TCP Throughput between client and server in the two-hop experiment

The Ideal curve corresponds to the maximum average throughput of this experiment if we assume perfect wireless channel before and after the link break and perfect detection and repair of the routing path by the routing protocol during the link break, that is, assuming that when the path is valid only one transmission is needed to correctly send packets between links, and when the link break occurs it takes no time to detect it and to repair the route. The average term refers to the value of the backoff counter: at every transmission, the mean value of the corresponding backoff counter is used to calculate the throughput. If we assume these conditions, TCP will not be aware of the link break and will not be affected. The only effect of the link break will be to reduce the throughput due to the increment in the hop count of the path. So, the curve will be a horizontal curve of 644845 Bytes per second during the one hop path and before the link break, and a horizontal curve of 322422 Bytes per second during the two hops path just after the link break and until the end of the experiment. See appendix "A.1 Cooperative Forwarding Calculations" for the calculations.

On the other hand, the ReTX+Coop curve corresponds to the maximum hardware-limited average throughput of our prototype when cooperation takes place, that is, mainly during the link break period. Due to limitations with the available hardware, cooperator nodes cannot interrupt the retransmission algorithm of the transmitter node. So, if they have to cooperate, they will wait until all retransmissions are done. In other words, if there is a link break between two adjacent nodes, before the route is repaired, cooperators will cooperate in order to mitigate the problems of the link break but with the unavoidable cost of having to wait to all the retransmissions of the transmitter for every packet. So, our prototype is not able to achieve the Ideal curve, even if the channel is perfect all the time except during the link break. However, it is important to bear in mind that these limitations are due to the fault of available hardware and not of the Cooperative Forwarding mechanism. With the correct hardware, the Cooperative Forwarding mechanism could achieve a throughput curve very close to the Ideal one. The value of this curve is 56083 Bytes per second. See appendix "A.1 Cooperative Forwarding Calculations" for the calculations.

Once the meaning of these curves is explained, we can now understand and compare the Cooperative and Non-cooperative curves. On the one hand, if we focus on the Non-cooperative curve we can see how, faced with the link break, the traditional forwarding mechanism (and under AODV in this case, although similar results would be obtained for any other existing routing protocol for MANETs) performs poorly and the TCP stream is enormously affected, causing a decrement of the throughput to 0 Bytes per second during 11 seconds approximately.

On the other hand, if we focus on the Cooperative curve we can see how, thanks to the Cooperative Forwarding mechanism, not only the TCP stream is not enough affected to cause a decrement of the throughput to 0 Bytes per second but also it is reestablished sooner than in the Non-cooperative case. So, the impact of the link break in the TCP traffic is greatly reduced thanks to the Cooperative Forwarding mechanism.

However, as we said before, the Cooperative curve is not always equal or close to the Ideal curve. The only reason for this is due to hardware limitations of the implementation of our prototype. If we look at the time period between second 31 and second 37 in Figure 67, we

will see that the ReTX+Coop curve explained before (the maximum hardware-limited average throughput that can be achieved by our prototype during cooperation) coincides with the time period in which the Cooperative curve is far away from the Ideal curve. This is the reason that causes the decrement of the throughput of the Cooperative case below the throughput of the Ideal case. In other words, the Cooperative Forwarding mechanism would be able to achieve the throughput over time curve close to the Ideal one if we could use more specific hardware. Note that just before the link break, the Cooperative curve (and the Non-cooperative one) is also far away from the Ideal curve. This is because the Ideal curve assumes that all packets are correctly sent in the first transmission until the link break occurs. In the experiment, before the link break, the wireless channel is not as bad as to cause a link break but more than one retransmission is needed to send packets through this link. So, both Cooperative and Non-cooperative curves will start to go far away from the ideal curve some seconds before the link break occurs.

In Figure 68 and Figure 69 we can see not only the throughput over time curve but also the number of retransmissions of every packet over time, for both the Cooperative and Non-cooperative cases respectively. It is important to note that the number of retransmissions plotted in these figures corresponds, before the link break, to the number of retransmissions used by the TCP client to send every packet to the TCP server, and after the link break, to the number of retransmissions used by the TCP client to send every packet to the intermediate node in the path between the TCP client and the TCP server. In Figure 68 we can see how the number of retransmissions per packet starts to grow up before the link break. When the link break occurs, the maximum number of retransmissions is used and none of them is useful. So, the throughput of the TCP stream starts to decrement to 0 Bytes per second. When the link break is detected and the route is repaired, the TCP stream is reestablished and we can see again the number of retransmissions needed to send every packet from the TCP client to the intermediate node of the route. In Figure 69 we can see a similar behavior to the one of Figure Figure 68. However, the most important fact in Figure 69 is that we can see that even during the cooperation period, and as explained before due to hardware limitations, all retransmissions are done. Since cooperators are not able to stop the retransmission algorithm of the transmitter (in this case, the TCP client), the wireless channel will be useless used and the throughput will be decremented, even with the cooperative actions of the cooperators. Nevertheless, the throughput will not reach 0 Bytes per second thanks to the Cooperative Forwarding mechanism.

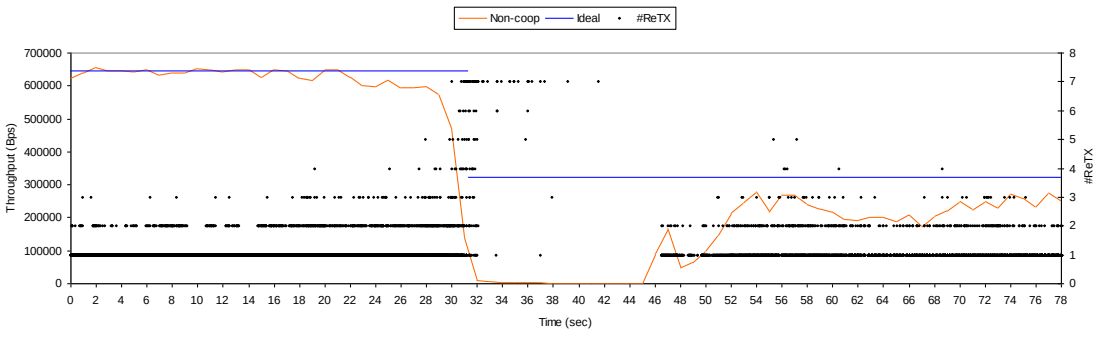

Figure 68. Number of Retransmissions in the Non-cooperative case

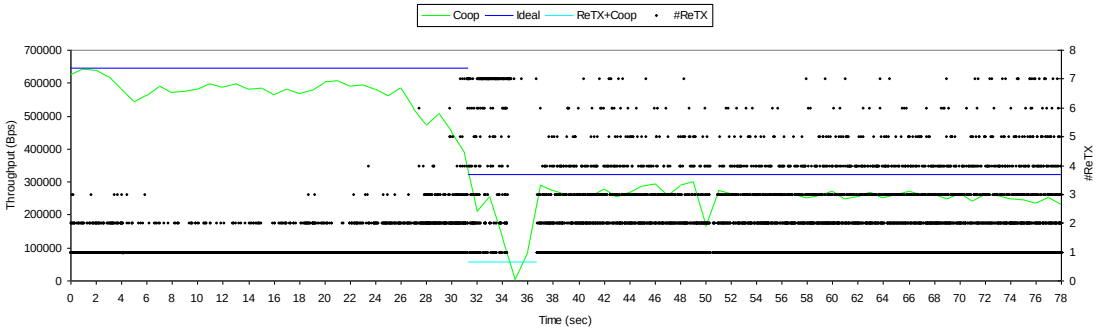

Figure 69. Number of Retransmissions in the Cooperative Case

In order to analyze in detail every curve and to be able to compare both curves each other, we will explain in detail the behavior of the TCP stream and of AODV in the experiment.

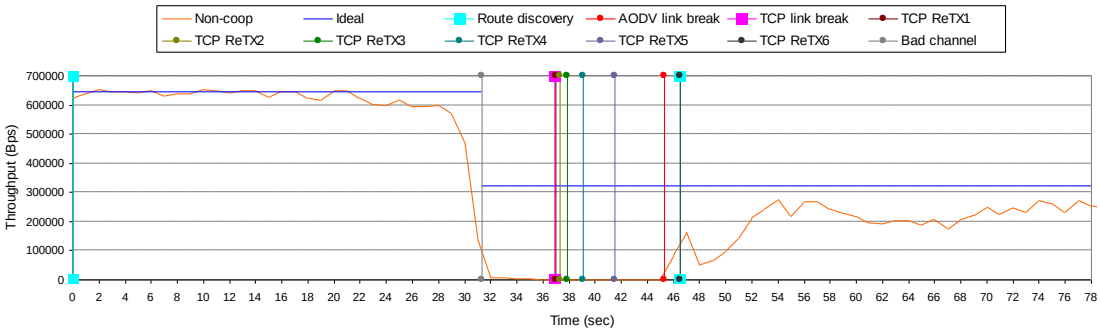

Figure 70. Milestones of the Non-cooperative Case

In Figure 70 we can see the Non-cooperative curve with some special marks or milestones:

- Route discovery: time when AODV starts a route discovery process.
- AODV link break: time when AODV decides that there is a link break, that is, when one of two neighboring nodes of a path has not received any HELLO message from its neighbor during more than “allowed_hello_loss·hello_interval” milliseconds.
- TCP link break: time when the link is not as bad as to cause an AODV link break, but is as bad as to cause enough losses to the TCP stream and to cause the corresponding “death” of TCP. Since TCP timers were designed for wired networks, the instability of the wireless channel has a great impact on the performance of TCP [**31**]. We will see that TCP usually dies before AODV detects the “real” link break, mainly due to delays and losses of packets that TCP understands as problems of congestion and not as problems of the wireless links as they are.
- TCP $ReTX_i$: Once the TCP link break has occurred, the TCP client starts to retransmit the last TCP packet, each time doubling the retransmission timeout.
- Bad link: time when, due to server mobility, the link between the client and the server starts to be of bad quality.

Concerning the Route discovery marks of Figure 70, we can see how there is one Route discovery process at time 0, that is, when the client starts the TCP connection, and another once the AODV link break has occurred. This is the Route discovery that will rediscover the two hops path between the client and the server, and in this case, through the laptop located at

point C of Figure 55. As we can see in Figure 70, it is important to note that the second Route discovery process does not occur immediately after the AODV link break. Instead, it occurs when, once the AODV link break has been detected, the node that has detected the link break (which, in this case is the client of the TCP stream but it could be also the server due to the two-way TCP traffic) has data to send. This is due to the reactive properties of the AODV protocol, which only discovers routes when they are needed.

The most important milestones of Figure 70 are the AODV and TCP link breaks. These are the points that will determine how much time TCP will rest stalled, that is, with throughput 0. As we said before, due to the instability of the wireless link, TCP will die before AODV detects the link break. The time difference between these two link breaks is what will determine the performance of the TCP stream. If we bear in mind that once the TCP stream has died it starts to retransmit a packet doubling the timeout at every retransmission, we can see how, in few seconds, the TCP retransmission timeout will become huge, and then, although the AODV link break would be detected, the TCP stream will not be reestablished until the big retransmission timeout expires. In summary, the more time difference between the two link breaks, the more time TCP will rest stalled. If we focus in the TCP retransmissions of Figure 70, we can see how the first retransmission is at time 36.980543 seconds. The second retransmission is at time 37.282084 seconds (~0.3 seconds after the last one), the third retransmission is at time 37.890121 seconds (~0.6 seconds after the last one), the fourth retransmission is at time 39.106202 seconds (~1.2 seconds after the last one), the fifth retransmission is at time 41.538358 seconds (~2.4 seconds after the last one), and the last retransmission (since the AODV link break has already been detected) is at time 46.505009 seconds (~4.8 seconds after the last one). After this retransmission, AODV will rediscover the route and the TCP stream will be reestablished.

In order to solve this problem, one solution is to tune the allowed_hello_loss and hello_interval parameters of AODV to reduce the time difference between the two link breaks, but there is always a tradeoff between reducing the time difference between the two link breaks and the reliability of the AODV link break: greater hello_interval will be better to reduce the time difference between the two link breaks but with the problem of incrementing the probability of detecting fake link breaks which will also affect the performance of TCP. Smaller hello_interval will reduce the probability of detecting fake link breaks but with the problem of the possibility of detecting the AODV link break a long time ago after the TCP link break. Nevertheless, we believe that parameter tuning is not the way to achieve good performance of TCP, since it is limited to the routing protocol in question. We believe that the solution cannot be fully achieved by the routing protocol, since it will never be able to react fast enough to mitigate the problems of the link break. We believe that the solution is to avoid the TCP link break, that is, to reduce the time difference between the two link breaks to 0. And as we will see in Figure 71, this can be achieved with the Cooperative Forwarding mechanism.

Note that in Figure 70 and after the AODV link break, the Non-cooperative curve starts to grow up before AODV performs the Route discovery. This behavior could be misunderstood by thinking that the TCP client starts to transmit data before the Route discovery. Of course, it is impossible, and the reason of this early increment is that the throughput over time curves are

plotted taking an interval of one second to calculate the throughput for each value of the x-axis.

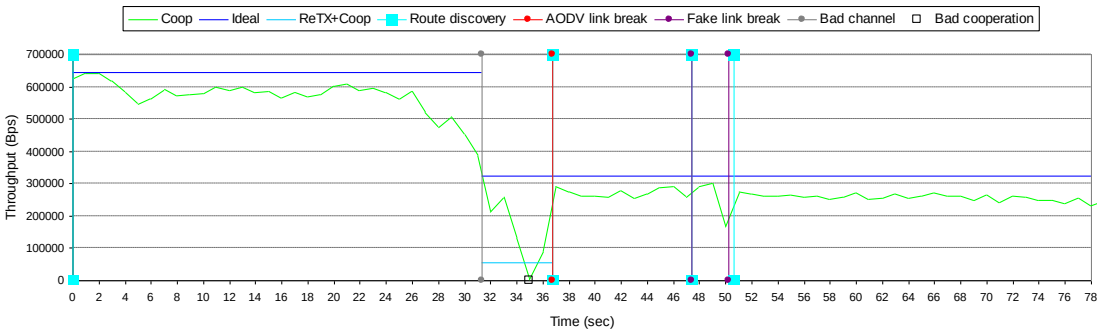

Figure 71. Milestones of the Cooperative Case

In Figure 71 we can see the Cooperative curve with some special milestones:

- Route discovery, AODV link break and Bad channel marks have the same meaning than in the Non-cooperative curve of Figure 70.
- Fake link break: it is an AODV link break which occurs when the link is not as bad as to be considered as a link break. Due to the instability of the wireless channel and the broadcast properties of the hello packets (i.e., no retransmission algorithm for broadcast packets), it is possible to loss all hello packets during more than "allowed_hello_loss·hello_interval" milliseconds and without enough losses of TCP in order to cause a TCP link break. We especially realized this behavior in the intermediate node of the two hops path. Since due to hardware-limitations we cannot use the RTS/CTS mechanism in the prototype, this intermediate node is a potential node to suffer the problems of the hidden node problem. Moreover, if we bear in mind that hello packets are broadcast messages and so, they are not retransmitted, it is possible that some fake link breaks can occur over time.
- Bad cooperation: due to hardware-limitations of our prototype, sometimes cooperation is needed and does not take place. As we will see, the problem exists when the PPA timeout is not well adjusted to its real value. In these cases, the heuristics of the cooperators will be set to erroneous values and cooperators will give up cooperating erroneously. It is important to bear in mind that these limitations are due to the fault of available hardware and not due to the Cooperative Forwarding mechanism itself.

Concerning the Route discovery marks of Figure 71, we can see how there is one Route discovery process at time 0, that is, when the client starts the TCP connection and another once the AODV link break has occurred. This is the Route discovery which will rediscover the two hops path between the client and the server, and in this case, through the laptop located at point D of Figure 55. Unlike the same Route discovery of the Non-cooperative case (Figure 70), it is important to note that in the Cooperative case it occurs immediately after the AODV link break. In fact, it occurs exactly for the same reason than in the Non-cooperative case, that is when, once the AODV link break has been detected, the node that has detected the link break (which, in this case is the server of the TCP stream but it could be also the client due to the two-way TCP traffic) has data to send. The difference is that in the Non-cooperative case the TCP stream is dead and there are no data ready to be sent, and in the Cooperative case,

the TCP stream is not dead and data can be sent immediately. In fact, as we said before, with the Cooperative Forwarding mechanism TCP packets can pass through the link, even when the channel is bad and the link break has not already been detected. Thanks to cooperation, TCP will not die during the link break, the TCP retransmission timeouts will not increment, and so, when AODV detects the link break TCP will be reestablished immediately.

Two other Route discoveries can be seen in Figure 71. They are because of the fake link breaks we explained before. The effect of these fake link breaks in the TCP stream will depend on the TCP state at the moment of the link break. If the intermediate node detects this fake link break, it will send RERR messages to the source of the TCP stream in order to notify the source that it needs to rediscover the route. Moreover, the intermediate node will discard all the packets that will arrive through the broken path. So, if the transmitter node has a lot of TCP packets waiting in the transmission buffer to be sent at the moment of the fake link break, the intermediate node will have to discard a lot of packets which will affect a lot the behavior of the TCP stream. In the first fake link break, 11 packets were discarded. In the second one, 15 were discarded. That is why the second fake link break is more visible in the throughput over time curve. However, this type of link break could occur not only in the Cooperative case but also in the Non-cooperative case.

The last point in Figure 71 we want to comment is the bad cooperation mark. As we said, this is also fault of the hardware-limitations of the prototype and not of the Cooperative Forwarding mechanism. In our implementation, cooperator nodes cooperate when they can cooperate, that is when they have the packet to cooperate and they have to cooperate. In other words, when the master node of the routing path is not able to forward the packet, and when they are good cooperators and they can deliver the packet to the next hop node. To know if they are good cooperators, they use heuristics based on the SNR with the next hop node and based on the correct reception of Cooperative Acknowledgment packets (CACK) after every cooperative action. Cooperators, in order to decide if they have received or not these CACKs, use the PPA timeout. If PPA expires and the corresponding CACK has not been received, they will consider that their cooperative action has failed, and then they will penalize their heuristics in order to let other possible better cooperators to cooperate. In the experiment, when the link break occurs, only the nodes located in the points C and D of Figure 55 are possible good cooperators. Nodes of points A and B do not fulfill the initial SNR requirements to become good cooperators of the link between the client and the server. Moreover, due to our heuristics, cooperator C has priority over cooperator D. For this reason, during the link break, cooperator C starts to cooperate with the corresponding decrement of the throughput due to hardware-limitations as we explained before. The problem occurs when, due to congestion in the transmission buffers of the node which has to send the corresponding CACKs, these CACKs are delayed a lot. Since the PPA timeout is not well adjusted in these cases, cooperator C will think that the CACKs it has to receive due to its cooperations will not be received, and so, it will penalize itself as a bad cooperator. In the experiment, this happened with three TCP cooperative packets that cooperator C sent to the server at times 34.491925, 34.509195 and 34.526948 seconds. Cooperator C received the three corresponding CACK packets with delay and then it gave up its cooperative action. At this point, cooperator D came into the scene. It became the most priority cooperator, and then it started to cooperate instead of cooperator C. The problem was that, due to instability in the

wireless channel, cooperator D sent to the server two TCP cooperative packets at times 34.554410 and 34.563647 seconds and it did not receive any of the two corresponding CACKs. At this moment, all possible good cooperators gave up their cooperative actions although they were still good cooperators. This is the reason why, in the bad cooperation mark, the throughput of the curve of Figure 11 seams to reach 0 Bytes per second (in fact, 3255 Bytes per second). But as we said before, this behavior is due to hardware-limitations and not to the Cooperative Forwarding mechanism itself.

Once the benefits of the Cooperative Forwarding mechanism in front of the non-cooperative or traditional forwarding mechanism have been demonstrated in this experiment, note that although both cases cannot be compared strictly, for example because the AODV link break in the cooperative case occurs at time 36.682115 seconds and in the non-cooperative case at time 45.311218 seconds, what it is important is the behavior of TCP in front of the link break when the Cooperative Forwarding mechanism is used or not. In short, the Cooperative Forwarding mechanism is able to avoid that TCP stalls.

Before finishing the analysis of the results, two other differences between the cooperative and non-cooperative curves of Figure 67 have to be explained. On the one hand, we can see how, when the TCP stream is reestablished after the link break, the cooperative curve achieves higher throughput than the non-cooperative curve. The reason is that after the link break, in the non-cooperative case the route is reestablished through the intermediate node located at point C of Figure 55, and in the cooperative case the route is reestablished through the intermediate node of point D of Figure 55. So, after the link break, two different paths are used for both cases and they cannot be strictly compared. Another justification to this behavior can be seen if we compare Figure 68 and Figure 69. In these figures we can see how the number of retransmissions used to send the packets between the TCP client and the corresponding intermediate node is totally different between the two cases. On the other hand, we can see also that from time 0 until the link break the cooperative curve achieves slightly lower throughput than the non-cooperative curve. As we saw in Figure 56, cooperative laptops were located over the non-cooperative laptops. The slightly space between the wireless cards of each laptop implies that both curves cannot be strictly compared and could cause this difference between the two curves.

However, one last important aspect which could explain these differences between both curves and that has to be studied is the overhead of the cooperative case in front of the non-cooperative case. Cooperation achieves better performance of TCP, but it has a cost. Our Cooperative Forwarding mechanism has cooperative acknowledgments (CACKs) and cooperative hello (CHELLO) packets, as well as useless or duplicated cooperative packets, which can let to an increment on the network usage.

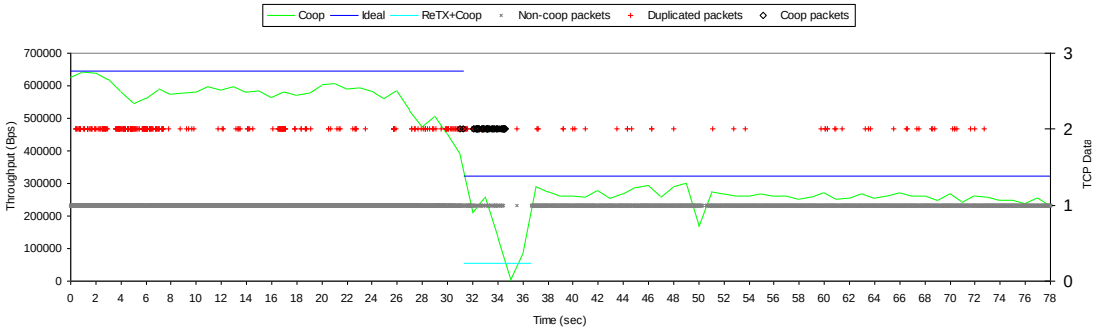

Figure 72. Cooperative Nature of the TCP Data Packets

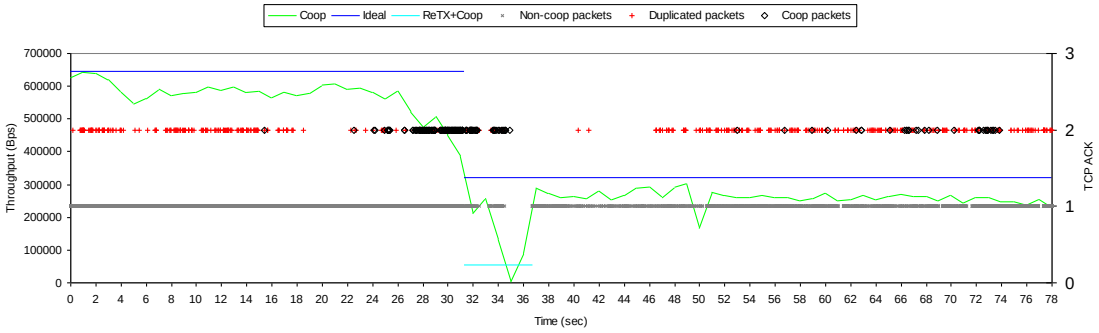

Figure 73. Cooperative Nature of the TCP ACK Packets

In Figure 72 we can see the throughput of the cooperative curve over time in conjunction with the cooperative nature of every TCP data packet over time (secondary y-axis). In Figure 73 we can see the same curves but for the TCP ACK packets, that is, for the other direction of the TCP traffic.

One packet can be delivered to the destination in three ways:

1. Non-coop packets: the packets that can reach the destination without the need of cooperation. The traditional forwarding mechanism is enough to deliver the packets.
2. Coop packets: the packets that have reached the destination thanks to the cooperative action of some cooperators. Without the Cooperative Forwarding mechanism these packets would not be delivered.
3. Duplicated packets: the packets that reach the destination by duplicated, that is, due to both the traditional forwarding mechanism and one useless cooperator, or due to several useless cooperators. Useless cooperators can appear when due to node mobility or the instability of the wireless channel they receive the data packet but not the corresponding acknowledgment, even when this acknowledgment has been sent. They believe that they have to cooperate, but this cooperative action is going to be useless.

In Figure 72 and Figure 73, we can see how the time period in which there are most of the useful cooperations (Coop packets) is during the link break period. However, several useless cooperations (Duplicated packets) can be seen along all the experiment. In order to take into account this extra overhead, we will calculate the efficiency ratio between the cooperative and

the non-cooperative cases. To do so, in the non-cooperative case we will divide the total amount of TCP data the TCP server has received by the total amount of TCP data and TCP ACKs that all nodes of the experiment have sent. In the cooperative case we will divide the total amount of TCP data the TCP server has received by the total amount of TCP data, Cooperative TCP data, TCP ACKs, Cooperative TCP ACKs, Cooperative ACKs and Cooperative Hellos that all nodes of the experiment have sent. If we count all values in Bytes we have:

$$Efficiency_{\ Non-coop} = \frac{TCPDATA_{ServerRX}}{(TCPDATA + TCPACK)_{AllNodesTX}} =$$

$$= \frac{24329896_{Server}}{24338364_{Client} + 704052_{Server} + 0_A + 0_B + 6214172_C + 0_D} = 0.77839$$

$$Efficiency_{\ Coop} =$$

$$= \frac{TCPDATA_{ServerRX}}{(TCPDATA + TCPDATA_{Coop} + TCPACK + TCPACK_{Coop} + CACK + CHELLO)_{AllNodesTX}} =$$

$$= \frac{26638736_{Server}}{26780026_{Client} + 854095_{Server} + 237454_A + 137148_B + 582814_C + 10252029_D} = 0.68579$$

Then,

$$Efficiency_{\ Ratio} = \frac{Efficiency_{\ Coop}}{Efficiency_{\ Non-coop}} = 0.881$$

The cooperative case was 11.9% less efficient than the non-cooperative case in terms of overhead. However, from the point of view of TCP and the upper-layer applications, this lower efficiency in terms of overheard is not nearly as important as the behavior of TCP as regards the link break, which is extremely more robust in the cooperative case.

15.1.4 Conclusions

Although our experiments have been done with AODV or LUNAR as routing protocol and that the link break detection and route repair (for the case of AODV) are strictly related with the routing protocol, we cannot forget that the Cooperative Forwarding mechanism is a mechanism which fits with any type of routing protocol. In other words, it is independent of the routing protocol.

Another important aspect of AODV is the availability or not of link layer feedback to detect link breaks. Since available hardware does not allow manage with this feature, we only used the hello message mechanism to detect link breaks. Of course, the results would be different if AODV had used link layer feedback. However, as before, the Cooperative Forwarding mechanism would continue being a technique improving the robustness of the network.

Since we adopted the idea of experimental work, we have used available hardware and standard and default parameters for most of the parts involved in the experiment. TCP tuning

is a part which could affect a lot the results of the experiment. If a TCP adapted to wireless networks had been used, better performance of the non-cooperative curve had been achieved. However, even with a tuned TCP which is able to manage link breaks without enter into a death state, the only way to not have to stop the data stream before detecting the link break when the link is bad and while repairing the route is with the Cooperative Forwarding mechanism, since it exploits the path diversity of possible cooperator nodes located in the vicinity of the two nodes affected by the link break.

The other aspects which will affect the results of the experiments are the hardware-limitations. There are hardware-limitations that without them the performance of the prototype would be improved. For example, as we explained, if cooperators had been able to stop the retransmission algorithm of the transmitter node, the cooperative curve would be near to the ideal one. Moreover, we believe that using a routing protocol specially designed bearing in mind the Cooperative Forwarding mechanism would improve a lot the results. Other limitations of our prototype are the lack of a rate adaptation algorithm in the MAC layer (remember that we used 7 retransmissions all at 6 Mbps) and the lack of the RTS/CTS mechanism in order to reduce collisions due to hidden node problems. However, we believe that all these limitations would affect the magnitude but not the conclusions of the results.

In summary, we have seen that path-robustness is highly improved, leading to increase performance in both cases of TCP and UDP traffic. The results show that in dense MANETs with pedestrian mobility, the Cooperative Forwarding mechanism is a promising technique to achieve such robustness, while keeping a low cost of overhead. All in all, we believe that the results presented provide interesting insights into the potential of cooperation in MANETs.

Cooperative Forwarding is an example of how some of the basic assumptions and mechanisms used in wired or infrastructure networks must be fundamentally modified when dealing with MANETs.

15.2 MStack

Currently, the MStack is under evaluation and every day new and better results are being obtained. A complete evaluation of the stack in all its layers in a short period of time should be only possible at a huge cost. For this reason, in this section we will only show a reduced set of experiments that both prove each of the layers of the stack and show the benefits of the stack versus the TCP/IP stack. A more complex and exhaustive evaluation of the MStack is out of the scope of this document.

15.2.1 Prototype

To test the efficiency and the improvements in the robustness of a MANET of the MStack we implemented two prototypes of the MStack, one using Symbian [**46**] (for Nokia smartphones) and another using Android [**47**] (for Google smartphones).

15.2.2 First Insights

In order to give the first insights on the performance of the MStack, we executed a set of simple experiments in four controlled scenarios.

Nokia N78 smartphones were used in these experiments. Although these devices are 802.11g enabled, we realized that they were able to achieve only ~8Mbps at maximum due to CPU limitations. Moreover, the Nokia N78 devices were slightly affected during the experiments by the log files used to record the experiments.

During the experiments, PTP was tuned in order to achieve certain limited data rates.

15.2.2.1 Scenario 1

This is the simplest scenario. Two Nokia N78 smartphones at one hop of distance and with perfect link between each other were used; see Figure 74. The test was realized in the university campus of the UPC, with a lot of Wi-Fi Access Points sharing the wireless channel.

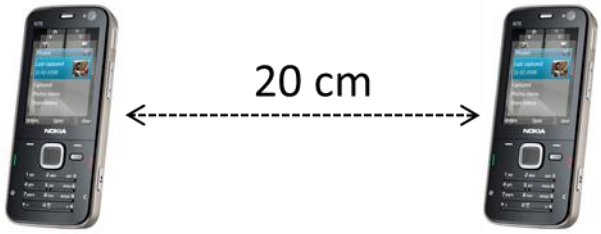

Figure 74. Scenario 1

In Figure 75 we can see the throughput achieved by PTP during the file transfer of 4MB. PTP was tuned to ~1.5Mbps. The variability in the throughput can be explained due to the channel conditions and to the internal use of the CPU of the smartphones.

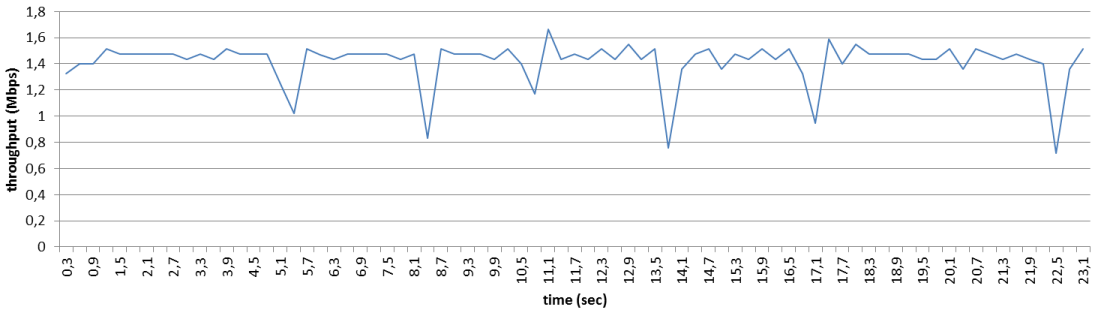

Figure 75. Throughput in Scenario 1

In Figure 76 we can see the behavior of PTP for the packets #0 to #50 from the point of view of the node receiving the file. From #13 to #25, PTP reduces (i.e., more slope in the curve) the data rate by sending request messages without resetting the data rate due to the fact that not enough data packets are received compared to the expected data packets that should be received.

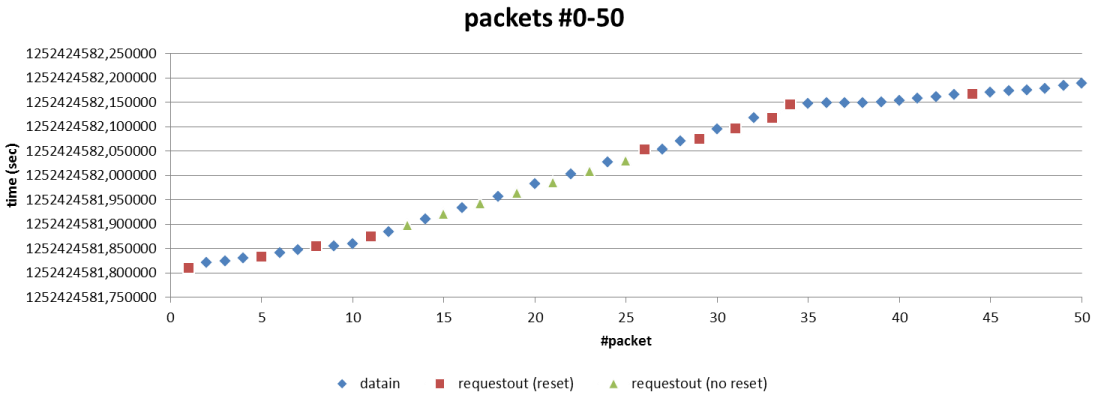

Figure 76. Packets #0 to #50 for the Receiver Node

In Figure 77 we can see the behavior of PTP for the packets #0 to #50 from the point of view of the node serving the file. From #9 to 12, PTP is not able to send the desired data packets, and then the node receiving the file acts as explained in the Figure 76, that is, reducing the data rate (i.e., more slope in the curve) and only reestablishing the data rate when the packet loses disappear (packet #26).

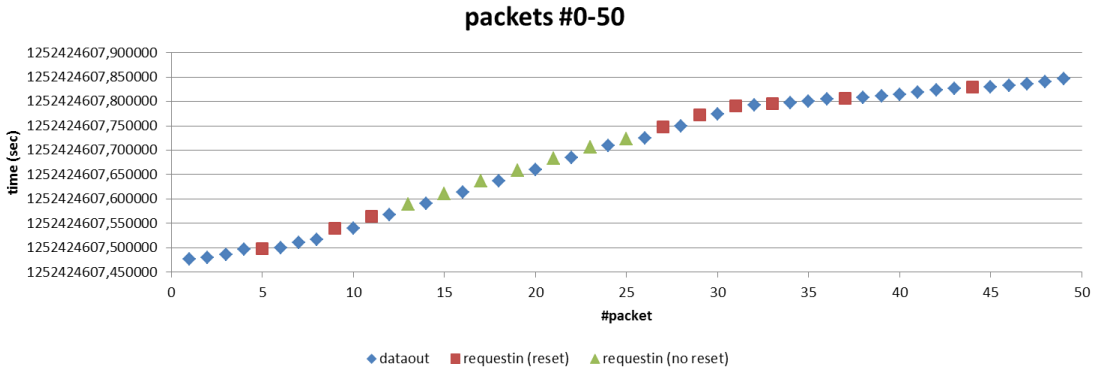

Figure 77. Packets #0 to #50 for the Sender Node

In Figure 78 we can see the behavior of PTP for the packets #100 to #150 from the point of view of the node receiving the file. No packet loses are detected and so, PTP keeps resetting the data rate to its maximum each time a request message is sent. However, PTP is not able to achieve its perfect behavior between data packets (we can see a certain level of variability in the slope of the datain packets). UDP buffers, NIC buffers, and wireless channel conditions impact on the behavior of PTP in the receiver side.

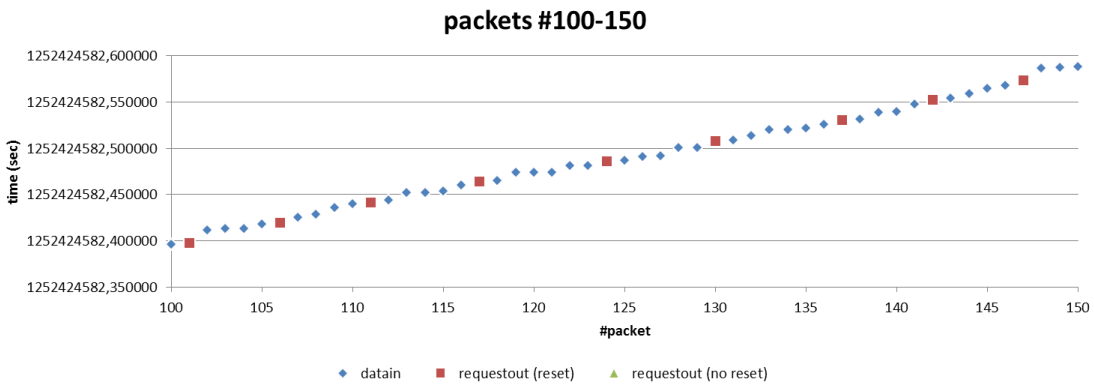

Figure 78. Packets #100 to #150 for the Receiver Node

In Figure 79 we can see the behavior of PTP for the packets #100 to #150 from the point of view of the node serving the file. Unlike the case of the node receiving the file (Figure 78), now PTP is able to achieve its perfect behavior between data packets (we can see an exponential slope between the dataout packets). In this case, no UDP buffers, no NIC buffers, and no wireless channel conditions impact on the behavior of PTP in the server side. Only CPU limitations could affect the behavior of PTP.

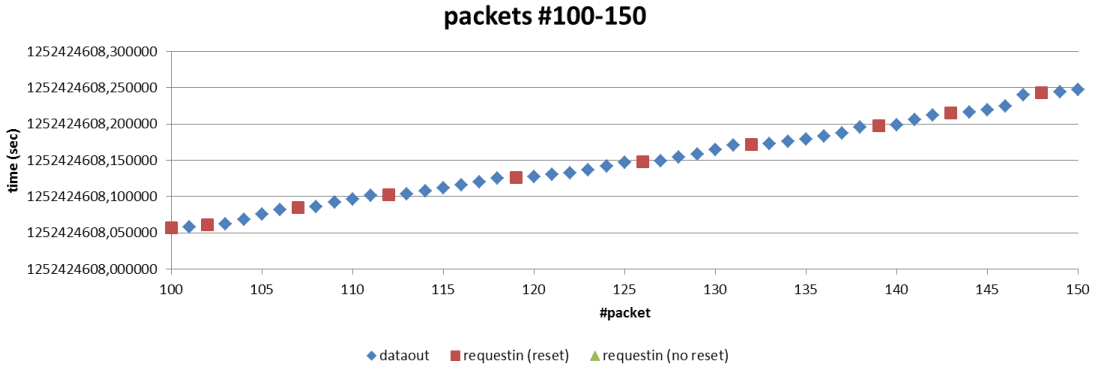

Figure 79. Packets #100 to #150 for the Sender Node

15.2.2.2 Scenario 2

Three Nokia N78 smartphones at one hop of distance and with perfect link between each other were used; see Figure 80. The test was realized in the university campus of the UPC, with a lot of Wi-Fi Access Points sharing the wireless channel.

In this experiment, at time ~3 sec, the originator of the information ran out of battery. A third node containing the same information was able to serve it without breaking the file transfer, thus exploiting the diversity properties of DELTOYA.

Figure 80. Scenario 2

In Figure 81 we can see the throughput achieved by PTP during the file transfer of 4MB. PTP was tuned to ~1.5Mbps.

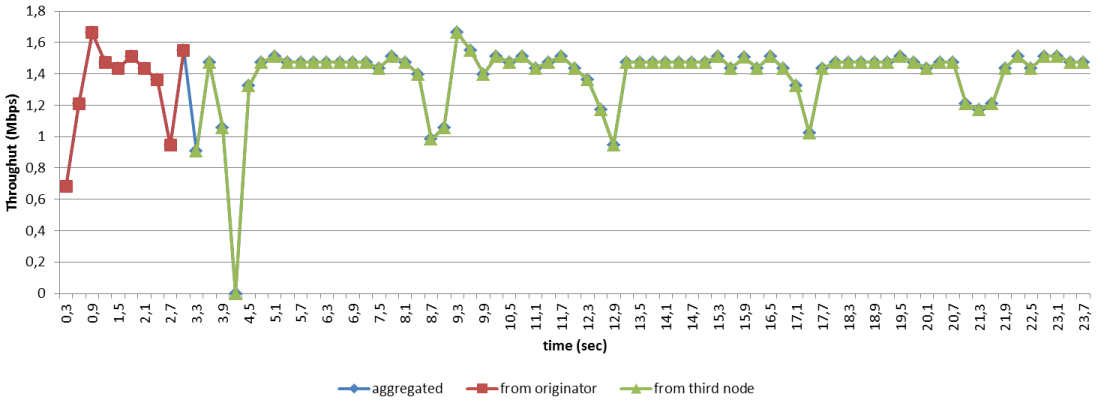

Figure 81. Throughput in Scenario 2

15.2.2.3 Scenario 3

Scenario 3 has the same properties than scenario 2 but with the difference that in scenario 3 we added mobility in the experiment. In Figure 82 we can see the map of a 40m x 20m building of the university campus of the UPC. The arrow together with the four time instants mark the mobility pattern of the experiment. The experiment consisted on sending a file between the originator node and the mobile receiver node while the mobile node moves around the building. A third node containing the same data that the mobile receiver is trying to get from the originator was placed in the other side of the building, in order to help the mobile receiver to receive the data.

At time ~10s, the originator of the information starts to have bad coverage with the receiver of the information, but as we can see in Figure 83 PTP is still able to maintain the file transfer. At time ~21s, the originator of the information becomes out of coverage and the file transfer is interrupted. However, at time ~26s, the third node containing the information is able to serve the information as soon as possible when a valid link between the third node and the mobile receiver appears, this exploiting the DELTOYA capabilities.

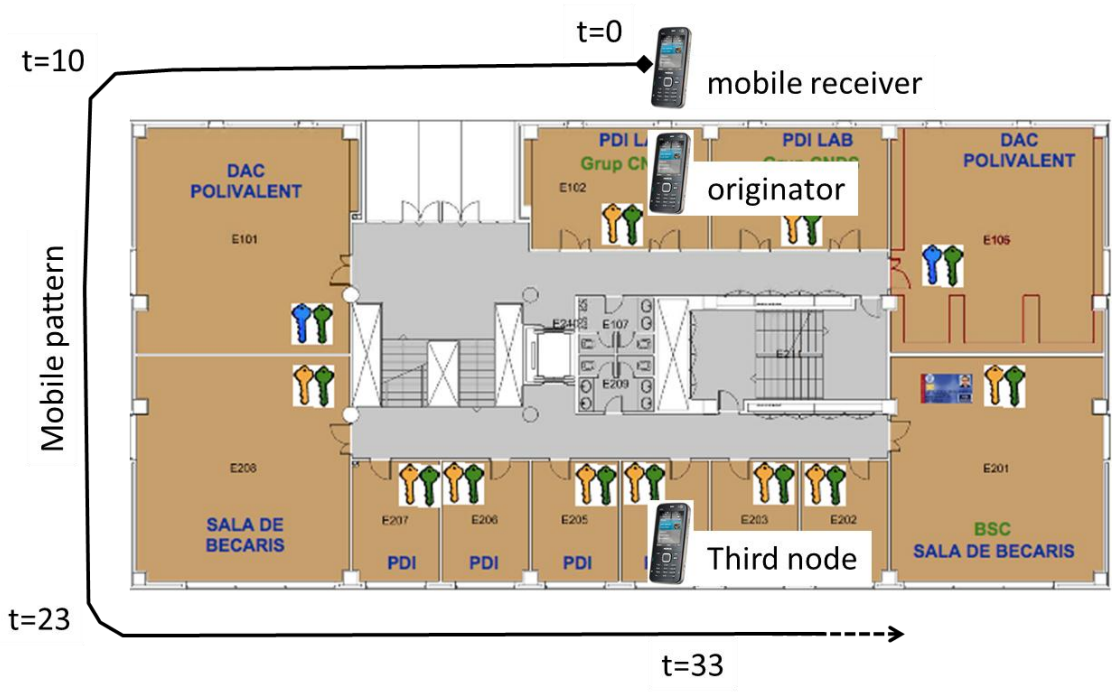

Figure 82. Scenario 3

In Figure 83 we can see the throughput achieved by PTP during a file transfer of 4MB. PTP was tuned to ~1.5Mbps.

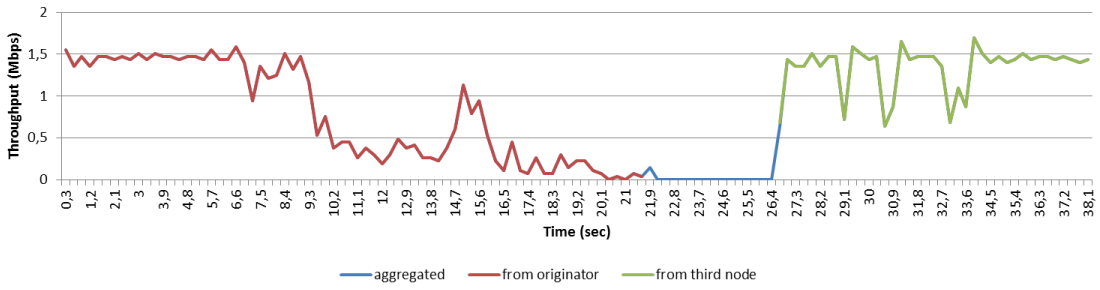

Figure 83. Throughput in Scenario 3

15.2.2.4 Scenario 4

In the scenario 4, placed at the same location of the scenario 3, we are trying to compare at high level the behavior of PTP versus TCP. We placed two originator nodes in the building and two other mobile receiver nodes were moved along the mobility pattern; see Figure 84. For the PTP case, two Nokia N78 smartphones were used. For the TCP case, two Toshiba laptops were used. The file transfer between the two smartphones was the same than the file transfer between the two laptops, meaning that the same file and data rate limits were used.

As we can see in Figure 84, the time instants have more than one value. That is because in order to finish the file transfer, we had to walk along the building more than one lap. The i-th value of the time instant corresponds to the time instant in that place for the i-th lap.

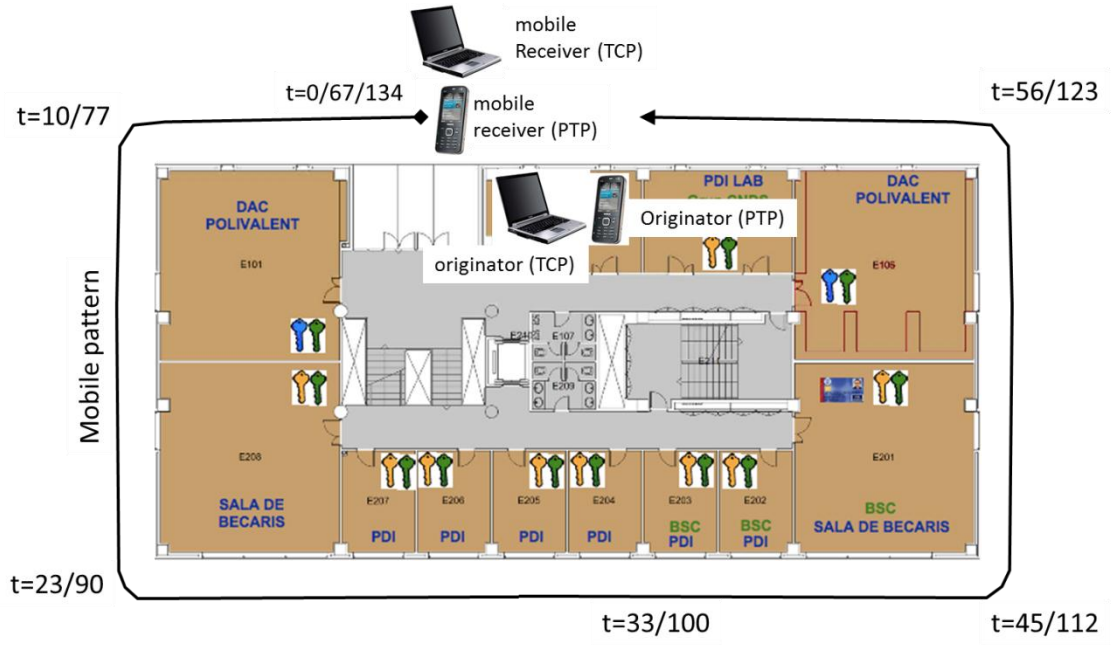

Figure 84. Scenario 4

During the experiment, two laps to the building were done. In Figure 85 we can see how during the first lap PTP was able to finish the file transfer (between 0s and 20s, and between 63s and 64s). On the other hand, TCP not only was not able to finish the file transfer during the first lap but also we had to wait 46 seconds (between 134s and 180s) stopped at the end of the second lap (i.e., near the originator node) in order to let TCP to restart the data flow.

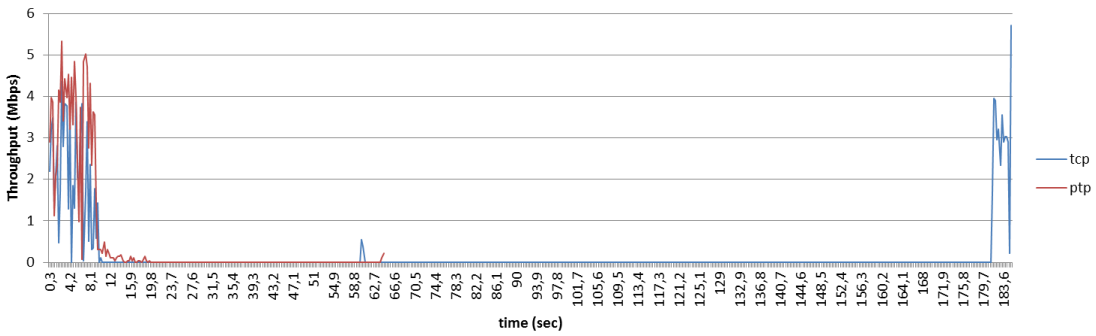

Figure 85. Throughput in Scenario 4

Moreover, as we can see in Figure 86 and in Figure 87, when the link starts to become bad (i.e., entering a grey area) and TCP dies due to timeout degeneration, PTP is still able to continue with the file transfer until the link really becomes break.

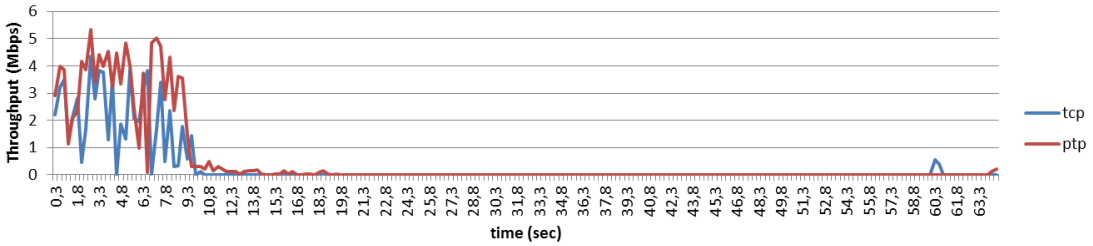

Figure 86. Grey Area in Scenario 4

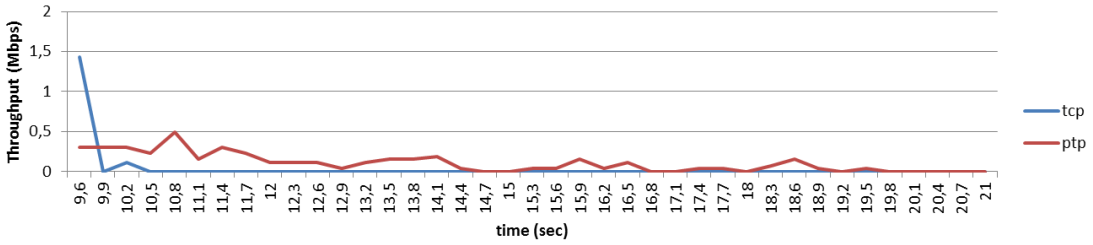

Figure 87. Zooming the Grey Area of Scenario 4

15.2.2.5 Conclusions

Although these results are just first insights on the performance of the MStack and more specifically, on the performance of DELTOYA and PTP (diversity and transport capabilities), they show a very good performance and an extremely improvement compared to the performance of TCP. With the MStack, it seems that we will be really able to transfer data flows in a real MANET, where mobility and disconnections are present.

15.2.3 Real Experiments in Real Scenarios

In order to show the benefits of the MStack not only in controlled scenarios but also in real scenarios, we performed a set of three experiments in different places of Barcelona.

We recorded the three experiments in three interesting videos available in YouTube. Although the videos are without voice, they are clear enough to be self-understood.

In these experiments we used from two to four HTC Nexus One smartphones with Android OS.

15.2.3.1 Network Layer

In the first video [**48**], the routing protocol of the MStack (AGENET) is showed. During the experiment, a file transfer between a source node and a destination node in a two hops path was used. As an intermediate node, two more mobile nodes were used. Both intermediate nodes were moving around during the experiment, causing intermittent link breaks in the routing paths, in order to simulate the mobility of the nodes in a real MANET.

15.2.3.2 Transport Layer

In the second video [**49**], a comparison between the MStack versus the TCP/IP stack in front of mobility is showed. In this scenario, two WLANs with public Wi-Fi Hotspots were used instead of a MANET. The idea of this experiment was to show the robustness of the MStack in front of the handovering process between different Wi-Fi access points. The same idea of mobility could be translated to the case of the MANETs.

In this experiment we started two file transfers of the same file from two different nodes. One was using the MStack and the other the TCP/IP stack. The file was downloaded from the Internet using the free internet connections of the open Wi-Fi Hotspots used in the experiments.

15.2.3.3 Session Layer

In the third video [**50**], the benefits of exploiting the diversity and delay-tolerant properties inherent in any MANET by the MStack are showed. In this experiment, three mobile devices were used. At the beginning of the video we can see how two mobile devices share the same file. Afterwards, the third static device starts to get this file from one of the two mobile devices. When this mobile device disappears from the range of the static device, the file is interrupted and resumed as soon as the other mobile device appears in the range of the static device, thus letting the static device resuming the file transfer as if a simple disconnection was occurred.

15.2.3.4 Conclusions

As the three videos show, the MStack is getting more and more performance in front of any of the main problems of the MANETs.

In the case of the network layer, the MStack performs very well in front of mobility, the main problem when dealing with multi-hop networks. In the case of the transport layer, the MStack performs perfectly in front of disconnections and interruptions of the data flows due to the intrinsic properties of the mobile networks. And in the case of the session layer, the MStack provides all the capabilities needed to build robust MANETs by exploiting the diversity and delay-tolerance properties of this type of networks.

16. Conclusions

In order to enable communications between nodes inside a network, a communications stack is needed in each node of the network, and each type of network needs its own specific communications stack.

Concerning MANETs, research in this area was always focused on optimizing the TCP/IP stack, the stack par excellence in the majority of our current networks. Instead of designing a complete new communications stack, researchers focused their work on improving already existing solutions coming from the wired networks. But the problem is that optimizing a communications stack specifically designed for wired networks will never be the perfect solution. A complete and new communications stack specifically designed for MANETs is needed if we want to achieve robust communications in this type of networks that have properties very different from what wired networks have.

Moreover, the problem of adapting the current solutions for MANETs is that they do not exploit the inherent properties of this type of networks: the broadcast properties of the wireless medium, the diversity of nodes in a MANET, and the delay-tolerance properties of a mobile environment. Exploiting the advantages of the broadcast medium of the wireless networks instead of the unicast medium of the wired networks, exploiting the diversity that the plurality of nodes of a MANET offer and exploiting the delay-tolerance properties that any mobile environment have seem to be obligatory for building a powerful communications stack for MANETs.

For all these reasons, in this thesis we present what nowadays is still missing: a novel communications stack specifically designed for mobile ad-hoc networks, the MStack.

The vast majority of the research work has been always based on results obtained via network simulators. Despite the evident advantage of using a network simulator concerning the operational costs of the experiments, this fact has caused an obtaining of results far away from reality. Not only network simulators are far away from simulating real conditions but also a lot of key properties for the correct operation of MANETs have been hided due to the fact of using simulators for the experimentation.

Have results based on simulations considered the problem that certain Wi-Fi devices, such as smartphones, have concerning the directivity of their Wi-Fi antennas? The simple fact of turning a smartphone in one direction or in another or the simple fact of having the smartphone in our pocket instead of in our hands can be of huge impact concerning the results of the experiment. This is important not only for evaluating the new proposed solutions but also for discovering this kind of details (only discoverable with real experimentation and not with simulations) that help a lot on improving our decisions of design.

Obviously, using real experimentation we cannot test one thousand mobile nodes in a five hundred square meters area. Only with network simulators we can test big scenarios. But this fact, far from being an advantage of the network simulators, is also another big error caused by the use of simulators. With difficulty a real MANET will have one thousand mobile nodes and will be located in a five hundred square meters area. A real MANET will usually have a few tens of nodes in a small area. And we should not forget that our task as researchers is not

designing better solutions than our research partners; our task is designing better solutions being able to work in the real world.

For all these reasons, our preference has been for real experimentation instead of for the more traditional evaluation through simulation, as we believe that current simulation models hardly capture with sufficient fidelity the particularities of wireless channels in complex scenarios.

Currently, the MStack is under evaluation and every day new and better results are being obtained. A complete evaluation of the stack in all its layers in a short period of time should be only possible at a huge cost. For this reason, we only showed a reduced set of experiments that both prove each of the layers of the stack and show the benefits of the stack versus the TCP/IP stack.

Although the results are just first insights on the performance of the MStack, they show a very good performance and an extremely improvement compared to the performance of TCP/IP. In the case of the network layer, the MStack performs very well in front of mobility, the main problem when dealing with multi-hop networks. In the case of the transport layer, the MStack performs perfectly in front of disconnections and interruptions of the data flows due to the intrinsic properties of the mobile networks. And in the case of the session layer, the MStack provides all the capabilities needed to build robust MANETs by exploiting the diversity and delay-tolerance properties of this type of networks.

The MStack is an example of how some of the basic assumptions and mechanisms used in wired or wireless infrastructure networks must be fundamentally modified when dealing with MANETs. All in all, we believe that the results presented in this thesis provide interesting insights into the potential of the MStack in MANETs.

16.1 Beyond the Conclusions

Independently of the contributions and the importance of this work, and independently of the quality of the results, I would like to add some conclusions concerning the lessons I learnt since I started this PhD.

The first time you decide to go through a more experimental work, you still do not know the implications that this decision will have in your progress. When you play in the simulation world, there are two main points: to design your protocols and to implement and simulate them in the network simulator to obtain the results; no hardware is involved in this process. However, when you play in the real world, you must start dealing with off-the-self hardware in order to implement and test your work. This is a critical aspect that for sure will affect your progress.

Not all wireless cards use the same wireless driver. And not all wireless drivers have the same capabilities. When you start hacking at this low layer, everything becomes unpredictable. Poor documentation, difficulty of debugging your work and other issues will keep bothering you. Moreover, when you start playing with mobile nodes such as Symbian, Android or iPhone smartphones instead of using open linux laptops, you will realize that all these closed operating systems will be a barrier for your implementations, especially for the case of setting

their wireless cards in ad-hoc mode. I've been wondering for so many time why these devices, with a standard Wi-Fi chipset (and so, including the ad-hoc or IBSS mode), have so many bugs or even do not support the ad-hoc mode. Is it because manufacturers do not see the business in using the Wi-Fi card in ad-hoc mode? Is it because there is not a power saving mode in the ad-hoc standard? Is it because the industry is afraid of some security issues related with the peer-to-peer nature of the ad-hoc mode? Or is it because of the tethering? Wi-Fi Direct is the answer for some of these questions, but of course, it is not a substitute of the ad-hoc mode.

The other difficulty you will find during your experimental work is the way in which you capture and test your experiments, especially in the multi-hop scenarios. In real experimentation we do not have the easy "out.log" file of the network simulators containing everything you wanted to capture from the experiment. Real experiments include several devices, and each of these devices sent real packets to the wireless medium. Moreover, these packets can reach their destinations or can be lost in the channel due to interferences or collisions, or even due to signal attenuation. Only wireless sniffers located in other devices and placed along the testbed scenario will be able to capture these issues. But how many sniffers do we need? And where exactly do we place the sniffers? If a packet is sent by a node and not captured by any sniffer, does this mean that the packet will not reach the destination node? And how we will synchronize all the captures of all the sniffers? A lot of difficult and hard work will be needed to analyze and merge these logs.

Concerning the testbed scenarios, a new set of difficulties will keep bothering you. If you want to test a mobile ad-hoc network of just ten nodes you will need ten people! And if you want or need to repeat the experiment again and again, you will have to start buying a lot of presents to your friends if you do not want to become alone during your experiments. Moreover, not only the people are important but also the scenario. During your experiments you will want to test your protocols in indoor and in outdoor scenarios. But the results in the indoor scenarios will vary a lot depending on the materials of the building! And when you deal with outdoor scenarios, sometimes you will want to have scenarios with line of sight (LOS) connectivity between your devices. But have you tried to find a scenario with LOS, without obstacles, and long enough to be able to go out of coverage between two devices? When I needed this scenario, the only place I found was a closed landing strip in a small airport, 150 kilometers far away from my office. Luckily, in those tests I did not need the ten friends.

And this is just a short summary of the difficulties you will find when you start dealing with the experimental work. Other aspects like the weather, power constraints, unpredictable but unrepeatable results which will let you a question mark on your face, and more and more issues will keep bothering you during your experimental work. But at the end, when you will have beaten all these difficulties and you will have noticed of the importance of your work, then you will realize that your work was not in vain.

Bibliography

[1] I. Chamtlac, M. Conti, and J. Liu, "Mobile ad hoc networking: imperatives and challenges," *Ad-Hoc Networks Journal*, vol. I, pp. 13-64, 2003.

[2] I. F. Akyildiz, X. Wong, and W. Wong, "Wireless mesh networks: a survey," *Computer Networks*, vol. 47, no. 4, pp. 445-487, March 2005.

[3] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A Survey on Sensor Networks," *IEEE Communications Magazine*, vol. 40, no. 8, pp. 102-114, August 2002.

[4] A. McMahon and S. Farrell, "Delay and Disruption Tolerant Networking," *IEEE Internet Computing*, vol. 13, no. 6, pp. 82-87, November 2009.

[5] H. Zimmermann, "OSI reference model - The ISO model for architecture for open systems interconnection," in *IEEE Transaction on Communications*, 1980.

[6] R. Stevens, *TCP/IP Illustrated, Vol. 1: The Protocols*.: Addison-Wesley, 1994.

[7] WAP Forum. [Online]. http://www.wapforum.com/

[8] RFC 791, "Internet Protocol - DARPA Internet Program Protocol Specification," 1981.

[9] RFC 793, "Transmission Control Protocol - DARPA Internet Program Protocol Specification," 1981.

[10] RFC 768, "User Datagram Protocol," 1980.

[11] J. García-Vidal, "The case for a cooperative stack for wireless multihop networks," in *Euroview*, Würzburg, 2007.

[12] C. Reis, R. Mahajan, M. Rodrig, D. Wetheral, and J. Zahorjan, "Measurement-based models of delivery and interference in static wireless networks," in *ACM SIGCOMM*, 2006, pp. 51-62.

[13] C. Perkins, E. Belding-Royer, and S. Das, "Ad hoc on-demand distance vector (AODV) routing," RFC 3561, 2003.

[14] C. Adjih et al., "Optimized link-state routing protocol," RFC 3626, 2003.

[15] M. K. Marina and S. R. Das, "Ad hoc on-demand multipath distance vector routing," in *ACM SIGMOBILE*, 2002, pp. 92-93.

[16] A. Miu, H. Balakrishna, and C. E. Koksal, "Improving loss resilience with multi-radio diversity in wireless networks," in *MOBICOM*, 2005.

[17] I. D. Aron and S. K. S. Gupta, "A witness-aided routing protocol for mobile ad-hoc networks with unidirectional links," in *Mobile Data Access First International Conference*,

1999, pp. 24-33.

[18] S. Chachulski, M. Jennings, S. Katti, and D. Katabi, "Trading structure for randomness in wireless opportunistic routing," in *ACM SIGCOMM*, 2007, pp. 169-180.

[19] C. E. Perkins and P. Bhagwat, "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers," in *ACM SIGCOMM*, 2004, pp. 234-244.

[20] D. Johnson, Y. Hu, and D. Maltz, "The dynamic source routing protocol (DSR) for mobile ad hoc networks for IPv4," in *RFC 4728*, 2007.

[21] A. Scaglione, D. L. Goeckel, and J. N. Laneman, "Cooperative communication in mobile ad hoc networks," *IEEE Signal Processing Magazine*, vol. 23, no. 5, pp. 18-29, September 2006.

[22] F. Fitzek and M. D. Katz, Eds., *Cooperation in wireless networks: principles and applications*. Dordrecht, Netherlands: Springer, 2006.

[23] S. Srinivasa and S. Krishnamurthy, "CREST: an opportunistic forwarding protocol based on conditional residual time," in *6th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, Rome, 2009.

[24] P. Basu and C. K. Chau, "Opportunistic forwarding in wireless networks with duty cycling," in *ACM Workshop on Challenged Networks*, 2008.

[25] C. K. Chau and P. Basu, "Exact analysis of latency of stateless opportunistic forwarding," in *IEEE INFOCOM*, 2009.

[26] P. Larsson, "Selection diversity forwarding in a multihop packet radio network with fading channel and capture," in *ACM SIGMOBILE*, 2001.

[27] S. Biswas and R. Morris, "Opportunistic routing in multi-hop wireless networks," in *ACM SIGCOMM*, 2004.

[28] S. Biswas and R. Morris, "ExOR: opportunistic multi-hop routing for wireless networks," in *ACM SIGCOMM*, 2005.

[29] P. A. Chou and Y. Wu, "Network coding for the internet and wireless networks," Microsoft Research, Tech. Rep. MSR-TR-2007-70, 2007.

[30] RFC 3550, *RTP: A Transport Protocol for Real-Time Applications*., 2003.

[31] H. Balakrishnan, V.N. Padmanabhan, S. Seshan, and R. H. Katz, "A comparison of mechanisms for improving TCP performance over wireless links," in *IEEE/ACM Transaction on Networking*, 1997, pp. 756-769.

[32] Y. Tian, K. Xu, and N. Ansari, "TCP in wireless environments: problems and solutions," *IEEE Communications Magazine*, vol. 43, no. 3, pp. 27-32, March 2006.

[33] E. Ayanoglu, S. Paul, T. F. LaPorta, K. K. Sabnani, and R. D. Gitlin, "AIRMAIL: A Link-Layer Protocol for Wireless Networks," *ACM Wireless Networks*, vol. 1, pp. 47-60, 1995.

[34] A. Bakre and B. R. Badrinath, "I-TCP: Indirect TCP for mobile hosts," in *15th International Conference on Distributed Computing Systems (ICDCS)*, 1995.

[35] RFC 3517, "A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP," 2003.

[36] RFC 2960, "Stream Control Transmission Protocol," 2000.

[37] J. Liu and S. Singh, "ATCP: TCP for Mobile Ad Hoc Networks," *IEEE*, vol. 19, no. 7, pp. 1300-1315, 2001.

[38] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming - The Sockets Networking API*.: Addison-Wesley, 2004.

[39] OMG 2004, "Common Object Request Broker Architecture (CORBA/IIOP)," version 3.0.3.

[40] Java RMI. [Online]. http://download.oracle.com/javase/tutorial/rmi/index.html

[41] Python Remote Objects (Pyro). [Online]. http://www.xs4all.nl/~irmen/pyro3/

[42] S. Kurkowski, T. Camp, and M. Colagrosso, "MANET simulation studies: the incredibles," in *ACM SIGMOBILE*, 2005.

[43] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click modular router," in *ACM Transactions on Computer Systems*, 2000.

[44] MadWifi: Multiband Atheros Driver for WiFi. [Online]. http://madwifi.org

[45] C. Tschudin and R. Gold, "LUNAR: lightweight underlay network ad-hoc routing," University of Basel, Switzerland, Tech. Rep. 2002.

[46] Nokia. Symbian OS. [Online]. http://symbian.org/

[47] Google. Android OS. [Online]. http://www.android.com/

[48] Miraveo. Network Layer Video. [Online]. http://www.youtube.com/watch?v=8rrOoeRGX-c

[49] Miraveo. Transport Layer Video. [Online]. http://www.youtube.com/watch?v=wskpZgOKMak

[50] Miraveo. Session Layer Video. [Online]. http://www.youtube.com/watch?v=oV3f-mUrGjg

Appendix A. Cooperative Forwarding Details

A.1 Cooperative Forwarding Calculations

Assuming 802.11a and OFDM as PHY layer, we have the following values of Table 13 and Table 14.

Table 13. 802.11a and OFDM parameters

$t_{PREAMBLE}$	t_{PLCP}	DIFS	SIFS	CW_{min}	Slot time (σ)
20 µs	4 µs	34 µs	16 µs	15	9 µs

Table 14. 802.11a and OFDM parameters (Cont.)

Dat Rate (Mbps)	Modulation	Coding Rate	Coded bits per subcarrier	Coded bits per OFDM symbol	Data bits per OFDM symbol
6	BPSK	1/2	1	48	24 (T_{SYM}=4 µs)
9	BPSK	3/4	1	48	36
12	QPSK	1/2	2	96	48
18	QPSK	3/4	2	96	72
24	16-QAM	1/2	4	192	96
36	16-QAM	3/4	4	192	144
48	16-QAM	2/3	6	288	192
54	64-QAM	3/4	6	288	216

In Figure 88 we can see the format and size of the packets implied in our calculations.

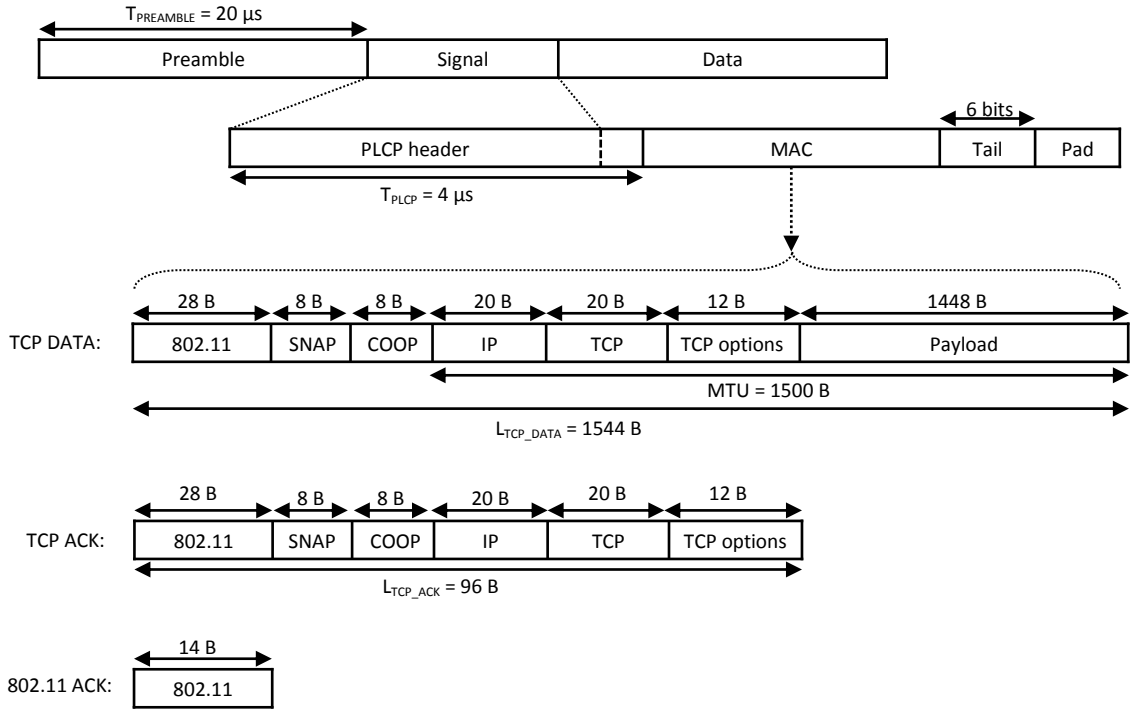

Figure 88. Format and Size of TCP Frames

In order to calculate the maximum average throughput that can be achieved in both one hop and two hops paths, we will assume:

- During the backoff period, no other station starts a transmission. The backoff counter will not be interrupted.
- The transmitter is the only node trying to access to the channel.
- Between retransmissions of the same frame, the transmitter does not switch its wireless card to reception mode.
- Propagation time = 0.
- The feedback of the TCP ACKs is not considered.
- Processing time (store and forward) = 0.

All these assumptions are correctly assumed since we want to get the maximum throughput in perfect channel conditions. Moreover, since we want to get the maximum average throughput, we will take the mean backoff counter at every retransmission; see Figure 89.

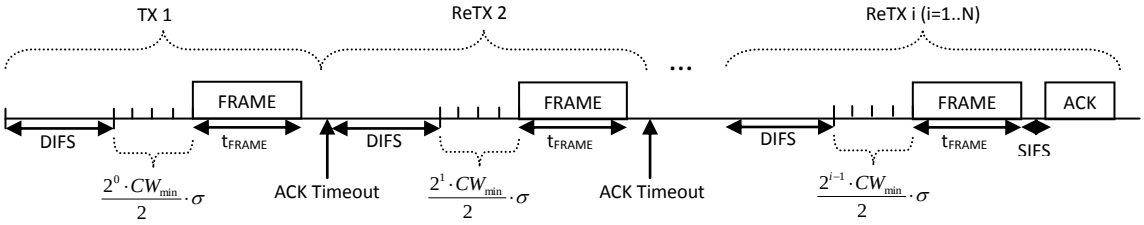

Figure 89. Mean Backoff Counter Per Retransmission

Then, the average time of n unacknowledged retransmissions will be:

$$E[t_{n\,\mathrm{ReTX}}] = \sum_{i=1}^{n} DIFS + \left(\frac{2^{i-1}\cdot CW_{\min}}{2}\cdot\sigma\right) + t_{FRAME} + t_{ACK\_TOUT} =$$

$$= n\cdot\left(DIFS + t_{FRAME} + t_{ACK\_TOUT}\right) + \left(CW_{\min}\cdot\sigma\cdot\sum_{i=1}^{n} 2^{i-2}\right) =$$

$$= n\cdot\left(DIFS + t_{FRAME} + t_{ACK\_TOUT}\right) + \left(CW_{\min}\cdot\sigma\cdot\frac{2^{n}-1}{2}\right)$$

where

$$t_{FRAME} = t_{PREAMBLE} + t_{PLCP} + L_{SYM(TCPDATA)}\cdot T_{SYM} =$$

$$= t_{PREAMBLE} + t_{PLCP} + \left\lceil\frac{L_{BYTES(TCPDATA)}\cdot 8 + tail\_bits}{Data\_bits\_per\_symbol}\right\rceil\cdot T_{SYM} =$$

$$= 20 + 4 + \left\lceil\frac{1544\cdot 8 + 6}{24}\right\rceil\cdot 4 = 20 + 4 + 515\cdot 4 = 2084\mu s$$

if the frame is a tcp data packet;

or

$$t_{FRAME} = t_{PREAMBLE} + t_{PLCP} + L_{SYM(TCPACK)} \cdot T_{SYM} =$$
$$= t_{PREAMBLE} + t_{PLCP} + \left\lceil \frac{L_{BYTES(TCPACK)} \cdot 8 + tail\_bits}{Data\_bits\_per\_symbol} \right\rceil \cdot T_{SYM} =$$
$$= 20 + 4 + \left\lceil \frac{96 \cdot 8 + 6}{24} \right\rceil \cdot 4 = 20 + 4 + 33 \cdot 4 = 156\mu s$$

if the frame is a tcp ack packet.

Then, the average time of a successful transmission that has needed i-1 retransmissions (i=1..n) will be:

$$E[t_{iTXOK}] = E[t_{i-1\,\mathrm{ReTX}}] + E[t_{i-thTX}] =$$
$$= (i-1) \cdot \left[(DIFS + t_{FRAME} + t_{ACK\_TOUT}) + \left(CW_{\min} \cdot \sigma \cdot \frac{2^{i-1} - 1}{2} \right) \right] +$$
$$+ \left[DIFS + \left(\frac{2^{i-1} \cdot CW_{\min}}{2} \cdot \sigma \right) + t_{FRAME} + SIFS + t_{80211ACK} \right]$$

where

$$t_{80211ACK} = t_{PREAMBLE} + t_{PLCP} + L_{SYM(80211ACK)} \cdot T_{SYM} =$$
$$= t_{PREAMBLE} + t_{PLCP} + \left\lceil \frac{L_{BYTES(80211ACK)} \cdot 8 + tail\_bits}{Data\_bits\_per\_symbol} \right\rceil \cdot T_{SYM} =$$
$$= 20 + 4 + \left\lceil \frac{14 \cdot 8 + 6}{24} \right\rceil \cdot 4 = 20 + 4 + 5 \cdot 4 = 44\mu s$$

Then, the maximum average throughput that can be achieved in one hop paths will be:

$$V_{ideal1hop} = \frac{Payload}{E[t_{1TXOK}]} = \frac{1448B}{34 + \left(\frac{15}{2} \cdot 9 \right) + 2084 + 16 + 44} = \frac{1448B}{2245{,}5\mu s} \approx 644845Bps$$

And the maximum average throughput that can be achieved in two hops paths will be:

$$V_{ideal2hops} \approx \frac{Payload}{2 \cdot E[t_{1TXOK}]} = \frac{1448B}{2 \cdot 2245{,}5\mu s} \approx 322422Bps$$

On the other hand, in order to calculate the maximum average throughput that our prototype, due to its hardware limitations, can achieve when cooperation takes place, we will assume:

- Cooperation takes place just after the N retransmissions of the transmitter node.
- The cooperator will correctly transmit the packet in the first transmission.

All these assumptions are correctly assumed since we want to get the maximum hardware-limited throughput in perfect channel conditions. Moreover, since we want to get the maximum hardware-limited average throughput, we will take the mean backoff counter at every retransmission; see Figure 90.

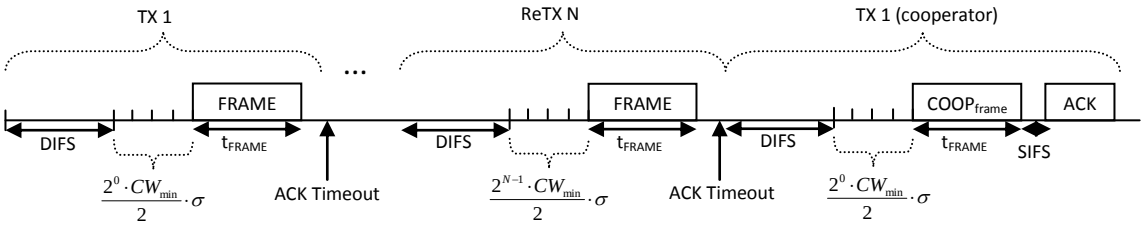

Figure 90. Mean Backoff Counter Per Retransmission (Cooperative case)

Then, the average time of a successful cooperation that has needed only one transmission will be:

$$E\left[t_{CoopTX}\right] = E\left[t_{N\,\mathrm{Re}TX}\right] + E\left[t_{1TXOK}\right]$$

And the maximum hardware-limited average throughput which can be achieved by our prototype in a link break will be:

$$V_{\mathrm{Re}TX+Coop} = \frac{Payload}{E\left[t_{CoopTX}\right]} =$$

$$= \frac{1448B}{7\cdot(34+2084+25)+\left(15\cdot 9\cdot\frac{2^7-1}{2}\right)+34+\left(\frac{15}{2}\cdot 9\right)+2084+16+44} =$$

$$= \frac{1448B}{25819\mu s} \approx 56083Bps$$

Appendix B. MICO Table Details

B.1 DTDO Table

The fields of the first sub table of the DTDO Table are:

- **DTDO_ID (uint32_t)**

  The unique identifier of the DTDO in the node where de DTDO was created

- **DTDO_pluginID (uint32_t)**

  The unique identifier of the application where the DTDO was created

- **DTDO_pluginTag (uint32_t)**

  The unique identifier of the DTDO in the scope of the application DTDO_pluginID and node DTDO_originator where the DTDO was created

- **DTDO_originator (uint64_t)**

  The unique ID of the node where the DTDO was created

- **DTDO_timestamp (struct timeval)**

  The time instant (in milliseconds from 1970 (unix time)) when the DTDO was created

- **DTDO_data (void*)**

  The actual data of the DTDO (if any)

- **DTDO_dataSize (uint32_t)**

  The size (in Bytes) of the data stored (or that will be stored) in the DTDO_data field

- **DTDO_dataTTL (uint32_t)**

  DTDO_dataTTL provides a mechanism for limiting the amount of time the DTDO_data of a DTDO is kept in the network (i.e. DTDO Table of remote nodes) once the DTDO has left its originator node

- **DTDO_dataTimeout (struct timeval)**

  The time instant when the DTDO_data of this DTDO has to be deleted from the DTDO Table

- **DTDO_useCounter (uint8_t)**

  The number of DTDO Requests simultaneously waiting for the obtaining of the DTDO_data of this DTDO entry

· **DTDO_isPublic (bool)**

A flag that indicates if the DTDO is accessible or not by other nodes of the network or even by the same node but by different application

Entries in this sub table, from the point of view of the node storing the sub table, can be divided into two implicit types: Local DTDOs, created by any application of the node (DTDO_originator == “my ID”), and Remote DTDOs, created by applications of other nodes of the network and so, obtained from the network (DTDO_originator ≠ “my ID”).

For local DTDOs, the key field of this sub table is the DTDO_ID, meaning that there cannot be two entries (i.e., two local DTDOs) with the same DTDO_ID. The DTDO_ID will be used by the applications for identifying DTDOs between them and DELTOYA. The DTDO_ID will be implemented as a cyclic sequence number maintained by DELTOYA. As said before, only local DTDOs will use the DTDO_ID field (i.e., DTDO_ID != 0); Remote DTDOs will always have DTDO_ID == 0. The scope of the uniqueness of this field is the node itself (i.e., DELTOYA itself). However, although the uniqueness of the DTDO_ID is guaranteed by DELTOYA for any local DTDO (i.e., for any DTDO of any application of the same node), we must ensure that applications, using the DTDO_ID, can only access (modify, publish, unpublish, delete, push, etc) DTDOs created by them, and forbid any access to either local DTDOs created by other applications of the same node or to remote DTDOs by means of the DTDO_ID.

On the other hand, the DTDO_pluginID, the DTDO_pluginTag, the DTDO_originator and the DTDO_timestamp will also be used together as key fields to uniquely identify a DTDO, meaning that there cannot be two entries (i.e., local or remote DTDOs) with the same DTDO_pluginID, DTDO_pluginTag, DTDO_originator and DTDO_timestamp fields. This extra identifier will be used by the applications for identifying local DTDOs created by other applications or for identifying remote DTDOs (i.e., for identifying DTDOs stored in other nodes of the network).

In fact, for local DTDOs, only DTDO_pluginID, DTDO_pluginTag and DTDO_originator will be used together as key fields, meaning that there cannot be two entries with the same DTDO_pluginID, DTDO_pluginTag and DTDO_originator fields, even if they have different DTDO_timestamp field. The reason is to force the user, when he wants to modify the DTDO_data of a DTDO, to either modify or delete/create the DTDO instead of creating a new one with the same fields without deleting the previous one.

Moreover, another restriction for both local and remote DTDOs has to be guaranted. New obtained DTDOs will be only cached if there is no newer version (same DTDO_pluginID, DTDO_pluginTag and DTDO_originator but more recent DTDO_timestamp) cached yet. Older versions (DTDO_timestamp in the past) will be deleted.

The DTDO_pluginID will be a unique identifier of the application, and it will be managed in order to guarantee the uniqueness of the field between all the applications of the same node.

The DTDO_pluginTag is a unique identifier of the DTDO in the scope of the application where the DTDO was created and so, in this case, the application itself is the entity in charge of guaranteeing the uniqueness of this field in its scope.

The DTDO_originator is the node that created the DTDO (see section 13).

The DTDO_timestamp is the time instant when the DTDO was created.

Note that for local DTDOs, DTDO_ID and DTDO_data fields have to be always used, and DTDO_dataTimeout and DTDO_useCounter have to never be used. For remote DTDOs, DTDO_ID have to never be used (i.e., set to 0), DTDO_data and DTDO_dataTimeout can either be used or not (depending on if we have discovered the DTDO and obtained its DTDO_data or only discovered the DTDO), and DTDO_useCounter have to be always used.

It is important to bear in mind that a problem for local and remote DTDOs being the same DTDO could appear. Imagine the case where we are the plugin P in node N, and we create a local DTDO. Implicitly, it will be able to be identified by {P,tag,N,timesamp}. Independently of the plugin, imagine now that node M requests this DTDO through the network and it obtains a copy. And then, node N requests the same DTDO (its own DTDO): node N is explicitly requesting one of its DTDOs, but maybe from another node. In this case, node N either will detect that he is asking for one of his local DTDOs and he will not ask it to the network during the DTDODM (in the case of requesting specific DTDOs; see appendix "C.7.2 Requesting specific DTDOs").

The DTDO_isPublic field can be true or false, meaning that a DTDO can be set as accessible or not to other nodes of the network or other applications of the same node (and independently of how we have obtained the DTDO, that is, via the methods `createDTDO`, `requestDTDOs` or `pickupDTDO`).

The DTDO_dataTTL field provides a mechanism for limiting the time the DTDO_data of a DTDO is kept in the SPAN network (i.e. DTDO Table of remote nodes). From the point of view of a node, local DTDOs will be always accessible (since they will be always cached) for any application of this node, independently of the local application that created them (as far as the application that created the DTDO publishes it and does not delete it). Moreover, the local application will set an initial value for the DTDO_dataTTL field of the local DTDO it created. From now on, these local DTDOs will be able to be obtained for other nodes of the network together with the DTDO_dataTTL value. From the point of view of these remote nodes, the DTDO_dataTTL value will be the amount of time in milliseconds that the remote node will maintain the DTDO_data of the DTDO in its DTDO Table and as accessible to other nodes. After this amount of time, as explained in the paragraph below (DTDO_dataTimeout and DTDO_useCounter fields), the remote node will delete the DTDO_data from the DTDO Table and, if applicable, also the DTDO entry from the DTDO Table. And, if during (and not after) this amount of time, this remote DTDO is asked by another node of the network, the DTDO_dataTTL that the node caching the remote DTDO will send to the third node will be the original DTDO_dataTTL value it received minus the amount of time the DTDO_data of the remote DTDO has spent in its DTDO Table. In this way, by updating the DTDO_dataTTL originated by the local node along all the remote nodes in which the DTDO_data of the DTDO is being "contaminated", DELTOYA provides a mechanism for limiting the time the DTDO_data of a DTDO is kept in the network (i.e., DTDO Table of remote nodes) once the DTDO has left its originator node.

Local DTDOs do not have timeout value, and they have to be deleted explicitly by the application that created them. Since viral time == cache time, DTDO_data of remote DTDOs has a timeout value of DTDO_dataTimeout = “insertion time plus DTDO_dataTTL” milliseconds. After this time instant, the DTDO_data of the DTDO will be automatically deleted from the DTDO Table. Concerning the DTDO entry, it will be deleted only when DTDO_data is NULL (due to DTDO_data for this DTDO has never been obtained or due to it has already expired) and when DTDO_useCounter is 0 (i.e., when there is not any DTDO Request trying to get the DTDO_data of this DTDO at this time).

DTDO_useCounter, as explained before, will be a field used to identify when DELTOYA can delete a DTDO entry of the DTDO Table or not. Entries for local DTDOs are created using the method `createDTDO` and are not affected by this field, since DTDO_useCounter is only valid for entries of remote DTDOs. For remote DTDOs, entries will always be created when a DTDODM (see section 9) of a currently active DTDO Request discovers a remote DTDO. At this time, the entry for the remote DTDO will be created (or updated) and the DTDO_useCounter field will be incremented by one, meaning that at least one DTDO Request is using this DTDO entry and so, warning that it does not have to be deleted. When the DTDO Request (section 6.2) (and not the DTDO Query (section 6.1)) will receive the DTDO_data requested or when the MRT of the DTDO Request will expire (in the case that not all the DTDO_data requested have been able to be obtained), then the DTDO_useCounter field of the involved DTDO entries (the DTDO entries of the DTDO_data received and the DTDO entries of the DTDO_data not received) will be decremented by one, meaning that this DTDO Request is no longer using these DTDO entries.

DTDO_ID (uint32_t) field can suffer overflow problems, meaning that if 2^32 values are used at the same time, no more values will be able to be used. If this occurs, DELTOYA must forbid creating new DTDOs if old ones are not deleted before.

DTDO_timestamp field can suffer overflow problems, meaning that after year 2038, unix time will not be valid anymore.

On the other hand, the fields of the second sub table of the DTDO Table are:

- **Destination node (uint64_t)**

  The unique ID of the destination node

- **Path (list{uint64_t,uint8_t,uint8_t})**

  The sequence of nodes (unique IDs) to reach the destination node (the last unique ID of the path field is always the unique ID of the destination node)

- **For each node in the Path field:**

  - **Level of battery**

    The known level of battery of this node when this routing information was added to the table

· **Timestamp (struct timeval)**

The time instant when this information was generated by its originator (i.e., by the destination node)

· **Status (uint8_t)**

0 → This routing path does not contain any unidirectional link

1 → This routing path contains one or more unidirectional links

· **Timeout (struct timeval)**

The time instant when this entry has to be deleted from the table

The key fields of this sub table are the destination node and the path (without the level of battery of each node of the path), meaning that there cannot be two entries with the same destination node and path fields. There can be more than one entry per destination node, and a maximum of MAXAGENETENTRIES_DEST entries per destination node.

The entries of this sub table have a timeout value of "insertion or update time plus TOUT_AGENETENTRY" milliseconds (see section appendix "I.3.2 Receiving AGROHellos"). If during TOUT_AGENETENTRY milliseconds the entry is not updated, they will be automatically deleted.

Finally, recipient relationships between entries of both sub tables can exist, meaning that an entry of the first sub table can be linked to zero, one or more entries of the second sub table, and at the same time, that an entry of the second sub table can be linked to zero, one or more entries of the first sub table. A recipient relationship indicates which nodes (destination nodes of the second sub table) are recipients of which DTDOs (entries of the first sub table), that is, in which nodes a DTDO could be found.

Each link has a TTL field. This TTL field indicates which DTDO_dataTTL value will have to be used once the DTDO_data of the DTDO is obtained from this recipient (if it is the case).

Note that the TTL field of each link is a field different from the DTDO_dataTTL field. While the TTL field of each link indicates only the value that, in the case of obtaining the DTDO from this recipient, will be used to set the DTDO_dataTTL field, the current DTDO_dataTTL field indicates the remaining time of our cached DTDO (if any).

Moreover, local DTDOs cannot be linked to any recipient of the second sub table (since they are never obtained from the network, meaning that we will never know other possible recipients of our DTDOs). Remote DTDOs can be linked to zero, one or more recipients of the second sub table (since they are always discovered through the network and so, from one or more recipients of the DTDO, although they can have zero recipients if the entries of their initially discovered recipients expire).

At the same time, recipients (or destination nodes, not entries) of the second sub table can be linked to zero, one or more entries of the first sub table (i.e. DTDOs), since usually, entries are

inserted due to a discovery process of a DTDO, but also for establishing communications with nodes in which MICO does not care about the DTDOs of these nodes.

The link between a DTDO and a destination node is maintained static as far as the DTDO or the destination node does not expire. If the DTDO entry expires or all the entries of a destination node expire, then the link between the DTDO and the destination node will be deleted.

B.2 DOCK Table

The fields of this table are:

- **DOCK_ID (uint32_t)**

  The unique identifier of the DOCK in the node where de DOCK was created

- **DOCK_pluginID (uint32_t)**

  The unique identifier of the application where the DOCK was created

- **DOCK_pluginTag (uint32_t)**

  The unique identifier of the DOCK in the scope of the application DOCK_pluginID and the node where the DOCK was created

- **DOCK_timestamp (struct timeval)**

  The time instant (in milliseconds from 1970 (unix time)) when the DOCK was created

- **DOCK_buffer (DOCK Buffer)**

  The buffer where the DTDOs pushed to this DOCK are saved until they are picked up from the DOCK

The key field of this table is the DOCK_ID, meaning that there cannot be two entries (i.e., two DOCKs) with the same DOCK_ID. The DOCK_ID will be used by the applications for identifying DOCKs between them and DELTOYA. The DOCK_ID will be implemented as a cyclic sequence number maintained by DELTOYA. The scope of the uniqueness of this field is the node itself (i.e., DELTOYA itself). However, although the uniqueness of the DOCK_ID is guaranteed by DELTOYA for any DOCK of any application of the same node, we must ensure that applications, using the DOCK_ID, can destroy or pick up only DOCKs created by them, and forbid it to DOCKs created by other applications of the same node by means of the DOCK_ID.

On the other hand, the DOCK_pluginID, the DOCK_pluginTag and the DOCK_timestamp will also be used together as key fields to uniquely identify a DOCK, meaning that there cannot be two entries with the same DOCK_pluginID, DOCK_pluginTag and DOCK_timestamp fields. This extra identifier will be used by the applications for identifying DOCKs created by other applications of the same node or for identifying DOCKs created by other nodes of the network.

In fact, only DOCK_pluginID and DOCK_pluginTag will be used together as key fields, meaning that there cannot be two entries with the same DOCK_pluginID and DOCK_pluginTag fields,

even if they have different DOCK_timestamp field. The reason is to differentiate the same DOCK (same DOCK_pluginID and DOCK_pluginTag) created twice: If a node A creates a DOCK, and another node B pushes some DTDOs to this DOCK, but while these DTDOs are travelling along the network, the node A destroys and creates again the same DOCK (with different DOCK_timestamp), then we do not want to put these DTDOs (when they arrive to node A) to the new DOCK, since in fact, it is a different DOCK although it has the same DOCK_pluginID and DOCK_pluginTag fields.

The DOCK_pluginID will be a unique identifier of the application, and it will be managed in order to guarantee the uniqueness of the field between all the applications of the same node.

The DOCK_pluginTag is a unique identifier of the DOCK in the scope of the application where the DOCK was created and so, in this case, the application itself is the entity in charge of guaranteeing the uniqueness of this field in its scope.

The DOCK_timestamp is the time instant when the DOCK was created.

DOCKs do not have timeout value, and they have to be deleted explicitly by the application that created them.

DOCK_ID (uint32_t) field can suffer overflow problems, meaning that if 2^32 values are used at the same time, no more values will be able to be used. If this occurs, DELTOYA must forbid creating new DOCKs if old ones are not deleted before.

DOCK_timestamp field can suffer overflow problems, meaning that after year 2038, unix time will not be valid anymore.

On the other hand, a DOCK Buffer entry has the following fields:

- **sender (uint64_t)**

  The unique ID of the node that has pushed to the DOCK the DTDO corresponding to this DOCK Buffer entry

- **CHAIN_ID (uint32_t)**

  The identifier of the CHAIN in which the DTDO corresponding to this DOCK Buffer entry belongs. If CHAIN_ID == 0, then the DTDO does not belong to any CHAIN (i.e., unchained DTDO)

- **CHAIN_pointer (uint32_t)**

  When CHAIN_ID $\neq 0$, CHAIN_pointer is the sequence number or order of this DTDO inside its CHAIN. Otherwise, CHAIN_pointer is a key number identifying the `pushDTDO` call that pushed this DOCK Buffer entry into the DOCK

- **timestamp (struct timeval)**

  The time instant when this entry was inserted to the DOCK (only needed when CHAIN_ID $\neq 0$)

· **trueDTDO (DTDO)**

The actual DTDO carried by this DOCK Buffer entry and the one to be returned when the application picks up from this DOCK

When a new entry has to be pushed to a DOCK Buffer (i.e., a DTDO has to be pushed to a DOCK), first of all we will have to check if there is enough space for this new entry in this DOCK Buffer, since DOCK Buffers have a maximum size of MAX_DOCKBUFFERSIZE or MAX_DOCKDISKSIZE (depending on if we are caching at RAM memory or at disk). Afterwards, if there is enough space, we will try to cache this new entry in the DOCK Buffer in the same way that new DTDOs are cached when they are created (see appendix "C.1 DTDO Creation").

Afterwards, if the new entry can be cached, in the case that this new entry corresponds to a chained DTDO (entry.CHAIN_ID≠0), we will check if a RXCHAIN Table entry corresponding to the chain "entry.sender,entry.CHAIN_ID" already exists in the RXCHAIN Table:

- If the CHAIN does not exist, we will create a new entry in the RXCHAIN Table.
- If this CHAIN already exists, we will check if the DOCK_ID of the RXCHAIN entry is equal to the DOCK_ID of the DOCK of this DOCK Buffer, and in the case that they are not equal, we will discard the DOCK Buffer entry (since it is not allowed to push different DTDOs of the same CHAIN to different DOCKs).

Once all checks are ok, we will proceed to insert the entry to the DOCK Buffer (and to the DOCK). A DOCK Buffer can be seen as a particular type of FIFO queue: we will maintain the entries of the DOCK Buffer ordered by arrival time between different senders and by CHAIN_pointer between entries of the same sender and CHAIN.

On the other hand, when an entry has to be picked up from a DOCK Buffer, we will start checking if the first entry can be picked up and we will keep checking entries until we found the first one that can be picked up or until we discover that no entry can be picked up at this moment.

For each entry, and starting from the first one, we will check if it belongs to a CHAIN or not:

- If it does not belong to a CHAIN, we can pick up the entry
- If it belongs to a CHAIN, then for sure there is an entry in the RXCHAIN Table corresponding to this CHAIN. We will check if the CHAIN_pointer of the entry is the current_CHAIN_pointer of the RXCHAIN entry:
    - In this case, we can pick up the entry
    - If the CHAIN_pointer of the entry is greater (it cannot be lower) than the current_CHAIN_pointer of the RXCHAIN entry, then we will check the CHAIN_resistance of the RXCHAIN entry:
        - If the CHAIN_resistance has expired ("now" > the timestamp of the DOCKBuffer entry + the CHAIN_resistance of the RXCHAIN entry) we can pick up the entry
        - If it has not expired, then we skip this entry and we will check the next one, until one entry valid to be picked up is found or until no more

entries are found (which in this case, the result of the pick up will be NULL)

B.3 CHAIN Table

Each entry of the CHAIN Table corresponds to a specific CHAIN of an application. A CHAIN has the following fields:

· **CHAIN_ID (uint32_t)**

The identifier of the CHAIN in the scope of the sender node

· **CHAIN_resistance (uint32_t)**

The amount of time in milliseconds that, in the case of having a gap in the CHAIN, we will wait for the missed DTDOs.

· **CHAIN_pointer (uint32_t)**

The current CHAIN_pointer to use with the next DTDO to push using this CHAIN

The key field of this table is the CHAIN_ID, meaning that there cannot be two entries with the same CHAIN_ID field. The CHAIN_ID will be used by the applications for identifying CHAINs of DTDOs. The CHAIN_ID will be implemented as a cyclic sequence number maintained by DELTOYA. The scope of the uniqueness of this field is the application that creates the CHAIN. We must ensure that applications, using the CHAIN_ID, can only push packets using CHAINs created by them, and not for other applications of the same node.

CHAINs do not have timeout and they have to be deleted from the CHAIN Table explicitly by the application using the `destroyCHAIN` call.

CHAIN_ID (uint32_t) field can suffer overflow problems, meaning that if 2^32 values are used at the same time, no more values will be able to be used. If this occurs, DELTOYA must forbid creating new CHAINs if old ones are not deleted before.

B.3.1 RXCHAIN

A RXCHAIN Table entry has the following fields:

· **sender (uint64_t)**

The unique ID of the node owning the CHAIN

· **CHAIN_ID (uint32_t)**

The identifier of the CHAIN in the scope of the sender node

· **DOCK_ID (uint32_t)**

The identifier of the DOCK where the DTDOs of this CHAIN are pushed. It is not allowed to push different DTDOs of the same CHAIN to different DOCKs

· **CHAIN_resistance (uint32_t)**

The amount of time in milliseconds that, in the case of having a gap in the CHAIN, we will wait for the missed DTDOs

· **current_CHAIN_pointer (uint32_t)**

The current expected CHAIN_pointer to pick up from the DOCK for this CHAIN

· **historyOfBrokenDTDOs (vector of uint32_t)**

A vector containing the CHAIN_pointers of the last MAXSIZE_HISTORYRXCHAIN DTDOs that really have broken this CHAIN

The key fields of this table are the sender and the CHAIN_ID, meaning that there cannot be two entries with the same sender and CHAIN_ID fields.

RXCHAIN Table entries have a timeout of TOUT_RXCHAIN milliseconds, meaning that if during this period of time no DTDO of this CHAIN is in the DOCK (i.e, in the DOCKBuffer of the DOCK identified by DOCK_ID), then the RXCHAIN Table entry will be deleted.

B.3.2 RXUNCHAIN

A RXUNCHAIN Table entry has the following fields:

· **sender (uint64_t)**

The unique ID of the node calling the `pushDTDO` call

· **CHAIN_pointer (uint32_t)**

The key number identifying the `pushDTDO` call that caused the creation of this RXUNCAIN Table

The key fields of this table are the sender and the CHAIN_pointer, meaning that there cannot be two entries with the same sender and CHAIN_pointer fields.

RXUNCHAIN Table entries do not have a timeout. Otherwise, we will set the maximum size of the RXUNCHAIN Table to MAXSIZE_RXUNCHAIN entries, and once the maximum size is reached, we will keep replacing the oldest entries with the new ones.

Appendix C. DELTOYA Details

C.1 DTDO Creation

DELTOYA provides a method to create DTDOs:

```
int8_t err = createDTDO(id,pluginID,pluginTag,data,dataSize,dataTTL);
```

Params (as defined in section 5.1):

- `id`

  output parameter

- `pluginID, pluginTag, data, dataSize, dataTTL`

  input parameters

`createDTDO` will check:

- `data` != NULL
- 1 <= `dataSize` <= MAXDATASIZE_DTDO
- MINTTL_DATADTDO <= `dataTTL` <= MAXTTL_DATADTDO
- There cannot exist a DTDO with DTDO_pluginID==`pluginID`, DTDO_pluginTag==`pluginTag` and DTDO_originator==”my ID” in the DTDO Table (note that although the DTDO_timestamp field is part of the key field of a DTDO, it is not checked).
- (when caching at RAM) current used space + `dataSize` <= Total space of the RAM of the device
- (when caching at disk) current used space + `dataSize` <= Total space of the disk of the device
- (when not caching a file) current size of cached data + `dataSize` <= MAXSIZE_DELTOYACACHE
- There are not more unused DTDO_IDs

In case of one of these checks fail, `createDTDO` will return -1,-2,-3,-4,-5, -6, -7 or -8 for each check respectively, or 0 if all checks are OK.

If all checks are OK, `createDTDO` will create an entry in the DTDO Table (sub table of local DTDOs) with:

- DTDO_ID (output parameter) = a new (not used) identifier (starting from 1)
- DTDO_pluginID = `pluginID`
- DTDO_pluginTag = `pluginTag`
- DTDO_originator = “my ID”
- DTDO_timestamp = “now”
- DTDO_dataSize = `dataSize`
- DTDO_dataTTL = `dataTTL`
- DTDO_dataTimeout = not used

- DTDO_useCounter = not used (0)
- DTDO_isPublic = false
- Neither a recipient will be created nor a link established to any already existing recipients

Moreover, caching of `data` will be performed in this way:

- If `data` is a file, we just cache a link to the file (hard disk).
- Else,
  - If `dataSize` <= THRESHOLDDATASIZE_CACHE, we cache a copy of `data` in the RAM memory of the device.
  - Else, we cache a copy of `data` in the hard disk of the device.

C.2 DTDO Modification

DELTOYA provides a method to modify an existing DTDO:

```
int8_t err = modifyDTDO(id,new_id,data,dataSize);
```

Params (as defined in section 5.1):

- `new_id`

  output parameter

- `id, data, dataSize`

  input parameters

`modifyDTDO` will check:

- if exists a DTDO with DTDO_ID==`id` (`id` >0) in the DTDO Table
- Applications only can modify local DTDOs created by them (applications cannot modify neither remote DTDOs nor local DTDOs of other applications)
- `data` != NULL
- 1 <= `dataSize` <= MAXDATASIZE_DTDO
- current used space + `dataSize` - current DTDO_dataSize of the DTDO with DTDO_ID==`id` <= Total space of the device
- current size of cached data + `dataSize` - current DTDO_dataSize of the DTDO with DTDO_ID==`id` <= MAXSIZE_DELTOYACACHE

In case of one of these checks fail, `modifyDTDO` will return -1,-2,-3,-4,-5 or -6 for each check respectively, or 0 if all checks are OK. Moreover, it will return the new DTDO_ID of the modified DTDO (`new_id`).

If all checks are OK, `modifyDTDO` will:

```
DTDO d = lookupDTDOTable(id);
createDTDO(new_id,d.pluginID,d.pluginTag,data,dataSize,
           d.DTDO_dataTTL);
```

```
If (d.DTDO_isPublic)
      publishDTDO(new_id);
EndIf
deleteDTDO(id);
return new_id;
```

C.3 DTDO Deletion

DELTOYA provides a method to delete DTDOs:

```
int8_t err = deleteDTDO(id);
```

`deleteDTDO` will check:

- If exists a DTDO with DTDO_ID==`id` (`id` >0) in the DTDO Table
- Applications only can delete local DTDOs created by them (applications cannot delete neither remote DTDOs nor local DTDOs of other applications)
- If `data` of the DTDO with DTDO_ID==`id` can be deleted at this time.

In case of one of these checks fail, `deleteDTDO` will return -1,-2 or - 3 for each check respectively, or 0 if all checks are OK.

If all checks are OK, `deleteDTDO` will:

- Release the cached data of this DTDO (only if we cached a copy of the DTDO in the time of creation. If it was a file, then we do not delete the file)
- Delete the corresponding entry in the DTDO Table (it cannot have any recipient linked)

Note that as explained in section 5.1, an implicit DTDO deletion can also happen. DTDOs stored in other nodes other than the node that created it (i.e. remote DTDOs) will be automatically deleted once the DTDO_dataTTL has expired and DTDO_useCounter is 0.

C.4 DTDO Publishing

DELTOYA provides a method to publish DTDOs:

```
int8_t err = publishDTDO(id);
```

`publishDTDO` will check:

- If exists a DTDO with DTDO_ID==`id` (`id` >0) in the DTDO Table
- Applications only can modify local DTDOs created by them (applications cannot modify neither remote DTDOs nor local DTDOs of other applications).

In case of this check fails, `publishDTDO` will return -1 or -2 for each check respectively, or 0 if all checks are OK.

If all checks are OK, `publishDTDO` will:

- DTDO_isPublic = true

From now on, this DTDO will be accessible to the rest of the applications of the network (or even to the other applications of the same node) in case they request for it.

C.5 DTDO Unpublishing

DELTOYA provides a method to unpublish DTDOs:

```
int8_t err = unpublishDTDO(id);
```

`unpublishDTDO` will check:

- If exists a DTDO with DTDO_ID==`id` (`id` >0) in the DTDO Table
- Applications only can modify local DTDOs created by them (applications cannot modify neither remote DTDOs nor local DTDOs of other applications).

In case of this check fails, `unpublishDTDO` will return -1 or -2 for each check respectively, or 0 if all checks are OK.

If all checks are OK, `unpublishDTDO` will:

- DTDO_isPublic = false

From now on, this DTDO will not be accessible to the rest of the applications of the network (or to the other applications of the same node) in case they request for it.

C.6 DTDO Querying

C.6.1 DTDO Query Queue (DTDOQQ)

The DTDO Query Queue (DTDOQQ) is the data structure used to buffer all the `queryDTDOs` calls done in a node. The DTDOQQ will be the responsible of trying to discover all the queried DTDOs (from the cache or from the network, and always using a DTDODM (see section 9)), and at the same time, the responsible of executing the `queryDTDOsCallback` when recipient is specified and a reply from this recipient is received, or in T_QUERYDTDOS milliseconds after the `queryDTDOs` was called, when recipient is not specified or recipient is specified but the reply is not received.

Each entry of the DTDOQQ identifies a `queryDTDOs` call, and it has the following fields:

· **pluginID, pluginTag, originator (uint32_t, uint32_t, uint64_t)**

The DTDOs to discover. pluginID will be ≠ NULL, and pluginTag and originator can be NULL or not (i.e., any pluginTag or any originator)

· **recipient (uint64_t)**

The unique ID of recipient from which try to discover the DTDOs. Recipient can be NULL (i.e. any recipient) or not

· **callbackInformation:**

pluginID, list{pluginTag,originator,timestamp}, idCallback

(uint32_t, list{uint32_t,uint64_t,struct timeval}, int32_t)

All the information needed to keep track of the results of the querying process (e.g., DTDOs discovered during the query period). This information will be the one to be returned in the queryDTDOsCallback.

Moreover, the idCallback of the queryDTDOs corresponding to this callback will also be returned.

The DTDOQQ does not have key fields, meaning that there can be two entries with the same pluginID, pluginTag, originator and recipient fields. When the same queryDTDOs call is called twice, then two processes for querying the same DTDOs will be started. However, each entry will have a unique idCallback which identifies the entry and that will be managed automatically (as a cyclic a sequence number) by the DTDOQQ itself.

During T_QUERYDTDOS milliseconds, the DTDOQQ entry will try to discover the queried DTDOs once the queryDTDOs call is started. In other words, during T_QUERYDTDOS milliseconds, the DTDODM will keep sending DTDODREQ messages using a Binary Exponential Backoff until a maximum of MAX_DTDODREQSS have been sent. After having sent MAX_DTDODREQSS DTDODREQ messages, then the Binary Exponential Backoff will automatically be restarted as if a new Binary Exponential Backoff was started (i.e. $DTDODREQ_{MAX\_DTDODREQSS}$ == $DTDODREQ_1$). See an example in Figure 91.

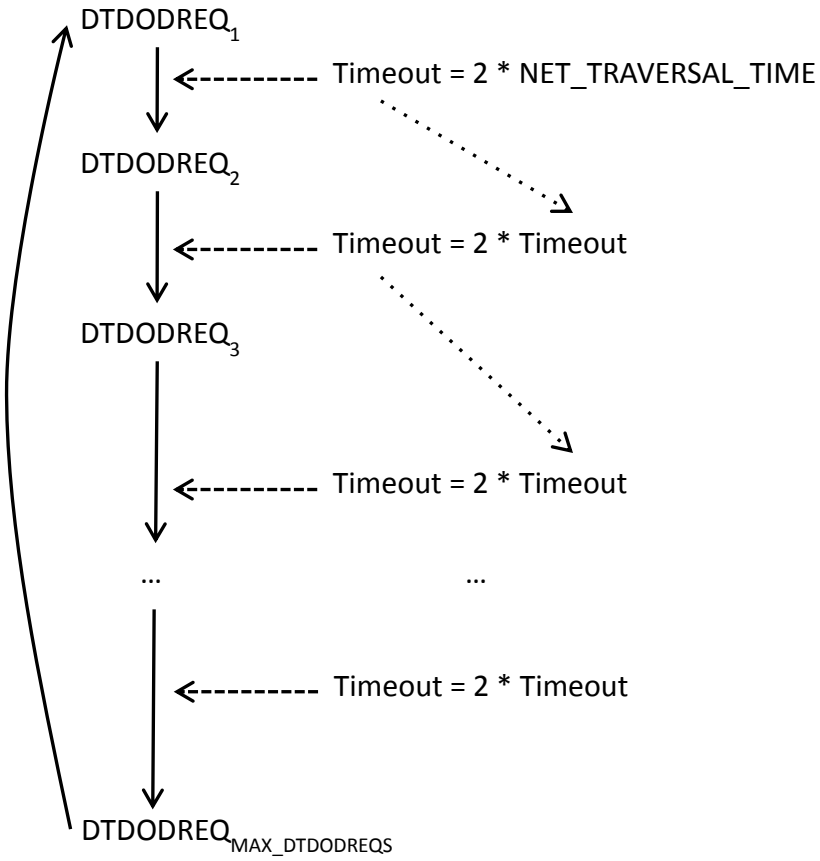

Figure 91. DTDO Querying Backoff

After a reply is received when the recipient is specified or after T_QUERYDTDOS milliseconds, the DTDOQQ entry will stop trying to discover more DTDOs, and the `queryDTDOsCallback` will be executed with the results obtained in the callbackInformation. Moreover, the DTDOQQ entry will be automatically deleted.

C.6.2 Querying DTDOs

DELTOYA provides a method to query DTDOs.

```
int32_t idCallback =

queryDTDOs(pluginID,pluginTag,originator,recipient);
```

Params (as defined in section 5.1):

- `pluginID`

  To identify the application where the DTDOs we want to obtain were created

- `pluginTag`

  To identify the DTDOs within the application where they were created. It can be NULL or not. If NULL, this field will not be used to decide if an specific DTDO has to be discovered or not, meaning that it will be discovered independently of its DTDO_pluginTag value and only depending on the other parameters which identify the DTDO (i.e. NULL value can be seen as '*' in a regular expression)

- `originator`

  Unique ID to identify the node where the DTDOs we want to discover were created. It can be empty or not. If empty, this field will not be used to decide if an specific DTDO has to be discovered or not, meaning that it will be discovered independently of its DTDO_originator value and only depending on the other parameters which identify the DTDO (i.e. empty can be seen as '*' in a regular expression)

- `recipient`

  Unique ID to identify the node where the DTDOs we want to discover have to be residing. It can be empty or not. If empty, this field will not be used to decide if the DTDOs have to be discovered or not from an specific recipient, meaning that they will be discovered independently of the recipient nodes that can contain the DTDOs (i.e. empty can be seen as '*' in a regular expression)

Table 15 shows all the possible `queryDTDOs` calls and their meaning.

Table 15. queryDTDOs calls

Request				Description
`pluginID`	`pluginTag`	`orig`	`recip`	We are querying DTDOs with DTDO_pluginID=`pluginID`,

				DTDO_pluginTag=`pluginTag` and DTDO_originator=`orig` (and independently of the DTDO_timestamp), and moreover specifically stored in the node `recip`.
`pluginID`	`pluginTag`	`orig`	*	Idem but they could be stored in any recipient.
`pluginID`	`pluginTag`	*	`recip`	We are querying DTDOs with DTDO_pluginID=`pluginID` and DTDO_pluginTag=`pluginTag` (and independently of the DTDO_originator and DTDO_timestamp), and moreover specifically stored in the node `recip`.
`pluginID`	`pluginTag`	*	*	Idem but they could be stored in any recipient.
`pluginID`	*	`orig`	`recip`	We are querying DTDOs with DTDO_pluginID=`pluginID` and DTDO_originator=`orig` (and independently of the DTDO_pluginTag and DTDO_timestamp), and moreover specifically stored in the node `recip`.
`pluginID`	*	`orig`	*	Idem but they could be stored in any recipient.
`pluginID`	*	*	`recip`	We are querying DTDOs with DTDO_pluginID=`pluginID` (and independently of the DTDO_pluginTag, DTDO_originator and DTDO_timestamp), and moreover specifically stored in the node `recip`.
`pluginID`	*	*	*	Idem but they could be stored in any recipient.

When a `queryDTDOs` is called, it will return `idCallback`. `idCallback` is the identifier of the query, and it can be used to cancel the query or to identify to which query corresponds each callback.

`queryDTDOs` will perform the actions needed to discover the queried DTDOs, independently if they are locally cached or if they have to be discovered through the network.

Once `queryDTDOs` is called, and after a reply is received when recipient is specified or after T_QUERYDTDOS milliseconds, a callback with the result of the requesting operation will be executed:

```
void queryDTDOsCallback(idCallback,pluginID,list{pluginTag,originator,
timestamp});
```

where `idCallback` is the identifier of the query that has caused this callback and `pluginID,list{pluginTag,originator,timestamp}` is the result of the querying operation, i.e., the list of discovered DTDOs in the network during the period of time when the `queryDTDOs` has been called. In the case of receiving different versions of the same DTDO (same pluginID, pluginTag and originator, but different timestamp), then only the newest version (more recent timesatamp) will be returned.

With all these assumptions, the operation of the `queryDTDOs` is as follows:

```
We create a new entry in the DTDOQQ for this queryDTDOs.

If (recipient==NULL)

      If ((pluginTag!=NULL) and (originator!=NULL))

            If (originator == "my ID")

                  DELTOYA checks if there exist some public
                  DTDOs=={pluginID,pluginTag,originator,*} in the sub
                  table of the local DTDOs (DTDO_data will always be
                  != NULL)

                  If (some found)

                        DTDOs discovered ready to be added to the
                        returning list of the callback.

                  EndIf

                  Now the callback for the queryDTDOs is ready to be
                  executed with all the DTDOs that have been
                  discovered from the cache

            Else

                  DELTOYA checks if there exist some public
                  DTDOs=={pluginID,pluginTag,originator,*} in the sub
                  table of the remote DTDOs with DTDO_data!=NULL

                  If (some found)

                        DTDOs discovered ready to be added to the
                        returning list of the callback.

                  EndIf

                  startDTDODM(pluginID,pluginTag,originator,*,
                  T_QUERYDTDOS,0)

                  Once T_QUERYDTDOS expires, the callback for the
                  queryDTDOs will be executed with all the DTDOs that
                  have been discovered (from the cache or from the
                  network during the DTDODM)

            EndIf

      ElseIf ((pluginTag==NULL) and (originator!=NULL))
```

Idem than the previous ElseIf but with DTDOs=={pluginID,*,originator,*} and **startDTDODM(pluginID,*,originator,*,T_QUERYDTDOS,0)**

ElseIf ((pluginTag!=NULL) and (originator==NULL))

DELTOYA checks if there exist some public DTDOs=={pluginID,pluginTag,*,*} in the sub table of the local DTDOs (DTDO_data will always be != NULL)

If (some found)

DTDOs discovered ready to be added to the returning list of the callback.

EndIf

DELTOYA checks if there exist some public DTDOs=={pluginID,pluginTag,*,*} in the sub table of the remote DTDOs with DTDO_data!=NULL

If (some found)

DTDOs discovered ready to be added to the returning list of the callback.

EndIf

startDTDODM(pluginID,pluginTag,*,*,T_QUERYDTDOS,0)

Once T_QUERYDTDOS expires, the callback for the queryDTDOs will be executed with all the DTDOs that have been discovered (from the cache or from the network during the DTDODM)

ElseIf ((pluginTag==NULL) and (originator==NULL)

Idem than the previous ElseIf but with DTDOs=={pluginID,*,*,*} and **startDTDODM(pluginID,*,*,*,T_QUERYDTDOS,0)**

EndIf

ElseIf (recipient == "my ID")

If ((pluginTag!=NULL) and (originator!=NULL))

If (originator == "my ID")

DELTOYA checks if there exist some public DTDOs=={pluginID,pluginTag,originator,*} in the sub table of the local DTDOs (DTDO_data will always be != NULL)

If (some found)

DTDOs discovered ready to be added to the returning list of the callback.

EndIf

Else

DELTOYA checks if there exist some public DTDOs=={pluginID,pluginTag,originator,*} in the sub table of the remote DTDOs with DTDO_data!=NULL

If (some found)

DTDOs discovered ready to be added to the returning list of the callback.

EndIf

EndIf

Now the callback for the queryDTDOs is ready to be executed with all the DTDOs that have been discovered from the cache

ElseIf ((pluginTag==NULL) and (originator!=NULL)

Idem than the previous ElseIf but with DTDOs=={pluginID,*,originator,*}

ElseIf ((pluginTag!=NULL) and (originator==NULL))

DELTOYA checks if there exist some public DTDOs=={pluginID,pluginTag,*,*} in the sub table of the local DTDOs (DTDO_data will always be != NULL)

If (some found)

DTDOs discovered ready to be added to the returning list of the callback.

EndIf

DELTOYA checks if there exist some public DTDOs=={pluginID,pluginTag,*,*} in the sub table of the remote DTDOs with DTDO_data!=NULL

If (some found)

DTDOs discovered ready to be added to the returning list of the callback.

EndIf

Now the callback for the queryDTDOs is ready to be executed with all the DTDOs that have been discovered from the cache

ElseIf ((pluginTag==NULL) and (originator==NULL))

Idem than the previous ElseIf but with DTDOs=={pluginID,*,*,*}

EndIf

Else

If ((pluginTag!=NULL) and (originator!=NULL))

startDTDODM(recipient,pluginID,pluginTag,originator,*, T_QUERYDTDOS,0)

```
            Once T_QUERYDTDOS expires or a reply is received:

            If (originator == "my ID")

                  The callback for the queryDTDOs will be executed
                  only with the DTDOs that have been discovered from
                  the network during the DTDODM and that are also
                  available in the DTDO Table of the node as public
                  local DTDOs

            Else

                  The callback for the queryDTDOs will be executed
                  with all the DTDOs that have been discovered (from
                  the network during the DTDODM)

            EndIf

      ElseIf ((pluginTag==NULL) and (originator!=NULL))

            Idem than the previous ElseIf but with
            startDTDODM(recipients,pluginID,*,originator,*,
            T_QUERYDTDOS,0)

      ElseIf ((pluginTag!=NULL) and (originator==NULL))

            startDTDODM(recipient,pluginID,pluginTag,*,*,T_QUERYDTDOS,
            0)

            Once T_QUERYDTDOS expires or a reply is received, the
            callback for the queryDTDOs will be executed with all the
            DTDOs that have been discovered (from the network during
            the DTDODM)

      ElseIf ((pluginTag==NULL) and (originator==NULL))

            Idem than the previous ElseIf but with
            startDTDODM(recipients,pluginID,*,*,*,T_QUERYDTDOS,0)

      EndIf

EndIf
```

C.6.3 Cancelling DTDO Queries

DELTOYA provides a method to cancel pending `queryDTDOs` calls:

```
int8_t err = cancelQueryDTDOs(idCallback);
```

`cancelQueryDTDOs` will check:

- If exists an entry in the DTDOQQ with idCallback==`idCallback`.

In case of this check fails, `cancelQueryDTDOs` will return -1, or 0 if all checks are OK.

If all checks are OK, `cancelQueryDTDOs` will:

- Cancel the DTDODM corresponding to this DTDOQQ entry
- Delete the DTDOQQ entry

From now on, the callback corresponding to the cancelled `queryDTDOs` will not be executed anymore.

C.7 DTDO Requesting

C.7.1 DTDO Request Queue (DTDORQ)

The DTDO Request Queue (DTDORQ) is the data structure used to buffer all the `requestDTDOs` calls done in a node. The DTDORQ will be the responsible of trying to obtain all the requested DTDOs (from the cache or from the network, and using a DTDO Discovery Mechanism or not), and at the same time, the responsible of executing the `requestDTDOsCallback` when all DTDOs are obtained or MRT expires.

Each entry of the DTDORQ identifies a `requestDTDOs` call, and it has the following fields:

- **pluginID, list{pluginTag, originator, timestamp} (uint32_t, list{uint32_t, uint64_t, struct timeval })**

  The DTDOs to discover. pluginID will be ≠ NULL, list.size will be > 0, and pluginTag, originator and timestamp will be all ≠ NULL for each entry of the list

- **MRT (uint32_t)**

  The Maximum Retrieval Time is the maximum amount of time in milliseconds during which DELTOYA will try to obtain the requested DTDOs once the `requestDTDOs` call is started. In other words, the time instant until the DTDODM corresponding to this DTDORQ entry can keep sending DTDODREQ messages (i.e. the DTDODM can keep going on), using a Binary Exponential Backoff (limited until a maximum of MAX_DTDODREQSS have been sent. After having sent MAX_DTDODREQSS DTDODREQ messages, then the Binary Exponential Backoff will automatically be restarted as if a new Binary Exponential Backoff was started (i.e. $DTDODREQ_{MAX\_DTDODREQSS} == DTDODREQ_1$)). See an example in Figure 92.

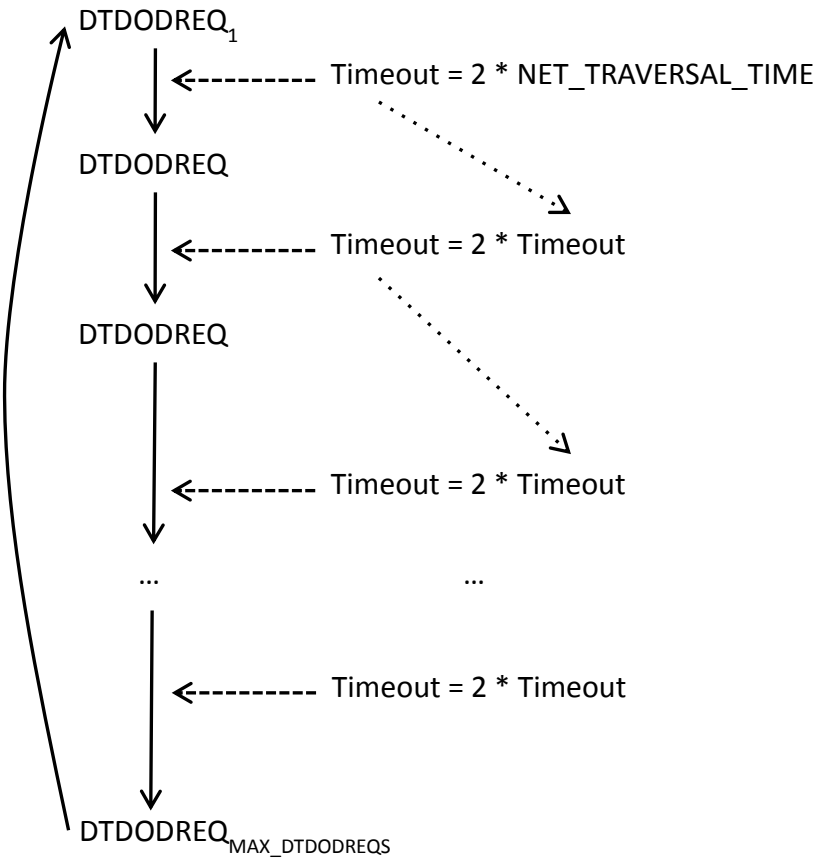

Figure 92. DTDO Requesting Backoff

If not all the requested DTDOs are obtained, after MRT milliseconds DELTOYA will stop trying to obtain them, and the `requestDTDOsCallback` will be executed with the partial results obtained. Moreover, the entry in the DTDORQ will be automatically deleted.

- **callbackInformation:**

  pluginID,list{pluginTag,originator,timestamp,data,dataSize,status}, idCallback

  (uint32_t,list of {uint32_t,uint64_t,

  struct timeval,void*,uint32_t,uint8_t},int32_t)

  All the information needed to keep track of the intermediate results of the requesting process (e.g., DTDOs already obtained during the MRT period). When all DTDOs are obtained, or when MRT expires, this information will be the one to be returned in the `requestDTDOsCallback`. This information includes a field to save a copy of the DTDO_data of the obtained DTDOs (and this copy will be the one to be returned in the callback). The fact of saving a copy of the DTDO_data in the DTDORQ entry immediately once the DTDO_data is obtained (from the cache or from the network) is because in this way DELTOYA avoids the problem of obtaining a DTDO_data, saving it in the

cache, but then, being deleted and so, being lost it before the callback of a `requestDTDOs` that wanted to use this DTDO_data is executed.

Moreover, for each DTDO_data obtained from a remote DTDO (from the cache or from the network), the DTDO_useCounter field of the DTDO entry in the DTDO Table will be immediately decremented by one (this will let to delete the DTDO entry of the DTDO Table as soon as possible). Moreover, in the case that MRT expires due to not all DTDOs have been obtained, the DTDO_useCounter field of all the DTDO entries of the DTDOs that have not been able to be obtained will also be decremented by one.

Moreover, the idCallback of the `requestDTDOs` corresponding to this callback will also be returned.

The DTDORQ does not have key fields, meaning that there can be two entries with the same pluginID and list{pluginTag, originator, timestamp} fields. When the same `requestDTDOs` call is called twice, then two processes for requesting the same DTDOs will be started. However, each entry will have a unique idCallback which identifies the entry and that will be managed automatically (e.g. cyclic a sequence number) by the DTDORQ itself.

For a DTDORQ entry, when all DTDOs are obtained, or when MRT expires, the `requestDTDOsCallback` corresponding to the `requestDTDOs` of this entry will be called, and the callbackInformation of the entry will be returned in the `requestDTDOsCallback`. Moreover, the entry will be automatically deleted.

C.7.2 Requesting specific DTDOs

DELTOYA provides a method for requesting a set of specific DTDOs (i.e., DTDOs specified by their pluginID, pluginTag, timestamp and originator), and independently of their recipient.

```
int32_t                          idCallback                          =
requestDTDOs(pluginID,list{pluginTag,originator,timestamp},MRT);
```

Params (as defined in section 5.1):

- `pluginID`

  To identify the application where the DTDOs we want to obtain were created

- `pluginTag`

  To identify the DTDO within the application where it was created

- `originator`

  Unique ID to identify the node where the DTDO we want to obtain was created

- `timestamp`

  To identify the timestamp of the DTDO we want to obtain when it was created

- `MRT (uint32_t)`

The Maximum Retrieval Time is the amount of time in milliseconds during which DELTOYA will try to obtain the requested DTDOs once the `requestDTDOs` call is started. After MRT milliseconds, even if not all the requested DTDOs are obtained, DELTOYA will stop trying to obtain them.

When the `requestDTDOs` is called, it will check:

- MIN_MRT <= `MRT` <= MAX_MRT
- `list{pluginTag,originator,timestamp}`.size > 0

In case of these checks fail, `requestDTDOs` will return -1 or -2 respectively, or `idCallback` if all checks are OK. `idCallback` is the identifier of the request, implemented as a cyclic sequence number, and it can be used to cancel the request or to identify to which request corresponds each callback.

If all checks are OK, `requestDTDOs` will perform the actions needed to obtain a copy of the requested DTDOs, independently if they are locally cached or if they have to be obtained through the network, and when all the requested DTDOs are obtained, or MRT has expired, a callback with the result of the requesting operation will be executed:

```
void
requestDTDOsCallback(idCallback,pluginID,list{pluginTag,originator,
timestamp,data,dataSize,status});
```

where `idCallback` is the identifier of the request that has caused this callback and `pluginID,list{pluginTag,originator,timestamp,data,dataSize,status}` is the result of the requesting operation. For each DTDO in the list, the `status` (`OK`=DTDO fully obtained, `KO`=DTDO not obtained, `NotFound`=DTDO not found in the network) of the DTDO after the request will be returned. And if `status`==OK, a copy of the data of the DTDO (`data,dataSize`) will also be returned.

With all these assumptions, the operation of the `requestDTDOs` is as follows:

```
Create an entry in the DTDO Request Queue (DTDORQ) for this
requestDTDOs.

listDTDOsToDiscover := empty

For each i-pluginTag,i-originator,i-timestamp of
list{pluginTag,originator,timestamp}

      If (i-originator == "my ID")

            DELTOYA checks if there exists a public DTDO={pluginID,i-
            pluginTag,i-originator,i-timestamp} in the sub table of
            the local DTDOs (DTDO_data will always be != NULL)

            If (found)

                  DTDO cached ready to be added (copied) to the
                  returning list of the callback.

            Else
```

```
                DTDO ready to be returned as not found
                (status=NotFound)

            EndIf

        Else

            DELTOYA checks if there exists a public DTDO={pluginID,i-
            pluginTag,i-originator,i-timestamp} in the sub table of
            the remote DTDOs.

            If (found with DTDO_data != NULL)

                DTDO cached ready to be added (copied) to the
                returning list of the callback.

            ElseIf (found with DTDO_data == NULL)

                DELTOYA checks if for this remote DTDO there are one
                or more recipients (i.e. links to destination nodes)
                in the DTDO Table.

                If (there is route)

                    Increment by 1 the DTDO_useCounter of this
                    remote DTDO (in this case, if the DTDODM
                    discovers this DTDO in the future, the DTDODM
                    will not increment the DTDO_useCounter of the
                    DTDO, even in the first time it discovers the
                    DTDO, since we have already incremented it
                    here. This is for assuring that PTP has the
                    lock of this DTDO even if the DTDODM does not
                    discovers this DTDO)

                    listDTDOsToDiscover.add(pluginID,
                    i-pluginTag,i-originator,i-timestamp)

                    [PTP] getDTDO(pluginID,i-pluginTag,
                          i-originator,i-timestamp,a pointer to the
                          DTDORQ entry,QoS)

                Else

                    listDTDOsToDiscover.add(pluginID,
                    i-pluginTag,i-originator,i-timestamp)

                EndIf

            ElseIf (no found)

                listDTDOsToDiscover.add(pluginID,i-pluginTag,
                i-originator,i-timestamp)

            EndIf

        EndIf

EndFor

If (listDTDOsToDiscover is not empty)

        startDTDODM(listDTDOsToDiscover,MRT,1)

EndIf
```

Once all requests to PTP finish or MRT expires, the callback will be executed with all the DTDOs which have been obtained (from the cache or from the network) (status=OK), with all the DTDOs which have been found but which have not been able to obtain (status=KO), and with all the DTDOs which have not been found (status=NotFound). Moreover, if there are one or more DTDOs with status==KO, **[PTP]** stopGetDTDO(a pointer to the DTDORQ entry) will be called to notify PTP to stop trying to obtain these DTDOs.

NOTE: it is important to bear in mind that given all the requests to PTP that can be queued and running in parallel, it has to be possible to identify to which requestDTDOss (one or more) correspond each request of PTP, in order to deliver the DTDOs obtained by PTP to the correct requests of DELTOYA and after this, to the correct requests of the applications (and the same for the DTDOs obtained directly from the cache). For this purpose, the DTDORQ (appendix "C.7.1 DTDO Request Queue (DTDORQ)") and the PTPRQ (appendix "D.2 PTP Request Queue (PTPRQ)")) will be used.

C.7.3 Requesting generic DTDOs

DELTOYA provides a method for requesting a set of generic or non-specific DTDOs (i.e., DTDOs not specified by all their pluginID, pluginTag, timestamp and originator key fields), and independently of their recipient.

```
int32_t idCallback = requestDTDOs(pluginID,pluginTag,originator,MRT);
```

Params (as defined in section 5.1):

- pluginID

  To identify the application where the DTDOs we want to obtain were created. It cannot be NULL

- pluginTag

  To identify the DTDO within the application where it was created. It can be NULL or not. If NULL, this field will not be used to decide if an specific DTDO has to be obtained or not, meaning that it will be obtained independently of its DTDO_pluginTag value and only depending on the other parameters which identify the DTDO (i.e. NULL value can be seen as '*' in a regular expression)

- originator

  Unique ID to identify the node where the DTDO we want to obtain was created. It can be empty or not. If empty, this field will not be used to decide if an specific DTDO has to be obtained or not, meaning that it will be obtained independently of its DTDO_originator value and only depending on the other parameters which identify the DTDO (i.e. empty can be seen as '*' in a regular expression)

- MRT (uint32_t)

  The Maximum Retrieval Time is the amount of time in milliseconds during which DELTOYA will try to obtain the requested DTDOs discovered during the first T_QUERYDTDOS milliseconds after the requestDTDOs call is called. After MRT

milliseconds, even if not all these requested DTDOs are obtained, DELTOYA will stop trying to obtain them.

Table 16 shows all the possible `requestDTDOs` calls and their meaning.

Table 16. requestDTDOs calls

Request			Description
`pluginID`	`pluginTag`	`orig`	We are querying DTDOs with DTDO_pluginID=`pluginID`, DTDO_pluginTag=`pluginTag` and DTDO_originator=`orig` (and independently of the DTDO_timestamp), and moreover stored in any recipient.
`pluginID`	`pluginTag`	*	We are querying DTDOs with DTDO_pluginID=`pluginID` and DTDO_pluginTag=`pluginTag` (and independently of the DTDO_originator and DTDO_timestamp), and moreover stored in any recipient.
`pluginID`	*	`orig`	We are querying DTDOs with DTDO_pluginID=`pluginID` and DTDO_originator=`orig` (and independently of the DTDO_pluginTag and DTDO_timestamp), and moreover stored in any recipient.
`pluginID`	*	*	We are querying DTDOs with DTDO_pluginID=`pluginID` (and independently of the DTDO_pluginTag, DTDO_originator and DTDO_timestamp), and moreover stored in any recipient.

When a `requestDTDOs` is called, it will check:

- MIN_MRT <= `MRT` <= MAX_MRT

In case of this check fails, `requestDTDOs` will return -1, or `idCallback` if all checks are OK. `idCallback` is the identifier of the request, implemented as a cyclic sequence number, and it can be used to cancel the request or to identify to which request corresponds each callback.

If all checks are OK, `requestDTDOs` will perform the actions needed to obtain a copy of the requested DTDOs (the ones discovered only during the first T_QUERYDTDOS milliseconds after the `requestDTDOs` call is called), independently if they are locally cached or they have to be obtained through the network, and when all the requested DTDOs are obtained, or MRT has expired, a callback with the result of the requesting operation will be executed:

```
void
requestDTDOsCallback(idCallback,pluginID,list{pluginTag,originator,
timestamp,data,dataSize,status});
```

where `idCallback` is the identifier of the request that has caused this callback and `pluginID,list{pluginTag,originator,timestamp,data,dataSize,status}` is the result of the requesting operation. For each DTDO in the list, the `status` (`OK`=DTDO fully obtained, `KO`=DTDO not obtained) of the DTDO after the request will be returned. And if `status`==OK, a copy of the data of the DTDO (`data,dataSize`) will also be returned.

With all these assumptions, the operation of the `requestDTDOs` is as follows:

```
We call queryDTDOs to make a photo:

queryDTDOs(pluginID,pluginTag,originator,*)


We wait the callback of the photo:

queryDTDOsCallback(l = list{pluginID,pluginTag,originator,timestamp})


And now, we can start a requestDTDOs of the DTDOs discovered in the
previous query, as if it was a request of specific DTDOs but without
the case of status==NotFound and with a DTDODM.MRT = MRT-T_QUERYDTDOS:

Create an entry in the DTDO Request Queue (DTDORQ) for this
requestDTDOs.

listDTDOsToDiscover := empty

For each i-pluginTag,i-originator,i-timestamp of l

      If (i-originator == "my ID")

            DELTOYA checks if there exists a public DTDO={pluginID,i-
            pluginTag,i-originator,i-timestamp} in the sub table of
            the local DTDOs (DTDO_data will always be != NULL)

            If (found)

                  DTDO cached ready to be added (copied) to the
                  returning list of the callback.

            EndIf

      Else

            DELTOYA checks if there exists a public DTDO={pluginID,i-
            pluginTag,i-originator,i-timestamp} in the sub table of
            the remote DTDOs.

            If (found with DTDO_data != NULL)

                  DTDO cached ready to be added (copied) to the
                  returning list of the callback.

            ElseIf (found with DTDO_data == NULL)
```

```
                DELTOYA checks if for this remote DTDO there are one
                or more recipients (i.e. links to destination nodes)
                in the DTDO Table.

                If (there is route)

                      Increment by 1 the DTDO_useCounter of this
                      remote DTDO (in this case, if the DTDODM
                      discovers this DTDO in the future, the DTDODM
                      will not increment the DTDO_useCounter of the
                      DTDO, even in the first time it discovers the
                      DTDO, since we have already incremented it
                      here. This is for assuring that PTP has the
                      lock of this DTDO even if the DTDODM does not
                      discovers this DTDO)

                      listDTDOsToDiscover.add(pluginID,
                            i-pluginTag,i-originator,i-timestamp)

                      [PTP] getDTDO(pluginID,i-pluginTag,
                      i-originator,i-timestamp,a pointer to the
                      DTDORQ entry,QoS)

                Else

                      listDTDOsToDiscover.add(pluginID,
                      i-pluginTag,i-originator,i-timestamp)

                EndIf

          ElseIf (no found)

                listDTDOsToDiscover.add(pluginID,i-pluginTag,
                i-originator,i-timestamp)

          EndIf

     EndIf

EndFor

If (listDTDOsToDiscover is not empty)

     startDTDODM(listDTDOsToDiscover,MRT - T_QUERYDTDOS,1)

EndIf
```

Once all requests to PTP finish or MRT expires, the callback will be executed with all the DTDOs which have been obtained (from the cache or from the network) (status=OK) and with all the DTDOs which have been found but which have not been able to obtain (status=KO). Moreover, if there are one or more DTDOs with status==KO, **[PTP]** stopGetDTDO(a pointer to the DTDORQ entry) will be called to notify PTP to stop trying to obtain these DTDOs.

NOTE: it is important to bear in mind that given all the requests to PTP that can be queued and running in parallel, it has to be possible to identify to which requestDTDOss (one or more) correspond each request of PTP, in order to deliver the DTDOs obtained by PTP to the correct requests of DELTOYA and after this, to the correct requests of the applications (and the same for the DTDOs obtained directly from the cache). For this purpose, the DTDORQ (appendix "C.7.1 DTDO Request

Queue (DTDORQ)") and the PTPRQ (appendix "D.2 PTP Request Queue (PTPRQ)")) will be used.

C.7.4 Cancelling DTDO Requests

DELTOYA provides a method to cancel pending `requestDTDOs` calls:

```
int8_t err = cancelRequestDTDOs(idCallback);
```

`cancelRequestDTDOs` will check:

- If exists an entry in the DTDORQ with idCallback==`idCallback`.

In case of this check fails, `cancelRequestDTDOs` will return -1, or 0 if all checks are OK.

If all checks are OK, `cancelRequestDTDOs` will:

- Cancel the DTDODM corresponding to this DTDORQ entry
- Delete the DTDORQ entry

From now on, the callback corresponding to the cancelled `requestDTDOs` will not be executed anymore.

C.8 DTDO Sending

C.8.1 DTDO Sending Queue (DTDOSQ)

The DTDO Sending Queue (DTDOSQ) is the data structure used to buffer all the `pushDTDO` calls done in a node. The DTDOSQ will be the responsible of trying to push the specific DTDO to the specific DOCK, and at the same time, the responsible of executing the `pushDTDOCallback` when the DTDO is correctly pushed or MPT expires.

Each entry of the DTDOSQ identifies a `pushDTDO` call, and it has the following fields:

· **DTDOid (uint32_t)**

The DTDO_ID of the DTDO to push

· **pluginID, pluginTag, originator, timestamp**

(uint32_t, uint32_t, uint64_t, struct timeval)

The four fields identifying the DOCK where the DTDO DTDOid has to be pushed

· **MPT (uint32_t)**

The Maximum Pushing Time is the maximum amount of time in milliseconds during which DELTOYA will try to push the DTDO DTDOid to the DOCK {pluginID,pluginTag,originator,timestamp} once the `pushDTDO` call is started.

· **CHAINid (uint32_t)**

The CHAIN_ID of the CHAIN where the DTDO DTDOid belongs. If 0, the DTDO DTDOid does not belong to a CHAIN

· **CHAIN_pointer (uint32_t)**

Only valid when CHAINid ≠ 0. The pointer of the DTDO DTDOid inside the CHAIN CHAINid.

· **CHAIN_resistance (uint32_t)**

When CHAIN_ID ≠ 0, the CHAIN_resistance is the amount of time in milliseconds that, in the case of having a gap in the CHAIN, we will wait for the missed DTDOs

· **canRepush (bool)**

Field only valid when the `pushDTDO` corresponding to this DTDOSQ entry is not currently running (i.e., has finished). canRepush is a boolean indicating if the last `pushDTDO` corresponding to this DTDOSQ entry finished with status == 0 or 4 (canRepush == false) or with any other status (canRepush == true). It wil be used to allow retrying a `pushDTDO` (`retryPushDTDO`) only when canRepush is true. When status == 0 (the `pushDTDO` finished with no errors) or when status == 4 (we cannot push this chained DTDO since posterior DTDOs of the same CHAIN have been already read) we will not allow to execute a `retryPushDTDO` for this `pushDTDO`

· **callbackInformation: idCallback, status, entry (int32_t, uint8_t, DTDOSQ_entry)**

This information will be the one to be returned in the `pushDTDOCallback` and includes a `status` field indicating the result of the call and, in the case of status ≠ 0 and status ≠ 4, the data structure corresponding to this DTDOSQ entry, in order to be able to call the `retryPushDTDO` call if desired. Moreover, the `idCallback` of the `pushDTDO` corresponding to this callback will also be returned.

The DTDOSQ does not have key fields. When the same `pushDTDO` call is called twice, then two processes for pushing the same DTDO to the same DOCK will be started. However, each entry will have a unique `idCallback` which identifies the entry and that will be managed automatically as cyclic a sequence number by the DTDOSQ itself.

For a DTDOSQ entry, when the DTDO is finally pushed to the DOCK, or when MPT expires, the `pushDTDOCallback` corresponding to the `pushDTDO` of this entry will be called, and the callbackInformation of the entry will be returned in the `pushDTDOCallback`. Moreover, the entry will be automatically deleted.

C.8.2 DTDO Asking Queue (DTDOAQ)

The DTDO Asking Queue (DTDOAQ) is the data structure used to buffer all the `pushDTDO` calls received in a node. The DTDOAQ will be the responsible of trying to get the specific DTDO to the specific DOCK.

Each entry of the DTDOAQ identifies a `pushDTDO` call received, and it has the following fields:

· **DTDO_pluginID, DTDO_pluginTag, DTDO_originator, DTDO_timestamp, DTDO_dataSize, DTDO_dataTTL, DTDO_isPublic**

(uint32_t, uint32_t, uint64_t, struct timeval, uint32_t, uint32_t, bool)

The fields identifying the DTDO to get

· **DOCK_pluginID, DOCK_pluginTag, DOCK_originator, DOCK_timestamp**

(uint32_t, uint32_t, uint64_t, struct timeval)

The four fields identifying the DOCK where to push the DTDO

· **MRT (uint32_t)**

The Maximum Retrieval Time is the maximum amount of time in milliseconds during which DELTOYA will try to get the DTDO

· **CHAINid (uint32_t)**

The CHAIN where the DTDO belongs. If 0, the DTDO does not belong to a CHAIN

· **CHAIN_pointer (uint32_t)**

When CHAIN_ID $\neq 0$, CHAIN_pointer is the sequence number or order of this DTDO inside its CHAIN. Otherwise, CHAIN_pointer is a key number identifying the `pushDTDO` call that caused the creation of this DTDOAQ entry

· **key (uint32_t)**

A field to save the key field of the ASK2ME packet received in order to be able to use it in the case we need to send the corresponding ASK2ME_FINISH packet or to detect duplicated ASK2ME packets.

· **returnInformation: DTDO_data, status (char*,uint8_t)**

The information needed to keep track of the result of the asking process (i.e., the DTDO obtained during the MRT period). This

information includes a field to save a copy of the DTDO_data of the obtained DTDO (and this copy will be the one to be put in the DOCK).

Moreover, for the DTDO_data obtained from the remote DTDO, the DTDO_useCounter field of the DTDO entry in the DTDO Table will be immediately decremented by one (this will let to delete the DTDO entry of the DTDO Table as soon as possible). Moreover, in the case that MRT expires due to the DTDO has not been obtained, the DTDO_useCounter field of the DTDO will also be decremented by one.

The key fields of the DTDOAQ entries are the DTDO_originator (assuming that DTDO_originator will be always the sender of the ASK2ME message) and the key fields, meaning that there cannot be two entries with the same DTDO_originator and key fields.

For a DTDOAQ entry, when the DTDO is finally obtained and saved in the DOCK, or when MRT expires, the entry will be automatically deleted.

C.8.3 ASK2ME Header

The fields of the ASK2ME header are the following:

· **Type (8 bits (unsigned))**

Type == 0 → ASK2ME (These packets will be forwarded according to MIFO to the ID of the node owning the DOCK)

Type == 1 → ASK2ME_ACK (These packets will be forwarded according to MIFO to the ID of the node pushing the DTDO)

Type == 2 → ASK2ME_FINISH (These packets will be forwarded according to MIFO to the ID of the node pushing the DTDO)

Type == 3 → ASK2ME_FINISH_ACK (These packets will be forwarded according to MIFO to the ID of the node owning the DOCK)

· **Version (8 bits (unsigned))**

0

· **Reserved (8 bits (unsigned))**

Unused

· **Payload**

Type == 0 →

· **DTDO_pluginID, DTDO_pluginTag,**

DTDO_originator, DTDO_timestamp,

DTDO_dataSize, DTDO_dataTTL, DTDO_isPublic (32+32+64+64+32+32+8 bits)

The fields of the DTDO this ASK2ME packet wants to push

· DOCK_pluginID, DOCK_pluginTag,

DOCK_originator, DOCK_timestamp,

(32+32+64+64 bits)

The fields of the DOCK where this ASK2ME packet wants to push the DTDO

· CHAIN_ID (32 bits)

The CHAIN in which the DTDO belongs (if any)

· CHAIN_pointer (32 bits)

When CHAIN_ID == 0, the CHAIN_pointer is a key number identifying this `pushDTDO` call

When CHAIN_ID ≠ 0, the CHAIN_pointer is the pointer of the DTDO inside the CHAIN

· CHAIN_resistance (32 bits)

When CHAIN_ID ≠ 0, the CHAIN_resistance is the amount of time in milliseconds that, in the case of having a gap in the CHAIN, we will wait for the missed DTDOs

· MPT (32 bits)

The maximum amount of time (in milliseconds) to try to push this DTDO

· key (32 bits (unsigned))

A key number identifying this `pushDTDO` call

Type == 1 →

· **status (8 bits)**

0 → Everything is OK. I am going to get your DTDO

1 → I do not have the DOCK where to put this DTDO

2 → I do not have enough space for getting this DTDO

3 → I already have this `pushDTDO` in this DOCK

4 → CHAIN in the future

5 → DTDO already pushed (Everything was OK)

· **key (32 bits (unsigned))**

The key field of the ASK2ME that has triggered this ASK2ME_ACK packet

Type == 2 →

· **key (32 bits (unsigned))**

The key field of the ASK2ME that has caused this ASK2ME_FINISH packet

Type == 3 →

· **key (32 bits (unsigned))**

The key field of the ASK2ME_FINISH that has triggered this ASK2ME_FINISH_ACK packet

C.8.4 Sending DTDOs

DELTOYA provides a method for pushing a specific DTDO (DTDOid) to a specific DOCK (pluginID, pluginTag, originator and timestamp).

```
int8_t err = pushDTDO(idCallback,DTDOid,pluginID,pluginTag,originator,
timestamp,MPT,CHAINid);
```

Params (as defined in sections 5.1, 5.2 and 5.3):

- idCallback

  Output parameter

- DTDOid

  The DTDO to push

- pluginID,pluginTag,originator,timestamp

  The DOCK where to push the DTDO

- MPT (uint32_t)

  The Maximum Pushing Time is the amount of time in milliseconds during which DELTOYA will try to push the DTDO to the DOCK once the pushDTDO call is started. After MPT milliseconds, even if the DTDO has not completely pushed, DELTOYA will stop trying to push it.

- CHAINid

  The CHAIN where the DTDO belongs. CHAINid can be NULL (i.e., unchained DTDO)

When the `pushDTDO` is called, it will check:

- MIN_MPT <= `MPT` <= MAX_MPT
- There exists a DTDO with DTDO_ID==`DTDOid` in the DTDO Table
- There exists a CHAIN with CHAIN_ID==`CHAINid` in the CHAIN Table
- There is enough space to create a new entry in the DTDOSQ

In case of these checks fail, `pushDTDO` will return -1, -2, -3 or -4 respectively, or `idCallback` if all checks are OK. `idCallback` is the identifier of the `pushDTDO` call, and it can be used to reexecute the `pushDTDO` (see appendix "C.8.4 Sending DTDOs") or to identify to which `pushDTDO` corresponds each `pushDTDOCallback`.

If all checks are OK, `pushDTDO` will perform the actions needed to push (i.e. send) the DTDO into the DOCK of the remote node, and when the pushing process is finished (i.e., the DTDO is finally in the DOCK), or MPT has expired, a callback with the result of the pushing operation will be executed:

```
void pushDTDOCallback(idCallback,status);
```

where `idCallback` is the identifier of the push that has caused this callback and `status` is the result of the pushing operation:

- `status==0` (DTDO fully pushed)
- `status==1` (DOCK not found)
- `status==2` (DOCK full)
- `status==3` (MPT expired. DTDO not fully pushed)
- `status==4` (Old DTDO. The CHAIN has been reading in DTDOs next to this)

With all these assumptions, the operation of the `pushDTDO` is as follows:

Create an entry in the DTDO Sending Queue (DTDOSQ) for this pushDTDO (we will call this new entry e for simplification):

If (CHAINid ≠ 0)

The CHAIN_pointer of this new entry will be the CHAIN_pointer value of the CHAIN Table entry corresponding to the CHAINid

We will increment by 1 the CHAIN_pointer value of the CHAIN Table entry corresponding to the CHAINid

The CHAIN_reistance of this new entry will be the CHAIN_resistance value of the CHAIN Table entry corresponding to the CHAINid

Else

The CHAIN_pointer of this new entry will be the idCallback value of this new entry

EndIf

startRDM(originator,ourObserver);

We keep sending this ASK2ME packet every T_ASK2ME milliseconds:

- DTDO_{pluginID,pluginTag,originator,timestamp,

  dataSize,dataTTL,isPublic} =

  the {pluginID,pluginTag,originator,timestamp,

  dataSize,dataTTL,isPublic} of the DTDO DTDOid

- DOCK_{pluginID,pluginTag,originator,timestamp} =

  pluginID,pluginTag,originator,timestamp

- CHAIN_ID = CHAINid

- CHAIN_pointer = e.CHAIN_pointer

- CHAIN_resistance = e.CHAIN_resistance

- MPT = MPT

- key = e.idCallback

until we receive the corresponding ASK2ME_ACK packet (ASK2ME_ACK.key==e.idCallback) or the corresponding ASK2ME_FINISH packet (ASK2ME_FINISH.key==e.idCallback) or until a maximum of MPT milliseconds.

INIT:

If (an ASK2ME_ACK packet with ASK2ME_ACK.key==e.idCalback or an ASK2ME_FINISH packet with ASK2ME_FINISH.key==e.idCalback is received before the MPT expiration)

We stop sending ASK2ME packets (if it is the case)

If (ASK2ME_ACK)

```
            stopRDM(originator,ourObserver);

            If (ASK2ME_ACK.status == 0)

                  We do nothing. Just wait for the corresponding
                  ASK2ME_FINISH packet or for the MPT expiration →
                  GOTO INIT

            ElseIf (ASK2ME_ACK.status == 1)

                  e.canRepush = true

                  Ready to call the pushDTDOCallback with status==1
                  and with the entry e

                  We delete de DTDOSQ entry e

            ElseIf (ASK2ME_ACK.status == 2)

                  e.canRepush = true

                  Ready to call the pushDTDOCallback with status==2
                  and with the entry e

                  We delete de DTDOSQ entry e

            ElseIf (ASK2ME_ACK.status == 3)

                  e.canRepush = false

                  Ready to call the pushDTDOCallback with status==0
                  and with the entry set to NULL

                  We delete de DTDOSQ entry e

            ElseIf (ASK2ME_ACK.status == 4)

                  e.canRepush = false

                  Ready to call the pushDTDOCallback with status==4
                  and with the entry set to NULL

                  We delete de DTDOSQ entry e

            EndIf

      ElseIf (ASK2ME_FINISH)

            We send this ASK2ME_FINISH_ACK packet:

                  · key = e.idCallback

            stopRDM(originator,ourObserver);

            e.canRepush = false

            Ready to call the pushDTDOCallback with status==0 and with
            the entry set to NULL

            We delete de DTDOSQ entry e

      EndIf

Else
```

We stop sending ASK2ME packets (if it is the case)

stopRDM(originator,ourObserver);

e.canRepush = true

Ready to call the pushDTDOCallback with status==3 and with the entry e

We delete de DTDOSQ entry e

EndIf

Note that if we receive an ASK2ME_ACK packet and it does not exist an entry in the DTDOSQ with idCallback equal to the key field of the ASK2ME_ACK, then the ASK2ME_ACK packet will be just discarded.

Note that if we receive an ASK2ME_FINISH packet and it does not exist an entry in the DTDOSQ with idCallback equal to the key field of the ASK2ME_FINISH, then we have to keep sending the corresponding ASK2ME_FINISH_ACK until no more AKS2ME_FINISH packets are received (from the implementation point of view, a temporal data structure to keep this metadata will be needed).

C.8.4.1 Receiving ASK2ME Packets

When we receive an ASK2ME packet (we will call it a2m for simplification) then we will proceed in the following way:

startRDM(a2m.DTDO_originator,ourObserver);

We will check if we have a DOCK in the DOCK Table/Cache with:

- DOCK_pluginID == a2m.DOCK_pluginID
- DOCK_pluginTag == a2m.DOCK_pluginTag
- DOCK_originator == a2m.DOCK_originator
- DOCK_timestamp == a2m.DOCK_timestamp

If (it does not exist)

We send back this ASK2ME_ACK packet:

- status = 1
- key = a2m.key

stopRDM(a2m.DTDO_originator,ourObserver);

exit

EndIf

If (CHAINid ≠ 0)

We will check if exists an entry in the RXCHAIN Table with:

- sender == a2m.DTDO_originator

· CHAIN_ID == a2m.CHAIN_ID

If (it exists)

If (DOCK_ID of the entry ≠ the DOCK_ID of the DOCK={a2m.DOCK_pluginID,a2m.DOCK_pluginTag, a2m.DOCK_originator,a2m.DOCK_timestamp})

We send this ASK2ME_ ACK packet:

· status = 1

· key = a2m.key

stopRDM(a2m.DTDO_originator,ourObserver);

exit

EndIf

If (CHAIN_resistance of the entry ≠ a2m.CHAIN_resistance)

We send this ASK2ME_ ACK packet:

· status = 1

· key = a2m.key

stopRDM(a2m.DTDO_originator,ourObserver);

exit

EndIf

If ((current_CHAIN_pointer of the entry > a2m.CHAIN_pointer) && (historyOfBrokenDTDOs contains a2m.CHAIN_pointer))

We send this ASK2ME_ ACK packet:

· status = 4

· key = a2m.key

stopRDM(a2m.DTDO_originator,ourObserver);

exit

ElseIf ((current_CHAIN_pointer of the entry > a2m.CHAIN_pointer) && (historyOfBrokenDTDOs do not contain a2m.CHAIN_pointer))

We send this ASK2ME_ ACK packet:

· status = 5

· key = a2m.key

stopRDM(a2m.DTDO_originator,ourObserver);

exit

EndIf

Else

We create an entry in the RXCHAIN Table with:

· sender = a2m.DTDO_originator

· CHAIN_ID = a2m.CHAIN_ID

· DOCK_ID = the DOCK_ID of the

DOCK={a2m.DOCK_pluginID,a2m.DOCK_pluginTag,

a2m.DOCK_originator,a2m.DOCK_timestamp}

· CHAIN_resistance = a2m.CHAIN_resistance

· current_CHAIN_pointer = 0

EndIf

Else

We will check if exists an entry in the RXUNCHAIN Table with:

· sender == a2m.DTDO_originator

· CHAIN_pointer == a2m.CHAIN_pointer (or a2m.key because for unchained DTDOs a2m.CHAIN_pointer==a2m.key)

If (it exists)

We send this ASK2ME_ ACK packet:

· status = 5

· key = a2m.key

stopRDM(a2m.DTDO_originator,ourObserver);

exit

EndIf

EndIf

We will check if there exists a DOCKBuffer entry dbe in the DOCKBuffer of the DOCK={a2m.DOCK_pluginID,a2m.DOCK_pluginTag,a2m.DOCK_originator,

a2m.DOCK_timestamp} with:

```
        · dbe.sender == a2m.DTDO_originator
        · If (a2m.CHAIN_ID == 0)
              dbe.CHAIN_pointer == a2m.CHAIN_pointer
          Else
              dbe.CHAIN_ID == a2m.CHAIN_ID
              dbe.CHAIN_pointer == a2m.CHAIN_pointer
          EndIf
If (it exists)
      We send this ASK2ME_ ACK packet:
            · status = 3
            · key = a2m.key

      stopRDM(a2m.DTDO_originator,ourObserver);
      exit
EndIf

We will check if there is enough free space in the DOCKBuffer of the
DOCK={a2m.DOCK_pluginID,a2m.DOCK_pluginTag,a2m.DOCK_originator,
a2m.DOCK_timestamp} (maximum size of MAX_DOCKBUFFERSIZE)
If (there is not enough space)
      We send this ASK2ME_ ACK packet:
            · status = 2
            · key = a2m.key

      stopRDM(a2m.DTDO_originator,ourObserver);
      exit
EndIf

If (# of entries of the DTDOAQ == MAXNUM_DTDOAQENTRIES)
      We send this ASK2ME_ ACK packet:
            · status = 2
            · key = a2m.key
```

```
      stopRDM(a2m.DTDO_originator,ourObserver);

      exit

EndIf
```

If (an entry in the DTDOAQ with DTDO_originator == a2m.DTDO_originator and with key == a2m.key already exists)

We send this ASK2ME_ ACK packet:

- status = 0
- key = a2m.key

```
      stopRDM(a2m.DTDO_originator,ourObserver);

      exit

EndIf
```

At this point, all checks are performed and we can start to get the DTDO.

We send this ASK2ME_ ACK packet:

- status = 0
- key = a2m.key

```
stopRDM(a2m.DTDO_originator,ourObserver);
```

We create an entry in the DTDO Asking Queue (DTDOAQ) (we will call this new entry e for simplification) with the fields of the a2m packet and with e.MRT = a2m.MPT

```
// For sure that e.DTDO_originator != "my ID"
```

DELTOYA checks if there exists a DTDO d={e.DTDO_pluginID,e.DTDO_pluginTag,e.DTDO_originator, e.DTDO_timestamp} in the sub table of the remote DTDOs.

If (found with DTDO_data != NULL)

put2DOCK(e.DTDO_originator,e.CHANid,e.CHAIN_pointer,d, the DOCK_ID of the DOCK={e.DOCK_pluginID,e.DOCK_pluginTag, e.DOCK_originator,e.DOCK_timestamp})

ElseIf (found with DTDO_data == NULL)

DELTOYA checks if for this remote DTDO there are one or more recipients (i.e. links to destination nodes) in the DTDO Table.

```
      If (there is route)

            Increment by 1 the DTDO_useCounter of this remote DTDO (in
            this case, if the DTDODM discovers this DTDO in the
            future, the DTDODM will not increment the DTDO_useCounter
            of the DTDO, even in the first time it discovers the DTDO,
            since we have already incremented it here. This is for
            assuring that PTP has the lock of this DTDO even if the
            DTDODM does not discovers this DTDO)

            If (e.DTDO_isPublic)

                  startDTDODM(e.DTDO_pluginID,e.DTDO_pluginTag,
                  e.DTDO_originator,e.DTDO_timestamp,e.MRT,2)

            Else

                  startDTDODM(e.DTDO_originator,e.DTDO_pluginID,
                  e.DTDO_pluginTag,e.DTDO_originator,
                  e.DTDO_timestamp,e.MRT,3);

            EndIf

            [PTP] getDTDO(e.DTDO_pluginID,e.DTDO_pluginTag,
            e.DTDO_originator,e.DTDO_timestamp, a pointer to the DTDOAQ
            entry e,QoS)

      Else

            If (e.DTDO_isPublic)

                  startDTDODM(e.DTDO_pluginID,e.DTDO_pluginTag,
                  e.DTDO_originator,e.DTDO_timestamp,e.MRT,2)

            Else

                  startDTDODM(e.DTDO_originator,e.DTDO_pluginID,
                  e.DTDO_pluginTag,e.DTDO_originator,
                  e.DTDO_timestamp,e.MRT,3);

            EndIf

      EndIf

ElseIf (no found)

      If (e.DTDO_isPublic)

            startDTDODM(e.DTDO_pluginID,e.DTDO_pluginTag,
            e.DTDO_originator,e.DTDO_timestamp,e.MRT,2)

      Else

            startDTDODM(e.DTDO_originator,e.DTDO_pluginID,
            e.DTDO_pluginTag,e.DTDO_originator,e.DTDO_timestamp,
            e.MRT,3);

      EndIf

EndIf
```

If (PTP finishes and is able to obtain the DTDO before the MRT expiration)

We will check again if we have the DOCK in the DOCK Table/Cache with:

- DOCK_pluginID == a2m.DOCK_pluginID
- DOCK_pluginTag == a2m.DOCK_pluginTag
- DOCK_originator == a2m.DOCK_originator
- DOCK_timestamp == a2m.DOCK_timestamp

If (it does not exist)

We send this ASK2ME_ ACK packet:

- status = 1
- key = a2m.key

stopRDM(a2m.DTDO_originator,ourObserver);

exit

EndIf

We will check again if there exists a DOCKBuffer entry dbe in the DOCKBuffer of the DOCK={a2m.DOCK_pluginID,a2m.DOCK_pluginTag,a2m.DOCK_originator,

a2m.DOCK_timestamp} with:

- dbe.sender == a2m.DTDO_originator
- **If** (a2m.CHAIN_ID == 0)

  dbe.CHAIN_pointer == a2m.CHAIN_pointer

  Else

  dbe.CHAIN_ID == a2m.CHAIN_ID

  dbe.CHAIN_pointer == a2m.CHAIN_pointer

  EndIf

If (it exists)

We send this ASK2ME_ ACK packet:

- status = 3
- key = a2m.key

stopRDM(a2m.DTDO_originator,ourObserver);

exit

EndIf

We will check again if there is enough free space in the DOCKBuffer of the DOCK={a2m.DOCK_pluginID,a2m.DOCK_pluginTag,a2m.DOCK_originator,

a2m.DOCK_timestamp} (maximum size of MAX_DOCKBUFFERSIZE)

If (there is not enough space)

We send this ASK2ME_ ACK packet:

· status = 2

· key = a2m.key

stopRDM(a2m.DTDO_originator,ourObserver);

exit

EndIf

put2DOCK(e.DTDO_originator,e.CHANid,e.CHAIN_pointer,

the DTDO obtained by PTP,

the DOCK_ID of the

DOCK={e.DOCK_pluginID,e.DOCK_pluginTag,

e.DOCK_originator,e.DOCK_timestamp})

If (e.CHAINid == 0)

We add an entry in the RXUNCHAIN Table with:

· sender = e.DTDO_originator

· CHAIN_pointer = e.CHAIN_pointer

EndIf

startRDM(e.DTDO_originator,ourObserver);

We keep sending this ASK2ME_FINISH packet every T_ASK2ME_FINISH milliseconds:

· key = e.key

until we receive the corresponding ASK2ME_FINISH_ACK packet (ASK2ME_FINISH_ACK.key==e.key) or until the MRT expiration.

When we receive the corresponding ASK2ME_FINISH_ACK packet (ASK2ME_FINISH_ACK.key==e.key) or when MRT expires, then we will stop sending ASK2ME_FINISH packets and we will stopRDM(e.DTDO_originator,ourObserver)

The entry e will be automatically deleted.

Else

[PTP] stopGetDTDO(a pointer to the DTDOAQ entry e) will be called to notify PTP to stop trying to obtain this DTDO.

The entry e will be automatically deleted.

EndIf

C.8.4.2 put2DOCK

```
put2DOCK(sender_p,CHAIN_ID_p,CHAIN_pointer_p,trueDTDO_p,DOCK_ID_p)
```

{

We create a DOCKBuffer entry dbe with:

- sender = sender_p
- CHAIN_ID = CHAIN_ID_p
- CHAIN_pointer = CHAIN_pointer_p
- timestamp = "now"
- trueDTDO = trueDTDO_p

We add the DOCKBuffer entry dbe in the DOCKBuffer of the DOCK DOCK_ID_p following the sorting algorithm explained in section 5.2: inside the DOCKBuffer, all entries will be ordered depending on their arrival time in the DOCKBuffer and for each CHAIN of DTDOs, ordered by their CHAIN_pointer inside the CHAIN

Moreover, in the case that, just before adding this new DOCKBuffer entry dbe in the DOCKBuffer of the DOCK DOCK_ID_p, no DTDO can be picked up from this DOCK DOCK_ID_p (following the algorithm of appendix "B.2 DOCK Table"), and that this new entry dbe corresponds to a DTDO that can be picked up right now from the DOCK, then the DTDOsAvaibaleCallback(DOCK_ID_p) (see appendix "C.8.6.1 DTDOsAvailableCallback") will be called

}

C.8.5 Resending DTDOs

DELTOYA provides a method for retrying to push a chained DTDO when, the last time it was pushed, the status of the callback was different of 0 and 4:

```
int8_t err = retryPushDTDO(DTDOSQ_entry,MPT);
```

Params:

- DTDOSQ_entry

The information returned by the callback of the `pushDTDO` we want to reexecute

- `MPT`

  A new Maximum Pushing Time for this retry of `pushDTDO`

retryPushDTDO will check:

- There cannot exist a DTDOSQ entry with idCallback==DTDOSQ_entry.idCallback
- DTDOSQ_entry.canRepush==true
- MIN_MPT <= MPT <= MAX_MPT
- There exists a DTDO with DTDO_ID==DTDOSQ_entry.DTDOid in the DTDO Table
- There exists a CHAIN with CHAIN_ID==DTDOSQ_entry.CHAINid in the CHAIN Table
- There is enough space to move the DTDOSQ_entry to the DTDOSQ

In case of one of these checks fail, `retryPushDTDO` will return -1, -2, -3, -4, -5 or -6 for each check respectively, or 0 if all checks are OK.

If all checks are OK, `retryPushDTDO` will:

```
We move the DTDOSQ_entry to the DTDOSQ

startRDM(DTDOSQ_entry.originator,ourObserver);


We keep sending this ASK2ME packet every T_ASK2ME milliseconds:

      · DTDO_{pluginID,pluginTag,originator,timestamp,dataSize,
      dataTTL,isPublic} = the {pluginID,pluginTag,originator,
      timestamp,dataSize,dataTTL,isPublic} of the DTDO
      DTDOSQ_entry.DTDOid

      · DOCK_{pluginID,pluginTag,originator,timestamp} =
      DTDOSQ_entry.pluginID,DTDOSQ_entry.pluginTag,
      DTDOSQ_entry.originator,DTDOSQ_entry.timestamp

      · CHAIN_ID = DTDOSQ_entry.CHAINid

      · CHAIN_pointer = DTDOSQ_entry.CHAIN_pointer

      · CHAIN_resistance = DTDOSQ_entry.CHAIN_resistance

      · MPT = MPT

      · key = DTDOSQ_entry.idCallback

until we receive the corresponding ASK2ME_ACK packet
(ASK2ME_ACK.key==DTDOSQ_entry.idCalback) or the corresponding
ASK2ME_FINISH packet (ASK2ME_FINISH.key==DTDOSQ_entry.idCalback) or
until a maximum of MPT milliseconds.

INIT:

If (an ASK2ME_ACK packet with ASK2ME_ACK.key==DTDOSQ_entry.idCalback
or an ASK2ME_FINISH packet with
```

ASK2ME_FINISH.key==DTDOSQ_entry.idCalback is received before the MPT expiration)

We stop sending ASK2ME packets (if it is the case)

If (ASK2ME_ACK)

stopRDM(DTDOSQ_entry.originator,ourObserver);

If (ASK2ME_ACK.status == 0)

We do nothing. Just wait for the corresponding ASK2ME_FINISH packet or for the MPT expiration → **GOTO INIT**

ElseIf (ASK2ME_ACK.status == 1)

DTDOSQ_entry.canRepush = true

Ready to call the pushDTDOCallback with status==1 and with the entry DTDOSQ_entry

We delete de DTDOSQ entry DTDOSQ_entry

ElseIf (ASK2ME_ACK.status == 2)

DTDOSQ_entry.canRepush = true

Ready to call the pushDTDOCallback with status==2 and with the entry DTDOSQ_entry

We delete de DTDOSQ entry DTDOSQ_entry

ElseIf (ASK2ME_ACK.status == 3)

DTDOSQ_entry.canRepush = false

Ready to call the pushDTDOCallback with status==0 and with the entry set to NULL

We delete de DTDOSQ entry DTDOSQ_entry

ElseIf (ASK2ME_ACK.status == 4)

DTDOSQ_entry.canRepush = false

Ready to call the pushDTDOCallback with status==4 and with the entry set to NULL

We delete de DTDOSQ entry DTDOSQ_entry

EndIf

ElseIf (ASK2ME_FINISH)

We send this ASK2ME_FINISH_ACK packet:

· key = DTDOSQ_entry.idCallback

stopRDM(DTDOSQ_entry.originator,ourObserver);

DTDOSQ_entry.canRepush = false

Ready to call the pushDTDOCallback with status==0 and with the entry set to NULL

```
            We delete de DTDOSQ entry DTDOSQ_entry
      EndIf
Else
      We stop sending ASK2ME packets (if it is the case)
      stopRDM(DTDOSQ_entry.originator,ourObserver);
      DTDOSQ_entry.canRepush = true
      Ready to call the pushDTDOCallback with status==3 and with the
      entry DTDOSQ_entry
      We delete de DTDOSQ entry DTDOSQ_entry
EndIf
```

Note that if we receive an ASK2ME_ACK packet and it does not exist an entry in the DTDOSQ with idCallback equal to the key field of the ASK2ME_ACK, then the ASK2ME_ACK packet will be just discarded.

Note that if we receive an ASK2ME_FINISH packet and it does not exist an entry in the DTDOSQ with idCallback equal to the key field of the ASK2ME_FINISH, then we have to keep sending the corresponding ASK2ME_FINISH_ACK until no more AKS2ME_FINISH packets are received (from the implementation point of view, a temporal data structure to keep this metadata will be needed).

C.8.6 Receiving DTDOs

DELTOYA provides a method for picking up a DTDO from a DOCK:

```
int8_t rtat =
pickupDTDO(DOCKid,pluginID,pluginTag,originator,timestamp,data,dataSiz
e,CHAINid,chainResistanceExpired,chainBroken);
```

Params (as defined in sections 5.1, 5.2 and 5.3):

- `DOCKid`

  Input parameter

- `pluginID,pluginTag,originator,timestamp,data,dataSize,CHAINid`

  Output parameters. pluginID, pluginTag, originator, timestamp, data and dataSize are only valid when rtat == 0, and they are the fields of the DTDO picked up. If CHAINid is 0, the returned DTDO does not belong to any CHAIN. Otherwise, CHAINid is the CHAN_ID of the CHAIN in which the returned DTDO belongs.

- `chainResistanceExpired`

  Output parameter. Only valid when rtat == 0 and CHAINid ≠ 0. A bool indicating that, for this DTDO belonging to a CHAIN that we have picked up, the CHAIN

resistance has expired, meaning that there is a gap in the CHAIN or that there has been more than CHAIN_resistance time between two consecutive `pushDTDO` calls

- `chainBroken`

  Output parameter. Only valid when rtat == 0 and CHAINid ≠ 0. A bool indicating that there is a gap in the CHAIN of the DTDO that we have picked up

`pickupDTDO` will return -1 if the DOCK does not exist, -2 if the DOCK is empty, -3 if the DOCK is not empty but no DTDO can be picked up, or 0 otherwise.

`pickupDTDO` will perform in the following way:

```
Check if there exists a DOCK with DOCK_ID==DOCKid in the DOCK Table/Cache

If (it does not exist)

      return -1

EndIf


We start checking if the first entry of the DOCKBuffer of the DOCK with DOCK_ID==DOCKid can be picked up, and we will keep checking entries until we found the first one that can be picked up or until we discover that no entry can be picked up at this moment:

If (there is not any entry)

      return -2

EndIf


For each DOCKBuffer entry e of the DOCK d, and starting from the first one, we will check if it belongs to a CHAIN or not:

      If (e.CHAIN_ID == 0)

            We copy the entry to the output parameters:

                  · pluginID = e.trueDTDO.DTDO_pluginID

                  · pluginTag = e.trueDTDO.DTDO_pluginTag

                  · originator = e.trueDTDO.DTDO_originator

                  · timestamp = e.trueDTDO.DTDO_timestamp

                  · data = e.trueDTDO.DTDO_data

                  · dataSize = e.trueDTDO.DTDO_dataSize

                  · CHAINid = e.CHAIN_ID

                  · chainResistanceExpired = false

                  · chainBroken = false
```

```
        We delete the entry e

        return 0

Else

        For sure there is an entry rte in the RXCHAIN Table
        corresponding to the CHAIN e.CHAIN_ID and to the sender
        e.sender

        If (e.CHAIN_pointer == rte.current_CHAIN_pointer)

              rte.current_CHAIN_pointer++

              We copy the entry to the output parameters:

                    · pluginID = e.trueDTDO.DTDO_pluginID

                    · pluginTag = e.trueDTDO.DTDO_pluginTag

                    · originator = e.trueDTDO.DTDO_originator

                    · timestamp = e.trueDTDO.DTDO_timestamp

                    · data = e.trueDTDO.DTDO_data

                    · dataSize = e.trueDTDO.DTDO_dataSize

                    · CHAINid = e.CHAIN_ID

                    · If ("now" >

                      (e.timestamp+rte.CHAIN_resistance))

                          chainResistanceExpired = true

                      Else

                          chainResistanceExpired = false

                      EndIf

                    · chainBroken = false


              We delete the entry e

        return 0

        Else

              For sure e.CHAIN_pointer > rte.current_CHAIN_pointer

              If ("now">(e.timestamp+rte.CHAIN_resistance))

                    rte.current_CHAIN_pointer = e.CHAIN_pointer+1

                    We copy the entry to the output parameters:

                          · pluginID = e.trueDTDO.DTDO_pluginID

                          · pluginTag = e.trueDTDO.DTDO_pluginTag
```

```
                        · originator =
                      e.trueDTDO.DTDO_originator

                        · timestamp = e.trueDTDO.DTDO_timestamp

                        · data = e.trueDTDO.DTDO_data

                        · dataSize = e.trueDTDO.DTDO_dataSize

                        · CHAINid = e.CHAIN_ID

                        · chainResistanceExpired = true

                        · chainBroken = true


                    We add (or replace since the
                    historyOfBrokenDTDOs field has a maximum size)
                    to the history of broken DTDOs all the DTDOs
                    of the CHAIN that have been assumed to be lost
                    due to this pickup

                    rte.historyOfBrokenDTDOs.add(from
                    rte.current_CHAIN_pointer to e.CHAIN_pointer-1)

                    We delete the entry e

                    return 0

                Else

                    continue

                EndIf

            EndIf

        EndIf

EndFor


If we are here, it means that no entry can be picked up from the DOCK:

return -3
```

C.8.6.1 DTDOsAvailableCallback

DELTOYA provides a callback for knowing when there are DTDOs available in a DOCK:

```
void DTDOsAvailableCallback(DOCKid);
```

Params (as defined in section 5.2):

- `DOCKid`

  The DOCK_ID of the DOCK that has DTDOs available to be picked up at this moment

`DTDOsAvailableCallback` will be executed as explained in appendix “C.8.4.2 put2DOCK”.

C.9 DOCKs and CHAINs

C.9.1 DOCK Creation

DELTOYA provides a method to create DOCKs:

```
int8_t err = createDOCK(id,pluginID,pluginTag);
```

Params (as defined in section 5.2):

- `id`

  Output parameter

- `pluginID, pluginTag`

  Input parameters

`createDOCK` will check:

- There cannot exist a DOCK with DOCK_pluginID==`pluginID` and DOCK_pluginTag==`pluginTag` in the DOCK Table (note that although the DOCK_timestamp field is part of the key field of a DOCK, it is not checked).
- There are not more unused DOCK_IDs

In case of one of these checks fail, `createDOCK` will return -1 or -2 for each check respectively, or 0 if all checks are OK.

If all checks are OK, `createDOCK` will create an entry in the DOCK Table with:

- DOCK_ID (output parameter) = a new (not used) identifier (starting from 0)
- DOCK_pluginID = `pluginID`
- DOCK_pluginTag = `pluginTag`
- DOCK_timestamp = “now”
- DOCK_buffer = empty()

C.9.2 DOCK Destruction

DELTOYA provides a method to destroy DOCKs:

```
uint8_t err = destroyDOCK(id);
```

Params (as defined in section 5.2):

- `id`

  Input parameter

`destroyDOCK` will check:

- If exists a DOCK with DOCK_ID==`id` in the DOCK Table
- Applications only can destroy DOCKs created by them (applications cannot destroy neither DOCKs of other nodes nor DOCKs of other applications of the same node).

In case of one of these checks fail, `destroyDOCK` will return -1 or -2 for each check respectively, or 0 if all checks are OK.

If all checks are OK, `destroyDOCK` will:

- Release the DOCKBuffer of this DOCK
- Delete the corresponding entry in the DOCK Table

C.9.3 DOCK Querying

C.9.3.1 DOCK Query Queue (DOCKQQ)

The DOCK Query Queue (DOCKQQ) is the data structure used to buffer all the `queryDOCKs` calls done in a node. The DOCKQQ will be the responsible of trying to discover all the queried DOCKs from the network, and always using a DOCKDM (see section 10), and at the same time, the responsible of executing the `queryDOCKsCallback` when originator is specified and a reply form this originator is received, or in T_QUERYDOCKS milliseconds after the `queryDOCKs` was called, when originator is not specified or it is specified but the reply is not received.

Each entry of the DOCKQQ identifies a `queryDOCKs` call, and it has the following fields:

· pluginID, pluginTag, originator (uint32_t, uint32_t, uint64_t)

The DOCKs to discover. pluginID will be ≠ NULL, and pluginTag and originator can be NULL or not (i.e., any pluginTag or any originator)

· callbackInformation:

pluginID, list{pluginTag,originator,timestamp}, idCallback

(uint32_t, list{uint32_t,uint64_t,struct timeval}, int32_t)

All the information needed to keep track of the results of the querying process (e.g., DOCKs discovered during the query period). This information will be the one to be returned in the `queryDOCKsCallback`.

Moreover, the idCallback of the `queryDOCKs` corresponding to this callback will also be returned.

The DOCKQQ does not have key fields, meaning that there can be two entries with the same pluginID, pluginTag and originator fields. When the same `queryDOCKs` call is called twice, then two processes for querying the same DTDOs will be started. However, each entry will have a unique idCallback which identifies the entry and that will be managed automatically (as cyclic a sequence number) by the DOCKQQ itself.

During T_QUERYDOCKS milliseconds, the DOCKQQ entry will try to discover the queried DOCKs once the `queryDOCKs` call is started. In other words, during T_QUERYDOCKS milliseconds, the DOCKDM will keep sending DOCKDREQ messages using a Binary Exponential Backoff until a

maximum of MAX_DOCKDREQS have been sent. After having sent MAX_DOCKDREQS DOCKDREQ messages, then the Binary Exponential Backoff will automatically be restarted as if a new Binary Exponential Backoff was started (i.e. $DOCKDREQ_{MAX\_DOCKDREQS}$ == $DOCKDREQ_1$). See an example in Figure 93.

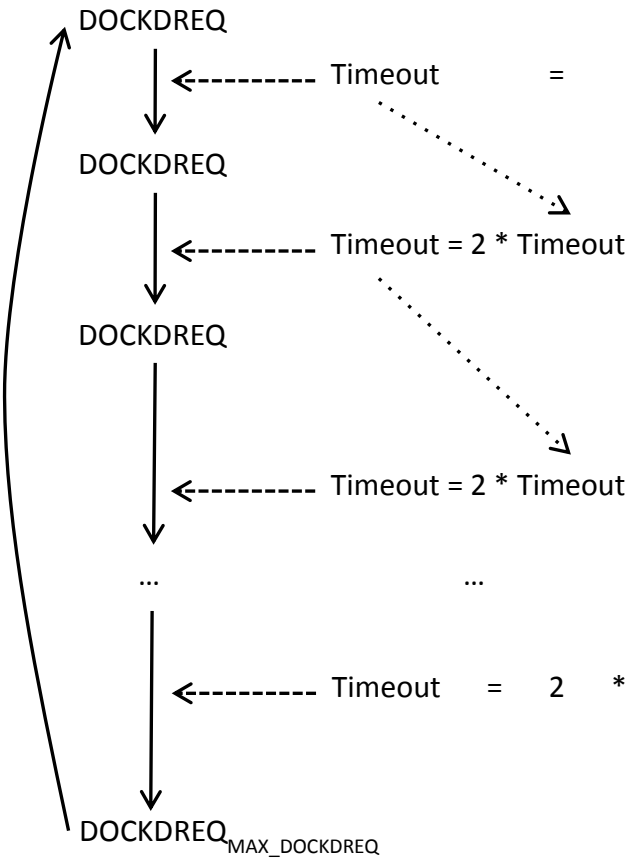

Figure 93. DOCK Querying Backoff

After a reply is received when originator is specified or after T_QUERYDOCKS milliseconds, the DOCKQQ entry will stop trying to discover more DOCKs, and the `queryDOCKsCallback` will be executed with the results obtained in the callbackInformation. Moreover, the DOCKQQ entry will be automatically deleted.

C.9.3.2 Querying DOCKs

DELTOYA provides a method to query DOCKs.

```
uint8_t err = queryDOCKs(idCallback,pluginID,pluginTag,originator);
```

Params (as defined in section DOCK Table):

- `idCallback`

  Output parameter. The identifier of the query (cyclic sequence number)

- `pluginID`

To identify the application where the DOCKs we want to obtain were created. It cannot be NULL

- `pluginTag`

  To identify the DOCKs within the application where they were created. It can be NULL or not. If NULL, this field will not be used to decide if an specific DOCK has to be discovered or not, meaning that it will be discovered independently of its DOCK_pluginTag value and only depending on the other parameters which identify the DOCK (i.e. NULL value can be seen as '*' in a regular expression)

- `originator`

  Unique ID to identify the node where the DOCKs we want to discover were created. It can be empty or not. If empty, this field will not be used to decide if an specific DOCK has to be discovered or not, meaning that it will be discovered independently of its DOCK_originator value and only depending on the other parameters which identify the DOCK (i.e. empty can be seen as '*' in a regular expression). If not empty, it must be different from the node itself

Table 17 shows all the possible `queryDOCKs` calls and their meaning.

Table 17. queryDOCKs calls

			Description
`pluginID`	`pluginTag`	`orig`	We are querying DOCKs with DOCK_pluginID=`pluginID`, DOCK_pluginTag=`pluginTag` and residing in the node `orig` (and independently of the DOCK_timestamp)
`pluginID`	`pluginTag`	*	We are querying DOCKs with DOCK_pluginID=`pluginID` and DOCK_pluginTag=`pluginTag` (and independently of the node where they are residing and DOCK_timestamp)
`pluginID`	*	`orig`	We are querying DOCKs with DOCK_pluginID=`pluginID` and residing in the node `orig` (and independently of the DOCK_pluginTag and DOCK_timestamp)
`pluginID`	*	*	We are querying DOCKs with DOCK_pluginID=`pluginID` (and independently of the DOCK_pluginTag, the node where they are residing and DOCK_timestamp)

When a `queryDOCKs` is called, it will return -1 if originator=="my iD", or 0 otherwise. Moreover, it will return `idCallback` as an output parameter. `idCallback` is the identifier

of the query, and it can be used to cancel the query or to identify to which query corresponds each callback.

`queryDOCKs` will perform the actions needed to discover the queried DOCKs through the network.

Once `queryDOCKs` is called, and after a reply is received when originator is specified or after T_QUERYDOCKS milliseconds, a callback with the result of the requesting operation will be executed:

```
void  queryDOCKsCallback(idCallback,pluginID,list{pluginTag,originator,
      timestamp});
```

where `idCallback` is the identifier of the query that has caused this callback and `pluginID,list{pluginTag,originator,timestamp}` is the result of the querying operation, i.e., the list of discovered DOCKs in the network during the period of time when the `queryDOCKs` has been called.

With all these assumptions, the operation of the `queryDOCKs` is as follows:

```
We create a new entry in the for this queryDOCKs.

If ((pluginTag!=NULL) and (originator!=NULL))

      startDOCKDM(pluginID,pluginTag,originator,T_QUERYDOCKS)

      Once T_QUERYDOCKS expires or a reply is received, the callback
      for the queryDOCKs will be executed with all the DOCKs that have
      been discovered from the network during the DOCKDM

ElseIf ((pluginTag==NULL) and (originator!=NULL))

      startDOCKDM(pluginID,*,originator,T_QUERYDOCKS)

      Once T_QUERYDOCKS expires or a reply is received, the callback
      for the queryDOCKs will be executed with all the DOCKs that have
      been discovered from the network during the DOCKDM

ElseIf ((pluginTag!=NULL) and (originator==NULL))

      startDOCKDM(pluginID,pluginTag,*,T_QUERYDOCKS)

      Once T_QUERYDOCKS expires, the callback for the queryDOCKs will
      be executed with all the DOCKs that have been discovered from
      the network during the DOCKDM

ElseIf ((pluginTag==NULL) and (originator==NULL)

      startDOCKDM(pluginID,*,*,T_QUERYDOCKS)

      Once T_QUERYDOCKS expires, the callback for the queryDOCKs will
      be executed with all the DOCKs that have been discovered from
      the network during the DOCKDM

EndIf
```

C.9.3.3 Cancelling DOCK Queries

DELTOYA provides a method to cancel pending `queryDOCKs` calls:

```
uint8_t err = cancelQueryDOCKs(idCallback);
```

`cancelQueryDOCKs` will check:

- If exists an entry in the DOCKQQ with idCallback==`idCallback`.

In case of this check fails, `cancelQueryDOCKs` will return -1, or 0 if all checks are OK.

If all checks are OK, `cancelQueryDOCKs` will:

- Cancel the DOCKDM corresponding to this DOCKQQ entry
- Delete the DOCKQQ entry
- From now on, the callback corresponding to the cancelled `queryDOCKs` will not be executed anymore.

C.9.4 CHAIN Creation

DELTOYA provides a method to create CHAINs:

```
uint8_t err = createCHAIN(id,CHAIN_resistance);
```

Params (as defined in section 5.3):

- id

  Output parameter

- CHAIN_resistance

  Input parameter

`createCHAIN` will check:

- MIN_CHAINRESISTANCE ≤ CHAIN_resistance ≤ MAX_CHAINRESISTANCE
- There are not more unused CHAIN_IDs

In case of one of these checks fail, `createCHAIN` will return -1 or -2 for each check respectively, or 0 if all checks are OK.

If all checks are OK, `createCHAIN` will create an entry in the CHAIN Table with:

- CHAIN_ID (output parameter) = a new (not used) identifier (starting from 0)
- CHAIN_resistance = `CHAIN_resistance`
- CHAIN_pointer = 0

C.9.5 CHAIN Destruction

DELTOYA provides a method to destroy CHAINs:

```
uint8_t err = destroyCHAIN(id);
```

Params (as defined in section 5.3):

- id

Input parameter

destroyCHAIN will check:

- If exists a CHAIN with CHAIN_ID==`id` in the CHAIN Table
- Applications only can destroy CHAINs created by them (applications cannot destroy neither CHAINs of other nodes nor CHAINs of other applications of the same node).

In case of one of these checks fail, `destroyCHAIN` will return -1 or -2 for each check respectively, or 0 if all checks are OK.

If all checks are OK, `destroyCHAIN` will:

- Delete the corresponding entry in the CHAIN Table

Appendix D. PTP Details

D.1 PTP Header

The fields of the PTP header are the following:

- **Type (8 bits (unsigned))**

  The type of the PTP PDU:

  Type == 0 → PTP Data

  Type == 1 → PTP Request

  Type == 2 → PTP Null

- **Version (8 bits (unsigned))**

  0

- **pluginID, pluginTag, originator, timestamp (32+32+64+64 bits)**

  Type == 0 → The set of fields identifying to which DTDO belongs the block of data of the Payload.

  Type == 1 → The set of fields identifying which DTDO we are requesting with this PTP Request message.

  Type == 2 → The set of fields identifying the DTDO that we requested and that for some reasons (see Payload field) it has become unavailable.

- **Reserved (8 bits (unsigned))**

  Unused

- **Payload**

  Type == 0 →

  · **dataRateFeedback (8 bits (unsigned))**

  The current data rate feedback used by the sender of this packet at the moment of sending it.

  · **dataOffset (32 bits (unsigned))**

  The offset (in terms of Bytes) of the block of data of the Portion field inside the DTDO this block of data belongs.

· **dataSize (16 bits (unsigned))**

The size (in Bytes) of the Portion field

· **Portion**

The block of data this PTP PDU is carrying.

Type == 1 →

· **dataRateFeedback (8 bits (unsigned))**

The data rate feedback advertised by this node to the node which is going to send the PTP Data messages of the requested portions.

· **bitmapSize (16 bits (unsigned))**

The size (in bits) of the bitmap field

· **bitmap**

A data structure that indicates which portions of the DTDO have been already received and which not and so, a data structure that tells the receiver of the PTP Request message which portions we are requesting.

Type == 2 →

· **Type (8 bits (unsigned))**

0 → DTDO not available (to detect possible deletions or expirations of the DTDO while the DTDO is being obtained)

1 → DTDO not possible to serve (for example, for low memory space in the recipient's device of the DTDO)

D.2 PTP Request Queue (PTPRQ)

Since several requests to PTP for one DTDO can be called at the same time while the DTDO is being obtained, a PTP Request Queue (PTPRQ) is needed to avoid duplicated data transfers of the same information. The PTPRQ is the data structure used to buffer all the `getDTDO` calls done in a node.

Each entry of the PTPRQ identifies a unique DTDO being retrieved (independently of the recipients used to obtain it), and each entry has the following fields:

· pluginID, pluginTag, originator, timestamp (uint32_t, uint32_t, uint64_t, struct timeval)

The DTDO to obtain

· list{observer} (list of uint32_t)

The list of observers still waiting for the data of this DTDO (i.e., the list of observers that have called a `getDTDO` call for this DTDO and that have not already expired). Only when this list becomes empty (there is not any observer interested in getting this DTDO), the process of getting the DTDO can be stopped and the PTPRQ entry can be deleted

· inPause (bool)

Used to indicate if the process of getting the DTDO_data corresponding to this PTPRQ entry is in pause (i.e. not getting the DTDO_data right now, but maintaing the metadata and partial data of the DTDO obtained in the past) or not

· currentRecipient (uint64_t)

The unique ID of the node we are obtaining the DTDO from

· currentTTL (uint32_t)

The TTL of the DTDO being obtained. For example, to be able to know the TTL of the DTDO when we receive the last packet of the DTDO and the recipient from which we obtained the DTDO has just disappeared from the DTDO Table (in this case, there is not link between the DTDO and the recipient, and so, we need to save the TTL of the obtained DTDO in this field in order to be able to use it if it is needed in the future)

· lastDataRateFeedback (uint8_t)

The data rate feedback of the last PTP Data message received

· lastDataRateFeedbackTime (struct timeval)

The instant of time when the last PTP Data message was received

· lastNumPTPDataRequested (uint16_t)

The number of PTP Data messages requested in the last PTP Request message sent

· **lastTimestampPTPRequest (struct timeval)**

The timestamp when the last PTP Request message was sent

· **bitmap**

A data structure to manage which portions of the DTDO have been already received and which not

· **bitmapCounter (uint16_t)**

A field to count the number of bits set to 0 in the bitmap

· **previousState (int32_t)**

A field to count the number of bits set to 1 in the bitmap at each T_CHECKSTATUSPTP expiration. If previousState == -1 then it means that it is undefined (there is no previous state)

· **changeRecipientCounter (uint8_t)**

A field to count the number of consecutive T_CHECKSTATUSPTP periods with 0 PTP Data messages received

· **buffer**

A temporary buffer where keep saving the partial data PTP is obtaining while retrieving a DTDO

The key fields of the PTPRQ are the pluginID, pluginTag, originator and timestamp fields, meaning that there cannot be two entries with the same pluginID, pluginTag, originator and timestamp fields.

When an already existent entry is added, only the list{observer} field of the entry will be updated adding the new observer to the list. Moreover, for each entry, when the whole data of the DTDO has been obtained or the list{observer} field becomes empty, the entry will be deleted.

D.3 PTP Serving Queue (PTPSQ)

Since several requests of PTP for one DTDO (or portion of the DTDO) can be received at the same time while the DTDO (or portion of the DTDO) is being served, a PTP Serving Queue (PTPSQ) is needed to manage with all these requests, which at the same time can also be received from the same or from several nodes. The PTPSQ is the data structure used to buffer all the PTP Request messages received in a node.

Each entry of the PTPSQ identifies a unique DTDO requested by a specific requester node, and each entry has the following fields:

· **pluginID, pluginTag, originator, timestamp (uint32_t, uint32_t, uint64_t, struct timeval)**

The DTDO to serve

· **Requester (uint64_t)**

The unique ID of the node that has sent this request

· **currentDataRateFeedback (uint8_t)**

The current data rate feedback used by this PTPSQ entry for sending the PTP Data messages of the DTDO we are serving

· **bitmap**

A data structure to manage which portions of the DTDO we have to serve or not

· **p1 (int32_t)**

A pointer indexing a specific position of the bitmap set to 0. p1==-1 means pointer not initialized. 0 ≤ p1 < size(bitmap)

· **timeout (struct timeval)**

The time instant when this PTPSQ entry has to be deleted

The key fields of the PTPSQ are the pluginID, pluginTag, originator, timestamp and Requester fields, meaning that there cannot be two entries with the same pluginID, pluginTag, originator, timestamp and Requester fields.

When an already existent entry is added, the currentDataRateFeedback and bitmap fields of the entry will be updated (if it is the case). Moreover, when the timeout expires, the entry will be deleted.

D.4 Pull Mode

D.4.1 getDTDO (Sending PTP Request messages)

PTP is needed when DELTOYA has discovered a new DTDO and it wants to get it through the network:

```
int8_t err =
getDTDO(pluginID,pluginTag,originator,timestamp,observer_O,[QoS])
```

where:

- `pluginID, pluginTag, originator, timestamp (uint32_t, uint32_t, uint64_t,` struct timeval`)`

  The fields identifying the DTDO PTP has to get

- `pointer(observer)`

  A field that allows PTP identifying which are the observers waiting for the data of the DTDO this `getDTDO` call will obtain

- `[QoS]`

  Unused

`getDTDO` will check:

- if `originator` != “my ID”
- if this DTDO has an entry in the sub table of the remote DTDOs
- if there is at least one recipient in the DTDO Table for this DTDO

In case of one of these checks fail, `getDTDO` will return -1, -2 or -3 respectively, or 0 if all checks are OK.

If all checks are OK, `getDTDO` will work as follows:

```
getDTDO checks if this DTDO in the sub table of the remote DTDOs has
DTDO_data == NULL or not (for sure that it will exist in the sub table
of the remote DTDOs due to the check #2)

If (DTDO_data != NULL)

      getDTDO will notify the observer_O with the DTDO found in the
      cache

Else

      For sure that we will have at least one recipient for this DTDO
      in the DTDO Table (due to the check #3)

      Start()

EndIf


Start()

{

      getDTDO will check if an entry with pluginID, pluginTag,
      originator and timestamp already exists in the PTPRQ.

      If (exists)

            The observer_O will be added to the list of the entry (if
            we are not already in the list).
```

If (inPause field of the entry == true)

Ready()

Else

Let the process of getting the DTDO continue doing the job with the current state

return

EndIf

Else

getDTDO will create a new entry in the PTPRQ with the corresponding pluginID, pluginTag, originator and timestamp fields, with list{observer} **=** {observer0}, with inPause = false, and with previousState = -1.

Now we can start the algorithm to retrieve the data until the whole data is obtained (the only possibility to stop the algorithm before the whole data is obtained is in the case that, from the point of view of DELTOYA, the MRT of the DTDORQ entry expires and it calls stopGetDTDO explicitly (and of course, there are not more DTDORQ entries interested in this DTDO))

Ready()

EndIf

}

Ready()

{

First of all, we have to choose which of the several recipients (if there are more than one) will be used to get the data of the DTDO (DTDO_data):

· If there is only one recipient, this will be the recipient to use

· If there are more than one recipient, PTP will check the Routing Heuristic (see section "3.2.1.2 Routing Heuristic (RH)") of all the routing paths of all these recipients, and PTP will choose the recipient with the best mean RH (i.e., PTP, for each recipient, will calculate the mean RH of all the RHs of all the routing paths of the recipient). In case of tie in the best mean RH, a random one will be chosen. This recipient will be the one to use to get the whole data of the DTDO.

· In the case that there are not recipients (only possible when inPause==true, since if inPause is false, then there is a previous check just before this (check number 3 of getDTDO) that checks that there is at least one recipient), then there is a process for getting the DTDO which was started but now is in pause. In this case, it has to continue in pause → **exit**

changeRecipientCounter = 0

If (inPause == true)

We can restart the process of getting this DTDO, since we have discovered a new recipient, and moreover, we can reuse the metada and partial data of the DTDO obtained in the past:

· inPause = false

· previousState = -1

· currentTTL = link(d,r).TTL

Continue1()

Else

In this case, there is not any process of getting the DTDO, neither paused nor running.

currentTTL = link(d,r).TTL

Go()

EndIf

}

Go()

{

At this point, we have an entry in the DTDO Table with the DTDO we want to obtain (we will call this DTDO d for simplification) and with d.DTDO_data==NULL. We also have a valid recipient from which to obtain d (The targetRecipient in the PTPRQ entry. We will call this recipient r for simplification).

Now, from the scope of the PTPRQ entry created (we will call this PTPRQ entry e for simplification), we will start the algorithm to get the d.DTDO_data from r:

We will create a bitmap (array of bits) of size

$$N = \left\lceil \frac{d.DTDO\_dataSize}{M} \right\rceil$$

where M is the maximum size of the Portion field of a PTP Data message (see section "2.3.1 PTP Segmentation"). We will call the bitmap b for simplification. It will be initialized to 0..0.

At the same time, the bitmapCounter field will be initialized to N, and it will account for the number of bits set to 0 in b. Every time a bit of b changes from 0 to 1 (i.e., we receive a new PTP Data message not yet

received), the bitmapCounter will be decremented by 1, and once the bitmapCounter reaches 0, we will be able to conclude the process of getting the DTDO, since we will have obtained all the d.DTDO_data.

Moreover, the lastDataRateFeedback field will be initialized to 0 (which means "invalid value"), and every time a PTP Data is received, the dataRateFeedback field (always greater than 0) of the Payload of the PTP Data message will be copied to the lastDataRateFeedback field of e. Also the time instant when this last data rate feedback was received will be saved in the lastDataRateFeedbackTime field of e.

The changeRecipientCounter field will be initialized to 0, and every T_CHECKSTATUSPTP period, it will be incremented by 1 if no PTP Data messages have been received, or set to 0 if one or more PTP Data messages have been received. In this way, if the counter reaches MAX_PTPCONSECUTIVELOSES, then PTP will decide to change of recipient in order to continue asking for the DTDO_data.

See next paragraphs for seeing how to manage/use these fields in more detail.

Assuming that b[i] (0 <= i < N) is the i-th bit of the bitmap b, we define that:

- if b[i] == 1, PTP has already received the PTP Data message carrying the portion of d.DTDO_data with offset == i*M and
  - size == M, if (i < N-1) or (i == N-1 and d.DTDO_dataSize % M == 0)
  - size == d.DTDO_dataSize % M, if (i == N-1 and d.DTDO_dataSize % M != 0)
- if b[i] == 0, PTP has not received the PTP Data message carrying this portion of d.DTDO_data

Moreover, we will allocate a buffer of d.DTDO_dataSize Bytes (buffer field of the PTPRQ entry). We will call this buffer buff for simplicity and it will be used to keep saving the portions or blocks of data of d.DTDO_data while they are being obtained. The dataOffset and dataSize fields of the Payload of the PTP Data messages of the DTDO d will be used to save the data in the correct place inside buff.

There exists a bijection between b and buff, meaning that if b[i] == 1, then buff is filled, for sure, with the portion of data of the DTDO starting at buff[i*M] and ending at buff[i*M+size-1], where size is set as defined before.

From now on, error and congestion control will be based on this bitmap b and order control based on this buffer buff.

The PTPRQ entry e will start a timer t that will expire every T_CHECKSTATUSPTP milliseconds (see Appendix A).

When t expires, the current state of b (number of bits set to 1 in b) will be compared with the state of b in the last expiration of t (i.e., the previous state of b saved in the previousState field of e). From the implementation point of view, every time t expires, we will have to save the current state of b in order to be able to use it in the next expiration of t as the previous state of b (previousState field of the PTPRE entry) → **Continue1()**.

}

Continue1()

/*

*** Continue1() is the callback of the timer t**

***/**

{

If (previousState == -1)

send_request(1)

Else

PTP will check the previous state with the current state of b:

RECEIVED_PTPDATA = current_state() – previousState

If $(RECEIVED\_PTPDATA == 0)$

changeRecipientCounter++

If (changeRecipientCounter >=

MAX_PTPCONSECUTIVELOSES)

change_recipient()

Else

Here we could set 1 or 0, depending on if we are not receiving PTP Data messages due to PTP Data loses (0 would be better), or due to PTP Request loses (1 would be better), or due to degradation of the T_DATATXPTP compared to T_CHECKSTATUSPTP (1 would be better), etc. We choose 1 for avoiding the last case.

send_request(1)

EndIf

Else

changeRecipientCounter = 0

EXPECTED_PTPDATA = "Appendix A" (calculated using the fields lastNumPTPDataRequested (i.e., NUM_REQPACKETS in Appendix A), lastTimestampPTPRequest (i.e., TIME_REQPACKETS in Appendix A), lastDataRateFeedback (i.e., F_x in Appendix A) and lastDataRateFeedbackTime (i.e., t_x in Appendix A) of the PTPRQ entry)

If $\left(RECEIVED\_PTPDATA \geq \left\lfloor \frac{EXPECTED\_PTPDATA}{2} \right\rfloor \right)$

send_request(1)

ElseIf $\left(0 < RECEIVED\_PTPDATA < \left\lfloor \frac{EXPECTED\_PTPDATA}{2} \right\rfloor \right)$

send_request(0)

EndIf

EndIf

EndIf

previousState = current_state()

t will be rescheduled again to T_CHECKSTATUSPTP → **Continue1()**

}

send_request(feedback_param={0|1})

{

First of all we have to check if the current recipient r is still in the DTDO Table (i.e., if there is at least one routing path to r).

If (exists)

PTP sends a PTP Request message to the current recipient r, requesting for all the portions of d.DTDO_data that have not been obtained yet and at the same time, derequesting all the portions of d.DTDO_data that have been already obtained (PTP Request messages will be forwarded according to MIFO to r):

· PTPHeader.Type = 1 (PTP Request message)

· PTPHeader.pluginID = d.DTDO_pluginID

· PTPHeader.pluginTag = d.DTDO_pluginTag

· PTPHeader.originator = d.DTDO_originator

· PTPHeader.timestamp = d.DTDO_timestamp

· PTPPayload.dataRateFeedback = feedback_param

· PTPPayload.bitmapSize = N

· PTPPayload.bitmap = b

We save in lastNumPTPDataRequested field the number of requested PTP Data messages in the previous PTP Request message. That is, the number of bits set to 0 in b, or in other words, the value of the bitmapCounter field:

e.lastNumPTPDataRequested = e.bitmapCounter

We save in lastTimestampPTPRequest field the timestamp in which we are sending this PTP Request message:

e.lastTimestampPTPRequest = "now"

return

Else

change_recipient()

EndIf

}

change_recipient()

{

changeRecipientCounter = 0

At this point, we realized that the current recipient r from which we are getting the d.DTDO_data (i.e. we are sending the PTP Request messages to r) has become a bad recipient.

First, we check if the recipient is still in the DTDO Table (i.e., if there is at least one routing path to r).

If (exists)

In this case, we want to change to another recipient, and if there are not more recipients for the DTDO d, then we want to keep using the recipient r as current recipient.

If (there are several recipients different from r)

We will choose the one with best mean RH. And in case of tie in the best mean RH, a random one will be chosen. This will be the new recipient to use as current recipient r.

Now we can continue getting the DTDO from the new recipient taking advantage of the partial DTDO_data we have already obtained in the past:

· currentTTL = link(d,r).TTL

send_request(1)

return

Else

In this case, there are no other recipients for this DTDO. We will keep using the recipient r as current recipient.

send_request(1)

return

EndIf

Else

In this case, we want to change to another recipient, and if there are not more recipients for this DTDO d, we will pause the process of getting the DTDO.

If (there are one or more recipients in the DTDO Table for d)

We will choose the one with best mean RH. And in case of tie in the best mean RH, a random one will be chosen. This will be the new recipient to use as current recipient r.

Now we can continue getting the DTDO from the new recipient taking advantage of the partial DTDO_data we have already obtained in the past:

· currentTTL = link(d,r).TTL

send_request(1)

return

Else

In the case that there are not remaining recipients, PTP will pause the process of getting the DTDO:

· **stop** timer t

· e.inPause = true

This pause will persist until a DTDODM adds new recipients for this DTDO (or of course, every DTDORQ entry requesting this DTDO calls stopGetDTDO, which in this case the process of getting the DTDO will be stopped and the PTPRQ entry deleted).

EndIf

EndIf

}

D.4.1.1 Receiving PTP Request messages

We will set r = the first unique ID of the SourceRoutingList field of the MIFO Header of the PTP Request message

Once a PTP Request message req is received, we will check if an entry with req.pluginID, req.pluginTag, req.originator, req.timestamp and Requester==r already exists in the PTPSQ (we will call this entry e).

If (**no** exists)

We will check if we have the DTDO d={req.pluginID,req.pluginTag,req.originator,req.timestamp} in the DTDO Table with d.DTDO_data!=NULL.

If (exists)

If (# of entries of PTPSQ == MAXNUM_PTPSQENTRIES)

We will send a PTP Null message to the requester r telling him that we do not have enough resources to serve the DTDO (PTP Null messages will be forwarded according to MIFO to r):

- PTPHeader.Type = 2 (PTP Null message)
- PTPHeader.pluginID = req.pluginID
- PTPHeader.pluginTag = req.pluginTag
- PTPHeader.originator = req.originator
- PTPHeader.timestamp = req.timestamp
- PTPPayload.Type = 1

exit

Else

We will create a new entry in the PTPSQ (which we will call e) with the corresponding req.pluginID, req.pluginTag, req.originator, req.timestamp and r fields.

If (req.Payload.dataRateFeedback == 0)

e.currentDataRateFeedback=1

Else

e.currentDataRateFeedback=

req.Payload.dataRateFeedback

EndIf

We will schedule e.timeout to T_PTPSQENTRYOUT= T_CHECKSTATUSPTP*(MAX_PTPCONSECUTIVELOSES+1)μs

We will copy the bitmap (array of bits) of the request to the bitmap of the entry e (we will call this bitmap b for simplification):

b = e.bitmap = req.bitmap (req.bitmap has size == req.bitmapSize bits)

In this way, assuming that b[i] (0 <= i < N) is the i-th bit of the bitmap b, we define that:

- if b[i] == 0, PTP still has to send the PTP Data message carrying the portion of d.DTDO_data with offset == i*M and
  - size == M, if (i < N-1) or (i == N-1 and d.DTDO_dataSize % M == 0)
  - size == d.DTDO_dataSize % M, if (i == N-1 and d.DTDO_dataSize % M != 0)
- if b[i] == 1, PTP does not has to send the PTP Data message carrying this portion of d.DTDO_data

Now, we initialize the pointer p1 to the position of a random bit of b equal to 0 (random and not the first one for avoiding the same problem of TCP when setting up the initial sequence number during the 3WHS)

Continue2()

EndIf

Else

We will send a PTP Null message to the requester r telling him that we do not have this DTDO available (PTP Null messages will be forwarded according to MIFO to r):

- PTPHeader.Type = 2 (PTP Null message)
- PTPHeader.pluginID = req.pluginID
- PTPHeader.pluginTag = req.pluginTag
- PTPHeader.originator = req.originator
- PTPHeader.timestamp = req.timestamp
- PTPPayload.Type = 0

exit

EndIf

Else

We will check if we have the DTDO d={req.pluginID,req.pluginTag,req.originator,req.timestamp} in the DTDO Table with d.DTDO_data!=NULL

If (exists)

We schedue e.timeout to T_PTPSQENTRYOUT = T_CHECKSTATUSPTP * (MAX_PTPCONSECUTIVELOSES+1) μs

We copy the bitmap (array of bits) of the request to the bitmap of the entry e:

e.bitmap = req.bitmap (req.bitmap has size == req.bitmapSize bits)

Now, we update (rounding b if it is necessary) the pointer p1 to the next position (i.e., starting from p1+1) of b equal to 0 (note that due to the overwrite of the bitmap it can be possible that the current position of p1 becomes 0 again):

```
For (i=p1+1; i<N; i++)
{
      If (b[i] == 0)
            p1 = i
            found!
      EndIf
}
If (i == N)
      For (i=0; i≤p1; i++)
      {
            If (b[i] == 0)
                  p1 = i
                  found!
            EndIf
      }
EndIf

If (req.Payload.dataRateFeedback > 0)
      e.currentDataRateFeedback =
```

req.Payload.dataRateFeedback

Continue2()

Else

If (timer t is not scheduled right now)

Continue2()

Else

Let timer t expire with its current scheduled time

EndIf

EndIf

Else

We will send a PTP Null message to the requester e.Requester telling him that we do not have this DTDO available (PTP Null messages will be forwarded according to MIFO to e.Requester):

· PTPHeader.Type = 2 (PTP Null message)

· PTPHeader.pluginID = req.pluginID

· PTPHeader.pluginTag = req.pluginTag

· PTPHeader.originator = req.originator

· PTPHeader.timestamp = req.timestamp

· PTPPayload.Type = 0

stop timer t (the PTPSQ entry e will be deleted by the e.timeout or reused for possible future PTP Request messages)

exit

EndIf

EndIf

Continue2()

/*

*** Continue2() is the callback of the timer t**

***/**

{

We will check if we have the DTDO d={req.pluginID,req.pluginTag,req.originator,req.timestamp} in the DTDO Table with d.DTDO_data!=NULL

If (exists)

We will send a PTP Data message to the requester e.Requester (PTP Data messages will be forwarded according to MIFO to e.Requester):

- PTPHeader.Type = 0 (PTP Data message)
- PTPHeader.pluginID = req.pluginID
- PTPHeader.pluginTag = req.pluginTag
- PTPHeader.originator = req.originator
- PTPHeader.timestamp = req.timestamp
- PTPPayload.dataRateFeedback =

  e.currentDataRateFeedback
- PTPPayload.dataOffset = p1*M
- **If** (p1 < (N-1))

  PTPPayload.dataSize = M

  ElseIf ((p1==(N-1)) && (d.DTDO_dataSize%M == 0))

  PTPPayload.dataSize = M

  ElseIf ((p1==(N-1)) && (d.DTDO_dataSize%M != 0))

  PTPPayload.dataSize = d.DTDO_dataSize%M

  EndIf
- PTPPayload.Portion = the block of data of d.DTDO_data with offset == PTPPayload.dataOffset and size == PTPPayload.dataSize

Moreover, PTP will ask MIFO the number of hops of the routing path used to send the previous PTP Data message:

nhops = MIFO.#hopsLastRoutingPathUsed(r)

If (nhops < 1)

In the case no routing path is found, PTP does not take any action, just continuing with the transmission algorithm as if a routing path of 3 was used. And if really the requester r has disappeared from the network, the timer e.timeout will do the job of deleting this PTPSQ entry

nhops = 3

EndIf

We will update the bitmap b:

b[p1] = 1

We will update the pointer p1 to the next bit set to 0 in b (rounding b if it is necessary):

```
p1_tmp = p1
For (i=p1+1; i<N; i++)
{
      If (b[i] == 0)
            p1 = i
            break
      EndIf
}
If (p1 == N)
      For (i=0; i<=p1; i++)
      {
            If (b[i] == 0)
                  p1 = i
                  break
            EndIf
      }
      If (i == p1+1)
            stop timer t (the PTPSQ entry e will be deleted by the e.timeout or reused for possible future PTP Request messages)
            exit
      EndIf
EndIf
```

$$TMP = \min(t_{TX\_DATA}) + \frac{(e.currentDataRateFeedback - 1)\min(t_{TX\_DATA})}{PTP\_FACTOR}$$

If (nhops == 1)

$$T\_DATATXPTP = TMP$$

ElseIf (nhops == 2)

$$T\_DATATXPTP = 2TMP$$

Else

$$T\_DATATXPTP = 3TMP$$

EndIf

e.currentDataRateFeedback++

We start a timer t that will expire in T_DATATXPTP milliseconds (see Appendix A) and that will execute **Continue2()** when it expires

Else

We will send a PTP Null message to the requester e.Requester telling him that we do not have this DTDO available (PTP Null messages will be forwarded according to MIFO to e.Requester):

· PTPHeader.Type = 2 (PTP Null message)

· PTPHeader.pluginID = req.pluginID

· PTPHeader.pluginTag = req.pluginTag

· PTPHeader.originator = req.originator

· PTPHeader.timestamp = req.timestamp

· PTPPayload.Type = 0

stop timer t (the PTPSQ entry e will be deleted by the e.timeout or reused for possible future PTP Request messages)

exit

EndIf

}

PTPSQEntry.timeout_expiration()

{

If the timeout of a PTPSQ entry e expires, the entry will be deleted:

· **stop** timer t

- **free** bitmap b
- And finally **delete** e

}

D.4.1.2 Receiving PTP Data messages

When a PTP Data message dat is received, we will check if there exists an entry e with dat.pluginID, dat.pluginTag, dat.originator and dat.timestamp in the PTPRQ (and for sure a DTDO d={dat.pluginID,dat.pluginTag,dat.originator,dat.timestamp} in the DTDO Table)

If (both exist)

We will update the bitmap b of the entry e by setting to 0 the bit corresponding to the portion of data that dat is carrying:

b[dat.Payload.dataOffset/M] = 1

We will update the buffer buff of the entry e by storing, in the correct place, the portion of data that dat is carrying:

memcpy(buff[dat.Payload.dataOffset],dat.Payload.Portion,

dat.Payload.dataSize)

e.bitmapCounter--

If (e.bitmapCounter == 0)

PTP detects that b has all bits set to 1. In this case, the whole d.DTDO_data has been obtained, and then:

If (d.DTDO_isPublic)

We will try to cache the data obtained (buff), in the DTDO Table

If (data is cacheable)

We will cache buff in the corresponding d.DTDO_data field

We will set d.DTDO_dataTTL to e.currentTTL (Note that in we use the TTL of the last recipient used to get the DTDO. But it would be possible to calculate, for example, a mean or the maximum of the TTLs of the several possible recipients that have been used to get the DTDO)

We will start the d.DTDO_dataTimeout

Else

We just do not cache buff (but we will not remove d from the DTDO Table, it will be done automatically by DELTOYA when nobody uses d)

EndIf

EndIf

We notify each observer of the list{observer} with a **copy** of buff.

We **stop** timer t, we delete b and buff and finally, the corresponding PTPRQ entry is deleted.

Else

e.lastDataRateFeedback=dat.Payload.dataRateFeedback

e.lastDataRateFeedbackTime="now"

EndIf

Else

Discard packet

EndIf

D.4.1.3 Receiving PTP Null messages

When a PTP Null message n is received, we will check if there exists an entry e with n.pluginID, n.pluginTag, n.originator and n.timestamp in the PTPRQ (and for sure a DTDO d={n.pluginID,n.pluginTag,n.originator,n.timestamp} in the DTDO Table)

If (both exist)

If (the sender of the PTP Null message (the first unique ID of the SourceRoutingList field of the MIFO Header of the PTP Null message) is equal to the current recipient r used by this PTPRQ entry to get the DTDO)

If (this current recipient r is in the DTDO Table)

If ((n.Payload.Type == 0) || (n.Payload.Type == 1))

stop timer t

e.inPause = true (but ready to restart it once the link is updated)

We delete the link between the DTDO d and r (if any)

Ready()

EndIf

```
            Else
                  stop timer t
                  e.inPause = true (but ready to restart it)
                  Ready()
            EndIf
      Else
            Discard packet
      EndIf
Else
      Discard packet
EndIf
```

D.4.2 stopGetDTDO

PTP will be notified when a DTDORQ entry (see section 9) expires and wants to stop (if possible) all the possible `getDTDO` calls of the DTDOs that have been requested but that have not been able to be obtained during its MRT period (i.e., the DTDO with status==KO, that is, discovered but not obtained):

`stopGetDTDO(observer)`

where:

- `observer`

  A field that allows PTP identifying which are the possible PTPRQ entries that have to be stopped because there is not any observer waiting for the DTDO they are trying to obtain

`stopGetDTDO` will stop any `getDTDO` call done by the `observer` and currently being executed (i.e., currently trying to obtain the DTDO), but only if there are not more observers waiting for this DTDO. In other words, the process of getting a DTDO by PTP is started by an observer, and during this process, other observers can become also interested in this DTDO. So, only when there is not any observer interested in a DTDO, the process of getting this DTDO will be able to be stopped. From the implementation point of view, `stopGetDTDO` will remove the value `observer` from the list{observer} of all the entries of the PTPRQ in which the value `observer` appears. And moreover, for each of these entries, if its list{observer} becomes empty, the process of getting the corresponding DTDO will be stopped and the PTPRQ entry deleted.

D.5 PTP Calculations

Assuming 802.11b and DSSS as PHY layer, we have the following values of Table 18 and Table 19.

Table 18. 802.11b and DSSS parameters

$t_{PREAMBLE}$	t_{PLCP}	DIFS	SIFS	CW_{min}	Slot time (σ)
144 μs	48 μs	50 μs	10 μs	31	20 μs

Table 19. 802.11b and DSSS parameters (Cont.)

Data Rate (Mbps)	Modulation Type	Coding	Bits/Sym	Symbol rate (MSym/sec)
1	BPSK	11 bit Barker Code	1	1
2	QPSK	11 bit Barker Code	2	1
5.5	QPSK	8 bit CCK	4	1.375
11	QPSK	8 bit CCK	8	1.375

In Figure 94 we can see the format and size of the frames involved in our calculations (assuming maximum size when a field has not static size).

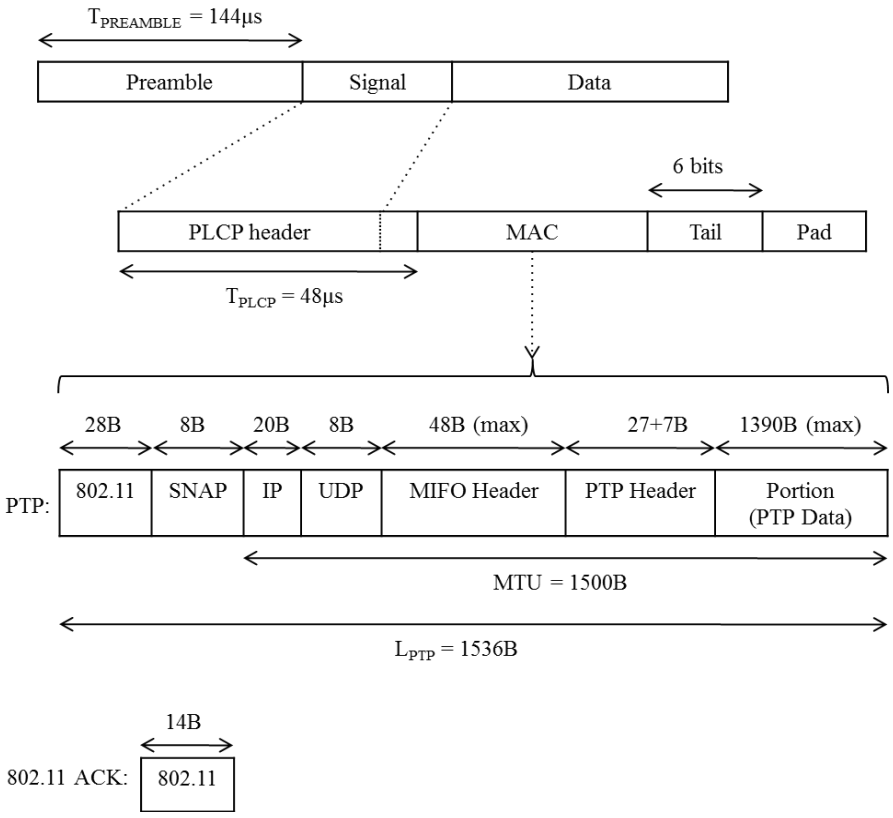

Figure 94. Format and Size of PTP frames

In Figure 95 we can see the 802.11 retransmission algorithm. For our calculations, we will take the mean backoff counter at every retransmission.

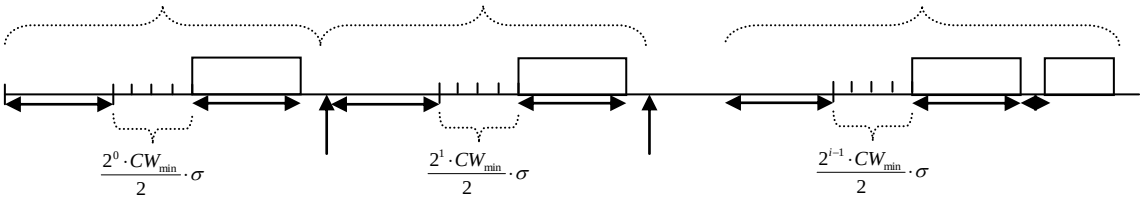

Figure 95. Mean Backoff Counter Per Retransmission

Moreover, for our calculations, we will assume:

- During the backoff period, no other station starts a transmission. The backoff counter will not be interrupted.
- The transmitter is the only node trying to access to the channel.
- Between retransmissions of the same frame, the transmitter does not switch its wireless card to reception mode.
- Propagation time = 0.
- Processing time (store and forward) = 0.

Then, the minimum time of n unacknowledged retransmissions (i.e., at maximum rate) of a PTP Data or Request message will be:

$$E[t_{n\,\mathrm{Re}TX}] = \sum_{i=1}^{n} DIFS + \left(\frac{2^{i-1} \cdot CW_{\min}}{2} ? \sigma \right) + t_{FRAME} + t_{ACK\_TOUT} =$$

$$= n \cdot (DIFS + t_{FRAME} + t_{ACK\_TOUT}) + \left(CW_{\min} \cdot \sigma \cdot \sum_{i=1}^{n} 2^{i-2} \right) =$$

$$= n \cdot (DIFS + t_{FRAME} + t_{ACK\_TOUT}) + \left(CW_{\min} \cdot \sigma \cdot \frac{2^n - 1}{2} \right)$$

where

$$t_{FRAME} = t_{PREAMBLE} + t_{PLCP} + L_{PTP(SYM)} \cdot T_{SYM} =$$

$$= t_{PREAMBLE} + t_{PLCP} + \left\lceil \frac{L_{PTP} \cdot 8 + tail\_bits}{Data\_bits\_per\_symbol} \right\rceil \cdot T_{SYM} =$$

$$= 144 + 48 + \left\lceil \frac{1536 \cdot 8 + 6}{8} \right\rceil \cdot 1.375 \approx 2305 \mu s$$

And the average time of a successful transmission which has needed i-1 retransmissions (i=1..N) will be:

$$E[t_{iTXOK}] = E[t_{i-1\,\mathrm{ReTX}}] + E[t_{i-thTX}] =$$
$$= (i-1)\cdot\left[\left(DIFS + t_{FRAME} + t_{ACK\_TOUT}\right) + \left(CW_{\min}\cdot\sigma\cdot\frac{2^{i-1}-1}{2}\right)\right] +$$
$$+\left[DIFS + \left(\frac{2^{i-1}\cdot CW_{\min}}{2}\cdot\sigma\right) + t_{FRAME} + SIFS + t_{80211ACK}\right]$$

where

$$t_{80211ACK} = t_{PREAMBLE} + t_{PLCP} + L_{80211ACK(SYM)}\cdot T_{SYM} =$$
$$= t_{PREAMBLE} + t_{PLCP} + \left\lceil\frac{L_{80211ACK}\cdot 8 + tail\_bits}{Data\_bits\_per\_symbol}\right\rceil\cdot T_{SYM} =$$
$$= 144 + 48 + \left\lceil\frac{14\cdot 8 + 6}{8}\right\rceil\cdot 1.375 \approx 212.28\mu s$$

Then, given all these parameters, assumptions and calculations, we can calculate, from the point of view of PTP, the minimum time to receive PTP_NUMPACKETS PTP Data messages once the PTP Request message asking for these PTP Data messages has been sent, considering the number of hops of the routing paths and the PTP Transmission algorithm, and assuming that the PTP Transmission algorithm starts at maximum rate. See Figure 96.

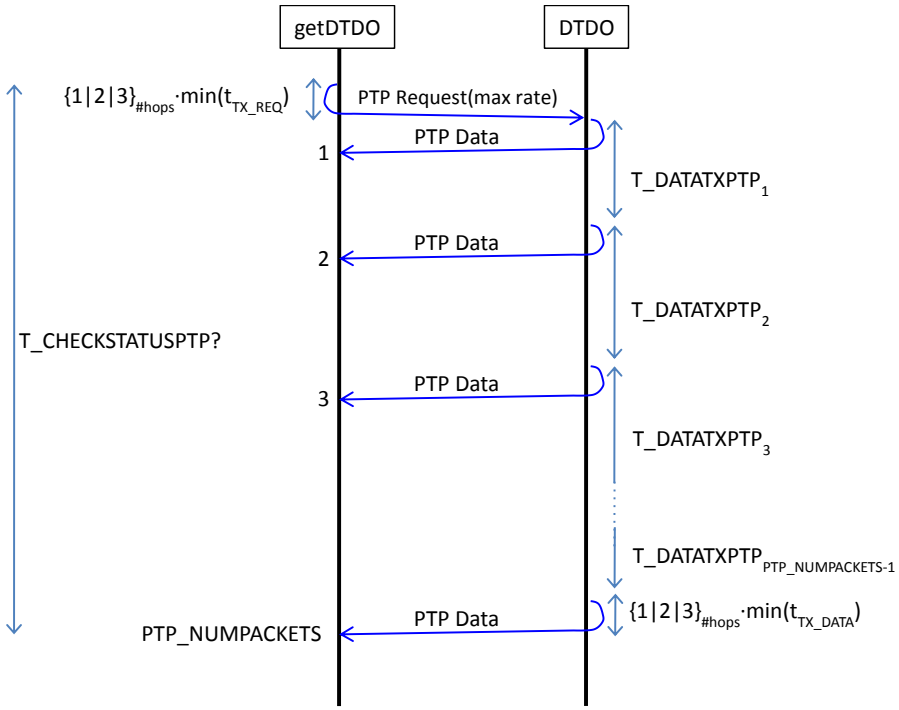

Figure 96. PTP Transmission Algorithm

$$T\_CHECKSTATUSPTP = \{1\,|\,2\,|\,3\}_{\#hops}\min(t_{TX\_REQ}) + T\_DATATXPTP_1 + \ldots$$
$$\ldots + T\_DATATXPTP_{PTP\_NUMPACKETS-1} + \{1\,|\,2\,|\,3\}_{\#hops}\min(t_{TX\_DATA})$$

where

$$T\_DATATXPTP_i = \{1 \mid 2 \mid 3\}_{\#hops} \min(t_{TX\_DATA}) + \frac{(i-1)\min(t_{TX\_DATA})}{PTP\_FACTOR}$$

By default, we took $\{1 \mid 2 \mid 3\}_{\#hops} = 2$ as 2 is the mean number of 1, 2 and 3 (although we know the recipient and we have a set of routing paths to this recipient, there is no reason these routing paths are the same than the ones used by the recipient to send us the PTP Data messages, and so, we cannot be sure if the routing paths used by the recipient are of one, two or more than two hops).

So,

$$T\_CHECKSTATUSPTP = 2\min(t_{TX\_REQ}) +$$
$$+ \sum_{i=1}^{PTP\_NUMPACKETS-1} \left(2\min(t_{TX\_DATA}) + \frac{(i-1)\min(t_{TX\_DATA})}{PTP\_FACTOR} \right) + 2\min(t_{TX\_DATA}) =$$
$$= 2\min(t_{TX\_REQ}) + 2PTP\_NUMPACKETS\min(t_{TX\_DATA}) +$$
$$+ \frac{\min(t_{TX\_DATA})}{PTP\_FACTOR} \times \sum_{i=1}^{PTP\_NUMPACKETS-1} (i-1) = 2\min(t_{TX\_REQ}) + 2PTP\_NUMPACKETS\min(t_{TX\_DATA}) +$$
$$+ \frac{\min(t_{TX\_DATA})}{PTP\_FACTOR} \left(\frac{(PTP\_NUMPACKETS-1)PTP\_NUMPACKETS}{2} - PTP\_NUMPACKETS + 1 \right) =$$
$$= 2\min(t_{TX\_REQ}) + \frac{PTP\_NUMPACKETS^2 + (4PTP\_FACTOR - 3)PTP\_NUMPACKETS + 2}{2PTP\_FACTOR} \min(t_{TX\_DATA})$$

And if for simplification,

$$\min(t_{TX\_REQ}) = \min(t_{TX\_DATA})$$

Then,

$$T\_CHECKSTATUSPTP =$$
$$= \frac{PTP\_NUMPACKETS^2 + (4PTP\_FACTOR - 3)PTP\_NUMPACKETS + 4PTP\_FACTOR + 2}{2PTP\_FACTOR} \min(t_{TX\_DATA})$$

where

$$\min(t_{TX\_DATA}) = E[t_{1TXOK}] = 50 + \left(\frac{31}{2} \times 20 \right) + 2305 + 10 + 212.28 = 2888.28\mu s$$

and

$$PTP\_FACTOR = 2$$

and

$$PTP\_NUMPACKETS = 5$$

A reasonable value of PTP_NUMPACKETS is 5 PTP Data messages, meaning that only when DTDOs are greater than 5 PTP Data messages, the timer T_CHECKSTATUSPTP will expire at least once, and transmission control will be done. Greater values have the problem of performing transmission control every too much time. Lesser values have the problem of performing transmission control too much frequently, and so, consuming too much processing in the device. Another aspect to bear in mind is that PTP_NUMPACKETS should be greater or equal than MAXAGENETENTRIES_DEST, meaning that at least, before taking any decision in the transmission control algorithm, all the possible routing paths to one destination will be tried.

So,

$$T\_CHECKSTATUSPTP = 43324\mu s$$

Additionally, with the previous calculations, we can calculate the ideal (not by PTP) maximum throughput that can be achieved in one hop paths:

$$V_{ideal1hop} = \frac{\mathrm{Re}al\_Data}{\min(t_{TX\_DATA})} = \frac{\max(size(PTPPDUPortion))}{E[t_{1TXOK}]} = \frac{1413B}{2888.28\mu s} \approx 3.9Mbps$$

And the maximum throughput that can be achieved in two hops paths:

$$V_{ideal2hops} \approx \frac{\mathrm{Re}al\_Data}{2 \cdot \min(t_{TX\_DATA})} \approx 1.95Mbps$$

And the maximum throughput that can be achieved in paths of more than two hops:

$$V_{ideal2hops} \approx \frac{\mathrm{Re}al\_Data}{3 \cdot \min(t_{TX\_DATA})} \approx 1.3Mbps$$

Additionally, with the previous calculations and with PTP_NUMPACKETS==5 and PTP_FACTOR==2, we can calculate the maximum throughput that can be achieved by PTP in one hop paths (assuming a transmission control refresh at maximum rate is done every T_CHECKSTATUSPTP period):

$$V_{ideal1hop} = \frac{\mathrm{Re}al\_Data}{Elapsed\_time} = \frac{7*\max(size(PTPPDUPortion))}{T\_CHECKSTATUSPTP} = \frac{7*1413B}{43324\mu s} \approx 1.8Mbps$$

And the maximum throughput that can be achieved in two hops paths:

$$V_{ideal1hop} = \frac{\mathrm{Re}al\_Data}{Elapsed\_time} = \frac{5*\max(size(PTPPDUPortion))}{T\_CHECKSTATUSPTP} = \frac{5*1413B}{43324\mu s} \approx 1.3Mbps$$

And the maximum throughput that can be achieved in paths of more than two hops:

$$V_{ideal1hop} = \frac{\mathrm{Re}al\_Data}{Elapsed\_time} = \frac{4*\max(size(PTPPDUPortion))}{T\_CHECKSTATUSPTP} = \frac{4*1413B}{43324\mu s} \approx 1Mbps$$

On the other hand, also given all these parameters, assumptions and calculations, we can calculate, from the point of view of PTP, the expected maximum number of PTP Data messages (EXPECTED_PTPDATA) that can be received between the last PTP Request message sent (TIME_REQPACKETS) and "now", knowing that PTP asked for NUM_REQPACKETS PTP Data messages in the last PTP Request message (theoretically sent T_CHECKSTATUSPTP μs before but practically sent at the time instant TIME_REQPACKETS) and that at time instant t_x (being t_x a time instant posterior to TIME_REQPACKETS) PTP received a PTP Data message with dataRateFeedback equal to F_x (this PTP Data message will be the last PTP Data message received, and can or cannot be the last PTP Data message transmitted by the node sending the DTDO_data). See Figure 97.

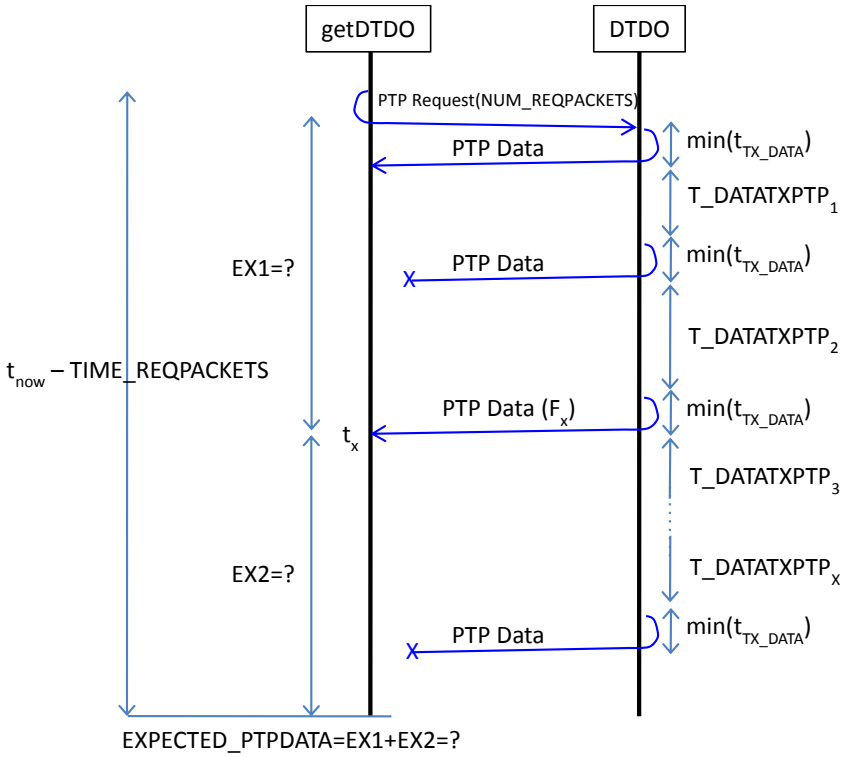

Figure 97. EXPECTED_PTPDATA

To calculate EXPECTED_PTPDATA, we will calculate the expected number of PTP Data messages between when the last PTP Request message was sent and t_x (EX1), and the expected number of PTP Data messages between t_x and "now" (EX2).

On the one hand, we have to find the maximum rounded-down integer value for EX1 such that EX1≥0, EX1<F_x, and it fulfills:

$$T\_DATATXPTP_1 + \cdots + T\_DATATXPTP_{EX1} \leq t_x - TIME\_REQPACKETS$$

where

$$T\_DATATXPTP_i = \{1\,|\,2\,|\,3\}_{\#hops}\min(t_{TX\_DATA}) + \frac{(F_x - i - 1)\min(t_{TX\_DATA})}{PTP\_FACTOR} =$$

$$= 2\min(t_{TX\_DATA}) + \frac{(F_x - i - 1)\min(t_{TX\_DATA})}{PTP\_FACTOR}$$

So,

$$\sum_{i=1}^{EX1} T\_DATATXPTP_i \le t_x - TIME\_REQPACKETS$$

$$\sum_{i=1}^{EX1}\left(2\min(t_{TX\_DATA}) + \frac{(F_x - i - 1)\min(t_{TX\_DATA})}{PTP\_FACTOR}\right) \le t_x - TIME\_REQPACKETS$$

$$2\min(t_{TX\_DATA})EX1 + \frac{\min(t_{TX\_DATA})}{PTP\_FACTOR}\sum_{i=1}^{EX1}(F_x - i - 1) \le t_x - TIME\_REQPACKETS$$

$$2\min(t_{TX\_DATA})EX1 + \frac{\min(t_{TX\_DATA})}{PTP\_FACTOR}\left(F_xEX1 - EX1 - \sum_{1}^{EX1} i\right) \le t_x - TIME\_REQPACKETS$$

$$2\min(t_{TX\_DATA})EX1 + \frac{\min(t_{TX\_DATA})}{PTP\_FACTOR}\left(F_xEX1 - EX1 - \frac{EX1(EX1+1)}{2}\right) \le$$

$$\le t_x - TIME\_REQPACKETS$$

$$2\min(t_{TX\_DATA})EX1 + \frac{-\min(t_{TX\_DATA})EX1^2 + 2\min(t_{TX\_DATA})F_xEX1 - 3\min(t_{TX\_DATA})EX1}{2PTP\_FACTOR} \le$$

$$\le t_x - TIME\_REQPACKETS$$

$$-\min(t_{TX\_DATA})EX1^2 +$$

$$+(4\min(t_{TX\_DATA})PTP\_FACTOR + 2\min(t_{TX\_DATA})F_x - 3\min(t_{TX\_DATA}))EX1 -$$

$$-2PTP\_FACTOR(t_x - TIME\_REQPACKETS) \le 0$$

$$-EX1^2 +$$

$$+(4PTP\_FACTOR + 2F_x - 3)EX1 -$$

$$-\frac{2PTP\_FACTOR(t_x - TIME\_REQPACKETS)}{\min(t_{TX\_DATA})} \le 0$$

And then, from

$$EX1 = \frac{-4PTP\_FACTOR - 2F_x + 3 \pm \sqrt{(4PTP\_FACTOR + 2F_x - 3)^2 - 4\left(\frac{2PTP\_FACTOR(t_x - TIME\_REQPACKETS)}{\min(t_{TX\_DATA})}\right)}}{-2}$$

we have to choose the maximum rounded-down integer root ≥ 0 and < F_x (we will always have a root fulfilling these properties).

On the other hand, we have to find the maximum rounded-down integer value for EX2 such that EX2≥0 and it fulfills:

$$T\_DATATXPTP_1 + \cdots + T\_DATATXPTP_{EX2-1} + \min(t_{TX\_DATA}) \le t_{now} - t_x$$

where

$$T\_DATATXPTP_i = \{1 \mid 2 \mid 3\}_{\#hops}\min(t_{TX\_DATA}) + \frac{(F_x + i - 2)\min(t_{TX\_DATA})}{PTP\_FACTOR} =$$

$$= 2\min(t_{TX\_DATA}) + \frac{(F_x + i - 2)\min(t_{TX\_DATA})}{PTP\_FACTOR}$$

So,

$$\sum_{i=1}^{EX2-1}\left(T\_DATATXPTP_i\right) + \min(t_{TX\_DATA}) \le t_{now} - t_x$$

$$\sum_{i=1}^{EX2-1}\left(2\min(t_{TX\_DATA}) + \frac{(F_x + i - 2)\min(t_{TX\_DATA})}{PTP\_FACTOR}\right) + \min(t_{TX\_DATA}) \le t_{now} - t_x$$

$$2\min(t_{TX\_DATA})(EX2-1) + \frac{\min(t_{TX\_DATA})}{PTP\_FACTOR}\sum_{i=1}^{EX2-1}\left(F_x + i - 2\right) + \min(t_{TX\_DATA}) \le t_{now} - t_x$$

$$2\min(t_{TX\_DATA})EX2 + \frac{\min(t_{TX\_DATA})}{PTP\_FACTOR}\left(F_x(EX2-1) - 2(EX2-1) + \sum_{i=1}^{EX2-1} i\right) \le$$

$$\le t_{now} - t_x + \min(t_{TX\_DATA})$$

$$2EX2 + \frac{1}{PTP\_FACTOR}\left(\frac{2F_xEX2 - 2F_x - 4EX2 + 4 + EX2^2 - EX2)}{2}\right) \le$$

$$\le \frac{t_{now} - t_x + \min(t_{TX\_DATA})}{\min(t_{TX\_DATA})}$$

$$EX2^2 +$$

$$+\left(4PTP\_FACTOR + 2F_x - 5\right)EX2 -$$

$$-2F_x + 4 - \frac{2PTP\_FACTOR(t_{now} - t_x + \min(t_{TX\_DATA}))}{\min(t_{TX\_DATA})} \le 0$$

And finally, from

$$EX2 = \frac{-4PTP\_FACTOR - 2F_x + 5 \pm \sqrt{(4PTP\_FACTOR + 2F_x - 5)^2 - 4\left(-2F_x + 4 - \frac{2PTP\_FACTOR(t_{now} - t_x + \min(t_{TX\_DATA}))}{\min(t_{TX\_DATA})}\right)}}{2}$$

we have to choose the maximum rounded-down integer root ≥ 0 (we will always have a root fulfilling these properties).

And finally,

```
If ((EX1 + EX2) > NUM_REQPACKETS)
      EXPECTED_PTPDATA = NUM_REQPACKETS
Else
      EXPECTED_PTPDATA = EX1 + EX2
EndIf
```

For example, if t_{now}=1000000 µs, t_x=969876 µs, TIME_REQPACKETS=956676 µs, PTP_FACTOR=2, F_x=3 and $\min(t_{TX\_DATA})$=2888.28 µs, then, between t_{now} and t_{now}-TCHECKSTATUSPTP we should have received EXPECTED_PTPDATA = EX1 + EX2 = 2 + 3 = 5, which in fact, it is PTP_NUMPACKETS, since the conditions in this example are the same conditions when we calculated T_CHECKSTATUSPTP for exactly PTP_NUMPACKETS.

In summary, there are two important parameters that define the behavior of PTP in terms of data rate:

- PTP_FACTOR (= 2)
- PTP_NUMPACKETS (= 5)

Appendix E. MIFO Details

E.1 MIFO Header

The fields of the MIFO header are the following:

- **Type (8 bits (unsigned))**

  0

- **Version (8 bits (unsigned))**

  0

- **Reserved (8 bits)**

  Unused

- **DestinationID (64 bits)**

  Unique ID of the destination node of the packet (or 0 in the case of destination broadcast)

- **Size of the SourceRoutingList (8 bits (unsigned))**

  The number of entries of the SourceRoutingList field

- **SourceRoutingList (64*Size bits)**

  The sequence of nodes (unique IDs) for which de unicast packet is travelling along, sorted by order of visit. Each node forwarding the packet will be the responsible for adding its own unique ID, at each hop, at the end of the SourceRoutingList of the packet being forwarded. I.e., the first entry of the SourceRoutingList will be the unique ID of the source node of the packet.

- **TTL (8 bits (unsigned))**

  The Time to Live value used to define the limit of the unicast packet along the network. It will be set initially to MAXALLOWED_HOPS by the source node, and decremented by one by each node in each hop. If TTL reaches 0, then the packet will not be forwarded (and so, discarded)

- **payloadSize (16 bits (unsigned))**

  The size in Bytes of the payload of the packet. payloadSize will be always less or equal to maxSize(UDP payload) - MAX(size(MIFO header)), meaning that no UDP fragmentation will be needed (in fact,

there does not exist UDP fragmentation, and sending a bigger packet would raise an UDP error)

· **Protocol (8 bits (unsigned))**

The protocol identifier of the subsequent upper layer protocol:

- PTP → Protocol == 0
- DTDODM → Protocol == 1
- AGENET → Protocol == 2
- MIMI → Protocol == 3
- RDM → Protocol == 4
- DOCKDM → Protocol == 5
- ASK2ME → Protocol == 6

E.2 MIFO Table

The fields of the MIFO Table are the following:

· **nodeID (64 bits (unsigned))**

The unique ID of a node

· **nodeIP (32 bits (unsigned))**

The currently known IP address of the node nodeID

· **Timeout (struct timeval)**

The time instant when this entry has to be deleted from the table

The key field of the MIFO table is the nodeID, meaning that there cannot be two entries with the same nodeID.

The entries of this table have a timeout value of "insertion or update time plus TOUT_MIFOENTRY" milliseconds. If during TOUT_MIFOENTRY milliseconds the entry is not updated, it will be automatically deleted.

E.3 Source Learning

Every packet received by MIFO will be used to update the MIFO table in this way:

```
When a packet is received, we will get the last unique ID of the
SourceRoutingField of the MIFO header of the packet, and we will check
if there is an entry in the MIFO Table with this unique ID.

If (exists)
```

We will update the nodeIP field of the entry with the source IP address of the IP header of the packet, and we will update the timeout of the entry to TOUT_MIFOENTRY milliseconds.

Else

We will create a new entry in the MIFO Table with nodeID==the last unique ID of the SourceRoutingField of the MIFO header of the packet, and with nodeIP== the source IP address of the IP header of the packet.

We will set the timeout of the entry to TOUT_MIFOENTRY milliseconds.

EndIf

Appendix F. DTDODM Details

The DTDO Discovery Mechanism (DTDODM) is the mechanism in charge of discovering a set of specified DTDOs in the network. The different types of DTDO Discovery Mechanisms (DTDODMs) are the following:

```
a) startDTDODM(pluginID,pluginTag,originator,*,T_QUERYDTDOS,0)

b) startDTDODM(pluginID,*,originator,*,T_QUERYDTDOS,0)

c) startDTDODM(pluginID,pluginTag,*,*,T_QUERYDTDOS,0)

d) startDTDODM(pluginID,*,*,*,T_QUERYDTDOS,0)


e) startDTDODM(recipient,pluginID,pluginTag,originator,*,
               T_QUERYDTDOS,0)
f) startDTDODM(recipient,pluginID,*,originator,*,T_QUERYDTDOS,0)

g) startDTDODM(recipient,pluginID,pluginTag,*,*,T_QUERYDTDOS,0)

h) startDTDODM(recipient,pluginID,*,*,*,T_QUERYDTDOS,0)


i) startDTDODM(listDTDOsToDiscover,MRT,1)

j) startDTDODM(listDTDOsToDiscover,MRT-T_QUERYDTDOS,1)


k) startDTDODM(pluginID,pluginTag,originator,timestamp,MRT,2)

l) startDTDODM(recipient,pluginID,pluginTag,originator,
               timestamp,MRT,2);
```

And in general:

```
err = startDTDODM(pluginID,pluginTag,originator,*,
                  remainingTime,type==0);
err = startDTDODM(recipient,pluginID,pluginTag,originator,*,
                  remainingTime,type==0);
err = startDTDODM(pluginID,list{pluginTag,originator,timestamp},
                  remainingTime,type==1);
err = startDTDODM(pluginID,pluginTag,originator,timestamp,
                  remainingTime,type==2);
err = startDTDODM(recipient,pluginID,pluginTag,originator,timestamp,
                  remainingTime,type==3);
```

Where the last param `type` is a uint8_t indicating if the DTDODM has been started by a `queryDTDOs` call (`type` == 0), by a `requestDTDOs` call (`type` == 1) or due to a `pushDTDO` call (`type` == 2 or 3)

The DTDODM will check:

- MIN_MRT <= `remainingTime` <= MAX_MRT

In case of this check fails, the call will return -1.

Otherwise, the call will:

- Start a flooding using DTDODREQ and DTDODREP messages in order to discover new recipients for specific DTDOs for a period of `remainingTime` milliseconds

F.1 Flooding (The Aggregator)

For each DTDODM started by any entry of the DTDOQQ/DTDORQ/DTDOAQ, DELTOYA will keep sending DTDODREQ messages following a Binary Exponential Backoff for each DTDODREQ, and only will stop the DTDODM when all the DTDOs have been obtained (only in the case of `requestDTDOs` calls and `pushDTDO` calls and not for `queryDTDOs` calls), or if not, when the `remainingTime` of the DTDODM expires.

However, this is from the point of view of the DTDODMs initiated by each DTDOQQ/DTDORQ/DTDOAQ entry. From the point of view of the network, the Aggregator will try to aggregate all the DTDODREQ and DTDODREP messages of the DTDODMs as much as possible, in order to reduce the number of transmissions in the network. See Figure 98.

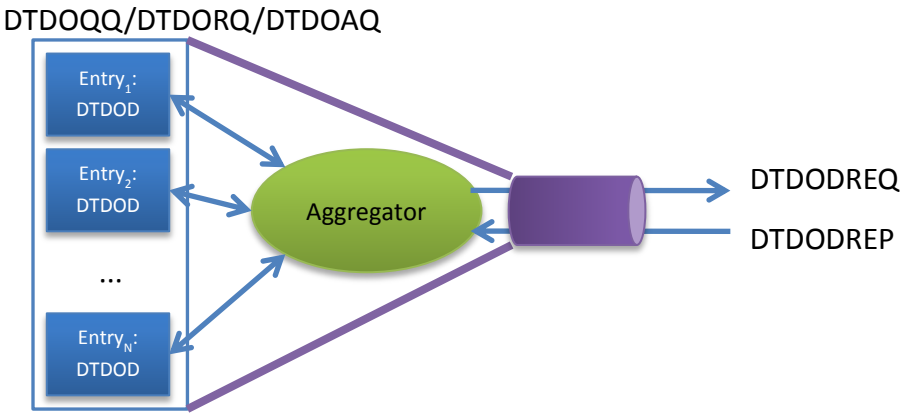

Figure 98. The DTDOQQ/DTDORQ/DTDOAQ Aggregator

For this reason, we have to define the concept of inclusion "⊆" between DTDODMs. For example, a DTDODM for the DTDOs = (pluginIDX,pluginTag=*,origY,timestamp=*,recips=*) already includes a DTDODM for the DTDOs = (pluginIDX,pluginTagZ,origY,timestamp=*,recips=*). In this way, two different entries will be added to the DTDOQQ/DTDORQ/DTDOAQ, but the DTDODM of the second one will be able to take advantage of the DTDODREQ messages of the DTDODM of the first one (e.g. when a DTDODM tells the Aggregator to send a DTDODREQ message for some DTDOs, and the

Aggregator knows that just less or equal than TIMEBETWEEN_INCLUDEDDTDODREQs milliseconds before it sent a DTDODREQ message for at least the same DTDOs, then it can decide to not send the second DTDODREQ, and use the possible DTDODREP of the first DTDODREQ as reply for also the second DTDODREQ).

In the same way, DTDODMs included in another DTDODM will be able to take advantage also of the DTDODREP messages of the first one. In this case, when a DTDODREP will be received, the Aggregator will have to check which of all the DTDOQQ/DTDORQ/DTDOAQ entries will need the DTDODREP.

F.2 DTDO Discovery Request (DTDODREQ)

DTDO Discovery Requests for queries of DTDOs are broadcast messages that will be flooded in the MANET in order to discover which DTDOs and in which nodes are available at that time in the network.

F.2.1 DTDODREQ for queryDTDOs

Each time a DTDODM of a `queryDTDOs` call (i.e., `type==0` in the `startDTDODM` call) wants to send a DTDODREQ, a DTDODREQ message of type 0 is created with the following fields:

- **type (8 bits (unsigned))**

  0

- **DTDODREQ_version (8 bits (unsigned))**

  0

- **DTDODREQ_ID (32 bits (unsigned))**

  A sequence number started from 0 and incremented by one each time a new DTDODREQ message is sent. When DTDODREQ_ID reaches the maximum value (2^32 - 1), the next time it has to be incremented, it will be set again to 0.

- **DTDODREQ_TTL (8 bits (unsigned))**

  The Time to Live value used to define the limit of the DTDODREQ message along the network. It will be set initially to MAXALLOWED_HOPS by the originator of the DTDODREQ, and decremented by one by each node in each hop. If DTDODREQ_TTL reaches 0, then the DTDODREQ message will be discarded and so, not forwarded.

- **Size of the SourceRouting List (8 bits (unsigned))**

  Number of entries of the SourceRoutingList

· **SourceRoutingList{node unique IDs} (Size of the SourceRoutinglist * (64))**

The sequence of nodes (unique IDs) for which the DTDODREQ message is travelling along, sorted by order of visit. Only the originator of the DTDODREQ message will not be included in this list, since it is already included in the last field of the DTDODREQ message. Each node forwarding the DTDODREQ message will be the responsible for adding this information (i.e. its own information), at each hop, at the end of the SourceRoutingList of the DTDODREQ being forwarded.

· **Reserved (32 bits)**

Unused

· **pluginID, pluginTag, originator, recipient (32+32+64+64 bits)**

The fields identifying the set of DTDOs we are querying.

NOTE: originator can never be the unique ID of the node sending the DTDODREQ, i.e. it is forbidden to ask for DTDOs in the network in which we are the originators. If we are the originators, we always and only check the DTDO Table, and even when the DTDO is not in the cache, we will assume that the DTDO does not have to be discovered in the network, since it has been deleted

· **OriginatorID (64 bits)**

The unique ID of the node owning the DTDODM that has generated the DTDODREQ message

DTDODREQ messages of type 0 will be forwarded according to MIFO to the destination node ID 0 (i.e., broadcast).

The operation when a node receives a DTDODREQ message of type 0 from the network is the following:

```
The node checks the SourceRoutingList and the OriginatorID of the
DTDODREQ message.

If (the SourceRoutinglist or the OriginatorID contain its own unique
ID)

      The node discards the DTDODREQ message.

Else

      The node checks if it already has forwarded
      MAXDTDODREQ_TOFORWARD DTDODREQ messages with the same
      DTDODREQ_ID during the last DTDODREQID_LIFE milliseconds

      If (true)
```

```
        The node discards the DTDODREQ message

Else

        If (recipient == NULL)

                If ((pluginTag!=NULL) and (originator!=NULL))

                        /* For sure that the originator does not
                        correspond to the ID of the node sending the
                        DTDODREQ */

                        listDiscoveredDTDOs := empty

                        DELTOYA checks if we have some public
                        DTDOs==(pluginID,pluginTag,originator,*)     in
                        the cache (DTDO_data!=NULL)

                        If (true)

                                listDiscoveredDTDOs.add(the     matching
                                DTDOs with ourTTL)

                        EndIf

                        If (listDiscoveredDTDOs is not empty)

                                generateDTDODREP(listDiscoveredDTDOs,0)

                        EndIf

                ElseIf ((pluginTag==NULL) and (originator!=NULL))

                        Idem than the previous ElseIf but with
                        (pluginID,*,originator,*)

                ElseIf ((pluginTag!=NULL) and (originator==NULL))

                        listDiscoveredDTDOs := empty

                        DELTOYA checks if we have some public
                        DTDOs==(pluginID,pluginTag,*,*) in the cache
                        (DTDO_data!=NULL)

                        If (true)

                                listDiscoveredDTDOs.add(only          the
                                matching DTDOs that have originator !=
                                the Originator ID of the DTDODREQ)

                        EndIf

                        If (listDiscoveredDTDOs is not empty)

                                generateDTDODREP(listDiscoveredDTDOs,0)

                        EndIf

                ElseIf ((pluginTag==NULL) and (originator==NULL))

                        Idem than the previous ElseIf but with
                        (pluginID,*,*,*)

                EndIf
```

```
DTDODREQ_TTL--

If (DTDODREQ_TTL > 0)

      The node forwards the DTDODREQ message
      updating the SourceRoutingList of the DTDODREQ
      by adding at the end of the list its own
      unique ID

Else

      The node discards the DTDODREQ message

EndIf

ElseIf (recipient != NULL)

      If (recipient != "my ID")

            DTDODREQ_TTL--

            If (DTDODREQ_TTL > 0)

                  The node forwards the DTDODREQ message
                  updating the SourceRoutingList of the
                  DTDODREQ by adding at the end of the
                  list its own unique ID

            Else

                  The node discards the DTDODREQ message

            EndIf

      Else

            /* In this case we answer even if the
            originator of the DTDOs corresponds to the
            originator of the DTDODREQ, since it will be
            the originator of this DTDODREQ the one to
            compare this answer with its cache */

            If ((pluginTag!=NULL) and (originator!=NULL))

                  listDiscoveredDTDOs := empty

                  DELTOYA checks if we have some public
                  DTDOs==(pluginID,pluginTag,originator,*)
                  in the cache (DTDO_data!=NULL)

                  If (true)

                        listDiscoveredDTDOs.add(the

                              matching DTDOs with ourTTL)

                  EndIf

                  generateDTDODREP(listDiscoveredDTDOs,0)

            ElseIf ((pluginTag==NULL) and

                  (originator!=NULL))
```

```
                    Idem than the previous ElseIf but with
                    (pluginID,*,originator,*)

                ElseIf ((pluginTag!=NULL) and

                    (originator==NULL)

                    Idem than the previous ElseIf but with
                    (pluginID,pluginTag,*,*)

                ElseIf ((pluginTag==NULL) and

                    (originator==NULL))

                    Idem than the previous ElseIf but with
                   (pluginID,*,*,*)

                EndIf

                The node discards the DTDODREQ message

            EndIf

        EndIf

    EndIf

EndIf
```

F.2.2 DTDODREQ for requestDTDOs

Unlike DTDODREQs for queries of DTDOs, DTDODREQs for requests of DTDOs will be also used in order to add routing information in the DTDO Table.

Each time a DTDODM of a `requestDTDOs` call (i.e., `type==1` in the `startDTDODM` call) wants to send a DTDODREQ (and if implemented, the Aggregator allows sending it), a DTDODREQ message of type 2 is created with the following fields:

- **type (8 bits (unsigned))**

  2

- **DTDODREQ_version (8 bits (unsigned))**

  0

- **DTDODREQ_ID (32 bits (unsigned))**

  A sequence number started from 0 and incremented by one each time a new DTDODREQ message is sent. When DTDODREQ_ID reaches the maximum value (2^32 - 1), the next time it has to be incremented, it will be set again to 0.

- **DTDODREQ_TTL (8 bits (unsigned))**

The Time to Live value used to define the limit of the DTDODREQ message along the network. It will be set initially to MAXALLOWED_HOPS by the originator of the DTDODREQ, and decremented by one by each node in each hop. If DTDODREQ_TTL reaches 0, then the DTDODREQ message will be discarded and so, not forwarded.

· **Size of the SourceRouting List (8 bits (unsigned))**

Number of entries of the SourceRoutingList

· **SourceRoutingList{node unique IDs,level of battery of the node,**

number of explosive neighbors of the node,

timestamp of the node}

(Size of the SourceRoutinglist * (64 + 8_unsigned + 8_unsigned + 64) bits where the 64 bits of the timestamp are 32 bits of timeval.tv_sec plus 32 bits of timeval.tv_usec)

The sequence of nodes (unique IDs) for which the DTDODREQ message is travelling along, sorted by order of visit. Only the unique ID of the originator of the DTDODREQ message will not be included in this list, since it is already included in the last four fields of the DTDODREQ message. Moreover, for each node, the level of battery, the number of explosive neighbors, and a timestamp will be added to the list at the moment when the DTDODREQ message is forwarded. Each node forwarding the DTDODREQ message will be the responsible for adding this information (i.e. its own information), at each hop, at the end of the SourceRoutingList of the DTDODREQ being forwarded.

· **Reserved (32 bits)**

Unused

· **list{pluginTag,originator,timestamp}.size,**

pluginID, list{pluginTag,originator,timestamp},

(16_unsigned + 32 +

list{pluginTag,originator,timestamp}.size*(32+64+64) bits)

The fields identifying the set of DTDOs we are requesting.

NOTE: originator can never be the unique ID of the node sending the DTDODREQ, i.e. it is forbidden to ask for DTDOs in the network in which we are the originators. If we are the originators, we always and only check the DTDO Table, and even when the DTDO is not in the cache, we will assume that the DTDO does not have to be discovered in the network, since it has been deleted

· **OriginatorID (64 bits)**

The unique ID of the node owning the DTDODM that has generated the DTDODREQ message

· **Level of battery of the OriginatorID node (8 bits (unsigned))**

Level of battery of the originator node at the moment of sending the DTDODREQ message

· **Number of explosive neighbors of the OriginatorID node (8 bits (unsigned))**

Number of explosive neighnors of the originator node at the moment of sending the DTDODREQ message

· **Timestamp of the OriginatorID node (64 bits: 32 bits of timeval.tv_sec plus 32 bits of timeval.tv_usec)**

Time instant when the DTDODREQ message is sent

DTDODREQ messages of type 2 will be forwarded according to MIFO to the destination node ID 0 (i.e., broadcast).

The operation when a node receives a DTDODREQ message of type 2 from the network is the following:

```
The node checks the SourceRoutingList and the OriginatorID of the
DTDODREQ message.

If (the SourceRoutingList or the OriginatorID contain its own unique
ID)

      The node discards the DTDODREQ message.

Else

      The node checks if it already has forwarded
      MAXDTDODREQ_TOFORWARD DTDODREQ messages with the same
      DTDODREQ_ID during the last DTDODREQID_LIFE milliseconds

      If (true)

            The node discards the DTDODREQ message

      Else
```

```
/* For sure that the originator does not correspond to the
ID of the node sending the DTDODREQ */

The node inserts/updates (if it is the case) to the DTDO
Table the routing information related with the routing path
to the originator node of the DTDODREQ message. All these
routes will be added with Status=1. In other words, from
the point of view of AGENET, the DTDODREQ message will be
processed as if it was an AGROHello

listDiscoveredDTDOs := empty

For each i-pluginTag,i-originator,i-timestamp of
list{pluginTag,originator,timestamp}

      DELTOYA checks if we have the public DTDO
      (pluginID,i-pluginTag,i-originator,i-timestamp) in
      the cache (DTDO_data!=NULL)

      If (true)

            listDiscoveredDTDOs.add(pluginID,

                        i-pluginTag,i-originator,

                        i-timestamp,ourTTL)

      EndIf

EndFor

If (listDiscoveredDTDOs is not empty)

      generateDTDODREP(listDiscoveredDTDOs,1)

      The node becomes Master of the RZ (see section 12)

EndIf

DTDODREQ_TTL--

If (DTDODREQ_TTL > 0)

      The node forwards the DTDODREQ message updating the
      SourceRoutingList of the DTDODREQ by adding at the
      end of the list its own (unique ID,level of
      battery,number of explosive neighbors,timestamp)

Else

      The node discards the DTDODREQ message

EndIf

   EndIf

EndIf
```

F.2.3 DTDODREQ due to pushDTDO

F.2.3.1 For Public DTDOs

Each time a DTDODM due to a `pushDTDO` call (i.e., `type==2` in the `startDTDODM` call) wants to send a DTDODREQ, a DTDODREQ message of type 4 is created with the following fields:

· **type (8 bits (unsigned))**

4

· **DTDODREQ_version (8 bits (unsigned))**

0

· **DTDODREQ_ID (32 bits (unsigned))**

A sequence number started from 0 and incremented by one each time a new DTDODREQ message is sent. When DTDODREQ_ID reaches the maximum value (2^32 - 1), the next time it has to be incremented, it will be set again to 0.

· **DTDODREQ_TTL (8 bits (unsigned))**

The Time to Live value used to define the limit of the DTDODREQ message along the network. It will be set initially to MAXALLOWED_HOPS by the originator of the DTDODREQ, and decremented by one by each node in each hop. If DTDODREQ_TTL reaches 0, then the DTDODREQ message will be discarded and so, not forwarded.

· **Size of the SourceRouting List (8 bits (unsigned))**

Number of entries of the SourceRoutingList

· **SourceRoutingList{node unique ID,level of battery of the node,**

number of explosive neighbors of the node,

timestamp of the node}

(Size of the SourceRoutinglist * (64 + 8_unsigned + 8_unsigned + 64) bits where the 64 bits of the timestamp are 32 bits of timeval.tv_sec plus 32 bits of timeval.tv_usec)

The sequence of nodes (unique IDs) for which the DTDODREQ message is travelling along, sorted by order of visit. Only the originator of the DTDODREQ message will not be included in this list, since it is already included in the last four fields of the DTDODREQ message. Moreover, for each node, the level of battery, the number of explosive neigbors, and a timestamp will be added to the list at the moment when the DTDODREQ message is forwarded. Each node forwarding the DTDODREQ message will be the responsible for adding this information

(i.e. its own information), at each hop, at the end of the SourceRoutingList of the DTDODREQ being forwarded.

· **Reserved (32 bits)**

Unused

· **pluginID, pluginTag, originator, timestamp (32+32+64+64 bits)**

The fields identifying the DTDO we are asking.

· **OriginatorID (64 bits)**

The unique ID of the node owning the DTDODM that has generated the DTDODREQ message

· **Level of battery of the OriginatorID node (8 bits (unsigned))**

Level of battery of the originator node at the moment of sending the DTDODREQ message

· **Number of explosive neighbors of the OriginatorID node (8 bits (unsigned))**

Number of explosive neighnors of the originator node at the moment of sending the DTDODREQ message

· **Timestamp of the OriginatorID node (64 bits: 32 bits of timeval.tv_sec plus 32 bits of timeval.tv_usec)**

Time instant when the DTDODREQ message is sent

DTDODREQ messages of type 4 will be forwarded according to MIFO to the destination node ID 0 (i.e., broadcast).

The operation when a node receives a DTDODREQ message of type 4 from the network is the following:

```
The node checks the SourceRoutingList and the OriginatorID of the
DTDODREQ message.

If (the SourceRoutingList or the OriginatorID contain its own unique
ID)

      The node discards the DTDODREQ message.

Else

      The node checks if it already has forwarded
      MAXDTDODREQ_TOFORWARD DTDODREQ messages with the same
      DTDODREQ_ID during the last DTDODREQID_LIFE milliseconds

      If (true)
```

```
        The node discards the DTDODREQ message

    Else

        /* For sure that the originator does not correspond to the
        ID of the node sending the DTDODREQ */

        The node inserts/updates (if it is the case) to the DTDO
        Table the routing information related with the routing path
        to the originator node of the DTDODREQ message. All these
        routes will be added with Status=1. In other words, from
        the point of view of AGENET, the DTDODREQ message will be
        processed as if it was an AGROHello.


        DELTOYA checks if we have the public DTDO (pluginID,
        pluginTag,originator,timestamp) in the cache
        (DTDO_data!=NULL)

        If (true)

            generateDTDODREP(pluginID,pluginTag,originator,
            timestamp,ourTTL,2)

            The node becomes Master of the RZ (see section 12)

        EndIf


        DTDODREQ_TTL--

        If (DTDODREQ_TTL > 0)

            The node forwards the DTDODREQ message updating the
            SourceRoutingList of the DTDODREQ by adding at the
            end of the list its own (unique ID,level of
            battery,number of explosive neighbors,timestamp)

        Else

            The node discards the DTDODREQ message

        EndIf

    EndIf

EndIf
```

F.2.3.2 For Private DTDOs

Each time a DTDODM due to a `pushDTDO` call (i.e., `type==3` in the `startDTDODM` call) wants to send a DTDODREQ, a DTDODREQ message of type 6 is created with the following fields:

- **type (8 bits (unsigned))**

  6

- **DTDODREQ_version (8 bits (unsigned))**

0

· **DTDODREQ_ID (32 bits (unsigned))**

A sequence number started from 0 and incremented by one each time a new DTDODREQ message is sent. When DTDODREQ_ID reaches the maximum value (2^32 - 1), the next time it has to be incremented, it will be set again to 0.

· **DTDODREQ_TTL (8 bits (unsigned))**

The Time to Live value used to define the limit of the DTDODREQ message along the network. It will be set initially to MAXALLOWED_HOPS by the originator of the DTDODREQ, and decremented by one by each node in each hop. If DTDODREQ_TTL reaches 0, then the DTDODREQ message will be discarded and so, not forwarded.

· **Size of the SourceRouting List (8 bits (unsigned))**

Number of entries of the SourceRoutingList

· **SourceRoutingList{node unique ID,level of battery of the node,**

number of explosive neighbors of the node,

timestamp of the node}

(Size of the SourceRoutinglist * (64 + 8_unsigned + 8_unsigned + 64) bits where the 64 bits of the timestamp are 32 bits of timeval.tv_sec plus 32 bits of timeval.tv_usec)

The sequence of nodes (unique IDs) for which the DTDODREQ message is travelling along, sorted by order of visit. Only the originator of the DTDODREQ message will not be included in this list, since it is already included in the last four fields of the DTDODREQ message. Moreover, for each node, the level of battery, the number of explosive neigbors, and a timestamp will be added to the list at the moment when the DTDODREQ message is forwarded. Each node forwarding the DTDODREQ message will be the responsible for adding this information (i.e. its own information), at each hop, at the end of the SourceRoutingList of the DTDODREQ being forwarded.

· **Reserved (32 bits)**

Unused

· **pluginID, pluginTag, originator, timestamp (32+32+64+64 bits)**

The fields identifying the DTDO we are asking

· **OriginatorID (64 bits)**

The unique ID of the node owning the DTDODM that has generated the DTDODREQ message

· **Level of battery of the OriginatorID node (8 bits (unsigned))**

Level of battery of the originator node at the moment of sending the DTDODREQ message

· **Number of explosive neighbors of the OriginatorID node (8 bits (unsigned))**

Number of explosive neighnors of the originator node at the moment of sending the DTDODREQ message

· **Timestamp of the OriginatorID node (64 bits: 32 bits of timeval.tv_sec plus 32 bits of timeval.tv_usec)**

Time instant when the DTDODREQ message is sent

DTDODREQ messages of type 6 will be forwarded according to MIFO to the destination node ID 0 (i.e., broadcast).

The operation when a node receives a DTDODREQ message of type 4 from the network is the following:

```
The node checks the SourceRoutingList and the OriginatorID of the
DTDODREQ message.

If (the SourceRoutingList or the OriginatorID contain its own unique
ID)

      The node discards the DTDODREQ message.

Else

      The node checks if it already has forwarded
      MAXDTDODREQ_TOFORWARD DTDODREQ messages with the same
      DTDODREQ_ID during the last DTDODREQID_LIFE milliseconds

      If (true)

            The node discards the DTDODREQ message

      Else

            /* For sure that the originator does not correspond to the
            ID of the node sending the DTDODREQ */
```

The node inserts/updates (if it is the case) to the DTDO Table the routing information related with the routing path to the originator node of the DTDODREQ message. All these routes will be added with Status=1. In other words, from the point of view of AGENET, the DTDODREQ message will be processed as if it was an AGROHello and an EXFOHello at the same time (see sections "3.2.1.1.4 Receiving AGROHellos" and "3.2.2.1.5 Receiving EXFOHellos").

If (originator != "my ID")

DTDODREQ_TTL--

If (DTDODREQ_TTL > 0)

The node forwards the DTDODREQ message updating the SourceRoutingList of the DTDODREQ by adding at the end of the list its own (unique ID,level of battery,number of explosive neighbors,timestamp)

Else

The node discards the DTDODREQ message

EndIf

Else

DELTOYA checks if we have the DTDO (pluginID,pluginTag,originator,timestamp) in the cache (DTDO_data!=NULL)

If (true)

generateDTDODREP(pluginID,pluginTag, originator,timestamp,ourTTL,3)

The node becomes Master of the RZ (see section 12).

EndIf

The node discards the DTDODREQ message

EndIf

EndIf

EndIf

F.3 DTDO Discovery Reply (DTDODREP)

DTDO Discovery Replies are unicast messages that will be answered in reply to the corresponding DTDO Discovery Requests.

DTDODREP messages can be generated in the following way:

generateDTDODREP(listDiscoveredDTDOs,type)

Params:

- `listDiscoveredDTDOs`

  The list of the discovered DTDOs (i.e., the pluginID, pluginTag, originator, timestamp and **remaining** dataTTL of each discovered DTDO)

- `type (uint8_t)`

  It indicates if the DTDODREP message is generated due to a DTDODREQ message triggered by a `queryDTDOs` call (`type`==0), due to a DTDODREQ message triggered by a `requestDTDOs` call (`type`==1) or due to a DTDODREQ message triggered by a `pushDTDO` call (`type`==2 or 3)

F.3.1 DTDODREP for queryDTDOs

When, due to a reception of a DTDODREQ message of type 0, a `generateDTDODREP(list,0)` is called, a DTDODREP message of type 1 is created and sent with the following fields:

· type (8 bits (unsigned))

1

· DTDODREP_version (8 bits (unsigned))

0

· DTDODREP_ID (32 bits (unsigned))

Unchanged. The same value than in the DTDODREQ message. It will be used in the Aggregator (together with the OriginatorID field) to identify which of the started DTDODMs need this DTDODREP.

· Reserved (32 bits)

Unused **(**for future use)

· Size of the PathToFollow List (8 bits (unsigned))

Number of entries of the PathToFollow list

· PathToFollow{node unique ID} (Size of the PathToFollow list * (64))

The SourceRoutingList of the DTDODREQ but in the reverse order. This list is the sequence of nodes (unique IDs) to follow in order to send the DTDODREP message to the originator node of the DTDODREQ.

· Pointer(PathToFollow) (8 bits (unsigned))

The pointer (i.e. order in the list) to know, at each hop, which node (nexthop) of the PathToFollow the node parsing the DTDODREP message has to use in order to forward the DTDODREP message. The initial value is the first node of the PathToFollow list, and at each hop, it will be updated by the node forwarding the DTDODREP (i.e., incremented by one in order to point the next node in the list)

· OriginatorID (64 bits)

Unchanged. The same value than in the DTDODREQ message. It will be used to identify the last hop of the DTDODREP message, since the originator node is not included in the PathToFollow list.

· Size of the DTDO List (16 bits (unsigned))

Number of entries of the DTDO list

· pluginID, DTDOlist{pluginTag,originator,timestamp,}

(32 + Size of the DTDO list*(32+64+64)) bits)

The set of specific DTDOs that the DTDODREQ message has found in the cache of the node generating the DTDODREP message

DTDODREP messages of type 1 will be forwarded according to MIFO to the first node of the PathToFollow list.

The operation when a node receives a DTDODREP message of type 1 from the network is the following:

```
If (OriginatorID of the DTDODREP != "my ID")

      The node forwards the DTDODREP message updating the PathToFollow
      field of the DTDODREP by updating the level of battery, the
      number of explosive neighbors, and the timestamp corresponding
      to itself, and by moving the Pointer(PathToFollow) to the next
      node in the PathToFollow list in order to reach the originator
      node.

Else

      The DTDODREP message is received by the Aggregator (if used) and
      send to all the DTDOQQ entries that need it (to save the DTDOs
      discovered in the CallbackInformation).

      If (recipient == NULL)

            /* the query was not for an specific recipient */

            We continue with the retransmission algorithm of the
            DTDODM of the DTDOQQ entry until the T_QUERYDTDOS period
            expires

      Else
```

```
            Since we have already received the reply from the
            recipient, we can stop the retransmission algorithm of the
            DTDODM and call the queryDTDOsCallback without waiting
            until the T_QUERYDTDOS period expires
        EndIf
EndIf
```

F.3.2 DTDODREP for requestDTDOs

When, due to a reception of a DTDODREQ message of type 2, a `generateDTDODREP(list,1)` is called, a DTDODREP message of type 3 is created and sent with the following fields:

· type (8 bits (unsigned))

3

· DTDODREP_version (8 bits (unsigned))

0

· DTDODREP_ID (32 bits (unsigned))

Unchanged. The same value than in the DTDODREQ message. It will be used in the Aggregator (together with the OriginatorID field) to identify which of the started DTDODMs need this DTDODREP.

· Reserved (32 bits)

Unused **(**for future use)

· Size of the PathToFollow List (8 bits (unsigned))

Number of entries of the PathToFollow list

· PathToFollow{node unique ID,level of battery of the node,

number of explosive neighbors of the node,

timestamp of the node}

(Size of the PathToFollow list * (64 + 8_unsigned + 8_unsigned + 64) bits where the 64 bits of the timestamp are 32 bits of timeval.tv_sec plus 32 bits of timeval.tv_usec)

The SourceRoutingList of the DTDODREQ but in the reverse order. This list is the sequence of nodes (unique IDs) to follow in order to send the DTDODREP message to the originator node of the DTDODREQ.

The only fields that will change along the path will be the level of battery, number of explosive neighbors and timestamp of each node. These fields will be updated at each hop by each node forwarding the DTDODREP.

· Pointer(PathToFollow) (8 bits (unsigned))

The pointer (i.e. order in the list) to know, at each hop, which node (nexthop) of the PathToFollow the node parsing the DTDODREP message has to use in order to forward the DTDODREP message. The initial value is the first node of the PathToFollow list, and at each hop, it will be updated by the node forwarding the DTDODREP (i.e., incremented by one in order to point the next node in the list)

· targetRecipient (64 bits)

The unique ID of the recipient replying the DTDODREQ message with this DTDODREP. It will be used in each node when receiving the DTDODREP message, in order to reconstruct the routing path, since this node is not included in the PathToFollow list.

· Level of battery of the targetRecipient (8 bits (unsigned))

Level of battery of the targetRecipient node at the moment when the DTDODREP is created.

· Number of explosive neighbors of the targetRecipient (8 bits (unsigned))

Number of explosive neighbors of the targetRecipient node at the moment when the DTDODREP is created

· Timestamp of the targetRecipient (64 bits: 32 bits of timeval.tv_sec plus 32 bits of timeval.tv_usec)

Timestamp of the targetRecipient node at the moment when the DTDODREP is created

· OriginatorID (64 bits)

Unchanged. The same value than in the DTDODREQ message. It will be used to identify the last hop of the DTDODREP message, since the originator node is not included in the PathToFollow list.

· Size of the DTDO List (16 bits (unsigned))

Number of entries of the DTDO list

· pluginID, DTDOlist{pluginTag,originator,timestamp,dataTTL,dataSize}

(32 + Size of the DTDO list*(32+64+64+32+32)) bits)

The set of specific DTDOs that the DTDODREQ message has found in the cache of the node generating the DTDODREP message, together with the remaining dataTTL value for the DTDO_data field once this data is obtained and cached, and the dataSize of the data that this DTDO contains

DTDODREP messages of type 3 will be forwarded according to MIFO to the first node of the PathToFollow list.

The operation when a node receives a DTDODREP message of type 3 from the network is the following:

The node inserts/updates (if it is the case) to the DTDO Table the routing information related with the routing path to the targetRecipient node of the DTDODREP message. All these routes will be added with Status=1. In other words, from the point of view of AGENET, the DTDODREP message will be processed as if it was an AGROHello.

The node becomes Master of the RZ (see section 12).

If (OriginatorID of the DTDODREP != "my ID")

The node forwards the DTDODREP message updating the PathToFollow field of the DTDODREP by updating the level of battery, the number of explosive neighbors, and the timestamp corresponding to itself, and by moving the Pointer(PathToFollow) to the next node in the PathToFollow list in order to reach the originator node.

Else

An entry in the DTDO Table (sub table of remote DTDOs) for each DTDO of the DTDODREP message will be created (or updated if it is already in the DTDO Table), using the information contained in the DTDODREP message, together with a link (with the TTL field) to the targetRecipient node inserted in the DTDO Table by AGENET. These new DTDOs will be set as public (DTDO_isPublic==true).

The DTDODREP message is received by the Aggregator (if used) and send to all the DTDORQ entries (DTDODMs) that need it:

From the point of view of each DTDODM, for each DTDO of the DTDODREP message, the DTDO_useCounter field will be incremented by one the first time the DTDO has been discovered by this DTDODM (i.e., if it is the second time the DTDODM discovered the same DTDO, then the DTDODM does not have to increment the DTDO_useCounter of this DTDO), since until PTP will give an answer to the DTDORQ entry, or the MRT of the DTDORQ entry will

expire, the DTDORQ entry wants to block the DTDO entry in order to avoid a possible deletion while the DTDORQ entry is using the DTDO entry.

And finally, for each DTDO discovered in the DTDODREP messages that we have received, the DTDORQ entry (DTDODM) will call PTP in order to obtain the data of these recently discovered DTDOs (see section "2.3 Patient Transport Protocol") in order to fill the callbackInformation of the DTDORQ entry:

[PTP] getDTDO(pluginID,pluginTag,originator,timestamp, pointer(DTDORQ entry),QoS)

EndIf

F.3.3 DTDODREP due to pushDTDO

F.3.3.1 For Public DTDOs

When, due to a reception of a DTDODREQ message of type 4, a `generateDTDODREP(DTDO,2)` is called, a DTDODREP message of type 5 is created and sent with the following fields:

- **type (8 bits (unsigned))**

  5

- **DTDODREP_version (8 bits (unsigned))**

  0

- **DTDODREP_ID (32 bits (unsigned))**

  Unchanged. The same value than in the DTDODREQ message. It will be used in the Aggregator (together with the OriginatorID field) to identify which of the started DTDODMs need this DTDODREP.

- **Reserved (32 bits)**

  Unused **(**for future use)

- **Size of the PathToFollow List (8 bits (unsigned))**

  Number of entries of the PathToFollow list

- **PathToFollow{node unique IDs,level of battery of the node,**

  number of explosive neighbors of the node,

  timestamp of the node}

(Size of the PathToFollow list * (64 + 8_unsigned + 8_unsigned + 64) bits where the 64 bits of the timestamp are 32 bits of timeval.tv_sec plus 32 bits of timeval.tv_usec)

The SourceRoutingList of the DTDODREQ but in the reverse order. This list is the sequence of nodes (unique IDs) to follow in order to send the DTDODREP message to the originator node of the DTDODREQ.

The only fields that will change along the path will be the level of battery, number of explosive neighbors and timestamp of each node. These fields will be updated at each hop by each node forwarding the DTDODREP.

· Pointer(PathToFollow) (8 bits (unsigned))

The pointer (i.e. order in the list) to know, at each hop, which node (nexthop) of the PathToFollow the node parsing the DTDODREP message has to use in order to forward the DTDODREP message. The initial value is the first node of the PathToFollow list, and at each hop, it will be updated by the node forwarding the DTDODREP (i.e., incremented by one in order to point the next node in the list)

· targetRecipient (64 bits)

The unique ID of the recipient replying the DTDODREQ message with this DTDODREP. It will be used in each node when receiving the DTDODREP message, in order to reconstruct the routing path, since this node is not included in the PathToFollow list.

· Level of battery of the targetRecipient (8 bits (unsigned))

Level of battery of the targetRecipient node at the moment when the DTDODREP is created

· Number of explosive neighbors of the targetRecipient (8 bits (unsigned))

Number of explosive neighbors of the targetRecipient node at the moment when the DTDODREP is created

· Timestamp of the targetRecipient (64 bits: 32 bits of timeval.tv_sec plus 32 bits of timeval.tv_usec)

Timestamp of the targetRecipient node at the moment when the DTDODREP is created

· OriginatorID (64 bits)

Unchanged. The same value than in the DTDOREQ message. It will be used to identify the last hop of the DTDODREP message, since the originator node is not included in the PathToFollow list.

· **pluginID, pluginTag, originator, timestamp, dataTTL, dataSize**

(32+32+64+64+32+32 bits)

The DTDO that the DTDOREQ message has found in the cache of the node generating the DTDODREP message, together with the remaining dataTTL value for the DTDO_data field once this data is obtained and cached, and the dataSize of the data that this DTDO contains

DTDODREP messages of type 5 will be forwarded according to MIFO to the first node of the PathToFollow list.

The operation when a node receives a DTDODREP message of type 5 from the network is the following:

The node inserts/updates (if it is the case) to the DTDO Table the routing information related with the routing path to the targetRecipient node of the DTDODREP message. All these routes will be added with Status=1. In other words, from the point of view of AGENET, the DTDODREP message will be processed as if it was an AGROHello.

The node becomes Master of the RZ (see section 12).

If (OriginatorID of the DTDODREP != "my ID")

The node forwards the DTDODREP message updating the PathToFollow field of the DTDODREP by updating the level of battery, the number of explosive neighbors, and the timestamp corresponding to itself, and by moving the Pointer(PathToFollow) to the next node in the PathToFollow list in order to reach the originator node.

Else

An entry in the DTDO Table (sub table of remote DTDOs) for the DTDO of the DTDODREP message will be created (or updated if it is already in the DTDO Table), using the information contained in the DTDODREP message, together with a link (with the TTL field) to the targetRecipient node inserted in the DTDO Table by AGENET. These new DTDO will be set as public (DTDO_isPublic==true).

The DTDODREP message is received by the Aggregator (if used) and sent to all the DTDOAQ entries (DTDODMs) that need it:

From the point of view of each DTDODM, for the DTDO of the DTDODREP message, the DTDO_useCounter field will be incremented by one the first time the DTDO has been discovered by this DTDODM (i.e., if it is the second time the DTDODM discovered the same DTDO, then the DTDODM does not have to increment the DTDO_useCounter of this DTDO), since until PTP will give an

answer to the DTDOAQ entry, or the MRT of the DTDOAQ entry will expire, the DTDOAQ entry wants to block the DTDO entry in order to avoid a possible deletion while the DTDOAQ entry is using the DTDO entry.

And finally, for the DTDO discovered in the DTDODREP message that we have received, the DTDOAQ entry (DTDODM) will call PTP in order to obtain the data of this recently discovered DTDO (see section "2.3 Patient Transport Protocol") in order to fill the returnInformation of the DTDOAQ entry:

[PTP] getDTDO(pluginID,pluginTag,originator,timestamp,

pointer(DTDOAQ entry),QoS)

EndIf

F.3.3.2 For Private DTDOs

When, due to a reception of a DTDODREQ message of type 6, a `generateDTDODREP(DTDO,3)` is called, a DTDODREP message of type 7 is created and sent with the following fields:

- **type (8 bits (unsigned))**

  7

- **DTDODREP_version (8 bits (unsigned))**

  0

- **DTDODREP_ID (32 bits (unsigned))**

  Unchanged. The same value than in the DTDODREQ message. It will be used in the Aggregator (together with the OriginatorID field) to identify which of the started DTDODMs need this DTDODREP.

- **Reserved (32 bits)**

  Unused **(**for future use)

- **Size of the PathToFollow List (8 bits (unsigned))**

  Number of entries of the PathToFollow list

- **PathToFollow{node unique ID,level of battery of the node,**

  number of explosive neighbors of the node,

  timestamp of the node}

(Size of the PathToFollow list * (64 + 8_unsigned + 8_unsigned + 64) bits where the 64 bits of the timestamp are 32 bits of timeval.tv_sec plus 32 bits of timeval.tv_usec)

The SourceRoutingList of the DTDODREQ but in the reverse order. This list is the sequence of nodes (unique IDs) to follow in order to send the DTDODREP message to the originator node of the DTDODREQ.

The only fields that will change along the path will be the level of battery, number of explosive neighbors and timestamp of each node. These fields will be updated at each hop by each node forwarding the DTDODREP.

· Pointer(PathToFollow) (8 bits (unsigned))

The pointer (i.e. order in the list) to know, at each hop, which node (nexthop) of the PathToFollow the node parsing the DTDODREP message has to use in order to forward the DTDODREP message. The initial value is the first node of the PathToFollow list, and at each hop, it will be updated by the node forwarding the DTDODREP (i.e., incremented by one in order to point the next node in the list)

· targetRecipient (64 bits)

The unique ID of the recipient replying the DTDODREQ message with this DTDODREP. It will be used in each node when receiving the DTDODREP message, in order to reconstruct the routing path, since this node is not included in the PathToFollow list.

· Level of battery of the targetRecipient (8 bits (unsigned))

Level of battery of the targetRecipient node at the moment when the DTDODREP is created

· Number of explosive neighbors of the targetRecipient (8 bits (unsigned))

Number of explosive neighbors of the targetRecipient node at the moment when the DTDODREP is created

· Timestamp of the targetRecipient (64 bits: 32 bits of timeval.tv_sec plus 32 bits of timeval.tv_usec)

Timestamp of the targetRecipient node at the moment when the DTDODREP is created

· OriginatorID (64 bits)

Unchanged. The same value than in the DTDODREQ message. It will be used to identify the last hop of the DTDODREP message, since the originator node is not included in the PathToFollow list.

· **pluginID, pluginTag, originator, timestamp, dataTTL, dataSize**

(32+32+64+64+32+32 bits)

The DTDO that the DTDODREQ message has found in the cache of the node generating the DTDODREP message, together with the remaining dataTTL value for the DTDO_data field once this data is obtained and cached, and the dataSize of the data that this DTDO contains

DTDODREP messages of type 7 will be forwarded according to MIFO to the first node of the PathToFollow list.

The operation when a node receives a DTDODREP message of type 7 from the network is the following:

The node inserts/updates (if it is the case) to the DTDO Table the routing information related with the routing path to the targetRecipient node of the DTDODREP message. All these routes will be added with Status=1. In other words, from the point of view of AGENET, the DTDODREP message will be processed as if it was an AGROHello.

The node becomes Master of the RZ (see section 12).

If (OriginatorID of the DTDODREP != "my ID")

The node forwards the DTDODREP message updating the PathToFollow field of the DTDODREP by updating the level of battery, the number of explosive neighbors, and the timestamp corresponding to itself, and by moving the Pointer(PathToFollow) to the next node in the PathToFollow list in order to reach the originator node.

Else

An entry in the DTDO Table (sub table of remote DTDOs) for the DTDO of the DTDODREP message will be created (or updated if it is already in the DTDO Table), using the information contained in the DTDODREP message, together with a link (with the TTL field) to the targetRecipient node inserted in the DTDO Table by AGENET. These new DTDO will be set as private (DTDO_isPublic==false) (only in the case that the DTDO already exists, then the DTDO_isPublic field will not be modified).

The DTDODREP message is received by the Aggregator (if used) and sent to all the DTDOAQ entries (DTDODMs) that need it:

From the point of view of each DTDODM, for the DTDO of the DTDODREP message, the DTDO_useCounter field will be incremented by one the first time the DTDO has been discovered by this DTDODM (i.e., if it is the second time the DTDODM discovered the same DTDO, then the DTDODM does not have to increment the DTDO_useCounter of this DTDO), since until PTP will give an answer to the DTDOAQ entry, or the MRT of the DTDOAQ entry will

expire, the DTDOAQ entry wants to block the DTDO entry in order to avoid a possible deletion while the DTDOAQ entry is using the DTDO entry.

And finally, for the DTDO discovered in the DTDODREP message that we have received, the DTDOAQ entry (DTDODM) will call PTP in order to obtain the data of this recently discovered DTDO (see section "2.3 Patient Transport Protocol") in order to fill the returnInformation of the DTDOAQ entry:

[PTP] getDTDO(pluginID,pluginTag,originator,timestamp, pointer(DTDOAQ entry),QoS)

EndIf

Appendix G. DOCKDM Details

The DOCK Discovery Mechanism (DTDODM) is the mechanism in charge of discovering a set of specified DOCKs in the network. The different types of DOCK Discovery Mechanisms (DOCKDMs) are the following:

a) `startDOCKDM(pluginID,pluginTag,originator,T_QUERYDOCKS)`

b) `startDOCKDM(pluginID,*,originator,T_QUERYDOCKS)`

c) `startDOCKDM(pluginID,pluginTag,*,T_QUERYDOCKS)`

d) `startDOCKDM(pluginID,*,*,T_QUERYDOCKS)`

And in general:

```
startDOCKDM(pluginID,pluginTag,originator,T_QUERYDOCKS);
```

Where the params have the same meaning than in `queryDOCKs` (see appendix "C.9.3 DOCK Querying").

`startDOCKDM` will start a flooding using DOCKDREQ and DOCKDREP messages in order to discover DOCKs in the network for a period of `T_QUERYDOCKS` milliseconds.

G.1 Flooding (The Aggregator)

For each DOCKDM started by any entry of the DOCKQQ, DELTOYA will keep sending DOCKDREQ messages following a Binary Exponential Backoff for each DOCKDREQ, and only will stop the DOCKDM when the T_QUERYDOCKS period of time of the DOCKDM expires.

However, this is from the point of view of the DOCKDMs initiated by each DOCKQQ entry. From the point of view of the network, the Aggregator will try to aggregate all the DOCKDREQ and DOCKDREP of the DOCKDMs as much as possible, in order to reduce the number of transmissions in the network. See Figure 99.

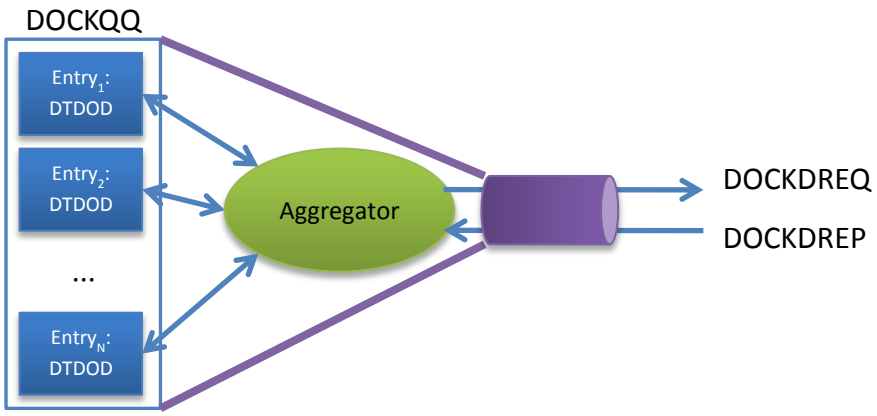

Figure 99. The DOCKQQ Aggregator

For this reason, we have to define the concept of inclusion "⊆" between DOCKDMs. For example, a DOCKDM for the DOCKs = (pluginIDX,pluginTag=*,origY) already includes a

DOCKDM for the DOCKs = (pluginIDX,pluginTagZ,origY,). In this way, two different entries will be added to the DOCKQQ, but the DOCKDM of the second one will be able to take advantage of the DOCKDREQ messages of the DOCKDM of the first one (e.g. when a DOCKDM tells the Aggregator to send a DOCKDREQ message for some DOCKs, and the Aggregator knows that just less or equal than TIMEBETWEEN_INCLUDEDDOCKDREQs milliseconds before it sent a DOCKDREQ message for at least the same DOCKs, then it can decide to not send the second DOCKDREQ, and use the possible DOCKDREP of the first DOCKDREQ as reply for also the second DOCKDREQ).

In the same way, DOCKDMs included in another DOCKDM will be able to take advantage also of the DOCKDREP messages of the first one. In this case, when a DOCKDREP will be received, the Aggregator will have to check which of all the DOCKQQ entries will need the DOCKDREP.

G.2 DOCK Discovery Request (DOCKDREQ)

DOCK Discovery Requests are broadcast messages that will be flooded in the MANET in order to discover which DOCKs and in which nodes are available at that time in the network.

Each time a DOCKDM wants to send a DOCKDREQ, a DOCKDREQ message is created with the following fields:

- **type (8 bits (unsigned))**

  0

- **DOCKDREQ_version (8 bits (unsigned))**

  0

- **DOCKDREQ_ID (32 bits (unsigned))**

  A sequence number started from 0 and incremented by one each time a new DOCKDREQ message is sent. When DOCKDREQ_ID reaches the maximum value (2^32 - 1), the next time it has to be incremented, it will be set again to 0.

- **DOCKDREQ_TTL (8 bits (unsigned))**

  The Time to Live value used to define the limit of the DOCKDREQ message along the network. It will be set initially to MAXALLOWED_HOPS by the originator of the DOCKDREQ, and decremented by one by each node in each hop. If DOCKDREQ_TTL reaches 0, then the DOCKDREQ message will be discarded and so, not forwarded.

- **Size of the SourceRouting List (8 bits (unsigned))**

  Number of entries of the SourceRoutingList

· **SourceRoutingList{node unique ID} (Size of the SourceRoutinglist * (64))**

The sequence of nodes (unique IDs) for which the DOCKDREQ message is travelling along, sorted by order of visit. Only the originator of the DOCKDREQ message will not be included in this list, since it is already included in the last field of the DOCKDREQ message. Each node forwarding the DOCKDREQ message will be the responsible for adding this information (i.e. its own information), at each hop, at the end of the SourceRoutingList of the DOCKDREQ being forwarded.

· **Reserved (32 bits)**

Unused

· **pluginID, pluginTag, originator (32+32+64 bits)**

The fields identifying the set of DOCKs we are querying

· **OriginatorID (64 bits)**

The unique ID of the node owning the DOCKDM that has generated the DOCKDREQ message

DOCKDREQ messages will be forwarded according to section MIFO to the destination node ID 0 (i.e., broadcast).

The operation when a node receives a DOCKDREQ message from the network is the following:

```
The node checks the SourceRoutingList and the OriginatorID of the
DOCKDREQ message.

If (the SourceRoutingList or the OriginatorID contain its own unique
ID)

      The node discards the DOCKDREQ message.

Else

      The node checks if it already has forwarded
      MAXDOCKDREQ_TOFORWARD DOCKDREQ messages with the same
      DOCKDREQ_ID during the last DOCKDREQID_LIFE milliseconds

      If (true)

            The node discards the DOCKDREQ message

      Else

            If (originator == NULL)

                  If (pluginTag!=NULL)

                        DELTOYA checks if we have a DOCK with
                        DOCK_pluginID==pluginID                  and
```

```
            DOCK_pluginTag==pluginTag    in    the    DOCK
            Table/Cache

            If (true)

                  generateDOCKDREP(the discovered DOCK)

            EndIf

      ElseIf (pluginTag==NULL)

            DELTOYA checks if we have some DOCKs with
            DOCK_pluginID==pluginID    in    the    DOCK
            Table/Cache

            If (true)

                  generateDOCKDREP(the discovered DOCKs)

            EndIf

      EndIf

      DOCKDREQ_TTL--

      If (DOCKDREQ_TTL > 0)

            The node forwards the DOCKDREQ message
            updating the SourceRoutingList of the DOCKDREQ
            by adding at the end of the list its own
            unique ID

      Else

            The node discards the DOCKDREQ message

      EndIf

ElseIf (originator != NULL)

      If (originator != "my ID")

            DOCKDREQ_TTL--

            If (DOCKDREQ_TTL > 0)

                  The node forwards the DOCKDREQ message
                  updating the SourceRoutingList of the
                  DOCKDREQ by adding at the end of the
                  list its own unique ID

            Else

                  The node discards the DOCKDREQ message

            EndIf

      Else

            If (pluginTag!=NULL)

                  DELTOYA checks if we have a DOCK with
                  DOCK_pluginID==pluginID              and
```

```
                    DOCK_pluginTag==pluginTag in the DOCK
                    Table/Cache

                    If (true)

                        generateDOCKDREP(the discovered
                        DOCK)

                    EndIf

                ElseIf (pluginTag==NULL)

                    DELTOYA checks if we have some DOCKs
                    with DOCK_pluginID==pluginID in the DOCK
                    Table/Cache

                    If (true)

                        generateDOCKDREP(the discovered
                        DOCKs)

                    EndIf

                EndIf

                The node discards the DTDODREQ message

            EndIf

        EndIf

    EndIf

EndIf
```

G.3 DOCK Discovery Reply (DOCKDREP)

DOCK Discovery Replies are unicast messages that will be answered in reply to the corresponding DOCK Discovery Requests.

As we have seen in the previous section, DOCKDREP messages can be generated in the following way:

```
generateDOCKDREP(listDiscoveredDOCKs)
```

Params (as defined in section 5.2):

- `listDiscoveredDOCKs`

  The list of the discovered DOCKs (i.e., the pluginID, pluginTag, originator, and timestamp of each discovered DOCK)

When, due to a reception of a DOCKDREQ message, a `generateDOCKDREP(list)` is called, a DOCKDREP message is created and sent with the following fields:

· type (8 bits (unsigned))

1

· **DOCKDREP_version (8 bits (unsigned))**

0

· **DOCKDREP_ID (32 bits (unsigned))**

Unchanged. The same value than in the DOCKDREQ message. It will be used in the Aggregator (together with the OriginatorID field) to identify which of the started DOCKDMs need this DOCKDREP.

· **Reserved (32 bits)**

Unused **(**for future use)

· **Size of the PathToFollow List (8 bits (unsigned))**

Number of entries of the PathToFollow list

· **PathToFollow{node unique ID}**

(Size of the PathToFollow list * (64))

The SourceRoutingList of the DOCKDREQ but in the reverse order. This list is the sequence of nodes (unique IDs) to follow in order to send the DOCKDREP message to the originator node of the DOCKDREQ.

· **Pointer(PathToFollow) (8 bits (unsigned))**

The pointer (i.e. order in the list) to know, at each hop, which node (nexthop) of the PathToFollow the node parsing the DOCKDREP message has to use in order to forward the DOCKDREP message. The initial value is the first node of the PathToFollow list, and at each hop, it will be updated by the node forwarding the DOCKDREP (i.e., incremented by one in order to point the next node in the list)

· **OriginatorID (64 bits)**

Unchanged. The same value than in the DOCKDREQ message. It will be used to identify the last hop of the DOCKDREP message, since the originator node is not included in the PathToFollow list.

· **Size of the DTDO List (16 bits (unsigned))**

Number of entries of the DTDO list

· **pluginID, DTDOlist{pluginTag,timestamp}, originator**

(32 + Size of the DTDO list*(32+64) + 64) bits)

The set of specific DOCKs that the DOCKDREQ message has found in the DOCK Table/Cache of the node generating the DOCKDREP message (i.e., the node of the originator field)

DOCKDREP messages will be forwarded according to section MIFO to the first node of the PathToFollow list.

The operation when a node receives a DOCKDREP message from the network is the following:

```
If (OriginatorID of the DOCKDREP != "my ID")

      The node forwards the DOCKDREP message updating the PathToFollow
      field of the DOCKDREP by moving the Pointer(PathToFollow) to the
      next node in the PathToFollow list in order to reach the
      originator node.

Else

      The DOCKDREP message is received by the Aggregator (if used) and
      sent to all the DOCKQQ entries that need it (to save the DOCKs
      discovered in the CallbackInformation).

      If (originator == NULL)

            /* the query was not for an specific originator */

            We continue with the retransmission algorithm of the
            DOCKDM of the DOCKQQ entry until the T_QUERYDOCKS period
            expires

      Else

            Since we have already received the reply from the
            originator, we can stop the retransmission algorithm of
            the DOCKDM and call the queryDOCKsCallback without waiting
            until the T_QUERYDOCKS period expires

      EndIf

EndIf
```

Appendix H. RDM Details

The RDM is the mechanism in charge of discovering a specific node (identified by its unique ID) in the network.

H.1 RDM Queue (RDMQ)

Since several RDMs for one node can be called at the same time while the node is being discovered, a RDM Queue (RDMQ) is needed to avoid duplicated actions. The RDMQ is the data structure used to buffer all the `startRDM` calls done in a node.

Each entry of the RDMQ identifies a unique node being discovered, and each entry has the following fields:

- **node2discover (uint64_t)**

  The unique ID of the node to discover

- **list{observer}**

  The list of observers still waiting for the discovery of this node (i.e., the list of observers that have called a `startRDM` call for this node and are still interested in the response of the RDM). Only when this list becomes empty (there is not any observer interested in discovering this node), the process of discovering the node can be stopped and the RDMQ entry can be deleted

The key field of the RDMQ is node2discover field, meaning that there cannot be two entries with the same node2discover field.

When an already existent entry is added, only the list{observer} field of the entry will be updated adding the new observer to the list. Moreover, for each entry, when the node has been discovered or the list{observer} field becomes empty, the entry will be deleted.

H.2 startRDM

Each time a request for discovering a new node is called, the following function is executed:

```
startRDM(IDToDiscover,observer)
```

Params:

- `IDToDiscover (uint64_t)`

  ID of the node to discover

- `observer`

  The observer calling the `startRDM`

startRDM will :

Check if an entry in the RDMQ already exists.

If (it exists)

The `observer` will be added to the list{observer} of the RDMQ entry and then, we will wait for a possible answer due to the already started flooding mechanism.

Else

We will create a new entry in the RDMQ with node2discover = `IDToDiscover` and with list{observer} = `{observer}`.

Then, a flooding mechanism using RDREQ and RDREP messages in order to discover the specific node will be started:

The flooding mechanism will follow a Binary Exponential Backoff until a maximum of MAX_RDREQS expirations have occurred. After having occurred MAX_RDREQS expirations, then the Binary Exponential Backoff will automatically be restarted as if a new Binary Exponential Backoff was started (i.e. $RDREQ_{MAX\_RDREQS} == RDREQ_1$). For each expiration, the flooding mechanism will check if there is a routing path to `IDToDiscover` in the DTDO Table. If there is routing path, the flooding mechanism will do nothing. If there is not routing path, then the flooding mechanism will send a RDREQ message. See Figure 100.

EndIf

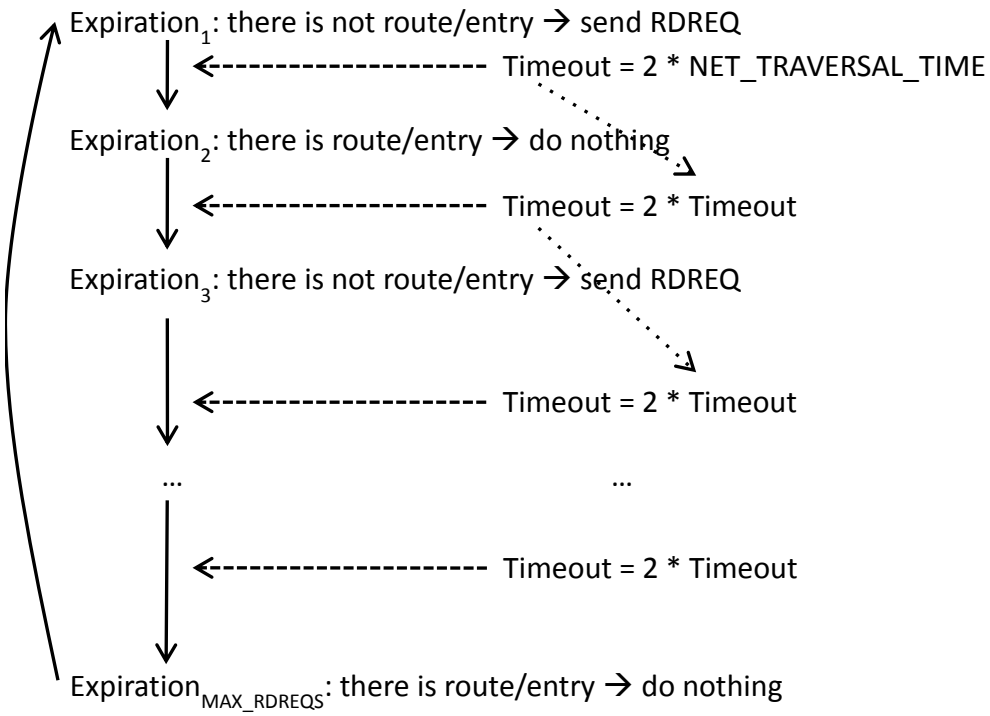

Figure 100. RDM Backoff

Only when the list{observer} of the corresponding RDMQ becomes empty, then the backoff mechanism will be stopped and the RDMQ entry will be deleted.

H.2.1 Route Discovery Request (RDREQ)

Route Discovery Requests are broadcast messages that will be flooded in the MANET in order to discover routing path between source and destination nodes.

Each time a RDM wants to send a RDREQ message, it will be created with the following fields:

· **type (8 bits (unsigned))**

0

· **RDREQ_version (8 bits (unsigned))**

0

· **RDREQ_ID (32 bits (unsigned))**

A sequence number started from 0 and incremented by one each time a new RDREQ message is sent. When RDREQ_ID reaches the maximum value (2^32 - 1), the next time it has to be incremented, it will be set again to 0.

· **RDREQ_TTL (8 bits (unsigned))**

The Time to Live value used to define the limit of the RDREQ message along the network. It will be set initially to MAXALLOWED_HOPS by the originator of the RDREQ, and decremented by one by each node in each hop. If RDREQ_TTL reaches 0, then the RDREQ message will be discarded and so, not forwarded.

· **Size of the SourceRouting List (8 bits (unsigned))**

Number of entries of the SourceRoutingList

· **SourceRoutingList{node unique ID,level of battery of the node,**

number of explosive neighbors of the node,

timestamp of the node}

(Size of the SourceRoutinglist * (64 + 8_unsigned + 8_unsigned + 64) bits where the 64 bits of the timestamp are 32 bits of timeval.tv_sec plus 32 bits of timeval.tv_usec)

The sequence of nodes (unique IDs) for which the RDREQ message is travelling along, sorted by order of visit. Only the originator of the RDREQ message will not be included in this list, since it is already included in the last four fields of the RDREQ message. Moreover, for each node, the level of battery, the number of explosive neigbors, and a timestamp will be added to the list at the moment when the RDREQ message is forwarded. Each node forwarding the RDREQ message will be the responsible for adding this information (i.e. its own information), at each hop, at the end of the SourceRoutingList of the RDREQ being forwarded.

· **Reserved (32 bits)**

Unused

· **IDToDiscover (64 bits)**

The unique ID of the node to discover

· **OriginatorID (64 bits)**

The unique ID of the node that has generated the RDREQ message

· **Level of battery of the OriginatorID node (8 bits (unsigned))**

Level of battery of the originator node at the moment of sending the RDREQ message

· **Number of explosive neighbors of the OriginatorID node (8 bits (unsigned))**

Number of explosive neighnors of the originator node at the moment of sending the RDREQ message

· **Timestamp of the OriginatorID node (64 bits: 32 bits of timeval.tv_sec plus 32 bits of timeval.tv_usec)**

Time instant when the RDREQ message is sent

RDREQ messages will be forwarded according to section MIFO to the destination node ID 0 (i.e., broadcast).

The operation when a node receives a RDREQ message from the network is the following:

```
The node checks the SourceRoutingList and the OriginatorID of the
RDREQ message.

If (the SourceRoutingList or the OriginatorID contain its own unique
ID)
```

```
        The node discards the RDREQ message.
Else
        The node checks if it already has forwarded MAXRDREQ_TOFORWARD
        RDREQ messages with the same RDREQ_ID during the last
        RDREQID_LIFE milliseconds
        If (true)
                The node discards the RDREQ message
        Else
                The node inserts/updates (if it is the case) to the DTDO
                Table the routing information related with the routing path
                to the originator node of the RDREQ message. All these
                routes will be added with Status=1. In other words, from
                the point of view of AGENET, the RDREQ message will be
                processed as if it was an AGROHello.
                If (RDREQ.IDToDiscover == "my ID")
                        generateRDREP()
                        The node becomes Master of the RZ (see section 12)
                Else
                        RDREQ_TTL--
                        If (RDREQ_TTL > 0)
                                The node forwards the RDREQ message updating
                                the SourceRoutingList of the RDREQ by adding
                                at the end of the list its own (unique
                                ID,level of battery,number of explosive
                                neighbors,timestamp)
                        Else
                                The node discards the RDREQ message
                        EndIf
                EndIf
        EndIf
EndIf
```

H.2.2 Route Discovery Reply (RDREP)

Route Discovery Replies are unicast messages that will be answered in reply to the corresponding Route Discovery Requests.

As we have seen in the previous section, RDREP messages can be generated in the following way:

```
        generateRDREP()
```

When, due to a reception of a RDREQ message, a `generateRDREP()` is called, a RDREP message is created and sent with the following fields:

· **type (8 bits (unsigned))**

1

· **RDREP_version (8 bits (unsigned))**

0

· **Reserved (32 bits)**

Unused **(**for future use)

· **Size of the PathToFollow List (8 bits (unsigned))**

Number of entries of the PathToFollow list

· **PathToFollow{node unique ID,level of battery of the node,**

number of explosive neighbors of the node,

timestamp of the node}

(Size of the PathToFollow list * (64 + 8_unsigned + 8_unsigned + 64) bits where the 64 bits of the timestamp are 32 bits of timeval.tv_sec plus 32 bits of timeval.tv_usec)

The SourceRoutingList of the RDREQ but in the reverse order. This list is the sequence of nodes (unique IDs) to follow in order to send the RDREP message to the originator node of the RDREQ.

The only fields that will change along the path will be the level of battery, number of explosive neighbors and timestamp of each node. These fields will be updated at each hop by each node forwarding the RDREP.

· **Pointer(PathToFollow) (8 bits (unsigned))**

The pointer (i.e. order in the list) to know, at each hop, which node (nexthop) of the PathToFollow the node parsing the RDREP message has to use in order to forward the RDREP message. The initial value is the first node of the PathToFollow list, and at each hop, it will be updated by the node forwarding the RDREP (i.e., incremented by one in order to point the next node in the list)

· **IDToDiscover (64 bits)**

The unique ID of the node replying the RDREQ message with this RDREP.

· **Level of battery of the node IDToDiscover (8 bits (unsigned))**

Level of battery of the node IDToDiscover at the moment when the RDREP is created

· **Number of explosive neighbors of the node IDToDiscover (8 bits (unsigned))**

Number of explosive neighbors of the node IDToDiscover at the moment when the RDREP is created

· **Timestamp of the node IDToDiscover (64 bits: 32 bits of timeval.tv_sec plus 32 bits of timeval.tv_usec)**

Timestamp of the node IDToDiscover at the moment when the RDREP is created

· **OriginatorID (64 bits)**

Unchanged. The same value than in the RDREQ message. It will be used to identify the last hop of the RDREP message, since the originator node is not included in the PathToFollow list.

RDREP messages will be forwarded according to section MIFO to the first node of the PathToFollow list.

The operation when a node receives a RDREP message from the network is the following:

```
The node inserts/updates (if it is the case) to the DTDO Table the
routing information related with the routing path to the IDToDiscover
node of the RDREP message. All these routes will be added with
Status=1. In other words, from the point of view of AGENET, the RDREP
message will be processed as if it was an AGROHello

The node becomes Master of the RZ (see section 12).

If (OriginatorID of the RDREP != "my ID")

      The node forwards the RDREP message updating the PathToFollow
      field of the RDREP by updating the level of battery, the number
      of explosive neighbors, and the timestamp corresponding to
      itself, and by moving the Pointer(PathToFollow) to the next node
      in the PathToFollow list in order to reach the originator node.

Else

      The RDREP message is received by the corresponding RDMQ entry.

EndIf
```

H.3 stopRDM

RDM provides a method to cancel currently running `startRDM` calls of specific observers:

```
stopRDM(IDToDiscover,observer)
```

Params:

- `IDToDiscover (uint64_t)`

  Unique ID of the node of the RDM to cancel

- `observer`

  The observer calling the `stopRDM`

stopRDM will:

Check if an entry in the RDMQ with node2discover==IDToDiscover exists.

- If it exists, then we will delete the `observer` from the list{observer} of the RDMQ entry
  - If the list becomes empty, then we will cancel its backoff mechanism and we will delete the RDMQ entry.
  - If the list does not become empty, we will not do anything.

If the RDMQ entry does not exist, we will not do anything.

Appendix I. AGENET Details

I.1 Routing Information: AGgressive ROuting Hellos (AGROHellos)

AGROHello messages are used to distribute routing information through the network. This distribution is performed by means of flooding, and it will be used to update the DTDO Table.

Nodes of a Routing Zone use periodic flooding of information, meaning that every time period of T_AGROHELLO milliseconds each node in the RZ will send an AGROHello message with its own information and with the information it has learnt from other nodes of the network. Since AGROHello messages could become large messages, the time period T_AGROHELLO has to be enough small to avoid work with obsolete information but enough large to avoid a lot of overhead and scalability problems. The information carried out by the AGROHellos messages will be used to update the DTDO Table.

I.2 Scalability: Routing Zone (RZ)

The Routing Zone (RZ) is a set of nodes of a MANET that are involved in the process of routing (for instance, sending routing information, forwarding data packets, etc). Outside the RZ, the nodes do not neither send any information related to the routing mechanism nor any data packet. In this way, scalability is achieved by limiting the geographical areas in which routing information is sent and routing information will be sent only when it is needed and where it is needed.

From the implementation point of view, AGENET has a flag indicating if the node is member of the RZ or not. And depending on if it is member or not, AGENET will proceed in one way or in another.

- **inTheRZ (uint8_t)**

  By default, set to 0 (not a member of the RZ)

I.2.1 Starting the RZ

When a source node wants to establish a communication path with a destination node, a DTDODM (see section 9) or a RDM (see section 11) has to be initialized.

The DTDODM of a `requestDTDOs` call and the RDM, as explained in appendix "F.3.2 DTDODREP for requestDTDOs" or in appendix "H.2.2 Route Discovery Reply (RDREP)" respectively, will be used to start the RZ.

All the nodes that forward (i.e., transmit, receive or transmit and receive) a DTDODREP of type 3 or a RDREP message automatically become members of the RZ. And from the point of view of the RZ, it is started when at least the RZ has one member.

I.2. 2 Maintaining the RZ

Every time a node transmits or receives a unicast packet (a PTP message, a DTDODREP message of type 3 or a RDREP message), it automatically becomes member of the RZ. This type of members of the RZ are called Masters (inTheRZ == 1).

Masters will start sending AGROHello messages that will contain the RZ information needed to tell other nodes in their vicinity to also become members of the RZ. A hop count value will be used inside the AGROHello messages to tell nodes that are at distance RZ_HOPS hops from the masters to become members of the RZ. Nodes at distance more than RZ_HOPS hops from the masters cannot become members of the RZ, even when they receive an AGROHello message. This new type of members of the RZ are called Cooperators (inTheRZ == 2).

Masters are the only members of the RZ that designate other nodes to become members of the RZ (and, in a first stage, as cooperators). Once these cooperators transmit or receive a unicast packet (a PTP message, a DTDODREP message of type 3 or a RDREP message), then they automatically become masters, and they can start to designate new cooperators for the RZ as well.

Based on this behavior, nodes that are not members of the RZ will never forward (transmit, receive or transmit and receive) a unicast data packet, since they will never be discovered by AGENET and so, they will never be included as possible next hops in any routing path.

In order to avoid possible inconsistence problems, the source node may decide to start more than once a RZ. For instance, it can start the RZ every time period of RZ_RESTART seconds. Another possibility is to restart the RZ when the source node detects that the paths are becoming too long, or too unreliable.

I.2.3 Ending the RZ

A timeout checking the inactivity of the node will be used to end the RZ. When a master does not transmit or receive any unicast packet (PTP messages, DTDODREP messages of type 3 or RDREP messages) during a time period of TOUT_RZ milliseconds, then the node leaves the RZ (inTheRZ = 0).

Analogously, when a cooperator does not receive any AGROHello message indicating it to become member of the RZ during a time period of TOUT_RZ milliseconds, then the node leaves the RZ (inTheRZ = 0).

In other words, when source nodes stop sending unicast data packets or stop starting DTDODMs, the RZ will automatically end.

I.3 AGROHello Format

Routing information is distributed through the network via AGROHello messages. Figure 101 shows the AGROHello message format.

Type,Version, flags…			Reserved		# of IDs		
RZ information			Level of battery of mine		MyTimestamp		
# of nodes	ID_1	Level of battery of ID_1	ID_2	Level of battery of ID_2	…	Timestamp	Status
# of nodes	ID_1	Level of battery of ID_1	ID_2	Level of battery of ID_2	…	Timestamp	Status

Figure 101. AGROHello message

Where:

· **Type (8 bits (unsigned))**

0

· **Version (8 bits (unsigned))**

0

· **Flags (8 bits)**

Unused

· **Reserved (32 bits)**

Unused

· **# of IDs (16 bits (unsigned))**

Number of routing paths that the AGROHello contains (always greater than 0 because an AGROHello always contain the information of the node sending it)

· **RZ information**

· **Hop count (8 bits (unsigned))**

A hop count value used to define the limits of the RZ

· **Reserved (24 bits)**

Unused

· **Level of battery of mine (8 bits (unsigned))**

The level of battery of the node sending the AGROHello

· **MyTimestamp (64 bits: 32 bits of timeval.tv_sec plus 32 bits of timeval.tv_usec)**

The time instant in which the node sends the AGROHello

And then, for every additional routing path (# of IDs–1) contained in the AGROHello:

· # of nodes (8 bits (unsigned))

Number of nodes in the routing path to reach this destination

· ID_1,ID_2,…,ID_N (64 bits * # of nodes)

The unique IDs of the nodes of the routing path to reach this destination. ID_N is the destination, and the sequence of these unique IDs sets the order to follow towards this destination (i.e., "myID"→ID_1→…→ ID_N)

· Level of battery of ID_1,ID_2,…,ID_N (8_unsigned bits * # of nodes)

Level of battery associated to every node of the routing path

· Timestamp (64 bits: 32 bits of timeval.tv_sec plus 32 bits of timeval.tv_usec)

The timestamp that this destination set when it sent the AGROHello containing this information

· Status (8 bits (unsigned))

0 → This routing path does not contain any unidirectional link

1 → This routing path contains one or more unidirectional links

I.3.1 Sending AGROHellos

Only nodes that belong to a Routing Zone (as masters or cooperators) will send AGROHello messages. AGROHello messages are sent with a time period of T_AGROHELLO milliseconds.

In each AGROHello message, masters add:

- Their level of battery and a timestamp equal to the time when they create the AGROHello message
- One entry (unique IDs of all the nodes in the path to this destination, level of battery of these nodes, timestamp which the destination assigned to this entry, and status that this path has in the sub table of the DTDO Table related with AGENET) in the AGROHello message for every entry they have in the sub table of the DTDO Table related with AGENET (i.e., for every routing path of every destination node)
- Hop Count = RZ_HOPS

In each AGROHello message, cooperators add:

- Their level of battery and a timestamp equal to the time when they create the AGROHello message

- One entry (unique IDs of all the nodes in the path to this destination, level of battery of these nodes, timestamp which the destination assigned to this entry, and status that this path has in the sub table of the DTDO Table related with AGENET) in the AGROHello message for every entry they have in the sub table of the DTDO Table related with AGENET (i.e., for every routing path of every destination node)
- Hop Count = the maximum Hop Count value received in any AGROHello message during the last T_AGROHELLO*2 milliseconds (that for sure it will be > 0) minus 1

AGROHello messages will be forwarded according to section MIFO to the destination node ID 0 (i.e., broadcast).

I.3.2 Receiving AGROHellos

AGROHello messages will be processed from two different points of view:

1. From the point of view of the RZ, masters do not process the AGROHello messages they receive. Cooperators and the rest of nodes only process AGROHello messages for which the RZ information of the AGROHello message designates them as cooperators, either if these messages confirm their status of cooperators or if designate them as new cooperators (i.e., if the Hop Count value of the AGROHello message is > 0). These cooperators and new cooperators will update the timeout of the RZ.

2. From the point of view of the routing information contained in the AGROHello, both masters and cooperators must process all the AGROHello messages they receive (even when cooperators receive AGROHello messages with the Hop Count value equal to 0). When a master or a cooperator processes the routing information of an AGROHello message, it will update its routing table in this way:

First, for the first entry of the AGROHello (i.e., the entry to the node sending the AGROHEllo):

If inside the AGROHello there is not a direct route to us (i.e., a route entry telling that we can reach us directly through the node sending the AGROHello, that is, a route with ID_1="ourID")

```
    status_tmp = 1
```

Else

```
    status_tmp = 0
```

EndIf

Now, the node checks if it has already an entry in the routing table for the same destination and with the same Path field:

If the node does not have an entry in the routing table for the same destination and with the same routing path, the node will check if no loop is created with this new route:

> **If** no loop is created (This condition is checked by determining whether the node receiving the AGROHello message already belongs to the list of nodes which conform the path to the destination of this new entry (note that, unlike in the MiMiHellos, in the

AGROHellos we have this check that avoids the need to have to use the valid/invalid concept for the entries of the DTDO Table in order to keep more time the metada (i.e. timestamp of the entry) to avoid reinserting the same entry in a ping pong way between two nodes even when the originator of this entry has disappeared fomm the network)

Now, if the node has less than MAXAGENETENTRIES_DEST entries for this destination in the DTDO Table, we add the new entry with Status==status_tmp. The timeout of the entry will be set to TOUT_AGENETENTRY milliseconds.

If we already have MAXAGENETENTRIES_DEST entries for this destination, and one of these MAXAGENETENTRIES_DEST entries have status==1, and status_tmp==0, we have to substitute, from the entries with status==1, the one with worst heuristic in the DTDO Table (see section "3.2.1.2 Routing Heuristic (RH)") by the new one. The timeout of the entry will be set to TOUT_AGENETENTRY milliseconds and Status to status_tmp.

If we already have MAXAGENETENTRIES_DEST entries for this destination in the DTDO Table, and one of these MAXAGENETENTRIES_DEST entries have status==1, and status_tmp==1, we will substitute, from the entries with status==1, the one with worst heuristic in the DTDO Table (see section "3.2.1.2 Routing Heuristic (RH)") by the new one, only if the new one has better heuristic. In case of adding, the timeout of the new entry will be set to TOUT_AGENETENTRY milliseconds and Status to status_tmp.

If we already have MAXAGENETENTRIES_DEST entries for this destination in the DTDO Table, and none of these MAXAGENETENTRIES_DEST entries have status==1, and status_tmp==1, we discard the new entry.

If we already have MAXAGENETENTRIES_DEST entries for this destination in the DTDO Table, and none of these MAXAGENETENTRIES_DEST entries have status==1, and status_tmp==0, we will substitute, from the entries with status==0, the one with worst heuristic in the DTDO Table (see section "3.2.1.2 Routing Heuristic (RH)") by the new one, only if the new one has better heuristic. In case of adding, the timeout of the new entry will be set to TOUT_AGENETENTRY milliseconds and Status to status_tmp.

EndIf

Else

If the timestamp in the AGROHello message is greater than the timestamp of the entry in the routing table

The node updates the entry (i.e. the number of explosive neighbors of every node in the path, the level of battery of every node in the path, and the timestamp), the timeout of the entry to TOUT_AGENETENTRY milliseconds, and the Status to status_tmp.

Else

The node will discard this entry of the AGROHello message.

```
      EndIf
EndIf
```

Second, for every entry in the AGROHello different from the first one, we will proceed as these entries were the the first entry of the AGROHello (the previous algorithm), but the status of the new entry (in case the entry has to be inserted) will be calculated in the following way:

If we do not have a direct route (a route telling that to reach a node the nexthop is the node itself) to the node sending the AGROHello message in the DTDO Table (very strange case)

```
      status_tmp = 1
```

ElseIf we have a direct route to the node sending the AGROHello message in the DTDO Table and with status==1

```
      status_tmp = 1
```

Else

```
      status_tmp = the status of the entry inside the AGROHello
```

EndIf

I.4 Routing Heuristic (RH)

The Routing Heuristic (RH) will be used to decide which of all the entries that may be learnt via AGROHello messages will be stored in the routing table.

As explained before, a maximum of MAXENTRIES_DEST entries can be stored in the routing table for each destination. The entries with the best RH will be the entries stored in the routing table.

For every entry in the sub table of the DTDO Table corresponding to AGENET, the node has the following routing information for a destination node:

- The destination node
- The path to follow in order to reach this destination
- The level of battery of each nexthop in the path
- Information about unidirectional links in the routing path
- The timestamp of this information (i.e., the time instant when this information was generated by the destination node)

The path to follow to the destination node includes the sequence of nexthops, which implicitly tells us the number of hops of the path.

The RH, for a specific routing path, will be calculated as follows:

$$RH = \left(\sum_{i=1}^{\#hops} f((NH_i)^b) \right) + TS$$

where

#hops = number of nodes in the Path field of the entry of the AGROHello we are calculating the RH

and

$$MAXTIMESTAMP < Timestamp \Rightarrow TS = 0$$
$$MAXTIMESTAMP \geq Timestamp \Rightarrow TS = \left(\frac{MAXTIMESTAMP - Timestamp}{2} \right)$$

where

MAXTIMESTAMP-Timestamp is in seconds and always with low round-off (e.g. if the result of *MAXTIMESTAMP-Timestamp* is 1.01 or 1.99 seconds, it will be rounded off to 1, and so *TS* will be 0.5)

and

$$MAXTIMESTAMP = \max_{i=1}^{\#entries} (T_i)$$

where

#entries = number of different entries for this destination in the DTDO Table

and

T_i = timestamp of the entries for this destination in the DTDO Table

and

Timestamp = timestamp of the entry of the AGROHello we are calculating the RH

and

$$f((NH_i)^b) = 1 + \left(\frac{100\% - b}{100\%} \right)$$

where

b = level of battery (in %) of the i-th nexthop NH_i of the entry of the AGROHello we are calculating the RH

Note that for every node in the routing path the heuristic penalizes with a value between 0 and 1 depending on the level of battery that the node has (if close to battery empty, the heuristic is penalized in 1. If full battery, the heuristic is not penalized). Moreover, for every 2

seconds of difference between timestamps the heuristic penalizes as if it was a path with one more hop. In this way, the lower the RH, the better the routing path.

Appendix J. MIMI Details

J.1 Miraveo Nodes (ID)

Each node of a MANET will have associated a unique ID of 64 bits. In fact, this unique ID will be the key field to uniquely identify a node.

The first two bytes of the ID will be set to 0 (reserved), and the subsequent six bytes will be set to the MAC address of the Wi-Fi chipset of the node.

J.1.1 IP

Each node of a MANET will have associated an IP address.

The IP address of a node will be randomly chosen in the range between the IP address 10.0.0.1 and the IP address 10.255.255.254.

We will assume that never two nodes will collide in the same MANET and at the same time with the same IP address.

J.1.2 Miraveo Applications

Miraveo Applications are the functionalities that a node can have installed. For example, a chat application, a bulletin board application, etc.

An application can be identified by its pluginID. In other words, pluginIDs will allow identifying which are the applications that a specific node has currently installed.

J.2 MIMITable

The MIMITable will be used to store some information of the current nodes of the MANET we are connected to.

Each entry identifies a node. The fields of each entry of the MIMITable are:

- **ID (uint64_t)**

  The unique ID of the node

- **GW (bool)**

  A boolean indicating if this node is a gateway to an external network (e.g., the Internet)

- **TTL (uint8_t)**

  The time to live of this information once we send it through the network via MIMIHellos

- **timestamp (struct timeval)**

  The time instant when this node generated this information

· **status (uin8_t)**

0 → the node is valid and has a bidirectional link with us

1 → the node is valid but has been discovered over a unidirectional link

2 → the node has expired and it is not valid.

· **timeout**

The timeout in charge of invalidating/deleting this entry in case of expiration

The key field of the MIMITable is the unique ID of each node, meaning that there cannot be two entries with the same ID.

Each entry is initially created as an entry with status==0 or status==1. The timeout of each entry will be set to TOUT_MVNODE milliseconds once the entry is inserted for the first time or updated. If the timeout expires, the entry will be invalidated (set status=2). From now on, the timeout of the entry will be set to TOUT_MVNODE*2 milliseconds. If the timeout expires, the entry will be finally deleted. If the entry is validated again (set status=0 or status=1) before timeout expiration, then the timeout of the entry will be set to TOUT_MVNODE milliseconds again as if the entry was inserted for the first time or updated.

Entries with status==0 are nodes discovered over only bidirectional links (in the sense of radio communication). Entries with status==1 are nodes discovered over one or more unidirectional links (in the sense of radio communication). Entries with status==2 are invalid nodes, meaning that these entries are only used to keep metadata of the expired nodes during a certain period of time (needed to avoid the problem of the ping pong hello between two nodes when the originator of the hello information has left the network).

J.3 MIMIHello

MIMIHellos are the messages used to distribute the node information to the rest of the nodes of the MANET. See Figure 102.

version	Reserved	M		
$mbox_0$	TTL_0		$timestamp_0$	
ID_1	$mbox_1$	TTL_1	$timestamp_1$	$status_1$
…				
ID_{M-1}	$mbox_{M-1}$	TTL_{M-1}	$timestamp_{M-1}$	$status_{M-1}$

Figure 102. MIMIHello

The fields of a MIMIHello are the following:

· **version (8 bits (unsigned))**

0

· **reserved (32 bits)**

0

· **M (16 bits (unsigned))**

The number of nodes this MIMIHello is carrying

· **ID_x (64 bits)**

The unique ID of the node X (X≠0)

· **$mbox_x$ (1 bit)**

1 if the node X is a gateway

· **TTL_x (7 bits (unsigned))**

The time to live of this information

· **$timestamp_x$ (64 bits)**

The time instant when node X generated this information

· **$status_x$ (8 bits)**

Only valid for X>0 (i.e., for all the nodes except for the node sending the MIMIHello). The status of the node X from the point of view of the node sending the MIMIHello

J.3.1 Sending MIMIHellos

Once a node connects to a MANET, MIMI starts sending a MIMIHello every T_MIMIHELLO milliseconds. In the MIMIHello, MIMI will put the information of its own node:

- $mbox_0$ = 1 if we are a gateway. 0 otherwise
- TTL_0 = MAXALLOWED_HOPS
- $timestamp_0$ = “now”

plus the information (ID_x, $mbox_x$, TTL_x, $timestamp_x$ and $status_x$) of all the nodes that there are in the MIMITable with $TTL_x > 0$ and with $status_x$==0 or $status_x$==1 but not with $status_x$==2.

MIMIHello messages will be forwarded according to MIFO to the destination node ID 0 (i.e., broadcast).

J.3.2 Receiving MIMIHellos

Each time a MIMIHello is received:

First, for the node sending the MIMIHello (i.e., the unique ID of the SourceRoutingList of the MIFO Header of the MIMIHello), that we will call it N_0, we will check if we (as node) are inside the MIMIHello:

If we are inside

stat_tmp = 0

add_node(N_0,stat_tmp)

Else

stat_tmp = 1

add_node(N_0,stat_tmp)

Endif

Second, for each node entry of the MIMIHello different from ourselves (and of course, different from the node sending the MIMIHello), that we will call them N_x (x>0):

If (stat_tmp == 1)

add_node(N_x,stat_tmp)

Else

add_node(N_x,N_x.status)

Endif

add_node(MIMIHello entry N, status st)

{

We will check if an entry in the MIMITable corresponding to the node N already exists:

· If it does not exist, we will add a new entry for this node in the MIMITable only if MIMI_GAP*0.75 consecutive MIMIHellos containing this node have been just previously recevied, i.e., if MIMI_GAP*0.75 MIMIHellos (including this MIMIHello) containing this node have been received in the last T_MIMIHELLO*(MIMI_GAP*0.75+1/2) milliseconds.

If added, the new entry will be added using the information contained in the MIMIHello entry N (or if the entry is N_0, the ID of the node will be obtained from the SourceRoutingList of the MIFO header of the MIMIHello instead of from the Payload of the MIMIHello) and we will set the timeout of the new entry to TOUT_MVNODE. For the TTL field, we will set this field to the value of the MIMIHello – 1. And for the status field, we will set this field to the st.

· If it exists and it has status==2, we will check if the timestamp contained in the MIMIHello is greater or equal (i.e. newer) than the timestamp that we have in the MIMITable for this node.

· If it is smaller, we will discard the information of this node contained in the MIMIHello.

· Otherwise, we will proceed as if no entry for the node N was found in the MIMITable: we will add a new entry for this node (in fact, we will move the already existent entry from status==2 to status==st) in the MIMITable only if MIMI_GAP*0.75 consecutive MIMIHellos containing this node have been just previously recevied, i.e., if MIMI_GAP*0.75 MIMIHellos (including this MIMIHello) containing this node have been received in the last T_MIMIHELLO*(MIMI_GAP*0.75+1/2) milliseconds.

If added (moved from status==2 to status==st), the new entry will be set using the information contained in the MIMIHello entry N (or if the entry is N_0, the ID of the node will be obtained from the SourceRoutingList of the MIFO header of the MIMIHello instead of from the Payload of the MIMIHello) and we will set the timeout of the new entry to TOUT_MVNODE. For the TTL field, we will set this field to the value of the MIMIHello – 1. And for the status field, we will set this field to the st.

· Otherwise (it exists and it has status 0 or 1), we will check if the timestamp contained in the MIMIHello is greater (i.e. newer) than the timestamp that we have in the MiMiTable for this node.

· If it is smaller, we will discard the information of this node contained in the MIMIHello.

· If it is equal:

· If the entry in the MIMITable corresponding to the node N has status 0 and st == 1, then we will discard the information of this node contained in the MIMIHello (and so, no timeout update will be performed for the corresponding MIMITable entry.

· If the entry in the MIMITable corresponding to the node N has status 0 and st == 0, then we will update all the fields of the entry of the MIMITable by the value of these fields in the MIMIHello. For the TTL field, we will update the field only when the value of the MIMIHello – 1 is greater than the value of the MIMITable. Moreover, we will set the timeout of the entry to TOUT_MVNODE. The status field does not need to be changed.

· If the entry in the MIMITable corresponding to the node N has status 1 and st == 1, then we will update all the fields of the entry of the MIMITable by the value of these fields in the MIMIHello. For the TTL field, we will update the field only when the value of the MIMIHello – 1 is greater than the value of the MIMITable. Moreover, we will set the timeout of the entry to TOUT_MVNODE. The status field does not need to be changed.

· If the entry in the MIMITable corresponding to the node N has status 1 and st == 0, then we will update all the fields of the entry of the MIMITable by the value of these fields in the MIMIHello (for the TTL field, we will always update the field, independently of its value in the MIMIHello). Moreover, we will set the timeout of the entry to TOUT_MVNODE and the status field to st.

· If it is greater, then we will update all the fields of the entry of the MIMITable by the value of these fields in the MIMIHello. Moreover, we will set the timeout of the entry to TOUT_MVNODE and the status field to st.

}

Appendix K. MStack Parameters

	#ifdef AdhocMode
AGENET_GAP	4
T_AGROHELLO	100 msecs
TOUT_AGENETENTRY	10000 msecs
MIMI_GAP	4
T_MIMIHELLO	250 msecs
TOUT_MVNODE	6000 msecs
EXFOHELLO_RATIO	2
EXFO_GAP	5
T_EXFOHELLO	100 msecs
T_CHECKNEXTHOP	1000 msecs
TOUT_NEXTHOP	T_EXFOHELLO * EXFO_GAP msecs
TOUT_EXPNEIGHBOR	TOUT_AGENETENTRY msecs
TOUT_RZ	TOUT_AGENETENTRY
MAXAGENETENTRIES_DEST	2
MAXDATASIZE_DTDO	200MB
MINTTL_DATADTDO	1 msec
MAXTTL_DATADTDO	1800000 msecs
MAXSIZE_DELTOYACACHE	32MB
THRESHOLDDATASIZE_CACHE	1MB
MIN_MRT	1000 msecs (MIN_MRT > T_QUERYDTDOS)
MAX_MRT	3600000 msecs
T_QUERYDTDOS	2000 msecs
NET_TRAVERSAL_TIME	200 msecs
TIMEBETWEEN_INCLUDEDDTDODREQs	NET_TRAVERSAL_TIME msecs
DTDODM_PORT	13005
PTP_PORT	13006
AGENET_PORT	13007
MIMI_PORT	13008
RDM_PORT	13009
DOCKDM_PORT	13010
ASK2ME_PORT	13011
BROADCAST_IP	10.255.255.255
MAXALLOWED_HOPS	4 (same as MiMi)
MAXDTDODREQ_TOFORWARD	2
DTDODREQID_LIFE	1000 msecs
RZ_HOPS	1
ALOTofEXPNEIGHBORS	5
MAX_EXPLOSIONS	2
MAX_PTPCONSECUTIVELOSES	2
MAXNUM_PTPSQENTRIES	5
T_MERGESPAN	TOUT_MVNODE / 2
MAX_DTDODREQS	5
MAX_RDREQS	4
MAX_DOCKBUFFERSIZE	Sizeof(DOCKBuffer entry with a DTDO of maximum size); where a DTDO of maximum size is sizeof(DTDO Table entry of dataSize == MAXDATASIZE_DTDO)
MIN_CHAINRESISTANCE	500 msecs
MAX_CHAINRESISTANCE	1800000 msecs
T_QUERYDOCKS	T_QUERYDTDOS
MAX_DOCKDREQS	MAX_DTDODREQS
TIMEBETWEEN_INCLUDEDDOCKDREQs	TIMEBETWEEN_INCLUDEDDTDODREQs
MAXDOCKDREQ_TOFORWARD	MAXDTDODREQ_TOFORWARD
DOCKDREQID_LIFE	DTDODREQID_LIFE
MIN_MPT	2*MIN_MRT
MAX_MPT	MAX_MRT
T_ASK2ME	NET_TRAVERSAL_TIME
T_ASK2ME_FINISH	T_ASK2ME
MAXNUM_DTDOAQENTRIES	MAXNUM_PTPSQENTRIES

TOUT_RXCHAIN	MAX_CHAINRESISTANCE + MIN_CHAINRESISTANCE
MAXSIZE_HISTORYRXCHAIN	10
MAXSIZE_RXUNCHAIN	10
MSERVER_ID	"static ID"
MSERVER_IP	"static public IP"
MSERVER_BATTERY	50%
TOUT_MIFOENTRY	60000 msecs