

# The Spanning Tree Protocol

Guangyu Dong, Jorg Liebeherr, MNG Group

Wednesday, March 30, 2005

This is a draft and not a final version.

© 2005-8. All rights reserved. All material is copyrighted by the authors.

## Table of Contents

<b>TABLE OF CONTENTS .....</b>	<b>1</b>
<b>1. INTRODUCTION .....</b>	<b>2</b>
<b>2. OVERVIEW OF SPANNING TREE PROTOCOL .....</b>	<b>3</b>
2.1 JOINING AND LEAVING .....	3
2.2 BUILDING SPANNING TREE .....	3
2.2.1 Overview .....	3
2.2.1 Measuring Link Quality .....	5
2.2.2 Algorithms of Choosing Ancestor .....	5
2.2.3 Asymmetric Link Problem .....	7
2.2.4 Count-To-Infinity Problem .....	8
2.3 MULTICAST ROUTING .....	9
2.3.1 Forwarded in Unicast Channel .....	9
2.3.2 Forwarded in Broadcast Channel .....	9
2.3.3 A Unified Solution .....	10
2.4 UNICAST ROUTING .....	10
2.4.1 Forwarding Table .....	11
2.4.2 Building On-Demand Unicast Path .....	11
2.4.3 Forwarding .....	14
<b>3. DETAILED PROTOCOL DESCRIPTION .....</b>	<b>15</b>
3.1 BASIC ELEMENTS .....	15
3.1.1 SPT ID .....	15
3.1.2 Physical Address .....	15
3.1.3 Neighbors .....	15
3.2 TABLES .....	15
3.2.1 Tree Information Table .....	15
3.2.2 Neighborhood Table .....	16
3.2.3 Backup Ancestor Table .....	16
3.2.4 Adjacency Table .....	16
3.2.5 Core Table .....	17
3.2.6 Forwarding Table .....	17
3.3 BEACON .....	17
3.3.1 Beacon Message .....	17
3.3.2 Sending Beacon .....	18
3.3.3 Receiving Beacon .....	18
3.3.4 Timeout Mechanism .....	20
3.3.5 Backup Ancestor .....	21
3.4 JOINING AND LEAVING .....	21
3.4.1 Joining .....	21
3.4.2 Leaving .....	22
3.5 MULTICAST DATA PACKET FORWARDING .....	22
3.5.1 Packet Header .....	22

3.5.2 Forwarding Decision.....	22
3.6 UNICAST DATA PACKET FORWARDING .....	22
3.6.1 Forwarding Table.....	22
3.6.2 On-Demand Route Maintenance .....	23
<b>4. SETTINGS.....</b>	<b>25</b>
4.1 TIMERS FOR MAINTAINING SPANNING TREE.....	25
4.2 TIMERS FOR MAINTAINING UNICAST PATH .....	26
4.3 OTHER SETTINGS .....	<b>ERROR! BOOKMARK NOT DEFINED.</b>
<b>5. MESSAGE FORMATS.....</b>	<b>26</b>
5.1 PROTOCOL MESSAGE FORMATS.....	27
5.1.1 Beacon Message.....	27
5.1.2 Goodbye Message.....	28
5.1.3 Route Request Message.....	28
5.1.4 Route Reply Message.....	28
5.2 DATA MESSAGE FORMATS .....	29
5.2.1 Multicast Packet Header.....	29
5.2.2 Unicast Packet Header .....	29
<b>6. INTERACTION WITH OVERLAY SOCKET.....</b>	<b>30</b>
6.1 INTERFACE FUNCTIONS .....	30
6.2 OPERATIONS OF FORWARDING ENGINE .....	31
<b>7. STATES AND STATE TRANSITIONS .....</b>	<b>32</b>
7.1 STATE DEFINITIONS .....	32
7.2 STATE TRANSITION DIAGRAM .....	32
7.3 ACTIONS OF SPT PROTOCOL .....	33

## 1. Introduction

In this document, we describe a network protocol which establishes and maintains a spanning tree connecting a group of mobile device in the wireless ad hoc network. The protocol is called Spanning Tree Protocol (or SPT) and has been implemented and tested as a part of the HyperCast 3.0 overlay software.

This Spanning Tree Algorithm is inspired by Perlman’s Spanning Tree Algorithm for bridges. Perlman’s spanning tree algorithm is used to prevent the existence of a loop in networks that contain parallel bridges. If there is a loop in the network, a packet will be forwarded by bridges for indefinite times, which can result in increased traffic and degradation in network performance. Since a tree has no loop, Perlman solves the loop problem by using a spanning tree algorithm to organize those bridges into a tree.

Similarly, in the ad hoc environment, after a spanning tree is built to connect a group of mobile devices, a packet can be always flooded into all members along the tree structure without loop and duplicated transmission. And a mechanism has been built to provide aid for unicasting a packet to some specific receiver without having to flood it to the whole network.

We refer to the protocol entities that execute the SPT protocol as nodes. Each node has a logical address and a physical address. The logical address in SPT protocol is a positive integer number, called SPT ID and set to 32bits. The SPT ID should be unique for each node in a SPT group. The physical address is for transmitting protocol messages and data packets between nodes, which consists of an IP address and a port number.

There's ordering between different SPT ID, which we define as the natural ordering of positive number.

## 2. Overview of Spanning Tree Protocol

### 2.1 Joining and Leaving

Since the Spanning Tree Protocol is implemented by building an overlay network, it must provide some rendezvous mechanism to enable nodes who want to join the overlay network to communicate with nodes in the overlay.

Basically there are three kinds of rendezvous mechanism for an overlay network: (1) Broadcast: non-members have a broadcast mechanism that is available to them. They use this to announce themselves to members of the overlay network. (2) Buddy List: non-members maintain a list of members that are likely to be in the overlay network (a "buddy list"). They use this list to contact members. (3) Server: non-members contact a well-known server that establishes communication between members and non-members of an overlay network.

In the SPT Protocol, we choose the first method, and it's done in an implicit way for the unique characteristics of wireless ad hoc network. Since a node in the overlay network will broadcast a beacon message periodically, and all adjacent nodes in its transmission range will get the beacon message. And since a no-member node can contact to a member node only when it's in its transmission range, the broadcasted beacon message is a natural way to find the member. So actually, the rendezvous process is done without any explicit additional mechanism.

**Join:** When a node wants to join a spanning tree network, it simply starts to periodically send out and receive beacon messages.

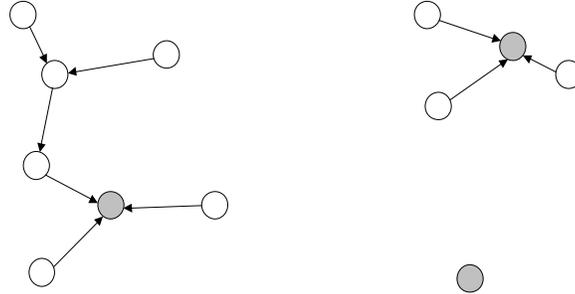
**Leave:** When a node wants to leave the network, it sends out a Goodbye message and stops sending and receiving beacon messages.

### 2.2 Building Spanning Tree

#### 2.2.1 Overview

A spanning tree is built by locally exchanging information between adjacent nodes. Two nodes are adjacent when they can communicate to each other directly, which means they are in the transmission range of each other in ad hoc network. For any two adjacent nodes, we say there is a single hop path between them. Here we don't assume wireless channels are always symmetric, but asymmetric channels will be discarded by some mechanism.

When we need to establish a spanning tree to connect a group of members, it's necessary that those members should be in a network partition already. That is, for any two nodes in the group, theoretically there exists a multi-hop path connecting them. When there are separated network partitions, a spanning tree will be formed for each partition by the protocol, as is shown in Figure 1.



**Figure 1: Network Partitions<sup>1</sup>**  
 When partitions exist, multiple spanning trees are formed

The basic algorithm of building and maintaining a spanning tree is that, beacon messages are exchanged periodically between adjacent nodes, and a node use the information stored in the beacons received to decide its ancestor. The basic fields of the beacon message are listed below:

*(Sender ID, Physical Address, Core ID, Ancestor ID, Cost, Path Metric, Adjacency Table)*

In a beacon message, the *Sender ID* is used to identify the sender node of the message and the *Physical Address* field gives the other nodes within wireless transmission range a way to send unicast message. *Core ID* field tells which node is the current spanning tree core of the message sender, and every node in the same SPT overlay network should finally agree on the same *Core ID* in order to be connected. The *Ancestor ID* tells which node is the current spanning tree ancestor of the sender, and the *Cost* field is the hop count to the core node. The *Path Metric* field is used by receivers as the part of the criteria to select their ancestor, and in the three ancestor selecting algorithms, there are different ways to compute the *Path Metric* field. A node's ancestor and all its children compose of its neighborhood table. At the end of each beacon message, there is an *Adjacency Table* listing the ID of all nodes that the beacon sender has recently heard from. A link quality value for each adjacent node is also included in the adjacency table. This table is used to avoid asymmetric wireless links, which are not unusual in the real world wireless network.

A SPT node in the overlay network periodically broadcasts the beacon messages to all adjacent nodes, not only to exchange the topology information, but also refresh its active state. If a node A has been not hearing beacons from another node B for some period of time, B will be removed from both the neighbor table and the adjacency table. The final function of beacon message is a probe of the wireless link quality, for which we will have detailed description in following sub-sections.

<sup>1</sup> The convention of drawing spanning tree figure used in all parts of the document is: 1) the gray node is the core of the tree; 2) An arrow pointing from node A to node B means B is the ancestor of A; 3) a pair of number  $(r, c)$  on the arrow pointing from node A to node B means that A's core is  $r$ , ancestor is B, the cost from A to the core is  $c$ .

## 2.2.1 Measuring Link Quality

SPT protocol uses the delivery ratio of beacon messages as the metric of link quality between adjacent nodes pair. The link quality measurement of each individual wireless link is used both as a way to avoid asymmetric link, and to compute the path metric value in some parent selecting algorithms that will be covered in following sub-sections.

Formally, the link quality from node  $A$  to  $B$  is defined as (given a parameter  $N$ ):

$$LQ(A,B) = \text{Delivery Ratio of Recent } N \text{ Beacons From } A \text{ to } B$$

And we define the bidirectional link quality value as:

$$BiLQ(A,B) = \min(LQ(A,B), LQ(B,A))$$

Node  $B$  calculates  $LQ(A,B)$  and stores it in table. And this value is put into the adjacency table at the end of the beacon message of  $B$  such that  $A$  can get it and calculates the bidirectional link quality value between  $A$  and  $B$ . It is not an exact calculation of  $BiLQ(A,B)$  since the beacon message from  $B$  can also be lost, but it is still a good estimation because if beacon from  $B$  is lost the  $LQ(B,A)$  will bring the  $BiLQ$  value down. By the way described above, every node keeps tracking of the bidirectional link quality between itself and all adjacent nodes. The link quality measurement is a value ranging from 0 to 1.

We place a threshold value as a parameter, such that any beacon message coming from a node that has a  $BiLQ$  lower than the threshold is dropped, to guarantee that the node only establish neighborhood relationship with nodes having reliable bidirectional wireless connectivity. A dropped beacon message is still used to calculate the link quality. When this threshold is only set to a small value larger than zero, it is simply used to eliminate the asymmetric link.

## 2.2.2 Algorithms of Choosing Ancestor

### 2.2.2.1 Basic Algorithm

Based on the beacon messages received from adjacent nodes, a node decides who should become its spanning tree ancestor. Firstly, the node in the network with the minimum ID becomes the core such that all nodes can easily make an agreement about who is the core node. Secondly, every node calculates a *Path Metric* value for the spanning tree path leading from the current node to the core, and this value is put in to  $B$ ,  $A$  computes the possible *Path Metric* value if  $B$  is selected as  $A$ 's ancestor, according to the *Path Metric* field in the beacon and some other rules depending on different algorithms.

When node  $A$  receives a beacon message  $M$  from  $B$ , it tries to determine if  $B$  is a better ancestor by the following rules:

(a) If  $M.CoreID < A.CoreID$ , then  $B$  is a better ancestor. Otherwise if  $M.CoreID = A.CoreID$ , continue the following judgments.

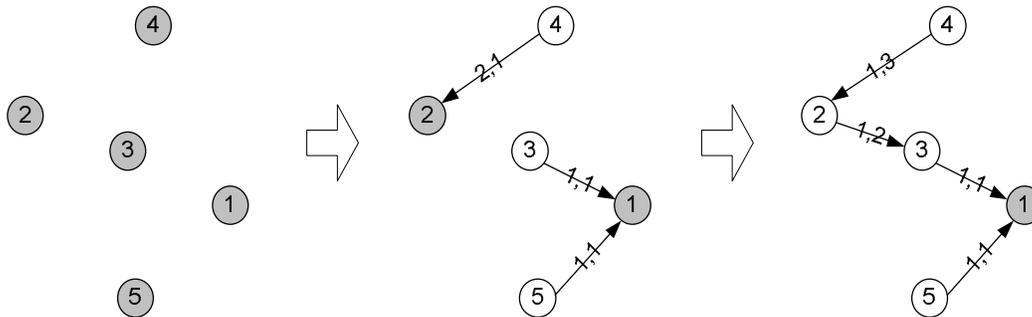
(b) If the *Path Metric* value computed based on  $M$  is larger than  $A$ 's current *Path Metric* by a preset threshold value (*Jumping Threshold*),  $B$  is a better ancestor. The

*Jumping Threshold* is used to prevent the spanning tree topology from being changed when the environment only has very little variance.

By the rules specified above, every node tries to optimize the *Path Metric* value of the path leading to the core node, though it would not necessarily choose the best path while limited by the *Jumping Threshold*. In our protocol to build a shared spanning tree, every node locally decides its ancestor in order to obtain a better path leading to core, and the result is a shared tree topology with better global metric. Then the quality of the tree is determined by how we define and compute the Path Metric value, which is the key difference between the algorithms described in following subsections.

When every node sees and sends unchanging beacon messages, the spanning tree is successfully established, and every node knows its ancestor, its descendants, and the core node. An example of the process for establishing spanning tree is show in Figure 2.

Tree structure is up to change when the network topology changes.



**Figure 2:** Process of Establishing Spanning Tree

### 2.2.2.2 Minimum Hop Count Algorithm

In this algorithm, the *Path Metric* of a node is defined as (- Hop Count to the Core), and the *Jumping Threshold* is set to 1. We put a negative sign because in our protocol, larger value means better for the *Path Metric*. The resulting spanning tree is a classic minimum cost tree. In fact, for this algorithm a *Path Metric* field in the beacon message is not needed since a *Cost* field already gives the enough information.

### 2.2.2.3 Link Quality Algorithm

In this algorithm, a node use the *BiLQ* to its ancestor as the *Path Metric*. To avoid loop, the node A would not choose node B as the ancestor if  $A.Cost \leq B.Cost - 2$  because B is probably A's descendent in this case. Basically, every node only tries to optimize the quality of the one hop link to its ancestor, under the condition that loop is impossible, regardless of what the resulting path length is. The *Path Metric* value of a node varies from 0 to 1. A moderate *Jumping Threshold* parameter is important to keep the topology relatively stable because usually the link quality between adjacent nodes always oscillates constantly even though everything is stationary. On the other hand, the *Jumping Threshold* cannot be too large such that it prevent node from choosing a better ancestor quickly.

Since there is almost intended control on the length of path to the core, the hop count tends to be larger and there is no defense against a spanning tree path going unnecessarily back and forth.

#### 2.2.2.4 Path Quality Algorithm

A product of all *BiLQ* values along the path to core node is evaluated as the *Path Metric*. To calculate the Path Metric, a node gets the product of the *BiLQ* to the ancestor and the *Path Metric* of the ancestor that can be obtained in the beacon messages. A node in the network intends to optimize quality of the path between itself and the core node. Same to the Link Quality algorithm, in this case the *Path Metric* is a value ranging from 0 to 1. The analysis of the *Jumping Threshold* parameter for this algorithm is similar to the Link Quality algorithm. On the other hand, the average hop count is expected to be shorter than Link Quality algorithm because a longer path tends to have smaller path metric since it is a product of more *BiLQ* values.

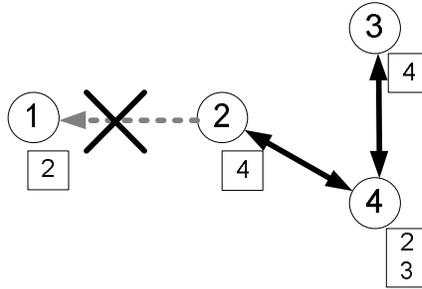
Although this algorithm seems to perform better in overall than the Link Quality approach with the ability to avoid unnecessarily long path, in some specific situations the later algorithm outbids. For example, when the communication happens majorly on two nodes on the same to-core path, the traffic doesn't go over the whole path therefore the Link Quality algorithm might exhibit better performance. However, since we assume that the traffic pattern is not predictable, we favor the Path Quality algorithm because it is expected to work better in average case.

#### 2.2.3 Asymmetric Link Problem

The SPT protocol is based on the assumption of symmetric channel. That is, if node A can hear from node B, the node B can also hear from node A, node A and node B have exactly the same transmission range. However, in the real wireless environment, this assumption is not necessarily the truth. Most the current ad hoc networks use 802.11 as the MAC layer. In 802.11, the broadcast operation doesn't require RTS/CTS operation or acknowledgement. As a result, it's quite possible that there exists an asymmetric channel between two nodes.

We use the *Bidirectional Link Quality* values of adjacent nodes to eliminate the asymmetric link. In SPT protocol, any adjacent node with a *BiLQ* value lower than a *Reliable Threshold* parameter is excluded from being an ancestor candidate. A decent threshold value can guarantee that only reliable link is used to form the topology. If we set this threshold value to 0.1, it is just used to avoid asymmetric link.

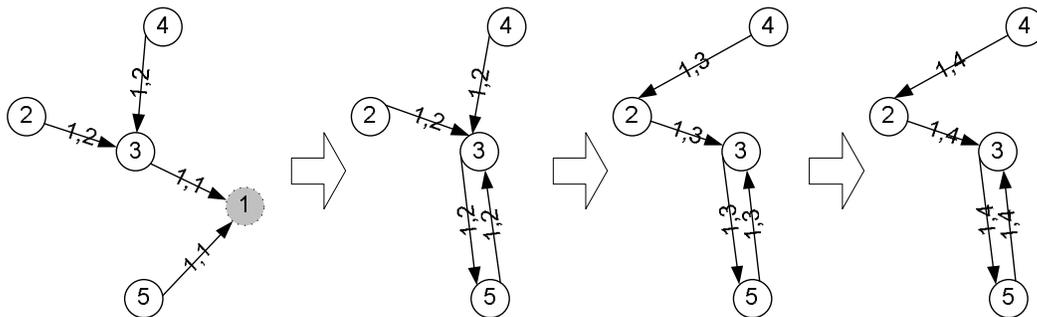
An example is shown in Figure 3. Since node 2 can find the *BiLQ* value of node 4 larger than the threshold and vice versa, a symmetric link is verified between these two nodes. On the other hand, since node 1 can't find itself in the Adjacency table of node 2, it will not process beacon message sent by 2 because *BiLQ* value of node 2 is zero.



**Figure 3:** Preventing Asymmetric Channel with Adjacency table  
The boxes are the adjacency table of each node

### 2.2.4 Count-To-Infinity Problem

Count-to-Infinity Problem will probably happen when a critical path leading to the core is destroyed for some reason and some part of the group has no longer any path to the core. For example, some node on the path leaves the group, crashes or move to somewhere else. The leaving node may be the core itself. Under some conditions the whole network will become unstable, as shown in Figure 5.



**Figure 4:** Count-to-Infinity Problem Caused by Leaving of Node 1

The cause of count-to-infinity problem is that, even though the core node has left, moved away or become unreachable, the core information announced by it is still being propagated through the network through some loop. A single node cannot detect this problem without additional information. So there should be a mechanism to make the core information stale. Simply checking whether the beacon message's Ancestor ID is the receiver itself will no solve the problem, since a loop may still form involving more than two nodes.

To solve this problem, in our implementation, a *Sequence Number* is added to the beacon message. The sequence number is only created by the core node and increments by 1 every time a beacon message is sent. For a node that is not the core, it stores the sequence number coming from the ancestor and sends it out in beacon message.

Every node maintains a hash table, called Core Table, which stores a list of (*Core ID*, *Sequence Number*, *Last Change Time*) entries. If the sequence number from some core

has not been increased for some specified period, any beacon message carrying the same or older sequence number will not be processed by the receiver.

## 2.3 Multicast Routing

### 2.3.1 Forwarded in Unicast Channel

After a spanning tree is established, multicast routing among all the tree members can be done along the tree. Since basically a node's location in the tree is unknown, how to forward a multicast data packet can not be decided by the source address of the packet. In our protocol, on receiving a multicast packet from one of its neighbors, a node will simply forward the packet to all other neighbors, if there is any.

When an underlying unicast wireless channel is used to transfer the data packet, the fields needed to be carried by the data packet to do the forwarding decision is a (*Source ID, Previous Hop*) pair.

An example is shown in the Figure 5 to show how node 3 forwards a multicast coming from node 4. As a result of our forwarding rules, the packet will be forwarded to node 1 and 2.

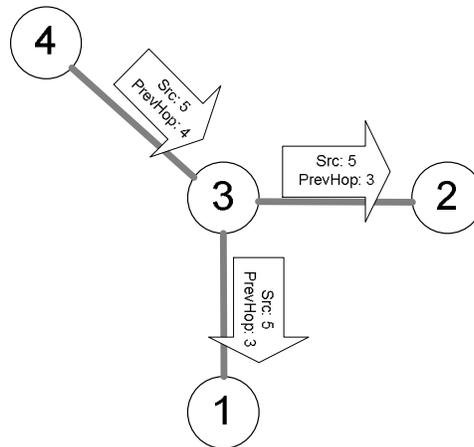


Figure 5: Forwarding Multicast Packet in Unicast Channel

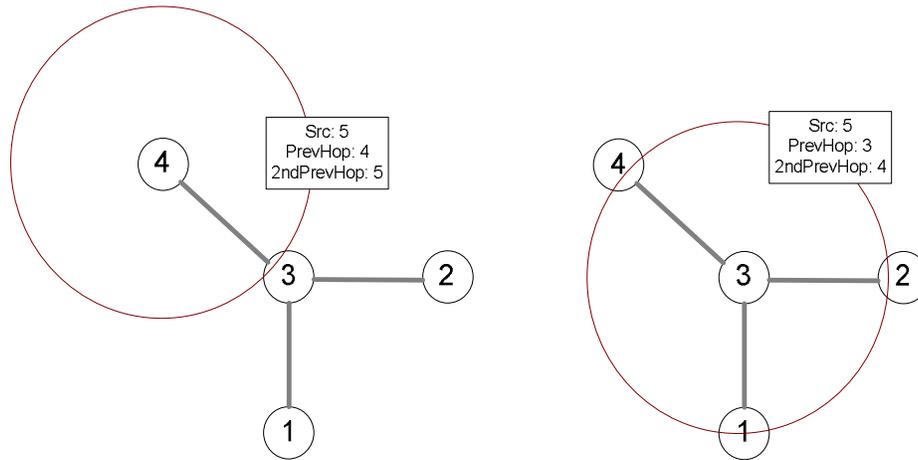
### 2.3.2 Forwarded in Broadcast Channel

When packet is forwarded by using the wireless broadcast channel, all adjacent nodes inside the transmission would have the chance to receive the packet. So some further control should be made to make it clear about which nodes are supposed to receive the packet.

In our protocol, when forwarded in underlying broadcast channel, a multicast data packet carries the fields of (*Source ID, Previous Hop, 2nd Previous Hop*). A node will only deliver and forward those multicast data packets coming from tree-neighbors and not listing the node's ID as *Previous Hop* or *2nd Previous Hop*. Data packet will be sent out at most once by each node forwarding it because a broadcast channel is being used.

Figure 6 shows an example about how node 3 forwards a multicast message received from node 4. Node 3 resends the message out by local broadcasting and all adjacent

nodes will receive it, including 1, 2, 4 and 3 itself. Node 3 will drop this message because its ID equals to *Previous Hop* of this message. Node 4 will drop this message because its ID equals the *2nd Previous Hop*. Node 1 and 2 will accept the message and do further forwarding when necessary.



**Figure 6:** Multicast Forwarding with Broadcasting Channel

### 2.3.3 A Unified Solution

By analyzing the solutions for forwarding multicast data packet in unicast or broadcasting channel, we found that they can be unified to a general scheme. In either solution, the data packet carries a list of nodes the packet has recently gone through (route record). However the difference is for forwarding in unicast channel, the size of list is 1, while in broadcast channel, the size of list is 2. When a node receiving a packet from a neighbor finds itself in the list, it will drop it.

Suppose the size of the list is  $N$ , then a routing loop involving  $N$  nodes can be avoided. For a spanning tree topology, in which only two-node-loop is possible,  $N=2$  is sufficient. However, for a protocol forming a mesh topology, to totally avoid loop,  $N$  has to equal to the *TTL* of the packet.

Thus, for forwarding multicast data packet, we got a single solution by adding a route record with limited size to the packet header, and let the size be a parameter of the network. When underlying unicast channel is used to forward data, the size should be at least 1 (previous hop), and when broadcast channel is used, size should be at least 2 (previous hop, 2<sup>nd</sup> previous hop). A node receiving the packet and seeing itself in the route record will not process it.

## 2.4 Unicast Routing

The difficulty of doing unicast in spanning tree is that the location of the destination in the tree is usually unknown. The exception only happens when the destination is the core node or it's one of the source node's neighbors. Certainly a unicast packet can be always flooded to the whole network as like it's a multicast packet, but it's very inefficient.

To route unicast data packet in an efficient way, we propose a unicast solution to build an on-demand unicast path between source and destination.

### 2.4.1 Forwarding Table

In our protocol, every node maintains a Forwarding Table containing a list of (*Destination, Next Hop*) pairs, which works as a routing cache. The node uses the forwarding table to lookup the next hop node to forward a unicast packet. How the forwarding table is updated is discussed in 2.4.2.

A forwarding entry will be removed from forwarding table if it expires or the next hop is no longer a neighbor.

As we have mentioned before, the forwarding table is not the only way to find the next hop leading to a destination. When the destination is core, or is a neighbor of the node forwarding the packet, the node also knows what the next hop is. On the other hand, even though the forwarding node can find the next hop from the forwarding table, if the next hop node is currently not its neighbor, it still cannot forwards the packet to the next hop node.

When a node doesn't know where to forward a unicast packet, it just forwards to all other neighbors except the previous hop, as like it's a multicast packet. So when the location of destination is unknown, the packet will be flooded to almost the whole network until it reaches the destination node.

### 2.4.2 Building On-Demand Unicast Path

#### 2.4.2.1 On-Demand Unicast Route Maintenance

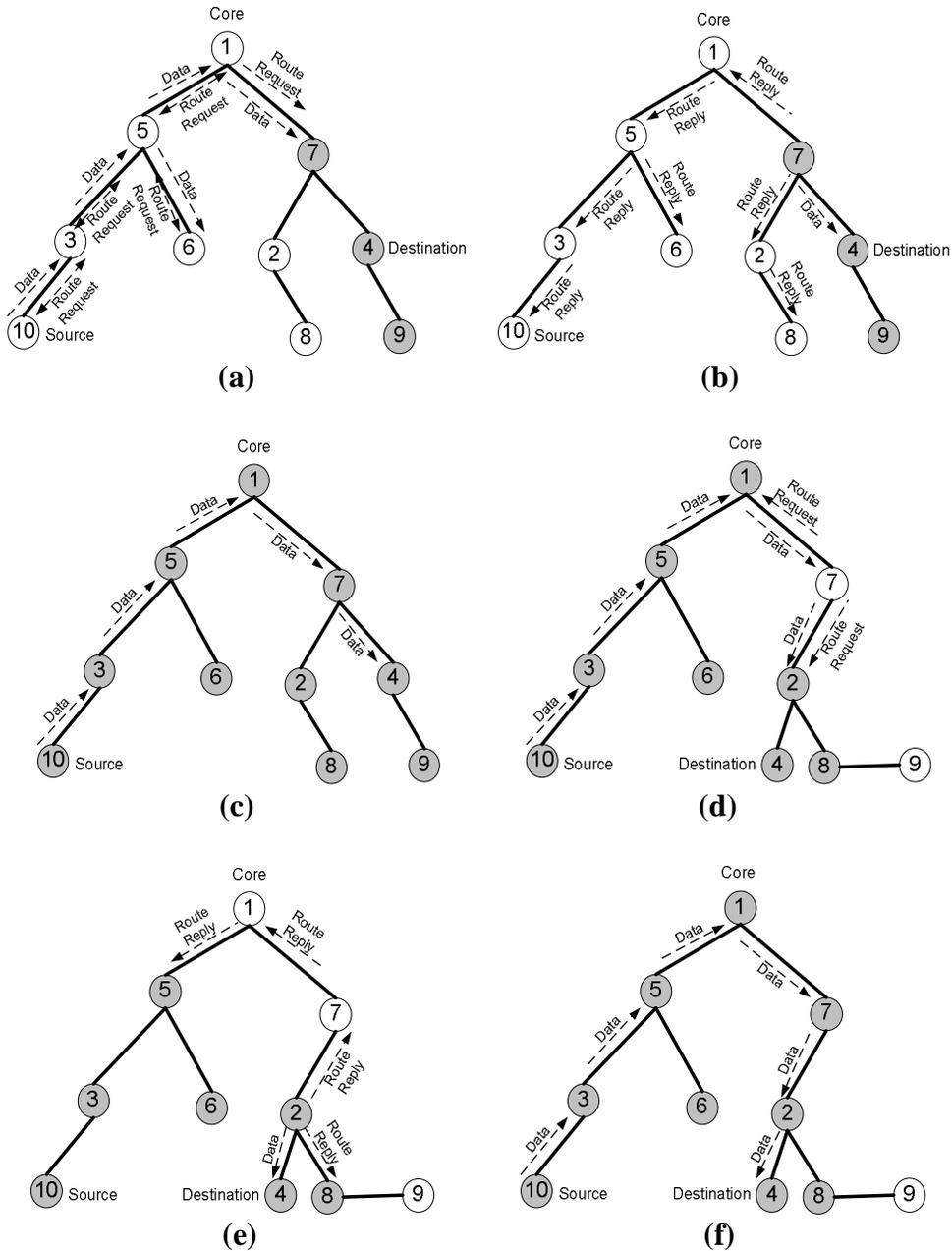
A unicast path built between the source and destination when all nodes on the route between them have a forwarding entry for that destination and carrying the right next hop information.

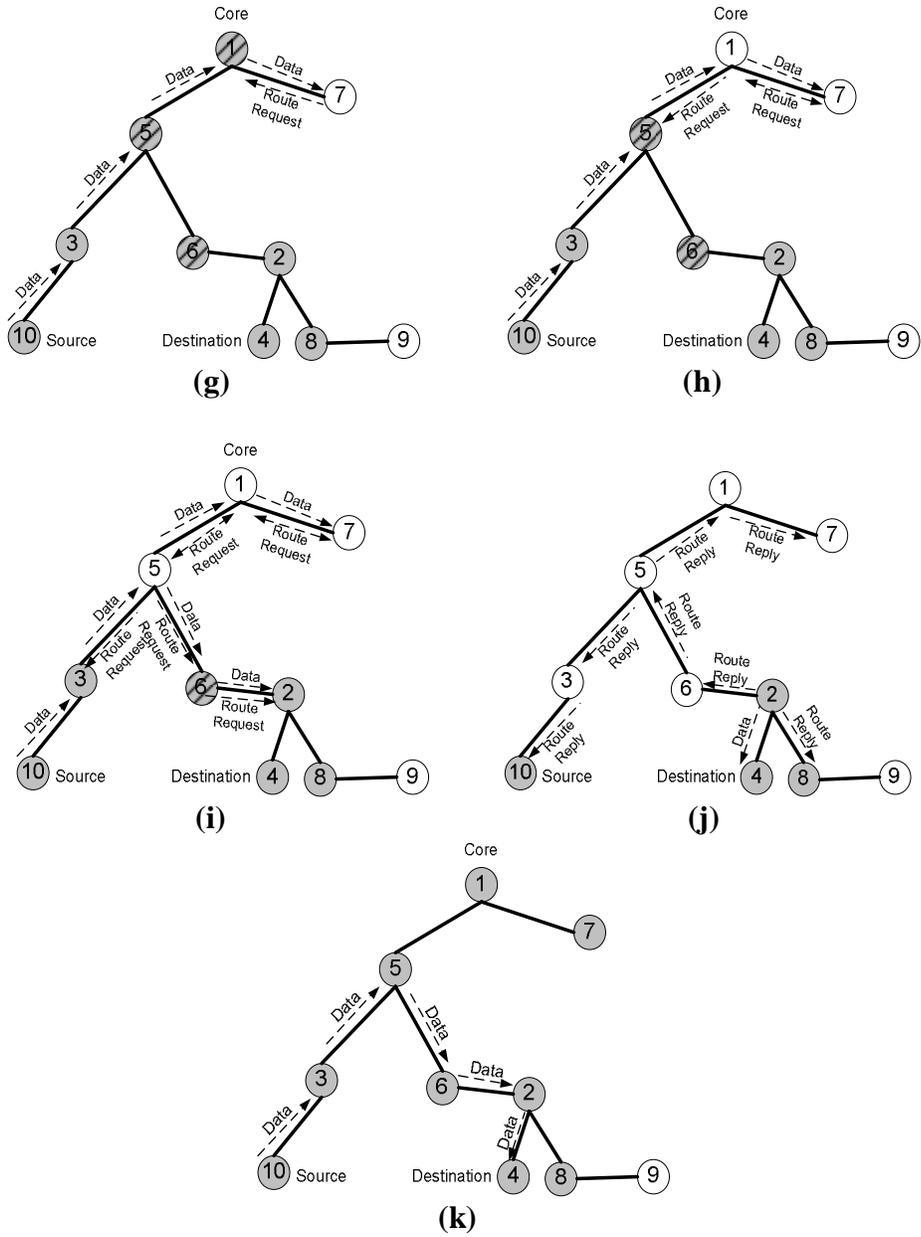
We suppose node  $S$  tries to send a unicast data packet to  $D$ , and the process to build and maintaining a unicast path is composed of several parts:

1. When a node tries to forward a unicast packet and doesn't know where to forward it, besides flooding the packet ahead, it also sends out a *RouteRequest* message to all its neighbors to inform them of its lack of the forwarding entry for  $D$ .
2. When a node holding a forwarding entry for  $D$  receives a *RouteRequest* from the next hop node, it learns that this forwarding entry is no longer valid so just removes it from the forwarding table.
3. When a node knowing where to forward packet to  $D$  (either  $D$  is one of its neighbors, or it holds a forwarding entry for  $D$ , or itself is  $D$ ) receives a *RouteRequest* for  $D$  from one of its neighbor, it sends out a *RouteReply* message to all neighbors excepts the next hop node.
4. When a node receives a *RouteReply* message for  $D$  from one of its neighbors, and it's not holding a forwarding entry for  $D$ , it update its forwarding table by adding the forwarding entry for  $D$  and set the next hop to the neighbor sending the message. If a forwarding entry for  $D$  has already existed, update the next hop field to the message sender. After then, if and only if a forwarding entry has been added or a forwarding entry has changed its next hop field due to receiving this *RouteReply*, the node will further forward this message to all other neighbors. The destination  $D$  itself will not process this message.

In the above situations, if a protocol message is being sent to multiple neighbors of a node (1, 3, & 4), it's sent in broadcast channel.

We illustrate our Route Discovery Algorithm in Figure 7 using an example. In our example, the source node is 10, and the destination is 4. In all of the sub-figures below, a node shown in gray color indicates that it knows where to forward the packet to destination 4.





**Figure 7: Route Discovery Example**

At the very beginning, all nodes in the network have empty forwarding tables, so the data is relayed by every node to all other neighbors. (a) shows that the packet is flooded to the network along the spanning tree until it reach node 7, who is 4's ancestor and will forward the data to 4 directly. In the mean time, because 7 receives a *RouteRequest* from 1, it will sends out a *RouteReply*, as shown in (b). Node 1, 5, 3 and 10 will further forward the *RouteReply*. As a result, in (c), a unicast path 10-3-5-1-7-4 is built. Node 2, 6 and 8 will also have the forwarding entries for *D*, because the *RouteReply* message is forwarded by each node not having a valid forwarding entry for *D*.

In (d), node 4 moves to another place and becomes a descendant of node 4, so the unicast path breaks at node 7. When some data packet reach node 7, node 7 found the 4 is no longer its neighbor, so it simply floods the data ahead to the remaining part of the spanning tree, along with a *RouteRequest* message to all neighbors. After the *RouteRequest* message reaches the node 1, 1 removes its forwarding entry for *D*. In (e), after node 2 receives the data packet and forwards to its descendant node 4, it will send a *RouteReply* back to repair the path. As shown in (f), a unicast path from 1 to 4 is repaired.

In (g), node 7 moves away, so node 2 has to find another new ancestor, which is 6, so the unicast path breaks again at 7. Situation is much more complicated. Note that this time, 1, 5 and 6 are all holding forwarding entries for *D* with wrong next hops. We will show how these wrong entries are removed or updated.

Similarly, when data reach node 7, 7 cannot relay it to 2 and doesn't have any other neighbor to forward the packet, so only a *RouteRequest* is sent to 1, which removes the forwarding entry on receiving it. In (h), another data packet reaches 1, and for 1 still has no forwarding entry, it floods the data to 7 and sends a *RouteRequest* to clear the forwarding entry in 5. Because these two data packet can never reach 4, they are lost.

In (i), when another data packet arrives, 5 will send *RouteRequest* to clear the forwarding entries of 3 and 6, and flood the data. This time, the data will reach node 2, and the destination 4 finally. In (j), similar to previous situations, node 2 sends back *RouteReply* to repair the path.

#### 2.4.2.2 Some Discussions

Some advantages of our route maintenance algorithm are listed below:

1. This solution builds the unicast path in an on-demand way, eliminating extra control overhead.
2. Since our protocol can detect the invalid unicast path and actively repair it, the timeout mechanism for forwarding entry becomes not very much necessary. We could even set the expiring time for a forwarding entry to an infinite value.
3. No information is exchanged periodically to maintain the path. If the spanning tree topology remains unchanged, no protocol message is needed to repair the path.
4. We try to restrict the forwarding of *RouteReply* message to prevent it from being flooded to the whole spanning tree unnecessarily.
5. We don't assume any pattern for the unicast data stream. It can vary from casual packet exchange, to heavy load traffic.

#### 2.4.3 Forwarding

When a node knows that a unicast path is passing through itself, it will forward the unicast data packet along the path, otherwise it will forward it like a multicast packet, that is, it will send it to all neighbors other than the one it received the packet from.

## 3. Detailed Protocol Description

### 3.1 Basic Elements

#### 3.1.1 SPT ID

Every node has a unique *ID*, which is a positive integer number. It can be a value set in the configuration file, a random number or derived from the IP address.

#### 3.1.2 Physical Address

Node's *Physical Address* is used to transmit protocol messages and data packets between nodes using underlying unicast channel. In our implementation, it's actually an address pair containing a physical address to transmit protocol message, and another one to forward data packets. A unicast physical address is in the form of "*IPAddress/Port*". The transfer protocol is UDP or TCP.

When a protocol message or data packet is transferred using underlying broadcasting channel, a multicast physical address is used to send the packet. This multicast physical address is not exchanged between nodes because it's well known by all nodes. The form of this multicast physical address is like "*MulticastIPAddress/Port*". The transfer protocol is UDP Multicast.

#### 3.1.3 Neighbors

Two nodes are neighbors if and only if there's a tree edge between them, which means for nodes *N1* and *N2*, *N1.Ancessor*=*N2* or *N2.Ancessor*=*N1*.

Nodes in a node's transmission range are called this node's adjacent nodes.

### 3.2 Tables

#### 3.2.1 Tree Information Table

*Tree Information Table* contains a single entry. The contents of the table are shown below:

Self ID	Physical Address	Core ID	Ancessor ID	Cost	Path Metric	Sequence Number
---------	------------------	---------	-------------	------	-------------	-----------------

**Table 1:** Tree Information Table

*Self ID:* The ID of this node

*Physical Address:* The physical address of this node. Physical address is used to forward data packets.

*Core ID:* The ID of the core.

*Ancessor ID:* The ID of the ancestor. If this node is core, *Ancessor ID*=*Self ID*.

*Cost:* The cost of the path from this node to the core. Metric is the hop count. If this node is core, *Cost*=0.

*Path Metric:* The Path Metric of the path between node and the core.

*Sequence Number:* This value is set to the *Sequence Number* recorded in the newest beacon message received from the ancestor, if the ancestor is not the node itself. Whenever a node whose ancestor is another node sends out a beacon, it put this value into the beacon message.

### 3.2.2 Neighborhood Table

*Neighborhood Table* contains a list of neighborhood entries corresponding to a node's neighbors in a spanning tree.

Neighbor ID	Physical Address	Core ID	Cost	Path Metric	Timestamp	IsAncestor?
...	...	...	...	...	...	...

**Table 2:** Neighborhood Table

*Neighbor ID:* The ID of the neighboring node. Every entry has a unique *Neighbor ID*.

*Physical Address:* The physical address of the neighboring node. The physical address contains both the unicast addresses for transferring protocol and data packets to this neighbor.

*Core ID:* The ID of the core of this neighboring node.

*Cost:* The neighboring node's cost to the core.

*Path Metric:* The Path Metric of the path between that neighbor and the core.

*Timestamp:* The last time this neighborhood entry has been updated.

*IsAncestor :* Whether this neighboring node is the ancestor. There should be at most one entry with this field set to 'YES'. An entry with *IsAncestor* set to 'YES' is called an 'Ancestor Entry', otherwise it's called 'Descendant Entry'.

### 3.2.3 Backup Ancestor Table

The *Backup Ancestor Table* is used to quickly find an alternative ancestor when the current ancestor has lost contact to the node. The entry in table is similar to that in the *Neighborhood Table*.

Neighbor ID	Physical Address	Core ID	Cost	Path Metric	Timestamp
...	...	...	...		...

**Table 3:** Backup Ancestor Table

This table has a limit of size, and entries in it are in the order of the extent to which they will be favored as an ancestor. An entry in the backup ancestor table is not necessarily in the neighborhood table. Every entry has a unique *Neighbor ID*.

### 3.2.4 Adjacency Table

The *Adjacency Table* is used to record the *Link Quality* values of all adjacent nodes. It stores a list of IDs of nodes it received beacons from recently. Every entry has a unique *Adjacent Node ID*.

Adjacent Node ID	Link Quality	Timestamp
...	...	...

**Table 4:** Adjacency Table

### 3.2.5 Core Table

The *Core Table* is used to solve the count-to-infinity problem. *Core Table* records the newest sequence numbers generated by different core nodes and the last time they have been increased. Every entry has a unique *Core ID*.

Core ID	Sequence Number	Last Change Time(ms)
...	...	...

**Table 5:** Core Table

### 3.2.6 Forwarding Table

*Forwarding Table* in a node functions like a Routing Cache, and is used to forward unicast data packet.

When a node is going to forward a unicast packet, it checks if the destination is the core or a neighbor, in which case it knows where to forward the packet.

If the result is false, it checks the *Forwarding Table*. If an entry is present and the *NextHop* is one of its neighbors, the packet is forwarded to the *NextHop*, otherwise, it will be forwarded to all neighbors other than the one the node received the packet from.

In 3.5.1, an solution to actively build and maintain the forwarding table is presented.

Destination ID	Next Hop	Timestamp
8	4	1065801553401
7	3	1065801553614
...	...	...

**Table 6:** Forwarding Table

*Destination ID:* The ID of the destination. Every entry has a unique *Destination ID*.

*Next Hop:* The ID of the next hop node leading to the destination

*Timestamp:* The last time this entry has been confirmed by a data packet sourced at the node with the *Destination ID*, or by a route reply packet containing that *Destination ID*.

## 3.3 Beacon

### 3.3.1 Beacon Message

Beacon Message is sent by each node to announce its presence and exchange tree information. The information contained in a Beacon Message is listed below:

(*Sender ID*, *Core ID*, *Ancestor ID*, *Cost*, *Sequence Number*, <*Adjacency Table*>)

<*Adjacency Table*>=<*Size*, *ID1*, *LinkQuality1*, *ID2*, *LinkQuality2* ...>

The beacon message contains the information store in the sender's Tree Information Table and Adjacency Table. For the *Sequence Number*, we have the following rule:

1. If  $Node.CoreID == Node.Self\ ID$ , Sequence Number in the beacon message is generated incrementally.
2. Otherwise, use the *Sequence Number* stored in the Tree Information Table.

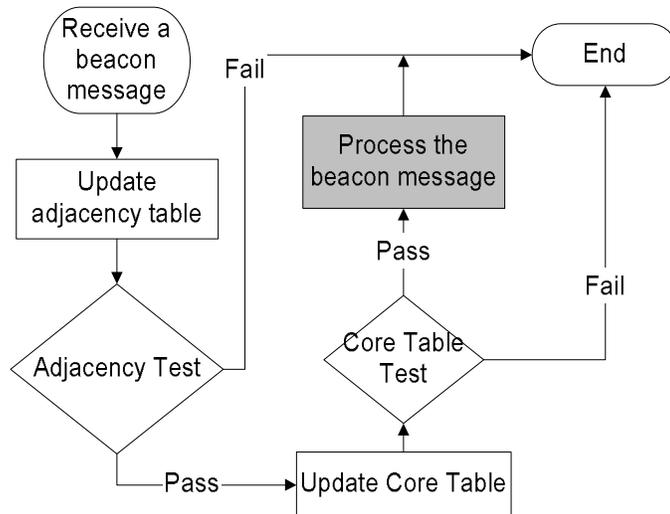
### 3.3.2 Sending Beacon

Beacon Message will be sent by every node in the network periodically, every [*Beacon-Period*] seconds. A beacon message is sent to a local broadcast channel so that every node in the transmission range will receive it.

### 3.3.3 Receiving Beacon

#### 3.3.3.1 Overall Operation

The overall operation diagram is shown in Figure 9:



**Figure 9:** Overall Processing of Beacon Message

#### 3.3.3.2 Adjacency and Reliability Test

Before a beacon message can be processed, the node uses it to update the link quality between node and beacon sender. That is, the node records how many beacons have been received during the last [*Ping-Buf-Size*] beacon periods. The *BiLQ* value of the beacon sender is computed using this *LinkQuality* and the information stored in the adjacency table in the beacon message.

After updating the *BiLQ* of the link, the node checks if the link is reliable ( $BiLQ \geq [Reliable-Threshold]$ ). If the result is YES, this node confirms that the beacon sender can also receive beacons from it, so there is a symmetric reliable link between them. Otherwise, there may exist an asymmetric or unreliable link.

When the testing result is NO, this beacon message will not be processed. However, no matter what the result is, the node will always use the ID of beacon sender to update its Adjacency Table.

### 3.3.3.3 Core Table Test

When a node receives a beacon message, it will try to find the matching entry in the Core Table using Core ID of the message as the index:

1. If no entry is found, the message is processed as usual and an entry is added to Core Table using the sequence number in the message.
2. If an entry is found, and the sequence number of the message is a new one, the message is processed as normal and the entry is updated using the sequence number in the message.
3. If an entry is found, and the sequence number of the message is the same one, don't update the entry. If the last updating time is not beyond [*Max-Message-Age*] ago, this message is processed as usual; otherwise it's dropped because the core information contained in this message is too stale.

### 3.3.3.4 Deciding Ancestor

After passing the adjacency test and core table test, a beacon message can be processed. A node will use the beacon message to decide whether the beacon sender should become its ancestor. The algorithm is listed below (*M* denotes the beacon message):

1. If  $M.CoreID < Node.CoreID$ , return TRUE.
2. If  $M.CoreID > Node.CoreID$ , return FALSE
3. If  $M.Cost \geq (Node.Cost + 2)$ , return FALSE
4. Computes the *NewPathMetric* assume that the beacon sender becomes the new ancestor. The way to compute the *NewPathMetric* value depends on the topology algorithm being used.
  - a) Minimum hop count algorithm:  $NewPathMetric = - M.Cost$
  - b) Link quality algorithm:  $NewPathMetric = BiLQ(M.SenderID)$
  - c) Path quality algorithm:  $NewPathMetric = BiLQ(M.SenderID) * M.PathMetric$
5.  $NewPathMetric \geq NodePathMetric + [Jump-Threshold]$ , return TRUE.
6. Otherwise, return FALSE;

A result of 'TRUE' indicates that the beacon sender should be the new ancestor of this node.

### 3.3.3.5 Maintaining Tree Information and Neighborhood Table

After getting the result to decide whether the beacon sender should be the new ancestor, we do some further steps to update the node's data structures:

1. If result==TURE, update the tree information and neighborhood table, and return:
  - a)  $Node.CoreID = M.CoreID$
  - b)  $Node.AncessorID = M.SenderID$
  - c)  $Node.Cost = M.Cost + 1$
  - d)  $Node.SequenceNumber = M.SequenceNumber$
  - e)  $Node.PathMetric = NewPathMetric$

- f) If there's a descendant entry in the table with  $NeighborID == M.SenderID$ , remove this entry.
- g) Remove the old ancestor entry (the one set  $IsAncestor$  to 'YES') from the neighborhood table, and add a new entry containing this new ancestor into the table; if old ancestor ID equals to new ancestor ID, only update the ancestor entry instead of removing it.
2. Otherwise, if  $M.SenderID == Node.AncestorID$ , update the tree information and neighborhood table:
  - a) If  $M.CoreID \geq Node.SelfID$ , reset the tree information, and return:
    - i.  $Node.CoreID = Node.SelfID$
    - ii.  $Node.Ancestor = Node.SelfID$
    - iii.  $Node.Cost = 0$
    - iv.  $Node.NewPathMetric = MaximumPathMetricValue$
    - v. Remove the ancestor entry from the neighborhood table.
  - b) Otherwise, update the tree information and ancestor entry in neighborhood table, and return:
    - i.  $Node.CoreID = M.CoreID$
    - ii.  $Node.AncestorID = M.SenderID$
    - iii.  $Node.Cost = M.Cost + 1$
    - iv.  $Node.PathMetric = NewPathMetric$
    - v.  $Node.SequenceNumber = M.SequenceNumber$
    - vi. Update the ancestor entry with this message.
3. Otherwise, if  $M.AncestorID == Node.SelfID$ , update the neighborhood table by adding or updating the corresponding descendant entry, and return.
4. Otherwise, if there exists a descendant entry in the neighborhood table, remove that entry.

### 3.3.4 Timeout Mechanism

#### 3.3.4.1 Timing out Adjacency Entries

After an adjacency entry has not been updated for more than [*Adjacency-Timeout*] seconds, it will be removed from the adjacency table of node.

#### 3.3.4.2 Timing out Core Entries

After a node has not received any beacon message carrying new *Sequence Number* for some *Core ID* for more than [*Core-Timeout*] seconds, the corresponding entry in Core Table will be removed.

#### 3.3.4.3 Timing out Neighborhood Entries

After a neighborhood entry has not been updated for more than [*Neighbor-Timeout*], it is removed from the neighborhood table.

If the neighborhood entry to be removed is an ancestor entry, tree information is reset:

1.  $Node.CoreID = Node.SelfID$
2.  $Node.Ancestor = Node.SelfID$
3.  $Node.Cost = 0$
4.  $Node.PathMetric = MaximumPathMetricValue$

After the tree information is *Reset*, the node thinks that Core is itself, as like it just begins to join the network.

### 3.3.5 Backup Ancestor

When the link between a node and its ancestor is broken, the timeout mechanism will detect the broken link and reset the tree information as like the node doesn't have an ancestor. The node will have chance to re-choose a new ancestor when it gets beacon messages from other adjacent nodes some time later. To eliminate the time gap between the time the node loses contact with its ancestor and the time the node finds another appropriate ancestor, a backup ancestor table is used.

#### 3.3.5.1 Updating Backup Ancestor Table

When the sender of a beacon message is decided by 3.3.3.4 and 3.3.3.5 to be neither the ancestor nor a descendant of the node, the message is used to update the backup ancestor table.

The entries in backup ancestor table are in the descending order of the extent they is favored by the node as an ancestor.

Different entries will not have duplicate different *Node ID*.

Nodes believed to be either the ancestor or a descendant of this node will be removed from the backup ancestor table.

Entries that have not been updated for more than [*Neighbor-Timeout*] will be removed from the backup ancestor table

#### 3.3.5.2 Finding Backup Ancestor

Every time before a node wants to reset its tree information table, that is, set itself as the core, it will check the backup ancestor table first to find whether it can find an alternative ancestor. The first entry in the backup ancestor table will be selected.

If there is no backup ancestor can be found, the node will reset the tree information table as usual.

## 3.4 Joining and Leaving

### 3.4.1 Joining

When a node wants to join a spanning tree network, it starts to do the following steps:

1. Reset the tree information by setting:
  - a.  $Node.CoreID = Node.SelfID$
  - b.  $Node.AncestorID = Node.SelfID$
  - c.  $Node.Cost = Node.Cost$
  - d.  $Node.PathMetric = MaximumPathMetricValue$ . The *MaximumPathMetricValue* depends on which algorithm to use:
    - i) Minimum hop count algorithm:  $MaximumPathMetricValue = 0$
    - ii) Link quality algorithm:  $MaximumPathMetricValue = 1$
    - iii) Path quality algorithm:  $MaximumPathMetricValue = 1$
2. Clear the Adjacency Table, Core Table, and Neighborhood Table.
3. Start to periodically send out beacon messages.

### 3.4.2 Leaving

When a node wants to leave the network, it simply stops sending and receiving beacon messages. To help the neighboring nodes to learn about the leaving of the node as soon as possible, besides the timeout mechanism, a *Goodbye* message is broadcasted by the leaving node to all nodes in its transmission range.

The *Goodbye* message only contains the ID of the leaving node. Any node receiving a *Goodbye* message will remove the sender from its neighborhood table.

For a node leaving the network because of unexpected failure, a *Goodbye* may not have chance to be sent out. In this case, timeout mechanism will help to remove this node from the neighborhood table of other nodes.

## 3.5 Multicast Data Packet Forwarding

### 3.5.1 Packet Header

The header of the multicast data packet should contain following fields:

(*Source ID*, *Route Record*)

The *Route Record* field is a list of node IDs the packet has recently gone through in the order of time they are put in the list. The size of the *Route Record* should be at least 1, and limited by a global parameter. The last element of *Route Record* should be always the previous hop of the packet. If underlying broadcast channel is used, the maximum size should be at least 2.

### 3.5.2 Forwarding Decision

When a node receives a multicast data packet, it does the forwarding decision by the following rules:

1. If the last node in the *Route Record* (previous hop) is not a neighbor of this node, the packet is dropped.
2. Otherwise if *Node.SelfID* is in the *Route Record*, drop the packet.
3. Otherwise, deliver this packet and forward it by doing:
  - a) Insert *Node.SelfID* into the last position of *Route Record*, and remove the first element if the record is already full.
  - b) If an underlying unicast channel is used, send message to all nodes listed in the neighborhood table other than the *Previous Hop*, by using the *Physical Address* stored in the neighborhood entry.
  - c) Otherwise Send out the message only once by local broadcasting.

## 3.6 Unicast Data Packet Forwarding

### 3.6.1 Forwarding Table

#### 3.6.1.1 Updating Forwarding Table

The entries in the forwarding table will be added or updated next hop field in the on-demand route maintenance scheme described in 3.6.2. Whenever an entry is added or updated, the time stamp field is updated to current time.

A forwarding entry is invalid when the next hop is no longer a neighbor. The node checks if an entry is valid when it wants to use the entry to forward a unicast data packet. If it finds the entry no longer valid, it removes it from the forwarding table, otherwise, the time stamp of the entry is updated to current time.

A forwarding entry will become invalid and be removed when the next hop node is removed from the neighborhood table, or the destination becomes one of the node's neighbors.

### 3.6.1.2 Timing out Forwarding Entries

A forwarding entry will be removed from *Forwarding Table* if it has not been updated for [*Route-Cache-Timeout*] seconds. The [*Route-Cache-Timeout*] could be a very large value, or even infinity, which means a forwarding entry will never be removed because it is too old. Since we have a mechanism to detect and correct invalid routing cache, we don't very much rely on the timeout mechanism to do the same work. On the other hand, because the time stamp of the forwarding entry is updated whenever a data packet is forwarded to the next hop, data traffic will help to prevent the corresponding forwarding entry from expiring.

## 3.6.2 On-Demand Route Maintenance

### 3.6.2.1 Forwarding Process

The header of the unicast data packet should contain following fields:

(*Source ID, Destination ID, Route Record*)

The *Route Record* field is a list of node IDs the packet has recently gone through in the order of time they are put in the list. The size of the *Route Record* should be at least 1, and limited by a global parameter. The last element of *Route Record* should be always the previous hop of the packet. If underlying broadcast channel is used, the maximum size should be at least 2.

On receiving a unicast packet with source *S* and destination *D*, a node do the following steps:

1. If it's *D* itself, deliver the packet.
2. Otherwise, if *D* is the core, forward the packet to its ancestor.
3. Otherwise, if *D* is one of its neighbors, forward the packet to *D*.
4. Otherwise, if there is a forwarding entry *E*, and  $E.DestinationID == D$ , and  $E.NextHop$  is one of the node's neighbors, forward the packet to  $E.NextHop$ .
5. Otherwise, send out a *RouteRequest* and forward the packet to all other neighbors except the previous hop. If there's a forwarding entry  $E.DestinationID == D$ , remove that entry.

For 1., 2., 3., or 4., we say the node know where to forward this packet. In 5., we say the node doesn't know where to forward this packet.

### 3.6.2.2 Sending Route Request

In **3.6.2.1**, if the node doesn't know where to forward the packet (5.), it will send a *RouteRequest* message to all neighbors using broadcasting channel, including the previous hop of the packet. The *RouteRequest* message contains the following information:

(*Sender*, *PathDestination*)

*Sender*: the sender of this message.

*PathDestination*: the destination of the unicast path, for which a route is requested for.

While sending the *RouteRequest* message, the node will add an entry containing the *PathDestination* and current time into the *Route Request History Table*.

If the destination of packet is core node, no *RouteRequest* is ever sent.

### 3.6.2.3 Receiving Route Request

The actions taken by a node on receiving a *RouteRequest* message *RReq* are listed below:

1. If *RReq.Sender* is not one of the neighbors, do nothing but return.

3. If *RReq.PathDestination*==*Node.CoreID*, do nothing but return.

3. Otherwise, if there's a forwarding entry *FE*, and *FE.Destination*==*RReq.PathDestination*, and *FE.NextHop*==*RReq.Sender*, removes *FE* from the forwarding table.

4. Otherwise, if *Node.SelfID*==*RReq.PathDestination*, or there's a neighbor entry *NE* with *NE.NeighborID*==*RReq.PathDestination*, or there's a valid forwarding entry *FE* with *FE.Destination*==*RReq.PathDestination*, and *FE.NextHop*!=*RReq.Sender*, sends out a *RouteReply* message (**3.6.2.4**).

### 3.6.2.4 Route Reply

*RouteReply* Message is sent by an intermediate node or the destination to announce its having a route to some destination. Neighboring nodes will rely on this message to update their forwarding table, so as to establish or maintain the unicast path to that destination. A *RouteReply* message contains the following information:

(*Sender*, *NextHop*, *PathDestination*)

*Sender*: the sender of this message.

*NextHop*: the next hop node from the view of the message sender. If the sender itself is the destination, this field is set to its ID; if the destination is one of the sender's neighbors, this field is set to the destination's ID; if the sender is having a valid forwarding entry for that destination, this field is set to the next hop field of that entry.

*PathDestination*: the destination of the unicast path.

A *RouteReply* message is triggered when a node receive *RouteRequest* message for some destination and happens to know where the destination is (**3.6.2.3**, 3.).

On receiving a *RouteReply* message *RR*, a node will do the following steps:

1. If *RR.Sender* is not a neighbor, do nothing but return.

2. Otherwise, if *RR.NextHop*==*Node.SelfID*, do nothing but return.

3. Otherwise, if *Node.SelfID*==*RR.PathDestination*, do nothing but return.

4. Otherwise, if *RR.PathDestination* is one of this node's neighbors, do nothing but return.
5. Otherwise, if there is no forwarding entry *FE* with *FE.Destination==RR.PathDestination*, add an forwarding entry *FE* to the forwarding table, and set *FE.Destination=RR.PathDestination*, and *FE.NextHop=RR.Sender*, and *FE.Timestamp=current time*.
6. Otherwise, if there a forwarding entry *FE* with *FE.Destination==RR.PathDestination* and *FE.NextHop!=RR.Sender*, set *FE.NextHop=RR.Sender*, and *FE.Timestamp=current time*.
7. Otherwise, if there a forwarding entry *FE* with *FE.Destination==RR.PathDestination* and *FE.NextHop==RR.Sender*, set *FE.Timestamp=current time*.
8. If either step **5.** or **6.** is executed, set *RR.NextHop=RR.Sender*, *RR.Sender=Node.SelfID*, and forward *RR* to all neighbors using broadcasting channel.

## 4. Settings

At the end of the description of each setting, the corresponding configuration attribute is listed.

### 4.1 Timers for Maintaining Spanning Tree

**[Beacon-Period]** Default = 1 second. This is the time period between two consecutive beacon messages sent by a node. (/Public/Node/SPT/BeaconTime)

**[Neighbor-Timeout]** Default = 5 seconds. A neighborhood entry will be removed if it has not been updated for this specified timeout value. (/Public/Node/SPT/NeighborTimeout)

**[Adjacency-Timeout]** Default = 20 seconds. An adjacency entry will be removed if it has not been updated for this specified timeout value. (/Public/Node/SPT/Adjacency-Timeout)

**[Core-Timeout]** Default = 20 seconds. A core table entry will be removed if the node has not received any beacon message carrying new *Sequence Number* for the *Core ID* of this entry for more than this specified timeout value. (/Public/Node/SPT/CoreHistoryTimeout)

**[Max-Message-Age]** Default = 5 seconds. If the *Sequence Number* from some *Core ID* has not changed for more than this specified time value, any beacon message carrying the same *Core ID* and *Sequence Number* will not be said to be too old and not be processed. (/Public/Node/SPT/MsgMaxAge)

**[Reliable-Threshold]** Default = 0.1. The threshold value that a link quality of a link must achieve to become a spanning tree edge. (/Public/Node/SPT/LinkQuality/ReliableThreshold)

**[Jump-Threshold]** Default = 1. A new ancestor candidate must improve the metric by at least this threshold value. (/Public/Node/SPT/JumpThreshold)

**[Ping-Buf-Size]** Default = 10. This is the size of the buffer the node uses to record the beacon recently received from an adjacent node. (/Public/Node/SPT/LinkQuality/PingBufSize)

## 4.2 Timers for Maintaining Unicast Path

[*Route-Cache-Timeout*] Default=20s. This timer is used to timeout an old forwarding entry which is believed to be probably invalid. It can be set to *Infinity*, which means that a forwarding entry can never be removed because of being too old. The reason about why we don't have to timeout a forwarding entry is that, we have another mechanism to detect and correct invalid route cache in an on-demand way (3.5.2).

(/Public/Node/SPT/RouteCacheTimeout)

## 4.3 SPT Logical Address Setting

There are three different way to decide the logical address of a SPT node, depending on how the configuration XML file is composed:

- 1) Use a specified SPT logical address

```
<SPT>
  <Coords>
    <FIXED>
      <coordinate>XX</coordinate>
    </FIXED>
  </Coords>
</SPT>
```

- 2) Use a random integer number smaller than the <base> as the logical address

```
<SPT>
  <Coords>
    <RANDOM>
      <base>XXX</base>
    </RANDOM>
  </Coords>
</SPT>
```

- 3) Use the IP address to generate a SPT logical address. The current system have to use TCP/IP to support this setting

```
<SPT>
  <Coords>
    <USE_IP/>
  </Coords>
</SPT>
```

## 5. Message Formats

This section list the detailed message formats used in the Spanning Tree Protocol, including the protocol messages and data messages. Since the SPT protocol is implemented as a part of the Hypercast software, the message headers listed here are all Hypercast-related.

## 5.1 Protocol Message Formats

The common format for all protocol messages is shown in Figure 10.



**Figure 10:** Protocol Message Format

The types of the SPT protocol message are:

Message Type	Type Field
Beacon	0
Goodbye	1
RouteRequest	2
RouteReply	3

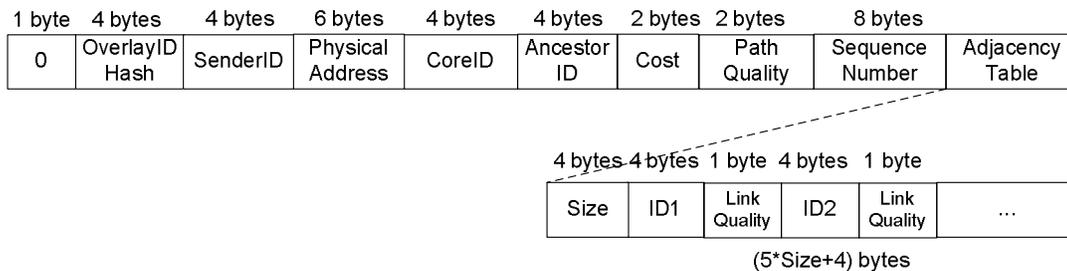
**Table 8:** Protocol Message Types

The *OverlayIDHash* is a 4-byte long hash value which is derived from the OverlayID. It's used to identify whether this protocol message is coming from a node inside the same overlay network. If no, the message will be dropped.

### 5.1.1 Beacon Message

*Beacon* message is sent by each node periodically to announce its existence and exchange tree information with adjacent nodes.

A *Beacon* Message is sent by using underlying broadcasting channel.



**Figure 11:** Beacon Message Format

*SenderID*: The ID of the sender of this beacon message.

*PhysicalAddress*: The physical address of the sender.

*CoreID*: The Core ID of the sender.

*AncestorID*: The ID of the sender's ancestor.

*Cost*: The path cost from the sender to the core.

*SequenceNumber*: The newest sequence number generated by the core and received by the sender.

*Adjacency Table:* This table copies the IDs listed in the beacon sender’s adjacency table. So the length of a beacon message is variable, depending on the size of its adjacency table. A node’s ID is in the adjacency table of a beacon message means that the beacon sender has received beacon message from this node recently.

### 5.1.2 Goodbye Message

The *Goodbye* Message is sent by a leaving node to inform its neighbors. Any node receiving a *Goodbye* message will remove the message sender from its neighborhood table immediately.

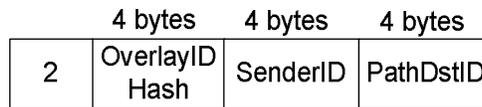
A *Goodbye* Message is sent by using underlying broadcasting channel.



**Figure12:** Goodbye Message Format

### 5.1.3 Route Request Message

A *RouteRequest* Message is sent to all neighbors when a node doesn’t know where to forward a unicast data packet and has to flood it ahead. The previous hop of this data packet will use this message to clear the forwarding entry, while other neighbor will be triggered to send *RouteReply* message by this message if it knows where the destination is.



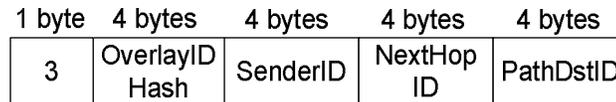
**Figure 13:** RouteRequest Message

*SenderID:* The ID of the sender of this message.

*PathDstID:* The ID of the destination, to which a route is requested.

### 5.1.4 Route Reply Message

A *RouteReply* Message is used to actively build or repair a unicast path when some node sees a *RouteRequest* coming in for some destination, for which it knows where to forward the unicast data.



**Figure 14:** Route Reply Message

*SenderID:* The ID of the sender of this message.

*NextHopID:* The next hop for the unicast destination from the view of the message sender. If the sender is the destination itself, this field will be set to the ID of itself. A

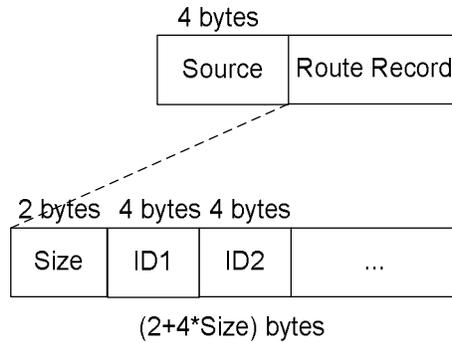
node receiving a *RouteReply* message with *NextHopID* set to it will simply ignore the message.

*PathDstID*: The ID of the destination of the unicast path to be built.

## 5.2 Data Message Formats

Here we present the header for the data packet forwarded in a spanning tree network. We only list the fields which are related to our protocol. In Hypercast software, a data packet header has a lot of common fields shared by different protocols, which are not listed here.

### 5.2.1 Multicast Packet Header



**Figure 15:** Multicast Data Packet Header

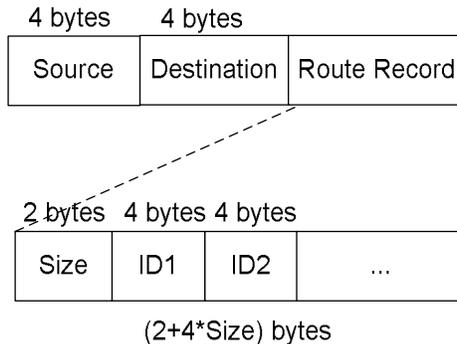
*Source*: The ID of the multicast source node.

*Route Record*: The list of nodes the packet has recently gone through. The *Size* of route record is limited by a maximum value, which is a parameter. The strategy of replacing when new node ID comes in and list is full, is FIFO.

A node receiving a multicast packet will drop it if it finds that:

1. The *Previous Hop* is not its neighbor.
2. Or, its node ID is in the *Route Record* of the packet.

### 5.2.2 Unicast Packet Header



**Figure 17: Unicast Data Packet Header**

*Source:* The ID of the unicast source.

*Destination:* The ID of the unicast destination.

*Route Record:* The list of nodes the packet has recently gone through. The *Size* of route record is limited by a maximum value, which is a parameter. The strategy of replacing when new node ID comes in and list is full, is FIFO.

A unicast packet can be forwarded in underlying broadcasting channel only when the forwarding node doesn't know how to forward it, and trying to forward it to all other neighbors except the previous hop.

A node receiving a unicast packet will drop it if it finds that:

1. The *Previous Hop* is not its neighbor.
2. Or, its node ID is in the *Route Record* of the packet.

## 6. Interaction with Overlay Socket

### 6.1 Interface Functions

In Hypercast framework, the spanning tree protocol is run in the protocol node, and it's the overlay socket that does the forwarding of data packets. On receiving a data packet from a neighbor, the overlay socket will query the protocol node about where to forward the packet by some specified interface functions, and do the forwarding accordingly.

So the Route Record scheme we have mentioned before is implemented in the overlay socket, rather in the protocol node. And most of the data forwarding steps are also in the overlay socket.

The interface functions related to data packet forwarding between the overlay socket and protocol node are listed below:

Functions	Comments
getParent(Dst)	<p>1. The overlay socket queries the node by this function to ask for the next hop to forward a unicast packet.</p> <p>2. The node can also use this function to learn when the socket needs to forward a unicast packet and what the destination is, and this the only information the node need to know for the unicast scheme.</p> <p>3. If the node knows where to forward the packet to destination, it simply returns the next hop; otherwise it returns a NULL value.</p> <p>4. If the overlay socket gets a NULL value</p>

	from the node, it will try to forward the packet to all other neighbors except the previous hop.
getChildren(Src)	<ol style="list-style-type: none"> <li>1. The overlay socket queries the node by this function to ask for a list of next hop neighbors to forward a multicast packet.</li> <li>2. The node can also use this function to learn when the socket needs to forward a multicast packet and what the multicast source is.</li> <li>3. The spanning tree protocol node always returns a NULL list here.</li> <li>4. Since the overlay socket always gets a NULL value by this function, it will try to forward the packet to all other neighbors except the previous hop.</li> </ol>
getAllNeighbors()	The overlay socket queries the node by this function to ask for a list of all neighbors.

**Table 9:** Interface Functions between Overlay Socket and Protocol Node

## 6.2 Operations of Forwarding Engine

The ways how the forwarding engine in OL\_Socket operates to forward data are listed as pseudo codes:

```

MulticastPacketReceived(p){
    If (p didn't come from a neighbor)return;
    If (my address is in the route record of p) return;
    Children_list=node.getChildren(p.Src);

    If(Children_list==null)
        Children_list=node.getAllNeighbors()-p.PrevHop;

    Add my address to the route record of p
    Forward p to each neighbor in the Children_list;

    Deliver p to the application;
}
UnicastPacketReceived(p){
    If (p didn't come from a neighbor) return;
    If (my address is in the route record of p) return;
    If (I am the destination){

```

```

        Deliver  $p$  to the application;
        return;
    }
    Nexthop=node.getParent( $p.Dst$ );
    If(Nexthop!=null){
        Add my address to the route record of  $p$ ;
        Forward  $p$  to Nexthop;
    }else{
        Nexthop_list=node.getAllNeighbors()- $p.PrevHop$ ;
        Add my address to the route record of  $p$ 
        Forward  $p$  to each neighbor in the Nexthop_list;
    }
}
}

```

## 7. States and State Transitions

### 7.1 State Definitions

State Name	State Definition
<i>Stopped</i>	The node is not running
<i>Core</i>	The node is the core of the spanning tree
<i>Not Core</i>	Te node is not core of the spanning tree

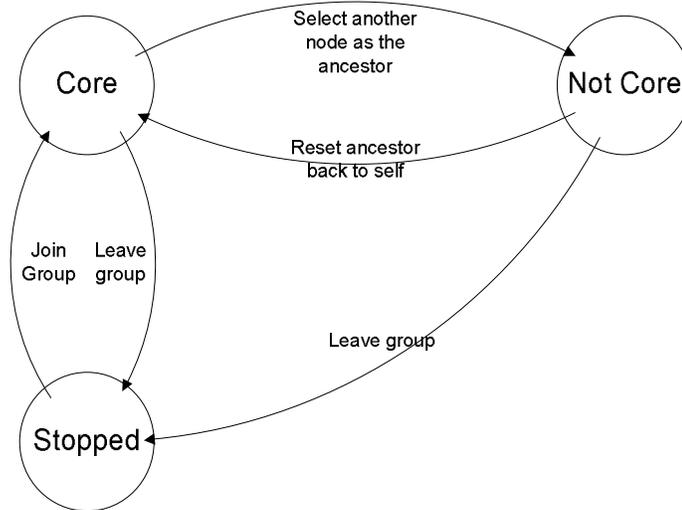
Table 10: Node States Definitions

*Stopped*: Nodes in this state do nothing. Before a node transits from another state to *Stopped* state, a *Goodbye* message will be sent out.

*Core*: After a node starts to join the network, it enters in this state. A node in *IsCore* state will periodically send out beacon message carrying an incremental sequence number.

*NotCore*: After a node in *IsCore* state receives a valid beacon message containing a Core ID smaller than its ID, it enters the *No Core* state. In this state, a node will periodically send out beacon message carrying the sequence number it got from its ancestor.

### 7.2 State Transition Diagram



**Figure 19:** Transition Diagram of Node States

### 7.3 Actions of SPT Protocol

The following tables summarize the actions taken by nodes when events like message arrival, timer expiration happen.

Please note that, when we say “insert an entry into a table”, we imply that if an entry already exists, we only update the information carried by that entry. Similarly, “remove an entry” implies that an entry will be removed only when it already exists in the table.

#### State: **Stopped**

Event	Action
Join Group	Reset tree information table, set ancestor to myself → <b>Core</b>

#### State: **Core**

Event	Action
Beacon message $m$ received from $s$	Update adjacency and reliability IF $m$ doesn't pass adjacency and reliability test <b>return</b> Update core table IF $m$ doesn't pass core table test <b>return</b> IF $s$ is a better ancestor Set ancestor to $s$ → <b>Not Core</b> <b>return</b> ELSE

	IF $s$ is my descendant Insert $s$ into neighborhood table ELSE Remove $s$ from neighborhood table
Goodbye message received from $s$	IF $s$ is one of the neighbors Remove $s$ from neighborhood table
Beacon Timer expires	Check neighborhood table, adjacency table, core table, backup ancestor table and forwarding table, to remove expiring or invalid entries.  Broadcast Beacon message $m$ to all adjacent nodes. The sequence number in $m$ is generated incrementally.

**State: Not Core**

Event	Action
Beacon message $m$ received from $s$	Update adjacency and reliability IF $m$ doesn't pass adjacency and reliability test <b>return</b> Update core table IF $m$ doesn't pass core table test <b>return</b> IF $s$ is a better ancestor Set ancestor to $s$ update tree information table remove old ancestor entry, insert $s$ as the ancestor entry into neighborhood table <b>return</b> ELSE IF $s$ is current ancestor IF $m$ carries a Core ID larger than my ID IF a backup ancestor is found Set ancestor to this backup ancestor ELSE Reset tree information table, set ancestor to myself <b>→ Core</b> ELSE Update ancestor information with $m$ ELSE IF $s$ is my descendant Insert $s$ into neighborhood table ELSE Remove $s$ from neighborhood table Insert $s$ into backup ancestor table
Goodbye message received from $s$	IF $s$ is one of the neighbors Remove $s$ from neighborhood table IF $s$ is current ancestor

	<p>IF a backup ancestor is found  Set ancestor to this backup ancestor  ELSE  Reset tree information table, set ancestor to myself  → <b>Core</b></p>
Beacon Timer expires	<p>Check neighborhood table, adjacency table, core table, backup ancestor table and forwarding table, to remove expiring or invalid entries.</p> <p>IF the ancestor entry is removed  IF a backup ancestor is found  Set ancestor to this backup ancestor  ELSE  Reset tree information table, set ancestor to myself  → <b>Core</b></p> <p>Broadcast Beacon message <math>m</math> to all adjacent nodes.  The sequence number in <math>m</math> uses the sequence number stored in tree information table</p>

**State: Core, Not Core**

Event	Action
Leave Group	<p>Broadcast Goodbye to all neighbors  → <b>Stopped</b></p>
Multicast data packet $d$ received	<p>IF myself is in the route record of <math>d</math>  <b>return</b>  IF <math>d</math> was not sent by one of my neighbors  <b>return</b></p> <p>Insert myself into the route record of <math>d</math>, while maintaining the maximum size of route record.</p> <p>IF using underlying unicast channel  Forward <math>d</math> to every neighbors except the previous hop  ELSE  IF has more than one neighbor  Broadcast <math>d</math> to all adjacent nodes</p>
Unicast data packet $d$ received	<p>IF myself is in the route record of <math>d</math>  <b>return</b></p> <p>IF <math>d</math> was not sent by one of my neighbors  IF I am the destination  Deliver <math>d</math>  <b>return</b></p>

	<p>IF I am the destination  Deliver <math>d</math>  <b>return</b></p> <p>Insert myself into the route record of <math>d</math>, while maintaining the maximum size of route record.</p> <p>IF destination is core  Forward <math>d</math> to my ancestor  <b>return</b></p> <p>IF destination is one of my neighbors  Forward <math>d</math> to destination node  <b>return</b></p> <p>IF can find the next hop for the destination in forwarding table  IF the next hop node is one of my neighbors  Update the timestamp of the entry  Forward <math>d</math> to the next hop node  <b>return</b></p> <p>ELSE  Remove the invalid forwarding entry</p> <p>Send RouteRequest message for the destination to all neighbors  Forward <math>d</math> to all neighbors except the previous hop</p>
RouteRequest message $rreq$ received from $s$	<p>IF the sender is not a neighbor  <b>return</b></p> <p>IF <math>rreq</math>'s path destination is core node  <b>return</b></p> <p>IF there is a forwarding entry for the path destination, and the next hop is equal to the sender of <math>rreq</math>  Remove the forwarding entry  <b>return</b></p> <p>IF I am the path destination  Send out RouteReply message  <b>return</b></p> <p>IF path destination is one of my neighbors  Send out RouteReply message  <b>return</b></p> <p>IF there's a valid and updated forwarding entry for the destination  Send out RouteReply message  <b>return</b></p>
RouteReply message $rr$ received from $s$	<p>IF the sender of <math>rr</math> is not one of my neighbors  <b>return</b></p>

	<p>IF <i>rr</i>'s path destination is core  <b>return</b></p> <p>IF <i>rr</i>'s next hop is myself  <b>return</b></p> <p>IF myself is the <i>rr</i>'s path destination  <b>return</b></p> <p>IF the <i>rr</i>'s path destination is one of my neighbors  <b>return</b></p> <p>IF there's no forwarding entry for the destination  Add an entry for that destination  Set the entry's next hop to <i>rr</i>'s sender  Forward <i>rr</i> to all neighbors  <b>return</b></p> <p>IF there's a forwarding entry for the destination  IF the entry's next hop is not <i>rr</i>'s sender  Set the entry's next hop to <i>rr</i>'s sender  Update the entry's timestamp  Forward <i>rr</i> to all neighbors  <b>return</b></p> <p>ELSE  Only update the entry's timestamp</p>