Resource Allocation using Multiple Edge-Sharing Multicast Trees

Abhishek Patil¹, A-H. Esfahanian¹, Li Xiao¹ and Yunhao Liu²

¹Dept of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824
{patilabh,esfahanian,lxiao}@cse.msu.edu

²Dept of Computer Science
Hong Kong Univ of Science and Technology
Kowloon, Hong Kong
liu@cs.ust.hk

Abstract - Implementing multicast in MANETs is a challenging task. A typical multicast network consists of a single tree, in which only a few internal nodes contribute most resources and are involved in performing the multicast functionality. This leads to an un-even utilization of network resources. This problem is more prominent in MANETs where network resources are limited. A possible solution to the problem is to split the multicast content over a number of trees. Multiple trees provide several paths for the multicast content and get more nodes involved in implementing the multicast functionality. However, in such a setup, not all the trees get to use the best weight edges, thus the overall multicast latency increases. This paper presents MEST, a distributed algorithm to construct multiple edge-sharing trees for small group multicast. MEST balances the resource allocation and delay constraints by choosing to overlap certain edges that have low weights. Our simulation results show that MEST is scalable and can generate multicast networks that have low delay and fair resource utilization.

Keywords: Resource Allocation, Graph Theory, Multicast, Mobile Ad Hoc Networks, MANET, Wireless Networks, Simulation, Mobile Networks.

I. INTRODUCTION

Mobile ad hoc networks (MANETs) are characterized by mobile nodes and constantly changing network topology. Implementing multicast in such a dynamic environment is a not an easy task. As pointed out by [17], traditional IP-layer multicast for MANETs has a lot of signaling overhead as it needs to take into account the network dynamics in addition to the (multicast) group dynamics. In recent years, several researchers have proposed the concept of application layer (overlay) multicast. Overlay multicast is gaining popularity due to its easy of implementation and flexibility. Overlay multicast relies on the underlying unicast protocols to adapt to the changing network topology. As a result, the application layer has to track only the group dynamic. Due to its ease of implementation and flexibility to adapt, overlay multicast networks (though not as efficient as IP layer multicast) are finding many practical applications in MANETs. AMRoute [4], PAST-DM [10], SOLONet [19] and LGT [6] are some of the overlay multicast protocols that have been proposed for MANETs. Similar to other multicast networks, overlay multicast also suffers from the resource utilization problem (also referred to as the fairness issue in some literature). This fairness issue results from the fact that in a typical multicast tree, only a small number of nodes and edges are actively involved in implementing the multicast functionality. For example, there are a few internal nodes that perform the task of duplicating and forwarding packets and a large number of leaf nodes that only act as receivers of packets. These leaf nodes do not contribute any resources to the multicast tree. MANETs are characterized by scarce network resources. Due to this uneven distribution, certain nodes will run out of resources (e.g. battery strength) faster than other nodes, leading to bottleneck nodes in the multicast trees. As explained by Wang and Gupta [23], the lifetime of a multicast tree depends on the lifetime of a bottleneck node. Proper resource allocation can help better utilize the available resource and extend the life of an overlay multicast network.

The delay in a multicast tree is equal to the time it takes for all the leaf nodes to receive the packets (data) sent by the source node. There are two main techniques that can help reduce this delay: source eccentricity and tree weight. The first approach tries to reduce the source node’s eccentricity. (The eccentricity of a node v in a connected graph G is length of the longest of all the shortest paths between v and every other point in G). A higher fan-out at interior nodes results in a low depth tree (low source eccentricity), thus having a shorter delay.

However, a tree with higher fan-out at interior nodes would have a large number of leaf nodes. For example, a binary tree has half the nodes as leaf nodes [15]. A tree with an average fan-out of 16 will be shorter (lower delay) but will have 10% internal nodes that perform the multicast functionality. Thus it is necessary to have a well constructed overlay multicast tree which would maintain a good balance between the multicast latency and utilization of network resources. There are many distributed algorithms [3, 5, 11, 16] for generating shortest path trees; however, most of them pay little attention to the network utilization issues.
The other method for reducing delay is by constructing low weight trees. (The weight of a tree is the sum of weights of its edges.) Low weight trees tend to have lower delays as they try to use low weight edges during their construction. In this paper, we concentrate on the second approach to reduce delay: constructing multiple low weight multicast trees.

In this paper, we present MEST—a distributed algorithm for building multiple edge-sharing multicast trees. MEST aims to uniformly distribute the multicast functionality across all the nodes in the group with little change in the multicast delay. Our MEST algorithm is designed to run on top of any distributed algorithm for constructing a minimum spanning tree of a given graph (e.g. [2, 8, 22]). In this paper, however, we describe MEST with respect to the GHS algorithm [8]. An interesting point to note here is that the concept of MEST can be applied to any kind of multicast network. Although this paper describes MEST with respect to overlay multicast, MEST algorithm can be used to improve the resource allocation (as well as keep the delay under control) in any wired multicast network or IP-based multicast in MANETS. MEST constructs several multicast trees in the same network. Each tree constitutes a separate sub-group and all the nodes participating in the multicast subscribe these sub-groups. The multicast content is split into smaller stripes and each stripe is then sent over one of the multicast trees (Figure 1). (Note that we chose to use the word stripes instead of fragments, to avoid confusion with ‘fragment of a graph’). Several papers propose the use of multiple multicast trees for mesh creation in order to have redundant links to improve reliability. Although improving reliability is not MEST’s primary objective, the MEST algorithm can also be used in building a reliable redundant mesh network. In such a network, the multicast content is not split into smaller stripes; instead, each MEST multicast tree will provide an alternative path to the multicast data. If one path fails, the nodes re-route the data over one of the other alternative paths.

We carried out several simulations to examine the scalability and performance of MEST. Overall, our simulation results indicate that MEST is highly scalable and has a lower delay as compared to a single tree multicast. We were also able to determine the best overlap threshold value for a given scenario. The simulation results also show significant reduction in the delay as the number of trees increase. However, we strongly believe that there would be other factors (like file fragmentation overhead) that need close examination in order to check if they offset this performance improvement.

The rest of the paper is structured as follows. Section II looks at previous work done in the area of uniform resource allocation. Section III gives a brief introduction of the GHS algorithm to our readers. Section IV presents detailed description of our MEST algorithm. In Section V we present simulation results for MEST. Finally, Section VI concludes the paper and presents directions for future research.

II. RELATED RESEARCH

Several overlay multicast protocols [4, 6, 10, 19] have been proposed and studied for MANETs in recent past. All of them are based on single tree multicast and are geared to address the efficiency issues of overlay multicast. In recent years, the issue of resource allocation has caught the attention of many researchers [14, 15, 18, 24].

Like MEST, SplitStream algorithm [15] tries to distribute the multicast load by constructing multiple multicast trees. However, SplitStream is based on a complex infrastructure (Pastry [20] and Scribe [21]) and requires the help of non-members to construct multicast trees. MEST on the other hand is much simpler and doesn’t require any help from the non-member nodes, thus making it compatible with any kind of underlying network. Although, SplitStream focuses on improving the resource utilization of the network, it doesn’t pay much attention to the delay constraints. The algorithm attempts to accommodate member nodes with different bandwidth capacities. In doing so, it makes use of the Scribe mechanism to limit the number of outgoing connections from a member node. As a result, a ‘child node’ might be forced to connect further down the tree resulting in higher delay.

Young [24] suggested a k-MST algorithm for building a reliable mesh structure. Their algorithm builds multiple edge-disjoint minimum spanning trees. The algorithm works by removing an edge from the graph once it is used by a tree. This tends to build trees that greatly vary in their weight and hence the multicast latency. During the later stages of such an algorithm, the newer trees end-up having the high weight edges (which were rejected by the earlier trees). The final tree would be the one with the highest delay since it is using all the high weight edges that were not used by the earlier trees. The k-MST algorithm differs from MEST since it tries to build edge disjoint multiple tree and has a primary goal of constructing a mesh network. In the k-MST algorithm, if one tree fails, data can be sent over another redundant tree. However, the multicast delay would increase since the data is being transferred over a tree with a high weight. It should be noted that such a method works well only for a mesh case. It cannot be used for address the resource allocation problem since the delay involved in simultaneous transfer of multicast content would be unacceptable. Some of the content would come over a tree that has the highest weight (maximum delay).
The strategy adopted in [14, 18] are different from ours essentially. They rely on accounting and charging methods to solve the unfairness problem, that is, group members served as a forwarder charges for its service to its children to make up its contribution to the system. Overcast [12] maintains a single tree and uses a dedicated server to optimize the bandwidth utilization across the network.

III. GHS ALGORITHM

Gallagher, Humblet, and Spira (GHS) [8] were one of the first to provide a solution to the problem of constructing a minimum spanning tree in a disturbed manner. The following two sub-sections give a brief description of their algorithm and the message exchange between nodes. Interested readers should refer to the GHS paper [8] for further details.

A. Overview

The GHS protocol works by combining fragments (disconnected components of a graph) along the shortest edge joining them. The protocol maintains a forest of trees, each identified by its fragment id \( \text{fragid} \) which consists of the fragment’s level number and fragment’s core node’s id. At start, all the nodes are at level 0 and every node is an individual fragment. Each fragment attempts to asynchronously find a minimum weight outgoing edge \( \text{mwoe} \) into another fragment. When such an edge is found the two fragments exchange ‘connect’ messages and attempt to combine to form a larger component. As fragments join, the level of the combined component increases by one (if certain conditions are satisfied). There are a few rules that need to be followed in order for this merger (and level increase) to happen:

1. Suppose a fragment \( F_l \) at level \( m > 0 \) encounters a \( \text{connect} \) message from a smaller fragment \( F_s \) at level \( m-p \) \( (p < m) \) at the other side of the \( \text{mwoe} \), the smaller fragment is absorbed as a part of the larger fragment. The level and \( \text{fragid} \) of the combined larger fragment stays unchanged i.e. \( F_l \) at \( m \).

2. If the two fragments are at the same level \( m \) and have the same \( \text{mwoe} \), then they combine to form a larger fragment at level \( m+1 \). The \( \text{mwoe} \) edge which joined the two fragments is absorbed as a part of the larger fragment. The level and \( \text{fragid} \) of the combined larger fragment stays unchanged i.e. \( F_l \) at \( m \).

3. On the other hand, if a fragment at level \( m \) encounters a larger fragment at level \( x > m \) along its \( \text{mwoe} \), it delays its \( \text{connect} \) message until it reaches a level at least equal to \( x \).

The rules for combining fragments ensure that a new fragment at level \( m+1 \) will be formed by the combination of at least two fragments at level \( m \). Thus, a fragment at level \( p \) will contain at least \( 2^p \) nodes. Therefore, \( \log_2 N \) is an upper bound on the fragment levels. The wait in rule 3 is essential since communication latency required to find it \( \text{mwoe} \) is proportional to the fragment size. If this delay is not implemented, it may result in a loop. An example is shown in Figure 2.

When the two fragments combine, the nodes in these fragments (other than the core nodes) are not immediately informed about the identity and level of the new fragment. In this example, let’s say \( F \) and \( F' \) (both at the same level) combine along their \( \text{mwoe} \) (in this case PQ). Now the new fragment is one level higher. The \( \text{fragid} \) has changed to the new \( \text{mwoe} \). Now, \( P \) and \( Q \) know about the new level and \( \text{fragid} \). However, there are several other nodes in each fragment (e.g. A and B) who won’t know about this change until they get some message (e.g. \( \text{initiate} \) message) from the \( P \) or \( Q \). At some point node A gets an initiate message from \( P \) and it updates its level. Node B on the other hand still has the old id and level. If the wait in the third rule is not implemented, node B might send a \( \text{connect} \) message to A thus causing a loop. The wait ensures that B waits till its level increases at least to that of A. When that happens, B realizes that they belong to the same component.

B. GHS Example

As an example of the GHS algorithm, consider Figure 3. At \( t=0 \), all nodes are at level 0. At some later time, nodes A and B merge to form a larger fragment at level 1. Similarly, nodes E and F combine on their \( \text{mwoe} \) to form \( F' \) at level 1. Further ahead, node C and node D get absorbed into \( F \) and \( F' \) respectively. After some more time, the two fragments merge on the edge BF to form a larger fragment \( F'' \) with level 2. At a later point in time, node G gets absorbed into this larger fragment.
IV. MEST ALGORITHM

Our MEST algorithm can run on top of any distributed algorithm for constructing a minimum spanning tree of a given graph (e.g., Async [22], Awerbuch [2], GHS [8]). In this paper, however, we describe MEST with respect to the GHS algorithm. MEST can be considered as several instances of the GHS algorithm running in a sequence one after the other. At start, all the nodes run the GHS algorithm to produce the first tree (T1), then the second run gives tree T2 and so on, till all the desired number of multicast trees have been generated.

A. Preliminaries

In case of multiple multicast trees, the highest weight multicast tree contributes the most delay and so, the delay of the overall system is determined by this (highest-weight) tree. Since edge disjoint multicast trees would use different edges in each tree, ideally, they would do a better job in solving the resource allocation problem. However, if edge disjoint trees are not constructed correctly, they may not be the best choice when it comes to delay constraints. An example is shown in Figure 4. For the given graph, it is possible to find many pairs of edge disjoint spanning trees. Note that each multicast tree has to be a spanning tree since it has to cover all the multicast member nodes. Two such pairs are shown (Figure 4b & 4c). In Figure 4b, T1 is a minimum spanning tree. However, other multicast tree (T2) has a very high latency since it is using all the high weight edges that were rejected by the minimum spanning tree (T1). With the two trees, the network resources are evenly used; however the overall multicast delay would be determined by the slowest amongst the two trees (T2). Unlike Figure 4b, in case of Figure 4c, the light weight edges are equally distributed between the two spanning trees and hence the overall delay is lower compared to the earlier case.

Now consider the case where two trees can share an edge. Figure 4d shows two spanning trees (one of them minimum) that have two common edges (DE and EF). Such a multicast tree may be obtained by having a condition that allows the overlap of edges that satisfy a certain condition. For example, edges having weight below a certain threshold or edges connected to nodes that have very low degree. This threshold should be wisely chosen. A value too high would lead to several overlapping edges, decreasing the network utilization and a value too low would increase the multicast latency. In the example, we choose the cut-off threshold for edge sharing as 5. In Section V, we show the best overlapping threshold for a particular scenario.

Other factors affecting the delay: In case of edge-sharing trees, a factor that might affect the delay is the number of times an edge is re-used. This is important because a shared edge has to carry packets for different multicast groups since it is part of several multicast trees. For example, in case of Figure 4d, the two common edges (DE and EF) have to carry packets for both the trees. A packet would be queued at a node if that node is currently sending or receiving another packet over the common edge. Depending on the queue length, this queuing might result in additional delay. In case of non-streaming applications, a packet would be queued only if an overlapping edge is at the same delay-distance from the source node in multiple trees. In case of streaming applications, this delay would always be present since there is constant stream of data to be sent by the node. In order to eliminate any further delays due to queuing, we suggest setting an upper bound on the number of trees that can share a particular low-weight edge.

Special cases: A bridge is an edge of a graph whose removal will result in a disconnected graph. For graphs that contain bridge edges, it is not possible to have multiple trees that are completely edge disjoint. Figure 5 shows such an example. In the example, edge BG is a bridge edge and will be present in any spanning tree of this graph. In our MEST algorithm, we mark a bridge edge with the ‘S’ tag to indicate that this is a special case edge, which has to be included in all the spanning trees.

B. Basic Operation

Let’s say we want to construct ‘m’ multicast trees (T1, T2, ..., Tm) which may have some overlapping edges. Since each node is running an instance of the GHS algorithm, at t=0 there are ‘m x n’ fragments. Each node maintains a variable called the fragid which stores the fragment id (fragid) of the components that the node is currently associated with. Nodes also maintain separate arrays (of length = ‘m’) for each of its edges.

![Figure 4: Edge disjoint vs edge overlapping trees](image-url)
The ‘m’ elements of the edge array correspond to the ‘m’
trees that the edge can potentially be part of. An element of the
edge array can have one the following values:

‘B’ = Edge forms a branch in the current tree.

‘R’ = Rejected edge - edge was by the current tree since it
joins another node of the same fragment.

‘U’ = Usable edge (Basic edge). This edge will be checked to
see if it’s the best edge for this node.

‘X’ = Edge was used by an earlier tree. An edge marked as X
will not be used in current tree.

‘S’ = Usable edge (Special case – bridge edge)

At start, edge arrays at each node are initialized so that
every element in it is marked as ‘U’. This allows the first run
of GHS to choose any edge it wants. Thus in a way, the first
tree is a true MST for that graph. During the formation of the
kth tree, a node will mark the kth element of an edge array as
‘B’ if the corresponding edge is being used (as a branch) in
that tree. If certain conditions are met, then the node marks the
k+1th and higher element of that edge as ‘X’. This is done so
that that edge is not reused in later trees. When a node is in
the n (n>k) GHS execution (formation of Tn), it will use an
edge only if it finds that the nth element of that edge array
marked as ‘U’. As explained earlier, certain edges are allowed
to overlap, in which case, the higher elements of those edges
are not marked as ‘X’. As the algorithm progresses more trees
are found and some of the previously used edges (elements in
the edge array) are marked as ‘X’, making them unavailable in
subsequent trees. There are two rule that determine which
used edge not to be marked as ‘X’:

i. If the edge weight is below the minimum weight threshold,
it may be shared by more than one multicast tree. In this
case, a node will not mark higher elements of a recently
used edge as unavailable (‘X’) for subsequent trees. Instead
these elements will remain as ‘U’ thus making them
available to the following GHS runs. However, if there is a
bound on the number if edges that can be shared (for
example a 3-edge overlapping multiple multicast trees),
then a node can mark higher elements in the array as ‘X’ if
this bound is reached.

ii. If the edge is a bridge connecting two components of the
graph (e.g. Figure 5), this edge will be shared amongst all
the multicast trees. A bridge edge is hard to detect. Section
D explains how this condition is detected and handled.

In order to lower queuing delay on shared edges (and also
to improve resource sharing), nodes in MEST will not re-use
an edge if they have the option of choosing an un-used edge
that falls below the threshold. This also gives rise to the
possibility that the new trees that are generated are edge-
disjoint. For example, if the cut-off threshold in Figure 4 was
set to 6, MEST could generate edge-disjoint trees similar to
Figure 4c.

There are no bounds on the number of trees that can be
generated using MEST for implementing a particular
multicast. The algorithm tries its best to select different set of
edges in every iteration (each tree). However, it should be
noted that beyond a certain point, there is no additional
improvement in the resource utilization since all the edges
have been used in at least one tree. Beyond a certain point,
additional trees will be reusing edges used in at least one tree
generated in an earlier iteration. This could possibly increasing
the delay due to congestion along the edges that are most used.

C. MEST Messages

Most of the message passing between the nodes is similar to
the one used in GHS algorithm except for some minor
modifications as explained here. Nodes in GHS belong to one
of the three states: Sleeping state, Find state and Found state.
A node is in the Sleeping state until it starts executing the
protocol or is awaken by a message from another node. The
two nodes adjacent to the core change their state to Find
and start a new iteration by broadcasting an initiate message
along the branches of the fragment. Upon receiving the initiate
message, a node enters the Find state and forwards the initiate
message to its neighbors (in the fragment). The initiate
message thus propagates to all the nodes in the fragment. The
initiate message contains the fragid and the fragment level. A
node in Find mode updates its fragid and level number to the
one in the initiate message. Next, the node picks its minimum
weight edge incident on a node in another fragment. To check
if an edge is going to a node in a different fragment, a node
sends a test message (containing the node’s fragid and level
number) across the edge. A node receiving a test message
responds back with an ‘accept’ or a ‘reject’ message
depending on whether it belongs to the same or different
fragment as the sending node.

Once a node identifies its ‘best edge’, it sends a report
message towards the fragment’s core. This report message
contains the id and weight of its best edge. Interior nodes hold
back their report message till it has received reports from all
of its child nodes. The internal node’s report contains the
smallest of its best edge or the outbound fragment branch on
which has the minimum best edge was found. There are two
interesting things to note. If a node is a leaf node, then its
report message contains the basic edge weight as infinity. If
no node has outgoing edge, then the algorithm is complete and
the fragment is an MST of the graph. After the two nodes
adjacent to the core have exchanged the report messages, the
mwoe for that iteration is determined. A Change-core message
is then sent over the branches of the fragment to indicate the
new core. A connect message is exchanged by the two
fragments over the mwoe before moving to the next level.

In the original GHS algorithm, a single degree node
responds back with a report message containing the ‘best
edge’ weight as infinity. In GHS, the node’s parent doesn’t do much with this information since its primary interest is in finding the smallest of its base edge or outgoing branch containing the smallest edge. In MEST, however, a node is able to examine only the edges that are marked usable (i.e. ‘U’). A node may finds that it is a single degree node for a particular run of GHS. This may be true for other trees (i.e. there might be other edges connecting that node to the rest of the graph). In MEST, a node will send negative infinity in its report message if it detects that it truly is a single degree node for the entire graph. Since there is only one edge connecting it to the rest of the graph, this edge has to be included in all the trees. When the node receives a negative infinity in a report from its child node, it will not mark all the elements in the edge array as ‘S’; thus making it available to subsequent trees.

D. Bridge Condition

In MEST, we introduce a fourth state called Completed state. This state is reached when a GHS run terminates, i.e., when no node has any outgoing edge for that fragment. In plain GHS, this condition is reached when all the nodes have been found (i.e. a multicast tree for the graph has been found). In MEST however, since not all edges get examined, bridge edges might be missed leading to one or more fragments that have reached the Completed state. When a fragment attains the Completed state, the nodes adjacent to the core broadcast an override message along the branches of the fragment. Upon receiving an override message, a node checks to see if any of its unavailable edges connect to a different fragment. This check is carried out in the same manner as in GHS - by sending a test message (containing the node’s fragid and level number) across the edge. Upon receiving a test message, the node on the other side replies back with a reject (same fragment) or accept (different fragment) message. An override message from the core is replied back with an override response message. When the node receives an accept message from the other side, it sends the weight of the outgoing edge along with the response. In case of a reject message, it replies with infinity as the weight. If several override response messages are received by the core nodes, they pick the edge that has the smallest weight.

E. MEST Example

![Figure 6: MEST example](image)

We illustrate the working of MEST by using Figure 6 (slight modification of Figure 4) as an example. The weight threshold for reusing edges is set to 5. At the end of the first GHS run, the edge array entries at various nodes are as shown in Table I. Edge BC is marked special, which edge DE and EF are re-usable in the second tree since they fall below the threshold. Table II shows the array entries after the second tree has been constructed. Although this is a simple example showing two multicast trees, in real network scenarios we may have a more complex graph with higher connectivity that gives several multicast trees.

V. Simulations

Simulations were carried out using Network Simulation (ns2.26) [1]. As of this writing, ns2 doesn’t have any extension for simulating overlay multicast in MANETs. With the help of C-programming and bash/tcl scripting, the traffic pattern generated by CMU’s cbrgen utility was modified to represent an overlay network. Additional modules were written to simulate our MEST algorithm (for multiple trees) and the Prim’s algorithm [7] (for generating a single MST).

In all the result graphs, MST corresponds to the results for a single tree multicast in which the multicast spanning tree is a minimum spanning tree built using Prim’s algorithm. The setdest utility was used to generate different node positions and movement patterns. The nodes in the simulation move according to the ‘random waypoint’ model [13]. Table III shows the default simulation parameters. In the following sub-sections, simulation parameters when not mentioned follow the one given in this table. All the simulation results are average of 25 random scenarios.

<table>
<thead>
<tr>
<th>Table III: Simulation Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation Area</td>
</tr>
<tr>
<td>Total number of nodes</td>
</tr>
<tr>
<td>Number of member nodes</td>
</tr>
<tr>
<td>Packet size</td>
</tr>
<tr>
<td>Total File size</td>
</tr>
<tr>
<td>Ad-hoc Routing Protocol</td>
</tr>
<tr>
<td>Pause Time</td>
</tr>
<tr>
<td>Max speed</td>
</tr>
</tbody>
</table>

![Image](image)
A. Threshold vs Latency/Leaves

Simulation results from Figure 7 show that there is a significant reduction in the multicast latency along with better utilization (fewer leaves) of the available network resources. Part of this improvement can be attributed to the parallel transmission over multiple trees. A close examination reveals that for this setup, the optimal threshold value is 75m. At 75m, the number of leaves in the overall multicast was minimum while the delay was near minimum. Another observation was that the performance curves tend to ‘flatten-out’ for higher threshold values. This is because at higher thresholds, more edges are allowed to overlap. When an edge is used in multiple trees, it encounters congestion delays since it has to handle packets for multiple trees. This delay is insignificant when the edge weight is small. However, when the threshold is high, longer edges are also allowed to overlap. In such cases, the congestion delays are comparable with the edge delays and hence, for higher threshold, the delay starts to increase. In the worst case, when the delay is set to infinity, all the MEST trees would correspond to the same tree.

B. Overlap Index

As seen from Figure 8, the edge overlap increased with higher number of trees. This means that there would be more edges that are being re-used in multiple trees. This could potentially create congestion issues since the edge has to ‘serve’ several trees. We also observed a slight increase in the overlapping as the threshold value increases. This is expected as higher thresholds allow more edges to overlap. Also Figure 8a & b show that there is a slight increase in the overlap index as the number of member nodes increases.

VI. CONCLUSION AND FUTURE WORK

We present MEST, a distributed algorithm for building multiple multicast trees in ad hoc environment. MEST is easy to implement and can work with any distributed algorithm for generating minimum spanning trees. Our simulation results show that MEST can significantly reduce the multicast delay and at the same time improve the network utilization. Although this paper describes MEST with reference to overlay multicast in MANETs, it should be noted however, that the concept of MEST can be easily extended to multicast in wired networks by considering stationary nodes or network layer (IP) multicast in MANETs.
In addition, even though the design of MEST was not meant to solve reliability issues, it can also be used to build a reliable mesh network - in which case, the same multicast data will be sent over different sub-trees (i.e. the multicast data won’t be split into smaller fragments).

Our current work was targeted for small group multicast networks. In the future, we will focus our attention in making modifications to MEST so that it can work with a larger (multicast) group size. In our present idea, we have reduced the multicast delay by building low weight spanning trees. In the future, we will attempt to build multicast trees where the source node has a low eccentricity. Such trees would have a low multicast delay since the source node would have a shortest path with every node in tree. Existing algorithms for generating shortest path (low source eccentricity) tree pay little attention to the network utilization issues.

REFERENCES